



Part A - Fall 2017.....	1
Part B - Fall 2017 Checkoffs and Homeworks.....	56
Part C - The Duckietown project .....	70
Part D - Duckumentation documentation.....	77
Part E - Exercises.....	110
Part F - Theory Background.....	130
Part G - Introduction to autonomy.....	137
Part H - Reference Material.....	156
Part I - Operation manual - Duckiebot .....	163
Part J - Duckiebot - DB17-1c configurations .....	282
Part K - Appendix.....	288
Part L - Fall 2017 projects .....	293

## PART A

### Fall 2017



Welcome to the Fall 2017 Duckietown experience.

#### UNIT A-1

### The Fall 2017 Duckietown experience



This is the first time that a class is taught jointly across 3 continents!

There are 4 universities involved in the joint teaching for the term:

- ETH Zürich (ETHZ), with instructors Emilio Frazzoli, Andrea Censi, Jacopo Tani.
- University of Montreal (UdeM), with instructor Liam Paull.
- TTI-Chicago (TTIC), with instructor Matthew Walter.
- National Chiao Tung University (NCTU), with instructor Nick Wang.

This part of the Duckiebook describes all the information that is needed by the students

of the four institutions.

At ETHZ, UdeM, TTIC, the class will be more-or-less synchronized. The materials are the same; there is some slight variation in the ordering.

Moreover, there will be some common groups for the projects.

The NCTU class is undergraduate level. Students will learn slightly simplified materials. They will not collaborate directly with the other classes.

## 1.1. The rules of Duckietown

### The first rule of Duckietown

The first rule of Duckietown is: you don't talk about Duckietown, *using email*.

Instead, we use a communication platform called Slack.

There is one exception: inquiries about “meta” level issues, such as course enrollment and other official bureaucratic issues can be communicated via email.

### The second rule of Duckietown

The second rule of Duckietown is: be kind and respectful, and have fun.

### The third rule of Duckietown

The third rule of Duckietown is: read the instructions carefully.

Do not blindly copy and paste.

Only run a command if you know what it does.

## UNIT A-2

## First Steps in Duckietown

### 2.1. Onboarding Procedure

Welcome aboard! We are so happy you are joining us at Duckietown!

This is your onboarding procedure. Please read all the steps and then complete all the steps.

If you do not follow the steps in order, you will suffer from unnecessary confusion.

#### 1) Github sign up

If you don't already have a Github account, sign up now.

→ [Github signup page](#)

Please use your full name when it asks you. Ideally, the username should be something like `FirstLast` or something that resembles your name.

When you sign up, use your university email. This allows to claim an educational discount that will be useful later.

## 2) Questionnaire

---

Taiwan For NCTU Students, complete this form:

[NCTU Student Questionnaire](#)

NCTU: Please complete by 10/30

For ETHZ, UdeM and TTIC fill in this [Preliminary Student Questionnaire](#).

Zurich: Please fill in questionnaire by Tuesday, September 26, 15:00 (extended from original deadline of 12:00).

**Point of contact:** if you have problems with this step, please contact Jacopo Tani <[tanj@ethz.ch](mailto:tanj@ethz.ch)>.

## 3) Accept invite to Github organization Duckietown

---

After we receive the questionnaire, we will invite you to the Duckietown organization. You need to accept the invite; until you do, you are not part of the Duckietown organization and can't access our repositories.

The invite should be waiting for you [at this page](#).

## 4) Accept the invite to Slack

---

After we receive the questionnaire, we will invite you to Slack.

The primary mode of online confabulation between staff and students is Slack, a team communication forum that allows the community to collaborate in making Duckietown awesome.

(Emails are otherwise forbidden, unless they relate to a private, university-based administrative concern.)

We will send you an invite to Slack. Check your inbox.

If after 24 hours from sending the questionnaire you haven't received the invite, contact HR representative Kirsten Bowser <[akbowser@gmail.com](mailto:akbowser@gmail.com)>.

**What is Slack?** More details about Slack are available [here \(master\)](#). In particular, remember to disable email notifications.

**Slack username.** When you accept your Slack invite, please identify yourself with first and last names followed by a “-” and your institution.

*example*

Andrea Censi - Zurich

**Slack picture.** Please add a picture (relatively professional, with duckie accessories encouraged).

**Slack channels.** A brief synopsis of all the help-related Slack channels is here: [Unit A-10 - Slack Channels](#).

Check out all the channels in Slack, and add yourself to those that pertain or interest you. Be sure to introduce yourself in the General channel.

## 2.2. (optional) Add Duckietown Engineering Linkedin profile

This is an optional step.

If you wish to connect with the Duckietown alumni network, on LinkedIn you can join the company “Duckietown Engineering”, with the title “Vehicle Autonomy Engineer in training”. Please keep updated your LinkedIn profile with any promotions you might receive in the future.

## 2.3. Laptops

If you do not have access to a laptop that meets the following requirements, please post a note in the channel `#help-laptops`.

You need a laptop with these specifications:

- Linux Ubuntu 16.04 installed natively (dual boot), not in a virtual machine. See [Subsection 2.3.1 - Can I use a virtual machine instead of dual booting?](#) below for a discussion of the virtual machine option.
- A WiFi interface that supports 5 GHz wireless networks. If you have a 2.4 GHz WiFi, you will not be able to comfortably stream images from the robot; moreover, you will need to adapt certain instructions.
- Minimum 50 GB of free disk space in addition to the OS. Ideally you have 200 GB+. This is for storing and processing logs.

There are no requirements of having a particularly good GPU, or a particularly good CPU. You will be developing code that runs on a Raspberry PI. Any laptop bought in the last 3 years should be powerful enough. However, having a good CPU / lots of RAM makes it faster to run regression tests.

### 1) Can I use a virtual machine instead of dual booting?

Running things in a virtual machine is possible, but **not supported**.

This means that while there is a way to make it work (in fact, Andrea develops in a VMWare virtual machine on OS X), we cannot guarantee that the instructions will

work on a virtual machine, and, most importantly, the TAs will *not* help you debug those problems.

The issues that you will encounter are of two types.

- There are performance issues. For example, 3D acceleration might not work in the virtual machine.
- Most importantly, there are network configuration issues. These come up late in the class, when you start connecting the laptop to the Duckiebot. At that point, ROS makes certain assumptions about subnets, that might not be satisfied by your virtual machine configuration. At that point, you need to be relatively skilled to fix it.

So, the required skill here is not “being able to install Ubuntu on a virtual machine”, but rather “Being able to debug network problems involving multiple real/virtual networks and multiple real/virtual adapters”.

Here’s a quiz: do these commands look familiar to you?

```
$ route add default gw 192.168.1.254 eth0
$ iptables -A FORWARD -o eth1 -j ACCEPT
```

If so, then things will probably work ok for you.

Otherwise, we strongly suggest that you use dual booting instead of a virtual machine.

## 2.4. Next steps for people in Zurich

### 1) Get acquainted with class journal and class logistics

---

At this point, you should be all set up, able to access our Github repositories, and, most important of all, able to ask for help on Slack.

You can now get acquainted to the class journal, to know the next steps.

→ [Unit A-11 - Zürich branch diary](#)

Also, in this page, we will collect the logistics information (lab times, etc.).

→ [Unit A-3 - Logistics for Zürich branch](#)

### 2) Make sure you can edit the Duckuments

---

To receive your Duckiebox on Wednesday Sep 27, you need to prove to be able to edit the Duckuments successfully.

→ See the instructions [in this section](#).

If you can’t come on Wednesday, please contact one of the TAs.

## 2.5. Next steps for people in Chicago

TODO: to write

### UNIT A-3 Logistics for Zürich branch

This section describes information specific to Zürich.

#### 1) The local staff

---

These are the local TAs:

- Shiying Li (shili@student.ethz.ch)
- Ercan Selçuk (ercans@student.ethz.ch)
- Miguel de la Iglesia Valls (dmiguel@student.ethz.ch)
- Harshit Khurana (hkhurana@student.ethz.ch)
- Dzenan Lapandic (ldzenan@student.ethz.ch)
- Marco Erni (merni@ethz.ch)

Please contact them on Slack, rather than email.

Also feel free to contact the TAs in Montreal and Chicago.

## 3.1. HR

Feel free to contact Ms. Kirsten Bowser (akbowser@gmail.com) if you have problems regarding accounts, permissions, etc.

## 3.2. Website / class journal

During the term, we are not going to update the website.

Rather, all important information, such as deadlines, is in the [class journal](#).

→ [Unit A-11 - Zürich branch diary](#)

## 3.3. Duckiebox

The point of contact for Duckiebox distribution is Shiying Li.

## 3.4. Duckietown room access

The local Duckietown room is ML J 44.2.

**TODO:** write opening hours and rules

+ comment

double check the room number -AC

### 3.5. Extra spaces

There will be extra lab space available.

Space-time coordinates TBD.

## UNIT A-4

### Logistics for Montréal branch

This unit contains all necessary info specific for students at Université de Montréal.

#### 4.1. Website

[This is the official course website](#). It contains links to the syllabus and description and other important info.

#### 4.2. Class Schedule

The authoritative class schedule will be tracked in [Unit A-12 - Montréal branch diary](#). This will contain all lecture material, homeworks, checkoffs, and labs.

#### 4.3. Lab Access

The lab room for the class is 2333 in Pavillion André-Aisenstadt. The code for the door is XXX. Please do not distribute the code for the door, we are trying to limit access to this room as much as possible.

#### 4.4. The Local Staff

The TA for the class is Florian Golemo. All communications with the course staff should happen through Slack.

The instructor is Prof. Liam Paull, whose office is 2347 Pavillion André-Aisenstadt.

#### 4.5. Storing Your Robot

It is preferable that you keep your robot for the semester. However, if you do not

have a secure location where you can store it, we can store it for you in Room XXX in Pavillion André-Aisenstadt. However, you will have to ask Prof. Liam Paull to access or store your robot there each time since we cannot give out access to this space to the students in the class.

## UNIT A-5

# Logistics for Chicago branch

Assigned to: Matt

This section describes information specific to TTIC and UChicago students.

### 1) Website

---

The [course website](#) provides a copy of the syllabus, grading information, and details on learning objectives.

### 2) Class Schedule

---

Classes take place on Mondays and Wednesdays from 9am-11am in TTIC Room 530. In practice, each class will be divided into an initial lecture period (approximately one hour), followed by a lab session.

The class schedule is maintained as part of the [TTIC Class Diary](#), which includes details on lecture topics, links to slides, etc.

### 3) Course Grading

---

The following is taken from the [course syllabus](#):

The class will assess your grasp of the material through a combination of problem sets, exams, and a final project. The contribution of each to your overall grade is as follows:

- 20%: Problem sets
- 10%: Checkoffs
- 20%: Participation
- 50%: Final project (includes report and presentation). The projects will be group-based, but we will assess the contribution of each student individually.

See the [course syllabus](#) for more information on how the participation and final project grades are determined.

### 4) Policy on Late Assignments and Collaboration

---

The following is taken from the [course syllabus](#):

Late problem sets will be penalized 10% for each day that they are late. Those submitted more than three days beyond their due date will receive no credit.

Each student has a budget of three days that they can use to avoid late penalties. It is up to the student to decide when/how they use these days (i.e., all at once or individually). Students must identify whether and how many days they use when they submit an assignment.

It is not acceptable to use code or solutions from outside class (including those found online), unless the resources are specifically suggested as part of the problem set.

You are encouraged to collaborate through study groups and to discuss problem sets and the project in person and over Slack. However, you must acknowledge who you worked with on each problem set. You must write up and implement your own solutions and are not allowed to duplicate efforts. The correct approach is to discuss solution strategies, credit your collaborator, and write your solutions individually. Solutions that are too similar will be penalized.

## 5) Lab Access

---

Duckietown labs will take place at TTIC in the robotics lab on the 4th floor.

**Note:** TTIC and U. Chicago students in Matthew Walter's research group use the lab as their exclusive research and office space. It also houses several robots and hardware to support them. Please respect the space when you use it: try not to distract lab members while they are working and please don't touch the robots, sensors, or tools.

## 6) The Local LAs

---

Duckietown is a collaborative effort involving close interaction among students, TAs, mentors, and faculty across several institutions. The local learning assistants (LAs) at TTIC are:

- Andrea F. Daniele (afdaniele@ttic.edu)
- Falcon Dai (dai@ttic.edu)
- Jon Michaux (jmichaux@ttic.edu)

## UNIT A-6

### Logistics for NCTU branch

Assigned to: Nick and Eric (Nick's student)

This section describes information specific to NCTU students.

#### 1) Website

---

The [Duckietown Taiwan Branch Website](#) provides some details about Duckietown Branch in NCTU-Taiwan and results of previous class in NCTU.

#### 2) Class Schedule

---

Classes take place on Thursday from 1:20pm~4:20pm in NCTU Engineering Building 5 Room 635. Each class will be divided into two sessions. In the first session, Professor Wang will give lessons on fundamental theory and inspire students to come up with more creative but useful ideas on final projects. In the second session, TAs will give practical lab on how to use Duckietown platform as their project platform and use ROS as their middleware toward a fantastic work.

The class schedule is maintained as part of the [NCTU Class Diary](#), which includes details on lecture topics, links to slides, etc.

### 3) Course Grading

---

The following is taken from the [course syllabus](#):

This course aims at developing software projects usable in real-world, and focuses on “learning by doing,” “team work,” and “research/startup oriented.”. The contribution of each to your overall grade is as follows:

- Class Participation, In Class Quiz, Problem Sets (10%)
- Midterm Presentation (30%)
- Final Presentation (30%)
- Project Report and Demo Video (30%)

See the [course syllabus](#) for more information on course object and grading policy.

### 4) Policy on Late Assignments and Collaboration

---

The following is taken from the [course syllabus](#):

Late problem sets will be penalized 10% for each day that they are late. Those submitted more than three days beyond their due date will receive no credit.

Each student has a budget of three days that they can use to avoid late penalties. It is up to the student to decide when/how they use these days (i.e., all at once or individually). Students must identify whether and how many days they use when they submit an assignment.

It is not acceptable to use code or solutions from outside class (including those found online), unless the resources are specifically suggested as part of the problem set.

You are encouraged to collaborate through study groups and to discuss problem sets and the project in person and over Slack. However, you must acknowledge who you worked with on each problem set. You must write up and implement your own solutions and are not allowed to duplicate efforts. The correct approach is to discuss solution strategies, credit your collaborator, and write your solutions individually. Solutions that are too similar will be penalized.

### 5) Lab Access

---

Duckietown labs will take place at NCTU in the same place with the lecture.

**Note:** The course focus on “learning by doing” which means that the lab session of each class is especially important.

## 6) The Local LAs

---

Duckietown is a collaborative effort involving close interaction among students, TAs, mentors, and faculty across several institutions. The local learning assistants (LAs) at NCTU are:

- Yu-Chieh ‘Tony’ Hsiao (tonycar12002@gmail.com)
- Pin-Wei ‘David’ Chen (ccpwearth@gmail.com)
- Chen-Lung ‘Eric’ Lu (eric565648@gmail.com)
- Yung-Shan ‘Michael’ Su (michael1994822@gmail.com)
- Chen-Hao ‘Peter’ Hung (losttime1001@gmail.com)
- Hong-Ming ‘Peter’ Huang (peterx7803@gmail.com)
- Tzu-Kuan ‘Brian’ Chuang (fire594594594@gmail.com)

## UNIT A-7

# Git usage guide for Fall 2017

## 7.1. Differences

There are some difference among the branches. These will be marked using the following graphical notation.

- ?? | This is only for Zurich.
- ?? | This is only for Montreal.
- ?? | This is only for Chicago.
- Taiwan | This is only for Taiwan.

## 7.2. Repositories

These are the repositories we use.

### 1) Software

---

The [Software](#) repository is the main repository that contains the software.

The URL to clone is:

```
git@github.com:duckietown/Software.git
```

In the documentation, this is referred to as `DUCKIETOWN_ROOT`.

During the first part of the class, you will only read from this repository.

## 2) Duckiefleet

---

The [duckiefleet](#) repository contains the data specific to this instance of the class.

The URL to clone is:

```
git@github.com:duckietown/duckiefleet.git
```

In the documentation, the location of this repo is referred to as `DUCKIEFLEET_ROOT`.

You will be asked to write to this repository, to update the robot DB and the people DB, and for doing exercises.

## 3) exercises-fall2017

---

For homework submissions, we will use the following URL:

```
git@github.com:duckietown/exercises-fall2017.git
```

As explained below, it is important that this repo is kept separate so that students can privately work on their exercises at schools where the homeworks are counted for grades.

## 4) Duckuments

---

The [Duckuments](#) repository is the one that contains this documentation.

The URL to clone is:

```
git@github.com:duckietown/duckuments.git
```

Everybody is encouraged to edit this documentation!

In particular, feel free to insert comments.

## 5) Lectures

---

The [lectures](#) repository contains the lecture slides.

The URL to clone is:

```
git@github.com:duckietown/lectures.git
```

Students are welcome to use this repository to get the slides, however, please note that

this is a space full of drafts.

## 6) Exercises

---

The [exercises](#) repository contains the solution to exercises.

The URL to clone is:

```
git@github.com:duckietown/XX-exercises.git
```

Only TAs have read and write permissions to this repository.

### 7.3. Git policy for homeworks (TTIC/UDEM)

?? | This does not apply to Zurich.

Homeworks will require you to write and submit coding exercises. They will be submitted using git. Since we have a university plagiarism policy ([UdeM's](#), [TTIC/UChicago](#)) we have to protect students work before the deadline of the homeworks. For this reason we will follow these steps for homework submission:

1. Go [here](#) and file a request at the bottom “Request a Discount” then enter your institution email and other info.
2. Go to [exercises-fall2017](#)
3. Click “Fork” button in the top right
4. Choose your account if there are multiple options
5. Click on the Settings tab of your repository, not your account
6. Under “Collaborators and Teams” in the left, click the “X” in the right for the section for “Fall 2017 Vehicle Autonomy Engineers in training”. You will get a popup asking you to confirm. Confirm.

If you have not yet cloned the duckietown repo do it now:

```
$ git clone git@github.com:duckietown/exercises-fall2017.git
```

Now you need to point the remote of your `exercises-fall2017` to your new local private repo. To do, from inside your already previously cloned `exercises-fall2017` repo do:

```
$ git remote set-url origin git@github.com:GIT_USERNAME/exercises-fall2017.git
```

Let's also add an `upstream` remote that points back to the original duckietown repo:

```
$ git remote add upstream git@github.com:duckietown/exercises-fall2017.git
```

If you type

```
$ git remote -v
```

You should now see:

```
origin git@github.com:GIT USERNAME/exercises-fall2017.git (fetch)
origin git@github.com:GIT USERNAME/exercises-fall2017.git (push)
upstream git@github.com:duckietown/exercises-fall2017.git (fetch)
upstream git@github.com:duckietown/exercises-fall2017.git (push)
```

Now the next time you push (without specifying a remote) you will push to your local private repo.

## 1) Duckiefleet file structure

You should put your homework files in folder at:

```
DUCKIEFLEET HOMEWORK ROOT/homeworks/XX homework name/YOUR ROBOT NAME
```

Some homeworks might not require ROS, they should go in a subfolder called `scripts`. ROS homeworks should go in packages which are generated using the process described here: [Minimal ROS node - pkg\\_name \(master\)](#). For an example see **DUCKIEFLEET HOMEWORK ROOT**/homeworks/01\_data\_processing/shamrock .

**Note:** To make your ROS packages findable by ROS you should add a symlink from your **DUCKIEFLEET HOMEWORK ROOT** to `duckietown/catkin_ws/src`.

## 2) To submit your homework

When you are ready to submit your homework, you should do **create a release** and **tag the Fall 2017 instructors/TAs group** to let us know that your work is complete. This can be done through the command line or through the github web interface:

Command line:

```
$ git tag XX homework name -m"@duckietown/fall-2017-instructors-and-tas homework complete"
$ git push origin --tags
```

Through Github:

1. Click on the “Releases” tab.
2. Click “Create a new Release”.
3. Add a version (e.g. 1.0).
4. Release title put **XX homework name**.
5. In the text box put “@duckietown/fall-2017-instructors-and-tas homework complete”.

6. Click “Publish release”.

You may make as many releases as you like before the deadline.

### 3) Merging things back

---

Once all deadlines have passed for all institutions, we can merge all the homework. We will ask to create a “Pull Request” from your private repo.

1. In your private `exercises-fall2017` repo, click the “New pull request button”.
2. Click “Create pull request” green button
3. The 4 drop down menus at the top should be left to right: (`base fork: duckietown/exercises-fall2017`, `base: master`, `head fork: YOUR_GIT_NAME/exercises-fall2017`, `compare: YOUR_BRANCH`)
4. Leave a comment if you like and click “Create pull request” green button below.
5. At some point a TA or instructor will either merge or leave you a comment.

### 4) For U de M students who have already submitted homework to the previous duckiefleet-2017 repo

---

?? These instructions assume that you are ok with losing the commit history from the first homework. If not, things get a little more complicated

Fork and clone the new “homework” repository using the process above. Followed by:

```
$ git clone git@github.com:GIT_USERNAME/exercises-fall2017.git
```

Copy over your homework files from the `duckiefleet-fall2017` repo into the `exercises-fall2017` repo.

```
git rm your folder from duckiefleet-fall2017 and commit and push.
```

```
git add your folder to exercises-fall2017 and commit and push.
```

Clone the new duckiefleet repo

```
$ git clone git@github.com:duckietown/duckiefleet.git
```

Update the symlink you created in your duckietown repo

```
$ ln -sf EXERCISES_FALL2017/homeworks $DUCKIETOWN_ROOT/catkin_ws/src/name-of-the-symlink
```

## 7.4. Git policy for project development

Different than the homeworks, development for the projects will take place in the

**Software** repo since plagiarism is not an issue here. The process is:

1. Create a branch from master
2. Develop code in that branch (note you may want to branch your branches. A good idea would be to have your own “`master`”, e.g. “`your_project-master`” and then do pull requests/merges into that branch as things start to work)
3. At the end of the project submit a pull request to master to merge your code. It may or may not get merged depending on many factors.

## UNIT A-8

# Organization

The following roster shows the teaching staff.

Andrea Censi



Liam Paull



Jacopo Tani



First Last



Emilio Fazzoli



First Last



First Last



First Last



First Last



First Last



First Last



First Last



First Last



First Last



First Last



Greta Paull



Kirsten Bowser



Staff: To add yourself to the roster, or to change your picture, add a YAML file and a jpg file to [the duckiefleet-fall2017 repository](#). in the [people/staff directory](#).

## 8.1. The Activity Tracker



### [Link to Activity Tracker](#)

The sheet called “Activity Tracker” describes specific tasks that you must do in a certain sequence. Tasks include things like “assemble your robot” or “sign up on Github”.

The difference between the Areas sheet and the Task sheet is that the Task sheet contains tasks that you have to do once; instead, the Areas sheet contains ongoing activities.

In this sheet, each task is a row, and each person is a column. There is one column for each person in the class, including instructors, TAs, mentors, and students.

You have two options:

- Only use the sheet as a reference;
- Use the sheet actively to track your progress. To do this, send a message to Kirsten with your gmail address, and add yourself.

Each task in the first column is linked to the documentation that describes how to perform the task.

The colored boxes have the following meaning:

- Grey: not ready. This means the task is not ready for you to start yet.
- Red: not started. The person has not started the task.
- Blue: in progress. The person is doing the task.
- Yellow: blocked. The person is blocked.
- Green: done. The person is done with the task.
- n/a: the task is not applicable to the person. (Certain tasks are staff-only.)

## 8.2. The Areas sheet



Please familiarize yourself with [this spreadsheet](#) and bookmark it in your browser.

The sheet called “Areas” describes the points of contact for each part of this experience. These are the people that can offer support. In particular, note that we list two

points of contact: one for America, and one for Europe. Moreover, there is a link to a Slack channel, which is the place where to ask for help. (We'll get you started on Slack in just a minute.)

## UNIT A-9

# Getting and giving help

## 9.1. Who to ask for help

### 1) Primary points of contact

---

The organization chart ([Section 8.2 - The Areas sheet](#)) lists the primary contact for each area.

### 2) Point of contacts for specific documents

---

Certain documents have specific points of contacts, listed at the top. These override the listing in the organization chart.

## 9.2. How to ask for help

The ways that we will support each other will depend on the type of situation. Here we will enumerate the different cases. Try to figure out which case is the most appropriate and act accordingly. These are ordered roughly in order of increasing severity.

### 1) Case: You find a mistake in the documentation

---

Action: Please fix it.

The goal for the instructions is that anybody is able to follow them. Last year, we managed to have two 15-year-old students reproduce the Duckiebot from instructions.

- How to edit the documentation is explained in [Part D - Duckumentation documentation](#). In particular, the notation on how to insert a comment is explained in [Section 3.3 - Notes and questions](#).

Note that because we use Git, we can always keep track of changes, and there is no risk of causing damage.

If you encounter typos, feel free to edit them directly.

Feel free to add additional explanations.

One thing that is very delicate is dealing with mistakes in the instructions.

A few times the following happened: there is a sequence of commands `cmd1;cmd2;cmd3` and `cmd2` has a mistake, and `cmd2b` is the right one, so that the sequence of commands

is `cmd1;cmd2b;cmd3`. In those situations we first just corrected the command `cmd2`.

However, that created a problem: now half of the students had used `cmd1;cmd2;cmd3` and half of the students had used `cmd1;cmd2b;cmd3`: the states had diverged. Now chaos might arise, because there is the possibility of “forks”.

Therefore, if a mistaken instruction is found, rather than just fixing the mistake, please add an addendum at the end of the section.

For example: “Note that instruction `cmd2` is wrong; it should be `cmd2b`. To fix this, please enter then command `cmd4`.”

Later, when everybody has gone through the instructions, the mistake is fixed and the addendum is deleted.

## 2) Case: You find the instructions unclear and you need clarification

---

Action: Ask for clarification on the appropriate Slack channel. For a list of slack channels that could be helpful see [Unit A-10 - Slack Channels](#). Once the ambiguity is clarified to your satisfaction, either you or the appropriate staff member should update the documentation if appropriate. For instructions on this see [Part D - Duckumentation documentation](#).

## 3) Case: You understand the instructions but you are blocked for some reason

---

Action: This is more serious than the previous. Open an issue [on the duckiefleet-fall2017 github page](#). Once the issue is resolved, either you or the appropriate staff member should update the documentation if appropriate. For instructions on this see [Part D - Duckumentation documentation](#).

## 4) Case: You are having a technical issue related to building the documentation

---

Action: Open an issue [on the duckuments github page](#) and provide all necessary information to reproduce it.

## 5) Case: You have found a well-defined defect in the software.

---

Action: open an issue [on the Software repository github page](#) and provide all necessary information for reproducing the bug.

## UNIT A-10

### Slack Channels

This page describes all of the helpful Slack channels and their purposes so that you can figure out where to get help.

## 10.1. Channels

You can also easily join the ones that you are interested in by [clicking the links in this message](#).

TABLE 10.1. DUCKIETOWN SLACK CHANNELS

Channel	Purpose
help-accounts	Info about necessary accounts, such as Slack, Github, etc.
help-assembly	Help putting your robot together
help-camera-calib	Help doing the intrinsic and extrinsic calibration of your camera
help-duckuments	Help compiling the online documentation
help-git	Help with git
help-infrastructure	Help with software infrastructure, such as Makefiles, unit tests, continuous integration, etc.
help-laptops	Help getting your laptop setup with Ubuntu 16.04
help-parts	Help getting the parts for the robot or replacement parts if you broke something
help-robot-setup	Help getting the robot setup to do basic things like be driven with a joystick
help-ros	Help with the Robot Operating System (ROS)
help-wheel-calib	Help doing your odometry calibration

+ comment

Note that we can link directly to the channels. (See list in the org sheet.) -AC

## UNIT A-11

### Zürich branch diary

## 11.1. Lectures and lab sessions

Lectures: Mon 13-15 HG F 26.5 Wed 10-12 HG E 22

Lab session: Fri 15-19 ML J 37.1

Duckielab: ML J 44.2

## 11.2. Wed Sep 20: Welcome to Duckietown!

This was an introduction meeting.

### 1) Material presented in class

These are the slides we showed:

- [PDF](#)
- [Keynote \(huge\)](#)

## 2) Feedback form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 3) Pointers to reading materials

---

Read about Duckietown's history and watch the Duckumentary.

- [Part C - The Duckietown project](#)

Start learning about Git and Github.

- [Source code control with Git \(master\)](#)

Montreal, Chicago? What's happening?

- [Unit A-1 - The Fall 2017 Duckietown experience](#)

## 11.3. Monday Sep 25: Introduction to autonomy

### 1) Material presented in class

---

These are the slides we presented:

a - Logistics: [Keynote](#), [PDF](#).

a - Autonomous Vehicles: [Keynote](#), [PDF](#).

c - Autonomous Mobility on Demand: [Keynote](#), [PDF](#).

d - Plan for the next months: [Keynote](#), [PDF](#).

### 2) Feedback form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

### 3) Questions and answers

---

*Q: MyStudies is not updated; I am still on the waiting list. What should I do?*

**Answer:** Nothing. Don't worry, if you have received the onboarding email, you are in the class, even if you still appear in the waiting list. We will figure this out with the department.

*Q: What version of Linux do I need to install?*

**Answer:** 16.04.\* (16.04.03 is the latest at time of this writing)

*Q: Do I need to install OpenCV, ROS, etc.?*

**Answer:** Not necessary. We will provide instructions for those steps.

**Q:** *My laptop is not ready. I'm having problems installing Linux on a partition.*

**Answer:** Don't worry, take a Duckie, and, take a breath. We have time to fix every issue. Start by asking for help in the #help-laptops channel in Slack. We will address the outstanding issues in the next classes.

**Q:** *How much space do I need on my Linux partition?*

**Answer:** At least 50 GB; 200 GB are recommended for easy processing of data (logs) later in the course. If you have less space (say ~100GB), it might be wise to acquire an external hard drive to use as storage.

**Q:** *Are there going to be Linux training sessions?*

**Answer:** Maybe. We didn't plan for it, but it seems that there is a need. Subject to figuring out the logistics, we might organize an extra "lab" session or produce a support video.

## 11.4. Monday Sep 25, late at night: Onboarding instructions

At some late hour of the night, we sent out the onboarding instructions.

→ [Section 2.1 - Onboarding Procedure](#)

Please complete the onboarding questionnaire by Tuesday, September 26, 15:00.

## 11.5. Wednesday Sep 27: Duckiebox distribution, and getting to know each other

Today we distribute the Duckieboxes and we name the robots. In other words, we perform the *Duckiebox ceremony*.

- getting to know each other;
- naming the robots;
- distribute the Duckieboxes.

**Note:** If you cannot make it to this class for the Duckiebox distribution, please inform the TA, to schedule a different time.

### 1) Preparation, step 1: choose a name for your robot

Before arriving to class, you must think of a name for your robot.

Here are the constraints:

- The name must work as a hostname. It needs to start with a letter, contains only letters and numbers, and no spaces or punctuation.
- It should be short, easy to type. (You'll type it a lot.)
- It cannot be your own name.

- It cannot be a generic name like “robot”, “duckiebot”, “car”. It cannot contain brand names.

## 2) Preparation, step 2: prepare a brief elevator pitch

---

As members of the same community, it is important to get to know a little about each other, so to know who to rely on in times of need.

During the Duckiebox distribution ceremony, you will be asked to walk up to the board, write your name on it, and introduce yourself. Keep it very brief (30 seconds), and tell us:

- what is your professional background and expertise / favorite subject;
- what is the name of your robot;
- why did you choose to name your robot in that way.

You will then receive a Duckiebox from our senior staff, a simple gesture but of semi-piternal glory, for which you have now become a member of the Duckietown community. This important moment will be remembered through a photograph. (If in the future you become a famous roboticist, we want to claim it's all our merit.)

Finally, you will bring the Duckiebox to our junior staff, who will apply labels with your name and the name of the robot. They will also give you labels with your robot name for future application on your Duckiebot.

## 3) Feedback form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 4) Material presented in class

---

- Duckiebot parts: [PowerPoint presentation](#), [PDF](#).

### 11.6. Thursday Sep 28: Misc announcements

---

We created the channel `#ethz-chitchat` for questions and other random things, so that we can leave this channel `#ethz-announcements` only for announcements.

We sent the final list to the Department; so hopefully in a couple of days the situation on MyStudies is regularized.

The “lab” time on Friday consists in an empty room for you to use as you wish, for example to assemble the robots together. In particular, it's on the same floor of the Duckietown room and the IDSC lab.

The instructions for assembling the Duckiebots are [here](#). Note that you don't have to do the parts that we did for you: buying the parts, soldering the boards, and reproducing the image.

Expected progress: We are happy if we see everybody reaching up to [Unit I-12 - RC+camera remotely](#) by Monday October 9. You are encouraged to start very early; it's

likely that you will not receive much help on Sunday October 8...

## 11.7. Sep 28: some announcements



A couple of announcements:

1. We created `#ethz-chitchat` for questions and other random things, so that we can leave this channel `#ethz-announcements` only for announcements.
2. MyStudies should be updated with everybody's names.
3. The "lab" time tomorrow consists in an empty room for you to use as you wish, for example to assemble the robots together. In particular, it's on the same floor of the Duckietown room and the IDSC lab.
4. The instructions for assembling the Duckiebots are [here](#). Note that we did for you step I-2 (buying the parts) and I-3 (soldering the boards); and I-6 is optional.
5. We are happy if we see everybody reaching I-13 by the Monday after next. I encourage you to start sooner than later.
6. I see only 30 people in this channel instead of 42. Please tell your friends that now all the action is on Slack.

## 11.8. Oct 01 (Mon): Announcement



It looks like that the current documentation is misleading in a couple of points. This is partially due to the fact that there is some divergence between Chicago, Montreal and Zurich regarding (1) the parts given out and (2) the setup environment (which networks are available). We did the simple changes (e.g. removing the infamous part 6), but we need some more time to review the other issues. At this point, the best course of action is that you enjoy your weekend without working on Duckietown, while we spend the weekend fixing the documentation.

## 11.9. Oct 02 (Mon): Networking, logical/physical architectures



### 1) Materials presented in class

---

- a - Logistics and other information: [Keynote](#), [PDF](#).
- b - Networking [Keynote](#), [PDF](#).
- c - System architecture basics [Keynote](#), [PDF](#).

# DUCKIETOWN WEEKLY NEWS Oct. 2, 2017

## ZÜRICH BRANCH OPENS

Duckies rejoice, hopeful of swift construction of AMOD service.



### Who are the men without the duckie?

Speculations run wild regarding the identity and motives of three men who were in the picture without a duckie on their head.



### Instruction confusion leads to assembling halt

Following the sudden realization that Montréal, Chicago, and Zürich have 3 different robot configurations, assembly and setup has been stopped while the instructions are being updated. Officials are hopeful that there will be a quick convergence of all branches to same configuration.

### Rumors of Chicago and Montréal branches opening

No photographic evidence to be found.

### Duckietown construction started

Construction suddenly halted when crew realized mats were upside down.



### Duckieboxes distributed

Commentators say boxes are observed to be similar to chunks of Swiss cheese. Holes planned in future upgrade.

### Popular vote dictates networking lecture



Non-anonymity of poll said to bias the results. Anonymous polls planned in the future.

### Duckietown critiqued over use of non-free software

Free software advocate Richard Stallman critiques Duckietown over the choice of Github, due to the website's use of non-free software.

Figure 11.1

## 2) Feedback form

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 11.10. Oct 04 (Wed): Modeling

### 1) Materials presented in class

- Modeling of a differential drive vehicle: [PowerPoint presentation](#), [PDF](#).

### 2) Feedback form

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 11.11. Oct 09 (Mon): Autonomy architectures and version control

### 1) Materials presented in class

a - Autonomy Architectures Basics: [Keynote](#), [PDF](#).

b - Version control with Git: [Keynote](#), [PDF](#).

## 11.12. Oct 11 (Wed): Computer vision and odometry calibration

### 1) Materials presented in class

a - Computer Vision Basics: [PDF](#), [PowerPoint presentation](#).

b - Odometry Calibration: [PDF](#), [PowerPoint presentation](#).

## 11.13. Oct 13 (Fri): new series of tasks out

### 1) Taking a video of the joystick control

Please take a video of the robot as it drives with joystick control, as described in [Section 1.7 - Upload your video](#) and upload it according to the instructions.

[Example of a great video, but with a nonconforming Duckiebot.](#)

### 2) Camera calibration

[Go forth and calibrate the camera!](#) And get help in `#help-camera-calib`.

### 3) Wheel calibration

This is not ready yet! will be ready in a day or so.

### 4) Taking a log check off

Follow [the instructions here](#) to learn how to take a log.

### 5) Data processing exercises

See [the list of exercises here](#).

Get help in `#ex-data-processing`.

TABLE 11.1. CURRENT DEADLINES FOR ZURICH

Robot assembly	overdue
Robot/laptop configuration	overdue
<a href="#">Subsection 11.13.1 - Taking a video of the joystick control</a>	Monday Oct 16
<a href="#">Subsection 11.13.2 - Camera calibration</a>	Friday Oct 20
<a href="#">Subsection 11.13.3 - Wheel calibration</a>	not ready yet
<a href="#">Subsection 11.13.4 - Taking a log check off</a>	Wed Oct 18
<a href="#">Subsection 11.13.5 - Data processing exercises</a>	Monday Oct 23

## 11.14. Oct 16 (Mon): Line detection

### 1) Materials presented in class

a - Line detection: [Keynote](#), [PDF](#).

b - Logistics: [Keynote](#), [PDF](#).

## 11.15. Oct 18 (Wed): Feature extraction

### 1) Materials presented in class

- 
- Feature extraction: [Keynote](#), [PDF](#).

## 11.16. Oct 20 (Fri): Lab session

### 1) Materials presented in class

- 
- ROS Main concepts: [PowerPoint presentation](#), [PDF](#).

## 11.17. Oct 23 (Mon) Filtering I

### 1) Materials presented in class

- 
- Lectures on filtering (Filtering I): [PowerPoint presentation](#), [PDF](#).

## 11.18. Oct 25 (Wed) Filtering II

a - Lectures (Particle Filter) [PowerPoint presentation](#), [PDF](#).

b - Lectures (Lane Filter) [PowerPoint presentation](#), [PDF](#).

## 11.19. Nov 1 (Wed) Control Systems

a - Lectures (Control Systems Module I) [PowerPoint presentation](#), [PDF](#).

b - Lectures (Control Systems Module II) [PowerPoint presentation](#), [PDF](#).

Points to be noted - Running what-the-duck on laptop and Duckiebot is mandatory. It helps save time in debugging errors and also is a standard way to ask for help from the staff. Keep repeating it periodically so as to keep the data up-to date - For the people lacking calibrated wheels, this serves as a reminder to calibrate the wheels and keep their duckiebot up-to date - It is advised to fill the lecture feedback form ([Feedback form](#)), so as to increase the effectiveness of the lectures - Always check the consistency of the camera calibration checkerboard before camera calibration (one has to check for the correct square size and correct distance between world and checkerboard reference)

## 11.20. Nov 6 (Mon) Project Pitches

Lecture Project Pitches [PDF](#).

## 11.21. Nov 8 (Wed) Motion Planing

Lecture Motion Planing [PDF](#).

A few references for planning of Andrea Censi:

- The Book on planning is the one by Lavalle, available for free here: <http://planning.cs.uiuc.edu/>.
- The mentioned robot BB-8: <http://starwars.wikia.com/wiki/BB-8>.
- The mentioned movie scene Donnie Darko: <https://www.youtube.com/watch?v=rPeGaos7DB4>.

## 11.22. Nov 13 (Mon) Project Team Assignments

- First Lecture: Project Team Assignments [PDF](#).
- Second Lecture: First meeting of the Controllers group -> Filling out the Preliminary Design Document

## 11.23. Nov 15 (Wed) Putting things together

- First Lecture: Putting things together [PDF](#).
- Second Lecture: Second meeting of the Controllers group -> Filling out the Preliminary Design Document

## 11.24. Nov 20 (Mon) Testing Autonomous Vehicles{#Zurich-2017-11-20}

- First Lecture: Testing Autonomous Vehicles [PDF](#).

## 11.25. Nov 22 (Wed) Fleet Control

- Lecture: Fleet Control in Autonomous Mobility on Demand [PDF](#).

## 11.26. Nov 27 (Mon) Inermediate design Report

- First Lecture: The intermediate Design report is introduced here [template-int-report](#). It is due on Monday 4th of December.
- Second Lecture was left for project discussion and interaction.

## 11.27. Nov 29 (Wed) Fleet Control

- First Lecture: Claudio finished Fleet Control in Autonomous Mobility on Demand [PDF](#).
- Second Lecture Julian Presented the state of the art in data driven vs Model driven robotics. [PDF](#)

## UNIT A-12

## Montréal branch diary

## 12.1. Wed Sept 6

Class (11:30)

Slides:

- Duckietown History Future ([keynote](#)) ([pdf](#))
- Duckietown Intro ([keynote](#)) ([pdf](#))
- Autonomy Overview ([keynote](#)) ([pdf](#))
- Autonomous Vehicles ([keynote](#)) ([pdf](#))

Book materials:

- [Part C - The Duckietown project](#)
- [Unit G-1 - Autonomous Vehicles \[draft\]](#)
- [Unit G-2 - Autonomy overview](#)

## 12.2. Friday Sept 8

Acceptance emails sent

## 12.3. Sun Sept 10

- Onboarding email sent to accepted students

## 12.4. Mon Sept 11

Class (10:30)

| **Note:** This class we are meeting in rm. 2333 Pavillion André Aisenstadt.

- Logistics
- [The Duckiebook](#)
- Slack ([Unit A-10 - Slack Channels](#))
- Git repos ([Unit A-7 - Git usage guide for Fall 2017](#))
- How to get and give help ([Unit A-9 - Getting and giving help](#))
- [Grading scheme](#)
- Student/Staff Introductions
- Duckiebox distribution.
- Go through the Duckiebox parts
- [Unit B-1 - Checkoff: Assembly and Configuration](#) initiated.

Deadline: Mon Sept. 25

## 12.5. Wed Sept 13

Class canceled. Continue working on [Unit B-1 - Checkoff: Assembly and Configuration](#).

## 12.6. Mon Sept 18

Class (10:30 - 11:30)

- General discussion (how are things going? Sorry I was away last week.. anyone need anything?)
- Intro to robotics - Modern robotic systems
- The robot as a system - System architectures
- Decomposing the robotics sytems into smaller pieces - autonomy architectures
- Agreeing on the language that the different pieces “talk” - representations
- Background on basic probability theory?

Slides:

- Modern Robotic Systems ([keynote](#)) ([pdf](#))
- System Architecture ([keynote](#)) ([pdf](#))
- Autonomy Architectures ([keynote](#)) ([pdf](#))
- Representations for Robotics ([keynote](#)) ([pdf](#))

Book Materials:

- [Modern Robotic Systems](#)
- [System Architecture Basics](#) ([master](#))
- [Autonomy Architectures](#)
- [Representations](#)
- [Probability Basics](#)

Lab (11:30 - 12:30)

- Liam will be in 2333 setting up network and building Duckietown.

## 12.7. Wed Sept 20

Class (11:30 - 12:30)

- Robotics middlewares - what are they and basic concepts
- Introduction to the Robot Operating System (ROS)

Slides:

- Software Architectures ([keynote](#)) ([pdf](#))
- Introduction to ROS ([keynote](#)) ([pdf](#))

Book Materials:

- [Introduction to ROS](#)
- [Middlewares](#) ([master](#))

Lab (12:30 - 1:30)

Homeworks and Checkoffs:

- [Unit B-1 - Checkoff: Assembly and Configuration](#) deadline extended to **Monday Sept 25**. Submit by clicking [here](#).

## 12.8. Mon Sept 25

Class (10:30-11:30)

- Microelectronics ([pptx](#)) ([pdf](#))
- Modern Signal Processing ([keynote](#)) ([pdf](#))
- Intro to Networking ([keynote](#)) ([pdf](#))

Book Materials (still rough drafts, will be completed soon):

Lab (11:30 - 12:30)

- Liam will be in 2333
- Any final help needed for [Unit B-1 - Checkoff: Assembly and Configuration](#)
- Finalize Duckietown map setup

Homeworks and Checkoffs:

- [Unit B-1 - Checkoff: Assembly and Configuration](#) due by 11pm.
- New checkoff: “Taking a log” will be linked later today (due Monday Oct 2)
- New homework: “Data processing” will be linked later today (due Monday Oct 2)

## 12.9. Wed Sept 27

Class (11:30 - 12:30) in Z-305

- USB drive distribution
- New checkoff announced: [Unit B-2 - Checkoff: Take a Log](#)
- New homework announced: [Unit B-3 - Homework: Data Processing \(UdeM\)](#)
- Let’s review the git policy for homeworks: [Section 7.3 - Git policy for homeworks \(TTIC/UDEM\)](#)
- Announcement regarding activity spreadsheet which is now embedded here: [Section 8.1 - The Activity Tracker](#). Feel free to just look or send Kirsten your gmail on Slack and she will give you access to it.

Slides:

- Duckiebot Modeling ([pptx](#)) ([pdf](#))

Book Material:

- [Coordinate systems \(master\)](#)
- [Reference frames \(master\)](#)
- [Unit G-6 - Duckiebot modeling](#)

## 12.10. Mon Oct 2

Class (10:30 - 11:30) in Z-210

Slides:

- Computer vision basics: ([keynote](#)) ([pdf](#))

Book Material:

- [Unit G-7 - Computer vision basics \[draft\]](#)

- [Unit G-8 - Camera geometry \[draft\]](#)
- [Unit G-9 - Camera calibration \[draft\]](#)
- [Unit G-10 - Image filtering \[draft\]](#)

Lab (11:30 - 12:30) in AA 2333

## 12.11. Wed Oct 4

Class (11:30 - 12:30) in Z-305

Slides:

- Computer Vision - Lane and Line Detection ([keynote](#)) ([pdf](#))

Book Material:

- [Unit G-10 - Image filtering \[draft\]](#)
- [Line Detection \(master\)](#)

Lab (12:30 - 1:30) in AA 2333

| **Note:** Checkoff and Homework due at 11pm

## 12.12. Mon Oct 9

Holiday no class!

## 12.13. Wed Oct 11

Class 11:30 in Z-305

- Computer Vision - Feature descriptors ([keynote](#)) ([pdf](#))

Book Material:

Lab 12:30 - 1:30 in AA2333

## 12.14. Friday Oct 13

New Checkoff Initiated: [Unit B-6 - Checkoff: Robot Calibration](#) Deadline is **Monday Oct. 23**. Deliverables are: - Screenshot of your robot passing the kinematic odometry test - PR to duckiefleet repo with your 3 robot calibrations (kinematics, camera intrinsics, camera extrinsics)

## 12.15. Monday Oct 16

Class 10:30-11:30 Z-205

- Reminder about checkoff.

- Intro to filtering ([pptx](#)) ([pdf](#))

## 12.16. Wednesday Oct. 18

Class 12:30-1:30 Z-310

Guest Lecture from Prof. James Forbes from McGill on Extended Kalman filter slides: ([pdf](#))

## 12.17. Friday Oct. 20

Homework [Unit B-5 - Homework: Augmented Reality](#) announced. Deadline is Friday Oct. 27 at 11pm

## 12.18. Monday Oct. 23

No class (Reading Week)

## 12.19. Wednesday Oct. 25

No class (Reading Week)

## 12.20. Monday Oct. 30

Class 10:30-11:30 Z-2015

Start of Introduction to SLAM

Slides: ([pdf](#)) ([keynote](#))

## 12.21. Wednesday Nov. 1

Class 11:30-12:30

End of Introduction to SLAM

Slides: ([pdf](#)) ([keynote](#))

## 12.22. Monday Nov. 6

Class 10:30-12:30

Project Pitches. [Link to slides](#)

## **12.23. Wednesday Nov. 8**

Class 11:30-12:30

- Motion Planning. ([pdf](#)), ([pptx](#))

[Checkoff Navigation](#) initiated. Deadline Nov 15 @ 11pm.

## **12.24. Thursday Nov. 9**

- [Filtering Homework](#) initiated. Deadline Nov 17 @ 11pm.
- Project groups announced

## **12.25. Monday Nov. 13**

Class 10:30-11:30

Control.

Module 1: ([pptx](#))

Module 2: ([pptx](#))

Module 3: ([pptx](#))

## **12.26. Wednesday Nov. 15**

Class 11:30-11:30

David Vazquez guest lecture

## **12.27. Oct 30 (Mon)**

Assigned to: XXX

## **12.28. Nov 01 (Wed)**

Assigned to: XXX

## **12.29. Nov 06 (Mon)**

Assigned to: XXX

## **12.30. Nov 08 (Wed)**

Assigned to: XXX

**12.31. Nov 13 (Mon)**

Assigned to: XXX

**12.32. Nov 15 (Wed)**

Assigned to: XXX

**12.33. Nov 20 (Mon)**

Assigned to: XXX

**12.34. Nov 22 (Wed)**

Assigned to: XXX

**12.35. Nov 27 (Mon)**

Assigned to: XXX

**12.36. Nov 29 (Wed)**

Assigned to: XXX

**12.37. Dec 04 (Mon)**

Assigned to: XXX

**12.38. Dec 06 (Wed)**

Assigned to: XXX

**12.39. Dec 11 (Mon)**

Assigned to: XXX

**12.40. (Template for every lecture) Date: Topic**

Assigned to: Name of TA

**1) Preparation**

Things that the students should do before class.

## 2) Material presented in class

Link to PDF and Keynote/Powerpoint materials.

## 3) Pointers to reading materials

Links to the units mentioned in the slides, and additional materials.

## 4) Questions and answers

Write here the FAQs that students have following the lecture or instructions.

### **12.41. (Template for every lecture) Date: Topic**

Assigned to: Name of TA

#### 1) Preparation

Things that the students should do before class.

#### 2) Material presented in class

Link to PDF and Keynote/Powerpoint materials.

#### 3) Pointers to reading materials

Links to the units mentioned in the slides, and additional materials.

#### 4) Questions and answers

Write here the FAQs that students have following the lecture or instructions.

### **UNIT A-13**

### **Chicago branch diary**

Classes take place on Mondays and Wednesdays from 9am-11am in TTIC Room 530.

### **13.1. Checkoffs:**

The following is a list of the current checkoffs, along with the date and time when they are due. Most submissions involve uploading videos to via Dropbox File Request using the link provided in the individual checkoff.

- Friday October 6, 5pm CT: [Unit B-1 - Checkoff: Assembly and Configuration](#)
- Wednesday October 18, 11:59pm CT: [Unit B-2 - Checkoff: Take a Log](#)
- Sunday October 22, 11:59pm CT: [Unit B-6 - Checkoff: Robot Calibration](#)
- Wednesday November 15, 11:59pm CT: [Unit B-8 - Checkoff: Navigation](#)

### 13.2. Problem Sets:

The following is a list of the current problem sets, along with the date and time when they are due.

→ [Section 7.3 - Git policy for homeworks \(TTIC/UDEM\)](#)

for instructions on how the homework should be submitted.

**Note:** Please keep track of how much time you spend on each problem set. We will ask you for this estimate along with other feedback at the end of each problem set.

- Friday October 13, 11:59pm CT: [Unit B-4 - Homework: Data Processing \(TTIC\)](#)
- Friday October 27, 11:59pm CT: [Unit B-5 - Homework: Augmented Reality](#)
- Friday November 17, 11:59pm CT: [Unit B-9 - Homework: Lane Filtering](#)

### 13.3. Monday September 25: Introduction to Duckietown

#### 1) Lecture Content

---

- Duckietown Course Intro ([Keynote](#), [PDF](#))

#### 2) Reading Material

---

- [Part C - The Duckietown project](#)
- [Unit G-2 - Autonomy overview](#)

#### 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

### 13.4. Tuesday September 26: Onboarding

Tonight, we sent out the onboarding instructions.

→ [Section 2.1 - Onboarding Procedure](#)

Please complete the onboarding questionnaire by Thursday, September 27, 5:00pm CT

### 13.5. Wednesday, September 27: Duckiebox Ceremony

Welcome to Duckietown! This lecture constitutes the *Duckiebox ceremony* during which we will distribute your Duckieboxes and ask you to name your yet-to-be built Duckiebots! We will also discuss logistics related to the course, but we admit that that isn't as exciting.

## 1) Lecture Content

---

- [The Duckiebook](#)
- Slack ([Unit A-10 - Slack Channels](#))
- Git repos ([Unit A-7 - Git usage guide for Fall 2017](#))
- How to get and give help ([Unit A-9 - Getting and giving help](#))
- [Unit B-1 - Checkoff: Assembly and Configuration](#) initiated
- Getting to know one another
- Naming your robots
- Distribute the Duckieboxes!

As you name your Duckiebots, please consider the following constraints:

- The name must work as a hostname. It needs to start with a letter, contains only letters and numbers, and no spaces or punctuation.
- It should be short, easy to type. (You'll type it a lot.)
- It cannot be your own name.
- It cannot be a generic name like "robot", "duckiebot", "car". It cannot contain brand names.

## 2) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.6. Monday, October 2: Modern Robotic Systems

### 1) Lecture Content

---

- Logistics ([Keynote, PDF](#))
- Autonomous Vehicles ([Keynote, PDF](#))
- Modern Robotic Systems ([Keynote, PDF](#))
- System Architecture Basics ([Keynote, PDF](#))

### 2) Reading Material

---

- [Unit G-1 - Autonomous Vehicles \[draft\]](#)
- [Unit G-2 - Autonomy overview](#)
- [Unit G-3 - Modern Robotic Systems \[draft\]](#)
- [System architectures basics \(master\)](#)

### 3) Assignments

---

- [Unit B-1 - Checkoff: Assembly and Configuration](#) is due by Friday 5pm CT. Note: The page provides the URL where you should upload your video(s).

### 4) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.7. Wednesday, October 4: Modern Robotic Systems (Continued)

Note: Today's lecture will take place in Room 501 due to the TTIC Board Meeting

### 1) Lecture Content

---

- Logistics ([Keynote](#), [PDF](#))
- Representations ([Keynote](#), [PDF](#))
- Software Architectures ([Keynote](#), [PDF](#))
- Networking ([Keynote](#), [PDF](#))

### 2) Reading Material

---

- [Unit G-5 - Representations \[draft\]](#)
- [Networking tools \(master\)](#)
- [System architectures basics \(master\)](#)

### 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.8. Monday, October 9: Modeling

### 1) Preparation

---

Important: Before starting these tutorials, make sure that you completed the following:

- [Unit I-6 - Installing Ubuntu on laptops](#)

Before coming to class, please read through the following tutorials:

- <https://tinyurl.com/ROS101-Intro>
- [http://wiki.ros.org/ROS/Tutorials#Beginner\\_Level](http://wiki.ros.org/ROS/Tutorials#Beginner_Level) Complete all the tutorials in Section 1.1 Beginner Level. There is no need to install ROS as the tutorial instructs, because you already installed as part of the [setup process](#). Sections 11 and 14 of this tutorial will ask you to implement simple ROS nodes using C++. You can skip them since Sections 12 and 15 are basically the same but using Python.

### 2) Lecture Content

---

- Modeling ([Powerpoint](#), [PDF](#))

### 3) Reading Material

---

- [Unit G-6 - Duckiebot modeling](#)

### 4) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.9. Wednesday, October 11: Introduction to Computer Vision

### 1) Lecture Content

- 
- Introduction to Computer Vision ([Keynote](#), [PDF](#))
  - Camera Models ([Keynote](#), [PDF](#))

### 2) Reading Material

- 
- [Unit G-7 - Computer vision basics \[draft\]](#)
  - [Unit G-8 - Camera geometry \[draft\]](#)
  - Richard Szeliski, *Computer Vision: Algorithms and Applications*, Chapters 1 and 2 (available [online](#))
  - David A. Forsyth and Jean Ponce, *Computer Vision: A Modern Approach*, Chapters 1 and 2

### 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.10. Monday, October 16: Camera Calibration and Image Filtering

**Note:** [The second checkoff](#) is due by 11:59pm CT Wednesday.

### 1) Lecture Content

- 
- Logistics ([Keynote](#), [PDF](#))
  - Calibration ([Keynote](#), [PDF](#))
  - Image Filtering ([Keynote](#), [PDF](#))

### 2) Reading Material

- 
- [Unit G-7 - Computer vision basics \[draft\]](#)
  - [Unit G-9 - Camera calibration \[draft\]](#)
  - [Unit G-10 - Image filtering \[draft\]](#)
  - Richard Szeliski, *Computer Vision: Algorithms and Applications*, Chapters 3 and 6 (available [online](#))
  - David A. Forsyth and Jean Ponce, *Computer Vision: A Modern Approach*, Chapters 5.3 and 7

### 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.11. Wednesday, October 18: Edge Detection and Lane Detection

**Note:** [The third checkoff](#) is due by 11:59pm CT Sunday. Note that camera calibration is necessary for the next problem set, which will be posted soon.

## 1) Lecture Content

---

- Logistics ([Keynote](#), [PDF](#))
- Image Filtering (Review) ([Keynote](#), [PDF](#))
- Image Gradients ([Keynote](#), [PDF](#))
- Edge Detection ([Keynote](#), [PDF](#))
- Line Detection ([Keynote](#), [PDF](#))
- Lane Detection ([Keynote](#), [PDF](#))

## 2) Reading Material

---

- [Line Detection \(master\)](#)
- Richard Szeliski, *Computer Vision: Algorithms and Applications*, Chapters 3 and 4 (available [online](#))
- David A. Forsyth and Jean Ponce, *Computer Vision: A Modern Approach*, Chapters 7 and 8

## 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.12. Monday, October 23: Feature Detection and Place Recognition

### 1) Lecture Content

---

- Logistics ([Keynote](#), [PDF](#))
- Robust Fitting ([Keynote](#), [PDF](#))
- Feature Detection ([Keynote](#), [PDF](#))
- Place Recognition ([Keynote](#), [PDF](#))

### 2) Reading Material

---

- Richard Szeliski, *Computer Vision: Algorithms and Applications*, Chapters 4 and 6.1.3 (available [online](#))

### 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.13. Wednesday, October 25: Filtering I

### 1) Lecture Content

---

- Place Recognition (continued) ([Keynote](#), [PDF](#))
- Introduction to Filtering ([Powerpoint](#), [PDF](#))

### 2) Reading Material

---

- Sebastian Thrun, Wolfram Burgard, and Dieter Fox, *Probabilistic Robotics*, Chapters 1 and 2

### 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.14. Monday, October 30: Filtering II

### 1) Lecture Content

---

- Nonparametric Filtering ([Powerpoint](#), [PDF](#))

### 2) Reading Material

---

- Sebastian Thrun, Wolfram Burgard, and Dieter Fox, *Probabilistic Robotics*, Chapter 4

### 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.15. Wednesday, November 1: Introduction to SLAM

### 1) Lecture Content

---

- SLAM Intro ([Keynote](#), [PDF](#))

### 2) Reading Material

---

- Sebastian Thrun, Wolfram Burgard, and Dieter Fox, *Probabilistic Robotics*, Chapters 9, 10, and 13

### 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.16. Monday, November 6: Introduction to Planning

### 1) Lecture Content

---

- Planning Intro ([Keynote](#), [PDF](#))
- Project Pitches ([Google Slides](#))

### 2) Reading Material

---

- Steven M. LaValle, *Planning Algorithms*, Chapters 3, 4, and 6. Available [online](#)

### 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.17. Wednesday, November 8: Introduction to Planning (Continued)

### 1) Lecture Content

- 
- Planning Intro ([Keynote](#), [PDF](#))

### 2) Reading Material

- 
- Steven M. LaValle, *Planning Algorithms*, Chapter 5. Available [online](#)

### 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.18. Monday, November 13: Introduction to Control

### 1) Lecture Content

- 
- Control Intro ([Powerpoint](#), [PDF](#))

### 2) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.19. Wednesday, November 15: Introduction to Control (Continued)

### 1) Lecture Content

- 
- Controls ([Powerpoint](#), [PDF](#))
  - Controls for Duckietown ([Powerpoint](#), [PDF](#))

### 2) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

## 13.20. Monday, November 20: Testing for Autonomous Vehicles

### 1) Lecture Content

- 
- Testing for Autonomous Vehicles ([Keynote](#), [PDF](#))

### 2) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

UNIT A-14

NCTU branch diary

Classes take place on Thursday from 1:20pm-4:20pm in NCTU Engineering Building 5 Room 635.

### 14.1. Checkoffs:

The following is a material of the current preview lecture, along with the date and time when they are due. Please make sure that you preview the materials every week before the class for insuring a good learning quality.

- Thursday October 19, 1:20pm: [Week5 Material](#)
- Thursday October 26, 1:20pm: [Week6 Material](#)

### 14.2. Course Material:

The following is a list of the course material. Please preview the course every week and better if you try the lab yourself.

→ [Course Material](#)

- Thursday October 12, 1:20pm : [Week5 Material](#)

### 14.3. Thursday September 14: Introduction to Duckietown and Creative Software Project

#### 1) Lecture Content

- 
- Duckietown Course Intro ([Week1 Material](#))

### 14.4. Thursday September 21: Project Ideas

How do you choose a good project idea? How about “writing” a good project idea?

→ [Week2 Material](#)

#### 1) Lecture Content

---

→ [Week2 Lecture](#)

#### 2) Weekly Lab

---

This is the first lab of the semester. We are going to teach you “git” which is essential when becoming a professional programmer and cooperating with professional team.

→ [Week2 Lab](#)

## 14.5. Thursday September 28: Robotics System

What is a robotics system? We'll introduce the concept of robotics system and some well known software architecture and middleware. Also, robotic operation system will be introduced in this class.

See” [Week3 Material](#)

---

### 1) Lecture Content

→ [Week3 Lecture](#)

---

### 2) Weekly Lab

This week we'll continue on the topic of git. Also, a new exciting chapter has been opened. We are going to prepare our own Duckiebot! Come and join us!

→ [Week3 Lab](#)

## 14.6. Thursday October 5: OpenCV, Python and Jupyter Notebook

Today we're going deeply into Duckiebot's “mind”. What is the algorithm behind lane following? What is the secret that the duckies are so smart to drive? Here comes the answer.

→ [Week4 Material](#)

---

### 1) Lecture Content

→ [Week4 Lecture](#)

---

### 2) Weekly Lab

Jupyter notebook is a very useful and convenient tool while dealing with python language. We will teach you how to use it. A part of lane following algorithm will be taught this week which is about line detector.

→ [Week4 Lab](#)

## 14.7. Thursday October 19: Camera and Wheel Calibration

This week we are going to do the camera and wheel calibration. We will teach the student the theory of camera calibration, including extrinsic and intrinsic.

→ [Week6 Material](#)

---

### 1) Lecture Content

→ [Week6 Lecture](#)

## 2) Weekly Lab

---

We will let them control the duckiebots by joystick to finish the wheel calibration. Also, we will give them chessboard for camera calibration.

→ [Week6 Lab](#)

### 14.8. (Template for every lecture) Date: Topic

What is this lecture all about?

#### 1) Preparation

---

Things that the students should do before class.

#### 2) Lecture Content

---

Link to PDF and Keynote/Powerpoint materials.

#### 3) Feedback Form

---

Please help us making the experience better by [providing feedback \(can be anonymous\)](#)

#### 4) Reading Material

---

Links to the units mentioned in the slides, and additional materials.

#### 5) Questions and Answers

---

FAQs that students have following the lecture or instructions.

## UNIT A-15

## Slack Channels

This page describes all of the helpful Slack channels and their purposes so that you can figure out where to get help.

### 15.1. Channels

TABLE 15.1. DUCKIETOWN SLACK CHANNELS

Channel	Purpose
help-accounts	Info about necessary accounts, such as Slack, Github, etc
help-assembly	Help putting your robot together
help-camera-calib	Help doing the intrinsic and extrinsic calibration of your camera
help-duckument	Help compiling the online documentation
help-git	Help with git
help-infrastructure	Help with software infrastructure, such as Makefiles, unit tests, continuous integration, etc.
help-laptops	Help getting your laptop setup with Ubuntu 16.04
help-parts	Help getting the parts for the robot or replacement parts if you broke something
help-robot-setup	Help getting the robot setup to do basic things like be driven with a joystick
help-ros	Help with the Robot Operating System (ROS)
help-wheel-calib	Help doing your odometry calibration

+ comment

Note that you can link directly to the channel. (See list in the org sheet.) -AC

## UNIT A-16

### Guide for TAs

#### 16.1. Dramatis personae

These are the TAs.

At ETHZ:

- Shiying Li (shili@student.ethz.ch)
- Ercan Selçuk (ercans@student.ethz.ch)
- Miguel de la Iglesia Valls (dmiguel@student.ethz.ch)
- Khurana Harshit (hkhurana@student.ethz.ch)
- Lapandic Dzenan (ldzenan@student.ethz.ch)
- Marco Erni (merni@ethz.ch)

At TTIC:

- Andrea F. Daniele (afdaniele@ttic.edu)
- Falcon Dai (dai@ttic.edu)
- Jon Michaux (jmichaux@ttic.edu)

At Montreal:

- Florian Golemo (fgolemo@gmail.com)

## 16.2. First steps

Here are the first steps for the TAs.

Note that many of these are not sequential and can be done in parallel.

### 1) Learn about Duckietown

---

Read about Duckietown's history; watch the Duckumentary.

→ [Part C - The Duckietown project](#)

### 2) Online accounts

---

You have to set up:

- A personal Github account
- A Twist account
- A Slack account
- A Google Docs account (Gmail address)

Send an email to Kirsten Bowser (akbowser@gmail.com), with your GMail address and your Github account. She will give you further instructions.

Point of contact: [Kirsten Bowser](#)

### 3) Install Ubuntu

---

Install Ubuntu 16.04 on your laptop, and then install ROS, Atom, LiClipse, etc.

→ [Unit I-6 - Installing Ubuntu on laptops](#)

### 4) Duckuments

---

Install the Duckuments system, so you can edit these instructions.

→ [Part D - Duckumentation documentation](#).

Point of contact: Andrea

### 5) Learn about Git and Github

---

Start learning about Git and Github. You don't have to read the entirety of the following references now, but keep them "on your desk" for later reference.

→ [Good book](#)

→ [Git Flow](#)

Point of contact: Liam?

## 6) Continuous integration

---

Understand the continuous integration system.

- [Documentation on continuous integration \(master\)](#).

Point of contact: Andrea

## 7) Duckiebot building

---

Build your Duckiebot according to the instructions.

- [Part I - Operation manual - Duckiebot](#)

Point of contact: Shiying (ETH)

Point of contact: ??? (UdeM)

Point of contact: ??? (TTIC)

As you read the instructions, keep open the Duckuments source, and note any discrepancies. You must note any unexpected thing that is not predicted from the instruction. If you don't understand anything, please note it.

The idea is that dozens of other people will have to do the same after you, so improving the documentation is the best use of your time, and it is much more efficient than answering the same question dozens of times.

## 8) Other documentation outside of the Duckuments

---

We have the following four documents outside of the duckuments:

1. [Organization chart](#): This is where we assign areas of responsibility.
2. [Lecture schedule](#)
3. [Checkoff spreadsheet](#)
4. [The big TODO list](#): Where we keep track of things to do.

### UNIT A-17

## Guide for mentors [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

### UNIT A-18

## Project proposals [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT A-19

# System Architecture

### 19.1. Preliminaries

Name of Project: System Architecture

Slack channel: #devel-heroes

Software development branch: devel-sonja (Software repo) or sonja-branch (Duckuments repo)

#### 1) Missions

---

The system architect project can be split into two missions:

1. Ensure that the development of the system goes smoothly (wooden spoon)
2. Develop a framework/tool to formally describe (and later optimize) the system (bronze, silver and gold)

### 19.2. Mission 1

Ensure that the development and integration of the projects into the system goes smoothly and that the resulting system makes sense, and is useful for future duck-iterations (duckie + generations).

#### 1) Problem Statement

---

Ensure that all teams know what their goal is and how it fits into the bigger picture

#### 2) Relevant Resources

---

- The functional diagram of the system
- Duckuments
- Other teams' preliminary project reports

#### 3) Deliverables (Goals)

---

The deliverables for Mission 1 will include the following:

- Functional diagram of the system
- Documentation of system architecture

Mission 1 is the “wooden spoon” level of the project.

#### 4) Proposed Approach

---

- Become one with the goals of Duckietown In order to make Duckietown a better place, one has to keep in mind what “better” means in Duckie terms.
- Be familiar with the current system architecture and track changes This can include having to update the functional diagram, for instance.
- Keep in close contact with teams This will be done by attending the meetings of some of the other teams (especially early meetings). Some teams’ meetings have been prioritized since many parts of the system are dependant on their work, namely:
  - Anti-instagram
  - Controllers
  - Navigators
  - Explicit coordination All teams will designate a contact person who can contact me whenever they change their project boundaries or have doubts/ need advice on their project’s boundaries/negotiating with other
- Offer nudges in a different direction if needed
- Acting as middleman/helper to facilitate negotiation of contracts between groups
- Monitor status of projects to find possible problems

#### 5) Logging and Testing Procedure

---

...

#### 6) Current status

---

Familiarisation with the current system status is under way.

Functional diagram has been updated to include multi-robot SLAM as alternative to single-robot SLAM to creating map.

#### 7) Tasks

---

- Familiarisation with existing system architecture
- Going to group meetings
- Identifying potential problems

#### 8) Timeline

---

...

#### 9) Meetings notes

---

...

### 19.3. Mission 2

Where there is a system, there is a want (nay, need) for optimisation. Describing a system's performance and resource requirements in a quantifiable way is a critical part of being able to benchmark modules and optimise the system.

Mission 2 is to formalise the description of the system characteristics, so that eventually the system performance can be optimised for some given resources.

## 1) Problem Statement

---

Find a way to describe all the module requirements, specifications, etc in a formal, quantifiable language.

Find a way to calculate the requirements and specifications of a whole system or subsystem, based on the requirements and specifications of the individual modules of the system.

Find a way to calculate the optimal system configuration, based on the desired requirements and specifications of the system.

## 2) Relevant Resources

---

- How the current system's characteristics are defined
- Which values/parameters are needed
- Possibly research on system description?
- Possibly graph theory?

## 3) Deliverables (Goals)

---

The different levels of Mission 2 are defined as follows:

- Bronze standard:
  - Formal, qualitative language to describe constraints/requirements between modules
- Silver standard:
  - each module has table of performance. Qualitative. Can compare and give yes/no queries. With given configuration  $x$ , is the cost/requirements possible with available resources?  $f(x)$  smaller equal to  $R_{max}$ ?
- Gold standard:
  - optimization is possible to find best implementation, given available resources. Given  $f(x)$  and  $R_{max}$ , find optimal configuration  $x$

The deliverables will then include:

- Documentation on the result of the project
- A description of the current system's characteristics (bronze)
- A program/tool that can give a qualitative answer (yes/no) to the question: Are these resources sufficient for this system configuration? (silver)
- A program/tool that will give an optimised system configuration, based on the given available resources (gold)

## 4) Proposed Approach

---

- Research on the topic of formal description of a system
- Find/develop a suitable language to describe module characteristics
- Require groups to compile a description of their respective modules' characteristics
- Find/develop functions to do mathematics on the language description of modules

#### 5) Logging and Testing Procedure

---

...

#### 6) Current status

---

Research is being done to identify some research areas that may be relevant and tools that may be helpful, in order to decide on an approach.

#### 7) Tasks

---

- Research into existing methods of system description
- Graph based databases?
- Perhaps graph theory can be useful later if the (suspiciously graph-looking) system can be described suitably.

#### 8) Timeline

---

...

#### 9) Meetings notes

---

...

## UNIT A-20

## Template of a project

Make a copy of this document before editing.

### 20.1. Preliminaries

Name of Project:

Team:

- Person 1 (UdeM) (Mentor)
- Person 2 (TTIC) (TA)
- ...

Slack channel: #XXX

## 20.2. Problem Statement

Summarize the mission for the team - What is the need that is being addressed? Do not focus on technical specifics yet.

## 20.3. Relevant Resources

List papers, open source code, pages in the Duckiebook, lecture slides, etc, that could be relevant in your quest.

## 20.4. Deliverables (Goals)

Anything that is going to be an output. These should be quantified in terms of functionalities and performance metrics where appropriate.

Example 1: A Duckiebot detection system (functionality) with minimum precision of 0.8, a minimum recall of 0.5, a maximum latency of 50ms with maximum CPU consumption of 80% of one core (performance).

Example 2: At least 20 hours of logs Duckiebots annotated

- Bronze standard:
- Functionality:
- Performance:
- Silver standard:
- Functionality:
- Performance:
- Gold standard:
- Functionality:
- Performance:

Part of the deliverables should be:

1. A new or improved functionality (demonstrated live and with a video) with well-documented code,
2. A approx 15-20min presentation about the functionality (with a slide deck + 1 slide overview poster to be shown at public demo),
3. A technical description of the underlying method in the form of a page in the Duckiebook,
4. Instructions for reproducing the functionality in the form of a page in the duckiebook.

## 20.5. Proposed Approach

After analysis of the resources and precise understanding of the problem you trying to solve, make a plan for how you will solve the problem. It is possible that at the

start you could explore several seemingly promising avenues. However, you should converge on Bronze standard before moving to Silver standard etc.

- Bronze standard:
- Silver standard:
- Gold standard:

## 20.6. Logging and Testing Procedure

A detailed description of the logs and procedure you will use to verify that the system is working the way you say it is working. In most cases this should include a regression test so that when someone changes something else, we can make sure that your thing still works as well as it used to.

## 20.7. Current status

Write here the current status. What works now, as opposed to what the goal is. The difference between these two is the work to be done.

**Note:** it is better to have something that does not work, and a good description of what should work and why it doesn't work, than to have something that kinda works, but nobody knows what the thing is supposed to do.

### 1) Functionality

---

Nothing implemented.

### 2) Performance

---

Infinitely slow.

## 20.8. Tasks

So and so should do such and such

## 20.9. Timeline

- This should be done by Nov 15
- That shoudl be done by Nov 16

## 20.10. Meetings notes

- Link 1
- Link 2

The map to be used in the Fall 2017 class is shown in [Figure 21.1](#).



Figure 21.1. The map to be used in the Fall 2017 class

The editable keynote file is in this directory of the duckuments repo. The ids on the signs correspond to the Apriltag IDs. For more details see [Signage \(master\)](#).

## PART B

# Fall 2017 Checkoffs and Homeworks

## UNIT B-1

### Checkoff: Assembly and Configuration

The first job is to get your Duckiebot put together and up and running.

#### 1.1. Pick up your Duckiebox

Slack channel: [#help-parts](#)

There is a checklist inside. You should go through the box and ensure that the parts that are supposed to be in it actually are inside.

If you are missing something, contact your local responsible and ask for help on Slack in the appropriate channel.

These sections describe the parts that are in your box.

→ [Unit I-2 - Acquiring the parts \(DB17-jwd\)](#)

→ [Unit J-1 - Acquiring the parts \(DB17-1c\)](#)

?? | You don't need to buy anything - you have all the parts that you need.

## 1.2. Soldering your boards

Depending on how kind your instructors/TAs are, you may have to solder your boards.

→ [Soldering boards \(DB17\) \(master\)](#)

→ [Unit J-2 - Soldering boards \(DB17-1\) \[draft\]](#)

?? | You don't need to solder anything.

## 1.3. Assemble your Robot

Slack channel: [#help-assembly](#)

You are ready to put things together now.

→ [Unit I-4 - Assembling the Duckiebot \(DB17-jwd\)](#)

→ [Unit J-4 - Bumper Assembly \[draft\]](#)

→ [Unit J-3 - Assembling the Duckiebot \(DB17-1c\) \[draft\]](#)

## 1.4. Optional: Reproduce the SD Card Image

If you are very inexperienced with Linux/Unix/networking etc, then you may find it a valuable experience to reproduce the SD card image to “see how the sausage is made”.

→ [Reproducing the image \(master\)](#)

?? | You probably don't want to see how the sausage is made.

## 1.5. Setup your laptop

Slack channel: [#help-laptops](#)

The only officially supported OS is Ubuntu 16.04. If you are not running this OS it is recommended that you make a small partition on your hard drive and install the OS.

Related parts of the book are:

→ [How to make a partition \(master\)](#) if you want to make a partition

→ [Unit I-6 - Installing Ubuntu on laptops](#)

## 1.6. Make your robot move

Slack channel: [#help-robot-setup](#)

Now you need to clone the software repo and run things to make your robot move.

First initialize the robot:

- [Unit I-7 - Duckiebot Initialization](#)

Then get it to move!

- [Unit I-9 - Software setup and RC remote control](#)

## 1.7. Upload your video

You should record a video demonstrating that your Duckiebot is up and running. Brownie points for creative videos. Please upload your videos via the following URL:

?? | Chicago: [upload your video](#)

?? | Zurich: [upload your video](#)

## UNIT B-2

### Checkoff: Take a Log

#### KNOWLEDGE AND ACTIVITY GRAPH

Requires: [Unit B-1 - Checkoff: Assembly and Configuration](#)

Results: A verified log in rosbag format uploaded to Dropbox.

Slack channel: [#help-logging](#)

?? | Montreal deadline: Oct 4, 11pm

?? | Zurich deadline: Oct 20, 17:00

## 2.1. Mount your USB drive

We will log to the USB drive that you were given.

- [Mounting USB drives \(master\)](#)

## 2.2. Take a Log

Take a 5 min log as you drive in Duckietown.

??

- | For Montreal this is rm. 2333.
- ?? | For Zurich this is ML J44. Ask the TA when it is available.
- ?? | For Chicago, we are still building the town, so feel free to do this at home or in the lab.
  - [Unit I-16 - Taking and verifying a log](#) for detailed instructions.

## 2.3. Verify your log

- [Section 16.5 - Verify a log](#) for detailed instructions.

## 2.4. Upload the log

- ?? | Upload the log [here](#)
- ?? | Upload the log [here](#)
- ?? | Upload the log [here](#)

## UNIT B-3

# Homework: Data Processing (UdeM)

## KNOWLEDGE AND ACTIVITY GRAPH

- | **Requires:** [Unit B-2 - Checkoff: Take a Log](#)
- | **Requires:** [Unit H-2 - ROS installation and reference](#)
- | **Results:** Ability to perform basic operations on images
- | **Results:** Build your first ROS package and node
- | **Results:** Ability to process imagery live

Slack channel: [#ex-data\\_processing](#)

Montreal deadline: Oct 4, 11:00pm

## 3.1. Follow the git policy for homeworks

- [Section 7.3 - Git policy for homeworks \(TTIC/UDEM\)](#)

for instructions on how the homework should be submitted.

### 3.2. Exercise: Basic image operations

Complete [Unit E-1 - Exercise: Basic image operations, adult version](#)

### 3.3. Exercise: Log decimation

Complete [Unit E-3 - Exercise: Bag in, bag out](#)

### 3.4. Exercise: Instagram filters

Complete [Unit E-5 - Exercise: Instagram filters](#)

### 3.5. Exercise: Live Instagram

Complete [Unit E-7 - Exercise: Live Instagram](#)

Call your package `dt-instagram-live_`*robot name* and call your node `dt-instagram-live_`*robot name*

When you are done, take a 5min log (See [Unit I-16 - Taking and verifying a log](#)) in Duckietown (2333 in Montreal) and upload [here](#)

## UNIT B-4

### Homework: Data Processing (TTIC)

#### KNOWLEDGE AND ACTIVITY GRAPH

Requires: [Unit H-2 - ROS installation and reference](#)

Results: Ability to perform basic operations on images

Results: Build your first ROS package and node

Results: Ability to process imagery live

Slack channel: [#help-data-processing](#)

TTIC deadline: Friday, October 13 11:59pm CT

### 4.1. Follow the git policy for homeworks

→ [Section 7.3 - Git policy for homeworks \(TTIC/UDEM\)](#)

for instructions on how the homework should be submitted.

## 4.2. Exercise: Basic image operations

Complete [Unit E-1 - Exercise: Basic image operations, adult version](#)

## 4.3. Exercise: Log analysis

Complete [Unit E-2 - Exercise: Simple data analysis from a bag](#)

## 4.4. Exercise: Log decimation

Complete [Unit E-3 - Exercise: Bag in, bag out](#)

## 4.5. Exercise: Video thumbnails

Complete [Unit E-4 - Exercise: Bag thumbnails](#)

## 4.6. Exercise: Instagram filters

Complete [Unit E-5 - Exercise: Instagram filters](#)

## 4.7. Exercise: Log Instagram

Complete [Unit E-6 - Exercise: Bag instagram](#)

## 4.8. Exercise: Live Instagram

Complete [Unit E-7 - Exercise: Live Instagram](#)

Call your package `dt-instagram-live_`**robot name** and call your node `dt-instagram-live_`**robot name**.

## 4.9. Exercise: Feedback

Complete the [exercise feedback form](#). You will receive points towards this exercise if you complete the form.

## UNIT B-5

## Homework: Augmented Reality

### KNOWLEDGE AND ACTIVITY GRAPH

Requires: [Unit H-2 - ROS installation and reference](#)

**Requires:** [Unit I-14 - Camera calibration](#)

**Results:** Ability to project fake things from an image back into the world

Slack channel: [#ex-augmented\\_reality](#)

?? | Montreal deadline: Oct 27, 11:00pm

?? | Chicago deadline: Oct 27, 11:59pm

?? | Zurich deadline: Oct ???, 11:59pm

## 5.1. Follow the git policy for homeworks

Please follow the instructions on how the homework should be submitted.

→ [Section 7.3 - Git policy for homeworks \(TTIC/UDEM\)](#)

## 5.2. Exercise: Augmented Reality

Complete [Unit E-8 - Exercise: Augmented Reality](#).

?? | Please hold on for Github instructions.

Note that you should do a pull in `Software` to get all the goodies and utils described in the exercise, and `exercises-fall2017` repos (in `exercises-fall2017` repo this means pulling from the `duckietown` remote:

```
$ git pull upstream master
```

if you have followed the instructions properly.

In the `exercises-fall2017` repository, you will find a template that you can use to make your own package. Basically everywhere you see `littleredcorvette` you would need to replace it with your `robot name`.

## 5.3. Submission

?? | Please upload the images requested in the homework to your git repository.  
When complete, please tag a release from your repo.

?? | Please upload the images requested in the homework to your git repository.  
When complete, please tag a release from your repo.

## 5.4. Bonus: Defining `intersection_4way.yaml`

The first student to do it (from any institution) gets notoriety and a bonus.

## UNIT B-6

# Checkoff: Robot Calibration

### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** [Unit B-1 - Checkoff: Assembly and Configuration](#)

**Requires:** That you have correctly cloned and followed the git procedure outline in [Unit A-7 - Git usage guide for Fall 2017](#)

**Requires:** That you have correctly setup your environment variables according to [Environment variables \(updated Sept 12\)](#) ([master](#))

**Results:** Your robot calibrations (wheels and camera (x2)) are merged to git through a PR

Slack channels: [#help-wheel-calib](#), [#help-camera-calib](#)

## 6.1. Pull and rebuild your Software repo on robot and laptop

Some of the services have changed and this requires a rebuild.

On both laptop and robot do:

```
$ cd Duckietown root
$ source environment.sh
$ make build-catkin-clean
$ make build-catkin-parallel
```

## 6.2. Make a branch in the duckiefleet repo

?? | Remember that the git policy has changed a bit. You are probably best to re-clone the duckiefleet repo. For details see [Unit A-7 - Git usage guide for Fall 2017](#) and particularly the section For U de M students who have already submitted homework to the previous duckiefleet-2017 repo

Don't forget that master is now protected in duckiefleet. So make a new branch right away and call it [GIT USERNAME](#)-devel

## 6.3. Kinematic calibration

Follow the procedure in [Wheel calibration \(master\)](#). Once you have successfully passed the automated test, take a screen shot and post it to the slack channel [#checkoffs](#) and we will all congratulate you.

## 6.4. Camera calibration

Follow the procedure in [Unit I-14 - Camera calibration](#) to do you intrinsic and extrinsic calibrations.

## 6.5. Visually verify the calibration is good in Duckietown

Take your robot to Duckietown. Put it in a lane.

On your robot execute

 \$ make demo-lane-following

On your laptop do (after setting ros master to your robot):

 \$ rqt\_image\_view

on your joystick you need to hit the top-right button (TODO: add picture). On the command line you should see the output `state_verbose = True`

in the drop down menu select `robot name/line_detector_node/image_with_lines`

on the display you should see all the color-coded line detections

now open the Rviz visualizer on your laptop (after setting ros master to your robot):

 \$ rviz

- click the `Add` button in the bottom left.
- then click the `By Topic` tab
- then click the triangle next to `/segment_list_markers` underneath `/duckiebot_visualizer`
- then double click on `MarkerArray`

On the display you should see the ground projected lines. Do they make sense? If not your calibration is wrong.

**TODO:** add a picture of what they should look like

## 6.6. Submit a PR

Don't forget at the end to submit a PR back to [duckiefleet repo](#)

UNIT B-7

Exercises: Data Processing (Zurich)

| **Requires:** [Unit H-2 - ROS installation and reference](#)

| **Results:** Ability to perform basic operations on images

| **Results:** Build your first ROS package and node

| **Results:** Ability to process imagery live

Slack channel to get help: `#ex-data-processing`

### 7.1. Git setup

First, make sure you are in the Github Zurich team.

If your name is not [here](#), contact Kirsten, and stop. You will not be able to do the next step.

Please clone this repository:

```
git@github.com:AndreaCensi/exercises-fall2017.git
```

using

```
$ git clone git@github.com:AndreaCensi/exercises-fall2017.git
```

This repository is writable by all Zurich people, but not readable by Chicago and Montreal. Because they grade the homework, we need to keep it secret.

We invite everybody to just push their exercises to this repository. (This is also compulsory to get help from TAs, so that the TAs can give comments that are useful for everybody.)

### 7.2. The exercises

We have created a series of exercises that are supposed to help somebody who doesn't know how to program in Python/Linux to get to a decent level.

We suggest the following:

1. First, read through the exercises and note the skills that are learned for each one.
2. Look at the last two: "Log Instagram" and "Live Instagram". Do you think you can do them? If so, just do those two and feel free to skip the rest.
3. Otherwise, you have a lot to catch up. No problem. Take your time. Start with the basic exercise. TAs are here to help.

## 7.3. Exercise: Basic image operations

- [Unit E-1 - Exercise: Basic image operations, adult version](#)

### 1) Exercise: Log analysis

---

- [Unit E-2 - Exercise: Simple data analysis from a bag](#)

### 2) Exercise: Log decimation

---

- [Unit E-3 - Exercise: Bag in, bag out](#)

### 3) Exercise: Video thumbnails

---

- [Unit E-4 - Exercise: Bag thumbnails](#)

### 4) Exercise: Instagram filters

---

- [Unit E-5 - Exercise: Instagram filters](#)

### 5) Exercise: Log Instagram [recommended for everybody]

---

- [Unit E-6 - Exercise: Bag instagram](#)

### 6) Exercise: Live Instagram [recommended for everybody]

---

- [Unit E-7 - Exercise: Live Instagram](#)

Call your package `dt-instagram-live_`*robot name* and call your node `dt-instagram-live_`*robot name*.

### 7) Exercise: Feedback form

---

**TODO:** provide link to exercise feedback form.

## UNIT B-8

### Checkoff: Navigation

#### KNOWLEDGE AND ACTIVITY GRAPH

Requires: [Unit B-6 - Checkoff: Robot Calibration](#)

Requires: [Unit B-2 - Checkoff: Take a Log](#)

Results: 2 logs of your robot autonomously navigating Duckietown

Montreal Deadline: Nov 15, 11pm

?? Chicago Deadline: Nov 15, 11pm

Slack channel: [#help-navigation](#)

## 8.1. Pull from master

As always - it's a good idea to pull from `master` often.

## 8.2. Lane Following

Place your robot on the Duckietown map somewhere on the “outer loop” (right hand lane so that it will follow the exterior of the map).

Launch the robot with the command from `DUCKIETOWN_ROOT`:



```
$ make demo-lane-following
```

Open a terminal on your laptop and set the ros master to your robot.

Toggle the `VERBOSE` flag by writing:

```
$ rosparam set /robot_name/line_detector_node/verbose true
```

Then open `rqt_image_view`. Look at the `.../image_with_lines` image output. Apply the **anti-instagram calibration** by pushing the `Y` button on the joystick (TODO: is it the same for the new joysticks?). You should see your image get corrected and the line detections become more correct. If nothing happens and your robot output complains of bad health, move the robot a little bit and try again.

You may also be interested to look at the `../_belief_img` to see the output of the histogram filter. It should be quite stable if your robot is not moving. You can move the robot around to see how the posterior is updating.

If everything is looking good then push the `START` button on the joystick and your robot should start to drive.

The robot operation should look like [this](#)

Follow the instructions [here](#) to take a **minimal** log of at least 5 mins of uninterrupted robot autonomous function.

?? Upload [here](#)

?? Upload [here](#)

## 1) Bonus

---

The student who uploads the longest log of uninterrupted robot autonomous lane following from any institution will get a great bonus.

### 8.3. Indefinite Navigation

Follow the exact same procedure above but instead of running the lane following demo run the “indefinite navigation” demo:

 \$ make indefinite-navigation

Your robot will now stop at the stop lines and then make a random turn through the intersection. If it is crashing a lot you may need to turn the trajectories it takes through the intersection. To do so you may need to edit the file [here](#):

```
turn_left: #time, velocity, angular vel
- [0.8, 0.43, 0]
- [1.8, 0.43, 2.896]
- [0.8, 0.43, 0.0]
turn_right:
- [0.6, 0.43, 0]
- [1.2, 0.3, -4.506]
- [1.0, 0.43, 0.0]
turn_forward:
- [0.8, 0.43, 0.4]
- [1.0, 0.43, 0.0]
- [1.0, 0.43, 0.0]
```

to make it more reliably traverse the intersections.

Follow the instructions [here](#) to take a **minimal**. You may use the **BACK** button to stop it from crashing and then return it to autonomous mode with the **START** button.

?? | Upload [here](#)

?? | Upload [here](#)

## 1) Bonus

---

The student who uploads the longest log of uninterrupted robot autonomous indefinite navigation from any institution will get a great bonus.

**UNIT B-9**

**Homework: Lane Filtering**

?? | Montreal deadline: Nov 17, 11:00pm

?? | Chicago deadline: Nov 17, 11:59pm

Slack channel: [#ex-filtering](#)

## 9.1. Follow the git policy for homeworks

Please follow the instructions on how the homework should be submitted.

→ [Section 7.3 - Git policy for homeworks \(TTIC/UDEM\)](#)

## 9.2. Pick your Poison

This homework is about filtering. Either replace the existing histogram lane filter with either an [Extended Kalman Filter](#) or a [Particle Filter](#). If you do both you will get a bonus.

## 9.3. Setup instructions

Pull from `master` in the `Software` repo

Pull from the Duckietown (`upstream`) remote in your `exercises-fall2017` repo.

We are providing a script to change all the instances of the default robot (in this case `shamrock`) with `YOUR_ROBOT_NAME` to save you time. To run navigate to the `homeworks/03_filtering` directory and run:

```
$ ./change_robot_name_everywhere.sh YOUR_ROBOT_NAME
```

(you're welcome...)

In the

```
'homeworks/03_filtering/YOUR_ROBOT_NAME/dt_filtering_YOUR_ROBOT_NAME'
```

folder, the files you need to worry about are the following:

1) `default.yaml`: this contains the parameters that will be loaded. Here's what it currently looks like:

```
# default parameters for lane_filter/lane_filter_node
# change to your robot name below
filter:
- dt_filtering_shamrock.LaneFilterPF
  - configuration:
    example1: 0.2
    # fill in params here

#uncomment below and comment above if you are doing EKF
# - dt_filtering_shamrock.LaneFilterEKF
# - configuration:
  example2: 0.3
  #fill in other params here
```

This parameter file tells your node to automatically load the right filter. If you are working on particle filter you can leave it the way it is and just add your parameters that you need under `configuration`. If you are working on EKF, comment or delete the lines for the PF and uncomment the lines for the EKF and then add your params as needed.

The other file you need to concern yourself with is in

```
include/dt_filtering_YOUR_ROBOT_NAME
```

You will need to fill in the functions that are setup for you.

## 9.4. Submission

As normal, tag the TAs and instructors in a release from your repo when you are ready for your work to be evaluated.

### PART C

## The Duckietown project

### UNIT C-1

## What is Duckietown?

### 1.1. Goals and objectives

Duckietown is a robotics education and outreach effort.

The most tangible goal of the project is to provide a low-cost educational platform for learning about autonomous systems, consisting of lectures and other learning material, the Duckiebot autonomous robots, and the Duckietowns, which constitute the

infrastructure in which the Duckiebots navigate.

We focus on the *learning experience* as a whole, by providing a set of modules, teaching plans, and other guides, as well as a curated role-play experience.

We have two targets:

1. For **instructors**, we want to create a “class-in-a-box” that allows people to offer a modern and engaging learning experience. Currently, this is feasible at the advanced undergraduate and graduate level, though in the future we would like to provide a platform that can be adapted to a range of different grade and experience levels.
2. For **self-guided learners**, we want to create a “self-learning experience” that allows students to go from having zero knowledge of robotics to a graduate-level understanding.

In addition, the Duckietown platform is also suitable for research.

## 1.2. Learn about the Duckietown educational experience

The video in [Figure 1.1](#) is the “Duckumentary”, a documentary about the first version of the class, during Spring 2016.



Figure 1.1. The Duckumentary, created by [Chris Welch](#).

The video in [Figure 1.2](#) is a documentary created by Red Hat on the current developments in self-driving cars.



Figure 1.2. The road to autonomy

If you'd like to know more about the educational experience, [\[1\]](#) present a more formal description of the course design for Duckietown: learning objectives, teaching

methods, etc.

### 1.3. Learn about the platform

The video in [Figure 1.3](#) shows some of the functionality of the platform.

If you would like to know more, the paper [\[2\]](#) describes the Duckiebot and its software. (With 30 authors, we made the record for a robotics conference!)



Figure 1.3. Duckietown functionality

## UNIT C-2 Duckietown history and future

### 2.1. The beginnings of Duckietown

The original Duckietown class was at MIT in 2016 ([Figure 2.1](#)).



Figure 2.1. Part of the first MIT class, during the final demo.

Duckietown was built by elves ([Figure 2.2](#)).



Figure 2.2. The elves of Duckietown

These are some advertisement videos we used.



Figure 2.3. The need for autonomy



Figure 2.4. Advertisement



Figure 2.5. Cool Duckietown by night

## 2.2. University-level classes in 2016

Later that year, the Duckietown platform was also used in these classes:

- [National Chiao Tung University 2016 \(master\)](#), Taiwan - Prof. Nick Wang;
- [Tsinghua University \(master\)](#), People's Republic of China - Prof. (Samuel) Qing-Shan Jia's *Computer Networks with Applications* course;
- [Rensselaer Polytechnic Institute 2016 \(master\)](#) - Prof. John Wen;



Figure 2.6. Duckietown at NCTU in 2016

## 2.3. University-level classes in 2017

In 2017, these four courses will be taught together, with the students interacting among institutions:

- [ETH Zürich 2017 \(master\)](#) - Prof. Emilio Frazzoli, Dr. Andrea Censi;
- [University of Montreal, 2017 \(master\)](#) - Prof. Liam Paull;
- [TTI/Chicago 2017 \(master\)](#) - Prof. Matthew Walter; and
- National Chiao Tung University, Taiwan - Prof. Nick Wang.

Furthermore, the Duckietown platform is used also in the following universities:

- Rensselaer Polytechnic Institute (Jeff Trinkle)
- National Chiao Tung University, Taiwan - Prof. Yon-Ping Chen's *Dynamic system simulation and implementation* course.
- Chosun University, Korea - Prof. Woosuk Sung's course;
- Petra Christian University, Indonesia - Prof. Resmana Lim's *Mobile Robot Design Course*
- National Tainan Normal University, Taiwan - Prof. Jen-Jee Chen's *Vehicle to Everything (V2X)* course; and
- Yuan Zhu University, Taiwan - Prof. Kan-Lin Hsiung's Control course.

## 2.4. Chile

**TODO:** to write

## 2.5. Duckietown High School

### 1) Introduction

---

DuckietownHS is inspired by the Duckietown project and targeted for high schools. The goal is to build and program duckiebots capable of moving autonomously on the streets of Duckietown. The technical objectives of DuckietownHS are simplified compared to those of the Duckietown project intended for universities so it is perfectly suited to the technical knowledge of the classes involved. The purpose is to create self-driving DuckiebotHS vehicles which can make choices and move autonomously on the streets of Duckietown, using sensors installed on the vehicles and special road signs positioned within Duckietown.

Once DuckiebotHS have been assembled and programmed to meet the specifications contained in this document and issued by the “customer” Perlatecnica, special missions and games will be offered for DuckiebotHS. The participants can also submit their own missions and games.

Just like the university project, DuckietownHS is an open source project, a role-playing game, a means to raise awareness on the subject and a learning experience for everyone involved. The project is promoted by the non-profit organization [Perlatecnica](#) based in Italy.

### 2) Purpose

---

The project has two main purposes:

- It is a course where students and teachers take part in a role play and they take the typical professional roles of an engineering company. They must design and implement a Duckietown responding to the specifications of the project, assemble DuckiebotHS (DBHS), and develop the software that will run on them. The deliverables of the project will be tutorials, how-to, source code, documentation, binaries and images and them will be designed and manufactured according to the procedures of the DTE.
- In respect of that mentioned above, special missions and games for DBHS will be introduced by the “customer” Perlatecnica.

### 3) Perlatecnica’s role

---

Perlatecnica assumes the role of the customer and commissions the Duckietown Engineering company to design and construct the Duckietown and Duckieboths. It will provide all necessary product requirements and will assume the responsibility to validate the compliancy of all deliverables to the required specifications.

### 4) The details of the project

---

The project consists in the design and realization of DuckiebotHS and DuckietownHS. They must have the same characteristics as the city of the University project as far as the size and color of the delimiting roadway bands is concerned but with a different type of management of the traffic lights system that regulates

the passage of DuckiebotHS at intersections. The DuckietownHS (DTHS) and DuckiebotHS (DBHS) are defined in the documentation and there is little room for the DTE to make its own choices in terms of design. The reason for this is that the DBHS produced by the different DTE's need to be identical from a hardware point of view so that the software development makes the difference.

## 5) Where to start

---

The purchase of the necessary materials is the first step to take. For both DTHS and DBHS a list of these materials is provided with links to possible sellers. Even though Amazon is typically indicated as a seller this is nothing more than an indication to facilitate the purchase for those less experienced. It is left to the individual DTE to choose where to buy the required parts. It is allowed to buy and use parts that are not on the list but this is not recommended as they will make the Duckiebot unfit to enter in official competitions. When necessary an assembly tutorial will be provided together with the list of materials. Once the DTHS city and the DBHS robots have been assembled, the next step will be the development of the software for the running of both the city and the DuckiebotHS. The city and the Duckiebot run on a board based on a microcontroller STM32 from STMicroelectronics the Nucleo F401RE that will be programmed via the online development environment mbed. Perlatecnica will not release any of the official codes necessary for the navigation of the DuckiebotHS as these are owned by the DTE who developed them. The full standard document is available on the project official web site.

Each DTE may release the source code under a license Creative Commons CC BY-SA 4.0.

## 6) The first mission of the Duckiebot

---

Once you have completed the assembly of all the parts that make up the Duckietown and DuckiebothS you should start programming the microcontroller so that the Duckiebot can move independently.

The basic mission of the DuckiebotHS is to move autonomously on the roads respecting the road signs and traffic lights, choosing a random journey and without crashing into other DuckiebotHS.

For the development of the code, there are no architectural constraints, but we recommend proceeding with order and to focus primarily on its major functions and not on a specific mission.

The main functions are those of perception and movement.

Moving around in DuckietownHS, the DuckiebotHS will have to drive on straight roads, make 90 degree curves while crossing an intersection but also make other unexpected curves. While doing all this the Duckiebot can be supported by a gyroscope that provides guidance to the orientation of the vehicle.

# Duckietown classes [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT C-4

### First steps [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## PART D

### Duckumentation documentation

#### UNIT D-1

#### Contributing to the documentation

##### 1.1. Where the documentation is

All the documentation is in the repository `duckietown/duckuments`.

The documentation is written as a series of small files in Markdown format.

It is then processed by a series of scripts to create this output:

- [a publication-quality PDF](#);
- [an online HTML version, split in multiple pages](#);
- [a one-page version](#).

##### 1.2. Editing links

The simplest way to contribute to the documentation is to click any of the “✎” icons next to the headers.

They link to the “edit” page in Github. There, one can make and commit the edits in only a few seconds.

## 1.3. Installing the documentation system



In the following, we are going to assume that the documentation system is installed in `~/duckuments`. However, it can be installed anywhere.

We are also going to assume that you have setup a Github account with working public keys.

- [Basic SSH config \(master\)](#).
- [Key pair creation \(master\)](#).
- [Adding public key on Github \(master\)](#).

We are also going to assume that you have installed the `duckietown/software` in `~/duckietown`.

### 1) Dependencies (Ubuntu 16.04)

---



On Ubuntu 16.04, these are the dependencies to install:

```
$ sudo apt install libxml2-dev libxslt1-dev  
$ sudo apt install libffi6 libffi-dev  
$ sudo apt install python-dev python-numpy python-matplotlib  
$ sudo apt install virtualenv  
$ sudo apt install bibtex2html pdftk  
$ sudo apt install imagemagick
```

### 2) Download the duckuments repo

---



Download the `duckietown/duckuments` repository in the `~/duckuments` directory:

```
$ git lfs clone --depth 100 git@github.com:duckietown/duckuments ~/duckuments
```

Here, note we are using `git lfs clone` – it's much faster, because it downloads the Git LFS files in parallel.

If it fails, it means that you do not have Git LFS installed. See [Git LFS \(master\)](#).

The command `--depth 100` tells it we don't care about the whole history.

### 3) Setup the virtual environment

---



Next, we will create a virtual environment using inside the `~/duckuments` directory. Make sure you are running Python 2.7.x. Python 3.x is not supported at the moment.

Change into that directory:

```
$ cd ~/duckuments
```

Create the virtual environment using `virtualenv`:

```
$ virtualenv --system-site-packages deploy
```

Other distributions: In other distributions you might need to use `venv` instead of `virtualenv`.

Activate the virtual environment:

```
$ source ~/duckuments/deploy/bin/activate
```

#### 4) Setup the `mcdp` external repository

---

Make sure you are in the directory:

```
$ cd ~/duckuments
```

Clone the `mcdp` external repository, with the branch `duckuments`.

```
$ git clone -b duckuments git@github.com:AndreaCensi/mcdp
```

Install it and its dependencies:

```
$ cd ~/duckuments/mcdp  
$ python setup.py develop
```

**Note:** If you get a permission error here, it means you have not properly activated the virtual environment.

Other distributions: If you are not on Ubuntu 16, depending on your system, you might need to install these other dependencies:

```
$ pip install numpy matplotlib
```

You also should run:

```
$ pip install -U SystemCmd==2.0.0
```

## 1.4. Compiling the documentation (updated Sep 12)

**Check before you continue**

Make sure you have deployed and activated the virtual environment. You can

check this by checking which `python` is active:

```
$ which python  
/home/user/duckuments/deploy/bin/python
```

To compile the master versions of the docs, run:

```
$ make master-clean master
```

To see the result, open the file

```
./duckuments-dist/master/duckiebook/index.html
```

If you want to do incremental compilation, you can omit the `clean` and just use:

```
$ make master
```

This will be faster. However, sometimes it might get confused. At that point, do `make master-clean`.

## 1) Compiling the Fall 2017 version only (introduced Sep 12)

To compile the Fall 2017 versions of the docs, run:

```
$ make fall2017-clean fall2017
```

To see the result, open the file

```
./duckuments-dist/master/duckiebook/index.html
```

For incremental compilation, use:

```
$ make fall2017
```

## 2) Single-file compilation

There is also the option to compile one single file.

To do this, use:

```
$ ./compile-single path to .md file
```

This is the fastest way to see the results of the editing; however, there are limitations:

- no links to other sections will work.
- not all images might be found.

## 1.5. The workflow to edit documentation (updated Sep 12)

This is the basic workflow:

1. Create a branch called `yourname`-branch in the `duckuments` repository.
2. Edit the Markdown in the `yourname`-branch branch.
3. Run `make master` to make sure it compiles.
4. Commit the Markdown and push on the `yourname`-branch branch.
5. Create a pull request.
6. Tag the group `duckietown/gardeners`.
  - Create a pull request from the command-line using [hub \(master\)](#).

## 1.6. Reporting problems

First, see the section [Unit D-7 - Markduck troubleshooting](#) for common problems and their resolution.

Please report problems with the duckuments using [the duckuments issue tracker](#). If it is urgent, please tag people (Andrea); otherwise these are processed in batch mode every few days.

If you have a problem with a generated PDF, please attach the offending PDF.

If you say something like “This happens for Figure 3”, then it is hard to know which figure you are referencing exactly, because numbering changes from commit to commit.

If you want to refer to specific parts of the text, please commit all your work on your branch, and obtain the name of the commit using the following commands:

```
$ git -C ~/duckuments rev-parse HEAD      # commit for duckuments  
$ git -C ~/duckuments/mcdp rev-parse HEAD # commit for mcdp
```

## UNIT D-2

### Basic Markduck guide

The Duckiebook is written in Markduck, a Markdown dialect.

It supports many features that make it possible to create publication-worthy materials.

## 2.1. Markdown

The Duckiebook is written in a Markdown dialect.

→ [A tutorial on Markdown.](#)

## 2.2. Variables in command lines and command output

Use the syntax “`![name]`” for describing the variables in the code.

*example*

For example, to obtain:

```
$ ssh robot name.local
```

Use the following:

For example, to obtain:

```
$ ssh ![robot name].local
```

Make sure to quote (with 4 spaces) all command lines. Otherwise, the dollar symbol confuses the LaTeX interpreter.

## 2.3. Character escapes

Use the string “`\$`,” to write the dollar symbol “`$`”, otherwise it gets confused with LaTeX math materials. Also notice that you should probably use “USD” to refer to U.S. dollars.

Other symbols to escape are shown in [Table 2.1](#).

TABLE 2.1. SYMBOLS TO ESCAPE

use <code>\\$</code> ;	instead of <code>\$</code>
use <code>\`</code> ;	instead of <code>`</code>
use <code>\&lt;</code> ;	instead of <code>&lt;</code>
use <code>\&gt;</code> ;	instead of <code>&gt;</code>

## 2.4. Keyboard keys

Use the `kbd` element for keystrokes.

*example*

For example, to obtain:

Press `a` then `Ctrl-C`.

use the following:

Press `<kbd>a</kbd>` then `<kbd>Ctrl</kbd>-<kbd>C</kbd>`.



## 2.5. Figures

For any element, adding an attribute called `figure-id` with value `fig:figure ID` or `tab:table ID` will create a figure that wraps the element.

For example:

```
<div figure-id="fig:figure ID">  
    figure content  
</div>
```

It will create HMTL of the form:

```
<div id='fig:code-wrap' class='generated-figure-wrap'>  
    <figure id='fig:figure ID' class='generated-figure'>  
        <div>  
            figure content  
        </div>  
    </figure>  
</div>
```

To add a caption, add an attribute `figure-caption`:

```
<div figure-id="fig:figure ID" figure-caption="This is my caption">  
    figure content  
</div>
```

Alternatively, you can put anywhere an element `figcaption` with ID `figure id:caption`:

```
<element figure-id="fig:figure ID">  
    figure content  
</element>  
  
<figcaption id='fig:figure ID:caption'>  
    This the caption figure.  
</figcaption>
```

To refer to the figure, use an empty link:

```
Please see [](#fig:figure ID).
```

The code will put a reference to “Figure XX”.

## 2.6. Subfigures

You can also create subfigures, using the following syntax.

```
<div figure-id="fig:big">
  <figcaption>Caption of big figure</figcaption>

  <div figure-id="subfig:first" figure-caption="Caption 1">
    <p style='width:5em;height:5em;background-color:#eef'>first subfig</p>
  </div>

  <div figure-id="subfig:second" figure-caption="Caption 2">
    <p style='width:5em;height:5em;background-color:#fee'>second subfig</p>
  </div>
</div>
```

This is the result:



(a) Caption 1



(b) Caption 2

Figure 2.1. Caption of big figure

By default, the subfigures are displayed one per line.

To make them flow horizontally, add `figure-class="flow-subfigures"` to the external figure `div`. Example:



(a) Caption 1 (b) Caption 2

Figure 2.2. Caption of big figure

## 2.7. Shortcut for tables

The shortcuts `col2`, `col3`, `col4`, `col5` are expanded in tables with 2, 3, 4 or 5 columns.

The following code:

```
<col2 figure-id="tab:mytable" figure-caption="My table">
  <span>A</span>
  <span>B</span>
  <span>C</span>
  <span>D</span>
</col2>
```

gives the following result:

TABLE 2.2. MY TABLE

A	B
C	D

### 1) labels-row1 and labels-col1

---

Use the classes `labels-row1` and `labels-col1` to make pretty tables like the following.

`labels-row1`: the first row is the headers.

`labels-col1`: the first column is the headers.

TABLE 2.3. USING CLASS="LABELS-COL1"

header A	B	C	1
header D	E	F	2
header G	H	I	3

TABLE 2.4. USING CLASS="LABELS-ROW1"

header A	header B	header C
D	E	F
G	H	I
1	2	3

## 2.8. Linking to documentation

### 1) Establishing names of headers

---

You give IDs to headers using the format:

```
### header title {#topic ID}
```

For example, for this subsection, we have used:

```
### Establishing names of headers {#establishing}
```

With this, we have given this header the ID “establishing”.

## 2) How to name IDs - and why it's not automated

---

Some time ago, if there was a section called

```
## My section
```

then it would be assigned the ID “my-section”.

This behavior has been removed, for several reasons.

One is that if you don't see the ID then you will be tempted to just change the name:

```
## My better section
```

and silently the ID will be changed to “my-better-section” and all the previous links will be invalidated.

The current behavior is to generate an ugly link like “autoid-209u31j”.

This will make it clear that you cannot link using the PURL if you don't assign an ID.

Also, I would like to clarify that all IDs are *global* (so it's easy to link stuff, without thinking about namespaces, etc.).

Therefore, please choose descriptive IDs, with at least two IDs.

E.g. if you make a section called

```
## Localization {#localization}
```

that's certainly a no-no, because “localization” is too generic.



```
## Localization {#intro-localization}
```

Also note that you don't *need* to add IDs to everything, only the things that people could link to. (e.g. not subsubsections)

### 3) Linking from the documentation to the documentation

---

You can use the syntax:

```
[](#topic ID)
```

to refer to the header.

You can also use some slightly more complex syntax that also allows to link to only the name, only the number or both ([Table 2.5](#)).

TABLE 2.5. SYNTAX FOR REFERRING TO SECTIONS.

See `[](#establishing)`.

See [Subsection 2.8.1 - Establishing names of headers](#)

See `<a class="only_name" href="#establishing"></a>`.

See [Establishing names of headers](#).

See `<a class="only_number" href="#establishing"></a>`.

See [2.8.1](#).

See `<a class="number_name" href="#establishing"></a>`.

See [Subsection 2.8.1 - Establishing names of headers](#).

### 4) Linking to the documentation from outside the documentation

---

You are encouraged to put links to the documentation from the code or scripts.

To do so, use links of the form:

```
http://purl.org/dth/topic ID
```

Here “`dth`” stands for “Duckietown Help”. This link will get redirected to the corresponding document on the website.

For example, you might have a script whose output is:

```
$ rosrun mypackage myscript  
Error. I cannot find the scuderia file.  
See: http://purl.org/dth/scuderia
```

When the user clicks on the link, they will be redirected to [The “scuderia” \(vehicle database\) \(master\)](#).

## UNIT D-3

# Special paragraphs and environments

## 3.1. Special paragraphs tags

The system supports parsing of some special paragraphs.

**Note:** some of these might be redundant and will be eliminated. For now, I am documenting what is implemented.

### 1) Special paragraphs must be separated by a line

A special paragraph is marked by a special prefix. The list of special prefixes is given in the next section.

There must be an empty line before a special paragraph; this is because in Markdown a paragraph starts only after an empty line.

This is checked automatically, and the compilation will abort if the mistake is found.

For example, this is invalid:

```
See: this book  
See: this other book
```

This is correct:

```
See: this book  
See: this other book
```

Similarly, this is invalid:

```
Author: author  
Maintainer: maintainer
```

and this is correct:

Author: author

Maintainer: maintainer

## 2) Todos, task markers

---

TODO: todo

**TODO:** todo

TOWRITE: towrite

To write: towrite

Task: task

**Task:** task

Assigned: assigned

| Assigned to: assigned

## 3) Notes and remarks

---

Remark: remark

| **Remark:** remark

Note: note

| **Note:** note

Warning: warning

| **Warning:** warning

#### 4) Troubleshooting

---

Symptom: symptom

| Symptom: symptom

Resolution: resolution

Resolution: resolution

#### 5) Guidelines

---

Bad: bad

\* bad

Better: better

✓ better

#### 6) Questions and answers

---

Q: question

*Q: question*

A: answer

Answer: answer

#### 7) Authors, maintainers, Point of Contact

---

Maintainer: maintainer

Maintainer: maintainer

Author: author

**Author:** author

Point of Contact: Point of Contact name

**Point of contact:** Point of Contact name

Slack channel: slack channel name

**Slack channel:** slack channel name

## 8) References

---

See: see

→ see

Reference: reference

→ reference

Requires: requires

**Requires:** requires

Results: results

**Results:** results

Next steps: next steps

**Next:** next steps

Recommended: recommended

## | Recommended: recommended

See also: see also

\* see also

## 3.2. Other div environments

For these, note the rules:

- You must include `markdown="1"`.
- There must be an empty line after the first `div` and before the closing `/div`.

### 1) Example usage

```
<div class='example-usage' markdown='1'>  
  
This is how you can use `rosbag`:  
  
$ rosbag play log.bag  
  
</div>
```

*example*

This is how you can use `rosbag`:

`$ rosbag play log.bag`

### 2) Check

```
<div class='check' markdown='1'>  
  
Check that you didn't forget anything.  
  
</div>
```

**Check before you continue**

Check that you didn't forget anything.

### 3) Requirements

```
<div class='requirements' markdown='1'>
```

List of requirements at the beginning of setup chapter.

```
</div>
```

## KNOWLEDGE AND ACTIVITY GRAPH

List of requirements at the beginning of setup chapter.

### 3.3. Notes and questions



There are three environments: “comment”, “question”, “doubt”, that result in boxes that can be expanded by the user.

These are the one-paragraph forms:

Comment: this is a comment on one paragraph.

+ comment

this is a comment on one paragraph.

Question: this is a question on one paragraph.

+ question

this is a question on one paragraph.

Doubt: I have my doubts on one paragraph.

+ doubt

I have my doubts on one paragraph.

These are the multiple-paragraph forms:

```
<div class='comment' markdown='1'>
```

A comment...

A second paragraph...

```
</div>
```

+ comment

A comment...

A second paragraph...

```
<div class='question' markdown='1'>  
A question...  
  
A second paragraph...  
</div>
```

+ question

A question...

A second paragraph...

```
<div class='doubt' markdown='1'>  
A question...  
  
Should it not be:  
  
$ alternative command  
  
A second paragraph...  
</div>
```

+ doubt

A question...

Should it not be:

\$ alternative command

A second paragraph...

## UNIT D-4

### Using LaTeX constructs in documentation



#### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Working knowledge of LaTeX.

#### 4.1. Embedded LaTeX

You can use \$LaTeX\$ math, environment, and references. For example, take a look at

$$\$ \$ x^2 = \int_0^t f(\tau) \text{d}\tau \$ \$$$

or refer to [Proposition 1 - Proposition example](#).

**Proposition 1.** (Proposition example) This is an example proposition:  $2x = x + x$ .

The above was written as in [Listing 4.1](#).

You can use `\LaTeX` math, environment, and references.  
For example, take a look at

```
\[  
x^2 = \int_0^t f(\tau) \text{d}\tau  
\]
```

or refer to `[](#prop:example)`.

```
\begin{proposition}[Proposition example]\label{prop:example}  
This is an example proposition: $2x = x + x$.  
\end{proposition}
```

Listing 4.1. Use of LaTeX code.

For the LaTeX environments to work properly you *must* add a `\label` declaration inside. Moreover, the label must have a prefix that is adequate to the environment. For example, for a proposition, you must insert `\label{prop:name}` inside.

The following table shows the list of the LaTeX environments supported and the label prefix that they need.

TABLE 4.1. LATEX ENVIRONMENTS AND LABEL PREFIXES

definition	def: <code>name</code>
proposition	prop: <code>name</code>
remark	rem: <code>name</code>
problem	prob: <code>name</code>
theorem	thm: <code>name</code>
lemma	lem: <code>name</code>

Examples of all environments follow.

*example*

```
\begin{definition} \label{def:lorem}  
Lorem  
\end{definition}
```

**Definition 1.** Lorem

```
\begin{proposition} \label{prop:lorem}  
Lorem  
\end{proposition}
```

**Proposition 2.** Lorem

```
\begin{remark} \label{rem:lorem}  
Lorem  
\end{remark}
```

### Remark 1. Lorem

```
\begin{problem} \label{prob:lorem}
Lorem
\end{problem}
```

### Problem 1. Lorem

```
\begin{example} \label{exa:lorem}
Lorem
\end{example}
```

### Example 1. Lorem

```
\begin{theorem} \label{thm:lorem}
Lorem
\end{theorem}
```

### Theorem 1. Lorem

```
\begin{lemma} \label{lem:lorem}
Lorem
\end{lemma}
```

### Lemma 1. Lorem

I can also refer to all of them:  
[](#def:lorem),  
[](#prop:lorem),  
[](#rem:lorem),  
[](#prob:lorem),  
[](#exa:lorem),  
[](#thm:lorem),  
[](#lem:lorem).

I can also refer to all of them: [Definition 1 -](#), [Proposition 2 -](#), [Remark 1 -](#), [Problem 1 -](#), [Example 1 -](#), [Theorem 1 -](#), [Lemma 1 -](#).

## 4.2. LaTeX equations



We can refer to equations, such as `\eqref{eq:one}`:

```
\begin{equation} 2a = a + a \label{eq:one} \tag{1} \end{equation}
```

This uses `align` and contains `\eqref{eq:two}` and `\eqref{eq:three}`.

```
\begin{align} a &= b \label{eq:two} \\ &= c \label{eq:three} \end{align}
```

We can refer to equations, such as `\eqref{eq:one}`:

```
\begin{equation} 2a = a + a \label{eq:one} \end{equation}
```

This uses `align` and contains `\eqref{eq:two}` and `\eqref{eq:three}`.

```
\begin{align} a &= b \label{eq:two} \\ &= c \label{eq:three} \end{align}
```

Note that referring to the equations is done using the syntax `\eqref{eq:name}`, rather than `[](#eq:name)`.

### 4.3. LaTeX symbols

The LaTeX symbols definitions are in a file called [docs/symbols.tex](#).

Put all definitions there; if they are centralized it is easier to check that they are coherent.

### 4.4. Bibliography support

You need to have installed `bibtex2html`.

The system supports Bibtex files.

Place `*.bib` files anywhere in the directory.

Then you can refer to them using the syntax:

```
[](#bib:bibtex ID)
```

For example:

```
Please see [](#bib:siciliano07handbook).
```

Will result in:

Please see [3].

## 4.5. Embedding Latex in Figures through SVG



### KNOWLEDGE AND ACTIVITY GRAPH



**Requires:** In order to compile the figures into PDFs you need to have Inkscape installed. Instructions to download and install Inkscape are [here](#).

To embed latex in your figures, you can add it directly to a file and save it as `file-name.svg` file and save anywhere in the `/docs` directory.

You can run:

```
$ make process-svg-figs
```

And the SVG file will be compiled into a PDF figure with the LaTeX commands properly interpreted.

You can then include the PDF file in a normal way ([Section 2.5 - Figures](#)) using `file-name.pdf` as the filename in the `<img>` tag.



Figure 4.1. Embedding LaTeX in images

It can take a bit of work to get the positioning of the code to appear properly on the figure.

## UNIT D-5

### Advanced Markduck guide



## 5.1. Embedding videos



It is possible to embed Vimeo videos in the documentation.

**Note:** Do not upload the videos to your personal Vimeo account; they must all be posted to the Duckietown Engineering account.

This is the syntax:

```
<dtvideo src="vimeo:vimeo ID"/>
```

### example

For example, this code:

```
<div figure-id="fig:example-embed">
  <figcaption>Cool Duckietown by night</figcaption>
  <dtvideo src="vimeo:152825632"/>
</div>
```

produces this result:



Figure 5.1. Cool Duckietown by night

Depending on the output media, the result will change:

- On the online book, the result is that a player is embedded.
- On the e-book version, the result is that a thumbnail is produced, with a link to the video;
- On the dead-tree version, a thumbnail is produced with a QR code linking to the video (TODO).

## 5.2. move-here tag

If a file contains the tag `move-here`, the fragment pointed by the `src` attribute is moved at the place of the tag.

This is used for autogenerated documentation.

Syntax:

```
# Node `node`  
<move-here src="#package-node-autogenerated"/>
```

## 5.3. Comments

You can insert comments using the HTML syntax for comments: any text between “`<!--`” and “`-->`” is ignored.

```
# My section  
  
<!-- this text is ignored --&gt;<br/>  
Let's start by...
```

## 5.4. Referring to Github files

You can refer to files in the repository by using:

```
See [this file](github:org=org,repo=repo,path=path,branch=branch).
```

The available keys are:

- `org` (required): organization name (e.g. `duckietown`);
- `repo` (required): repository name (e.g. `Software`);
- `path` (required): the filename. Can be just the file name or also include directories;
- `branch` (optional) the repository branch; defaults to `master`;

For example, you can refer to [the file `pkg\_name/src/subscriber\_node.py`](#) by using the following syntax:

```
See [this file](github:org=duckietown,repo=Software,path=pkg_name/src/subscriber_node.py)
```

You can also refer to a particular line:

This is done using the following parameters:

- `from_text` (optional): reference the first line containing the text;
- `from_line` (optional): reference the line by number;

For example, you can refer to [the line containing “Initialize”](#) of `pkg_name/src/subscriber_node.py` by using the following syntax:

```
For example, you can refer to [the line containing “Initialize”][link2]  
of `pkg_name/src/subscriber_node.py` by using the following syntax:
```

```
[link2]: github:org=duckietown,repo=Software,path=pkg_name/src/  
subscriber_node.py,from_text=Initialize
```

You can also reference a range of lines, using the parameters:

- `to_text` (optional): references the final line, by text;
- `to_line` (optional): references the final line, by number.

You cannot give `from_text` and `from_line` at the same time. You cannot give a `to....` without the `from....`.

For example, [this link refers to a range of lines](#): click it to see how Github highlights the lines from “Initialize” to “spin”.

This is the source of the previous paragraph:

For example, [this link refers to a range of lines][interval]: click it to see how Github highlights the lines from “Initialize” to “spin”.

[interval]: github:org=duckietown,repo=Software,path(pkg\_name/src/ subscriber\_node.py,from\_text=Initialize,to\_text=spin

## 5.5. Putting code from the repository in line

In addition to referencing the files, you can also copy the contents of a file inside the documentation.

This is done by using the tag `display-file`.

For example, you can put a copy of `pkg_name/src/subscriber_node.py` using:

```
<display-file src=""
    github:org=duckietown,
    repo=Software,
    path=pkg_name/src/subscriber_node.py
"/>
```

and the result is the following automatically generated listing:

```

#!/usr/bin/env python
import rospy

# Imports message type
from std_msgs.msg import String

# Define callback function
def callback(msg):
    s = "I heard: %s" % (msg.data)
    rospy.loginfo(s)

# Initialize the node with rospy
rospy.init_node('subscriber_node', anonymous=False)

# Create subscriber
subscriber = rospy.Subscriber("topic", String, callback)

# Runs continuously until interrupted
rospy.spin()

```

Listing 5.2. [subscriber\\_node.py](#)

If you use the `from_text` and `to_text` (or `from_line` and `to_line`), you can actually display part of a file. For example:

```

<display-file src=""
  github:org=duckietown,
  repo=Software,
  path(pkg_name/src/subscriber_node.py,
  from_text=Initialize,
  to_text=spin
  "/>

```

creates the following automatically generated listing:

```

# Initialize the node with rospy
rospy.init_node('subscriber_node', anonymous=False)

# Create subscriber
subscriber = rospy.Subscriber("topic", String, callback)

# Runs continuously until interrupted
rospy.spin()

```

Listing 5.3. [subscriber\\_node.py](#)

## UNIT D-6

### \*Compiling the PDF version



This part describes how to compile the PDF version.

**Note:** The dependencies below are harder to install. If you don't manage to do it, then you only lose the ability to compile the PDF. You can do `make compile` to compile the HTML version, but you cannot do `make compile-pdf`.

## 6.1. Installing nodejs

Ensure the latest version (>6) of `nodejs` is installed.

Run:

```
$ nodejs --version  
6.xx
```

If the version is 4 or less, remove `nodejs`:

```
$ sudo apt remove nodejs
```

Install `nodejs` using [the instructions at this page](#).

Next, install the necessary Javascript libraries using `npm`:

```
$ cd $DUCKUMENTS  
$ npm install MathJax-node jsdom@9.3 less
```

### 1) Troubleshooting nodejs installation problems

---

The only pain point in the installation procedure has been the installation of `nodejs` packages using `npm`. For some reason, they cannot be installed globally (`npm install -g`).

Do not use `sudo` for installation. It will cause problems.

If you use `sudo`, you probably have to delete a bunch of directories, such as: `~/duckuments/node_modules`, `~/.npm`, and `~/.node_modules`, if they exist.

## 6.2. Installing Prince

Install PrinceXML from [this page](#).

## 6.3. Installing fonts

Copy the `~/duckuments/fonts` directory in `~/.fonts`:

```
$ mkdir -p ~/.fonts    # create if not exists  
$ cp -R ~/duckuments/fonts ~/.fonts
```

and then rebuild the font cache using:

```
$ fc-cache -fv
```

## 6.4. Compiling the PDF

To compile the PDF, use:

```
$ make compile-pdf
```

This creates the file:

```
./duckuments-dist/master/duckiebook.pdf
```

# UNIT D-7

## Markduck troubleshooting

### 7.1. Changes don't appear on the website

For these issues, see [Unit D-8 - The Duckuments bot](#).

### 7.2. Troubleshooting errors in the compilation process

**Symptom:** “Invalid XML”

**Resolution:** “Markdown” doesn’t mean that you can put anything in a file. Except for the code blocks, it must be valid XML. For example, if you use “`>`” and “`<`” without quoting, it will likely cause a compile error.

**Symptom:** “Tabs are evil”

**Resolution:** Do not use tab characters. The error message in this case is quite helpful in telling you exactly where the tabs are.

**Symptom:** The error message contains `ValueError: Suspicious math fragment 'KEYMATHS000END-KEY'`

**Resolution:** You probably have forgotten to indent a command line by at least 4 spaces. The dollar in the command line is now being confused for a math formula.

## 7.3. Common mistakes with Markdown

Here are some common mistakes encountered.

### 1) Not properly starting a list

There must be an empty line before the list starts.

This is correct:

```
I want to learn:  
- robotics  
- computer vision  
- underwater basket weaving
```

This is incorrect:

```
I want to learn:  
- robotics  
- computer vision  
- underwater basket weaving
```

and it will be rendered as follows:

I want to learn: - robotics - computer vision - underwater basket weaving

### UNIT D-8

## The Duckuments bot

**Note:** This is an advanced section mainly for Liam.

## 8.1. Documentation deployment

The book is published to a different [repository called duckuments-dist](#) and from there published as [book.duckietown.org](http://book.duckietown.org).

## 8.2. Understand what's going on

There is a bot, called [frankfurt.co-design.science](http://frankfurt.co-design.science), which is an AWS machine (somewhere in Frankfurt).

Every minute, it tries to do the following:

1. It checks out the last version of `mcdp`;
2. It checks out the last version of `duckuments`;

3. It compiles the various versions;
4. It uploads the results to a repository called `duckuments-dist`.

This process takes 4 minutes for an incremental change, and about 10-15 minutes for a big change, such as a change in headers IDs, which implies re-checking all cross-references.

### 8.3. Logging

There are logs you can access to see what's going on.

[The high-level compilation log](#) tells you in what phase of the cycle the bot is. Scroll to the bottom.

Ideally what you want to see is something like the following:

```
Starting
Mon Sep 11 10:49:04 CEST 2017
  succeeded html
  succeeded fall 2017
  succeeded upload
  succeeded split
  succeeded html upload
  succeeded PDF
  succeeded PDF upload
Mon Sep 11 10:54:21 CEST 2017
Done.
```

This shows that the compilation took 5 minutes.

Every two hours you will see something like this:

```
automatic-compile-cleanup killing everything
```

and the next iteration will take longer because it starts from scratch.

[The last log](#) is a live version of the compilation log. This might not be tremendously informative because it is very verbose.

### 8.4. Debugging Github Pages problems

Sometimes, it's Github Pages that lags behind.

To check this, the bot also makes available the compilation output as a website called `book2.duckietown.org`. You can take any URL starting with `book.duckietown.org`, put `book2`, and you will see what is on the server.

This can identify if the problem is Github.

# UNIT D-9

## Documentation style guide

This chapter describes the conventions for writing the technical documentation.

### 9.1. General guidelines for technical writing

The following holds for all technical writing.

- The documentation is written in correct English.
- Do not say “should” when you mean “must”. “Must” and “should” have precise meanings and they are not interchangeable. These meanings are explained [in this document](#).
- “Please” is unnecessary in technical documentation.
  - ✗ “Please remove the SD card.”
  - ✓ “Remove the SD card”.
- Do not use colloquialisms or abbreviations.
  - ✗ “The pwd is `ubuntu`.”
  - ✓ “The password is `ubuntu`.”
  - ✗ “To create a ROS pkg...”
  - ✓ “To create a ROS package...”
- Python is capitalized when used as a name.
  - ✗ “If you are using `python`...”
  - ✓ “If you are using `Python`...”
- Do not use emojis.
- Do not use ALL CAPS.
- Make infrequent use of **bold statements**.
- Do not use exclamation points.

### 9.2. Style guide for the Duckietown documentation

- The English version of the documentation is written in American English. Please note that your spell checker might be set to British English.
  - ✗ `behaviour`
  - ✓ `behavior`
  - ✗ `localisation`

## ✓ localization

- It's ok to use "it's" instead of "it is", "can't" instead of "cannot", etc.
  - All the filenames and commands must be enclosed in code blocks using Markdown backticks.
- ✗ "Edit the `~/.ssh/config` file using `vi`."
  - ✓ "Edit the `~/.ssh/config` file using `vi`."
- `Ctrl`-`C`, `ssh` etc. are not verbs.
    - ✗ "`Ctrl`-`C` from the command line".
    - ✓ "Use `Ctrl`-`C` from the command line".
  - Subtle humor and puns about duckies are encouraged.

## 9.3. Writing command lines

Use either "`laptop`" or "`duckiebot`" (not capitalized, as a hostname) as the prefix for the command line.

For example, for a command that is supposed to run on the laptop, use:

```
laptop $ cd ~/duckietown
```

It will become:

 \$ cd ~/duckietown

For a command that must run on the Duckiebot, use:

```
duckiebot $ cd ~/duckietown
```

It will become:

 \$ cd ~/duckietown

If the command is supposed to be run on both, omit the hostname:

```
$ cd ~/duckietown
```

## 9.4. Frequently misspelled words

- "Duckiebot" is always capitalized.
- Use "Raspberry Pi", not "PI", "raspi", etc.

- These are other words frequently misspelled: 5 GHz WiFi

## 9.5. Other conventions

When the user must edit a file, just say: “edit `/this/file`”.

Writing down the command line for editing, like the following:

```
$ vi /this/file
```

is too much detail.

(If people need to be told how to edit a file, Duckietown is too advanced for them.)

## 9.6. Troubleshooting sections

Write the documentation as if every step succeeds.

Then, at the end, make a “Troubleshooting” section.

Organize the troubleshooting section as a list of symptom/resolution.

The following is an example of a troubleshooting section.

### 1) Troubleshooting

---

**Symptom:** This strange thing happens.

**Resolution:** Maybe the camera is not inserted correctly. Remove and reconnect.

**Symptom:** This other strange thing happens.

**Resolution:** Maybe the plumbus is not working correctly. Try reformatting the plumbus.

## UNIT D-10

### Learning in Duckietown [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT D-11

### Knowledge graph [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT D-12

### Translations [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## PART E

### Exercises

## UNIT E-1

### Exercise: Basic image operations, adult version

Assigned to: Andrea Daniele

#### 1.1. Skills learned

- Dealing with exceptions.
- Using exit conditions.
- Verification and unit tests.

#### 1.2. Instructions

Implement the program `dt-image-flip` specified in the following section.

This time, we specify exactly what should happen for various anomalous conditions. This allows to do automated testing of the program.

#### 1.3. `dt-image-flip` specification

The program `image-ops` expects exactly two arguments: a filename (a JPG file) and a directory name.

```
$ dt-image-flip file outdir
```

If the file does not exist, the script must exit with error code 2.

If the file cannot be decoded, the script must exit with error code 3.

If the file exists, then the script must create:

- `outdir/regular.jpg`: a copy of the initial file
- `outdir/flip.jpg`: the file, flipped vertically.
- `outdir/side-by-side.jpg`: the two files, side by side.

If any other error occurs, the script should exit with error code 99.



(a) The original picture. (b) The output `flip.jpg` (c) The output `side-by-side.jpg`

Figure 1.1. Example input-output for the program `image-ops`.

## 1.4. Useful APIs

### 1) Images side-by-side

+ comment

Good explanation, but shouldn't it go in the previous exercise? -AC

An image loaded using the OpenCV function `imread` is stored in memory as a [NumPy array](#). For example, the image shown above ([Figure 1.1a - The original picture.](#)) will be represented in memory as a NumPy array with shape `(96, 96, 3)`. The first dimension indicates the number of pixels along the `Y-axis`, the second indicates the number of pixels along the `X-axis` and the third is known as *number of channels* (e.g., Blue, Green, and Red).

NumPy provides a utility function called `concatenate` that joins a sequence of arrays along a given axis.

## 1.5. Testing it works with `image-ops-tester`

We provide 4 scripts that can be used to make sure that you wrote a conforming `dt-image-flip`. The scripts are `image-ops-tester-good`, `image-ops-tester-bad1`, `image-ops-tester-bad2`, and `image-ops-tester-bad3`. You can find them in the directory [/exercises/dt-image-flip/image-ops-tester](#) in the [duckietown/duckuments](#) repository.

The script called `image-ops-tester-good` tests your program in a situation in which we expect it to work properly. The 3 “bad” test scripts (i.e., `image-ops-tester-bad1` through `image-ops-tester-bad3`) test your code in situations in which we expect your program to complain in the proper way.

Use them as follows:

```
$ image-ops-tester-scenario candidate-program
```

**Note:** The tester scripts must be called from their own location. Make sure to change your working directory to `/exercises/dt-image-flip/image-ops-tester` before launching the tester scripts.

If the script cannot be found, `image-ops-tester-scenario` will return 1.

`image-ops-tester-scenario` will return 0 if the program exists and conforms to the specification ([Section 1.3 - dt-image-flip specification](#)).

If it can establish that the program is not good, it will return 11.

## Bottom line

Things that are not tested are broken.

## UNIT E-2

### Exercise: Simple data analysis from a bag

Assigned to: Andrea Daniele

#### 2.1. Skills learned

- Reading Bag files.
- Statistics functions (mean, median) in Numpy.
- Use YAML format.

#### 2.2. Instructions

Create an implementation of `dt-bag-analyze` according to the specification below.

#### 2.3. Specification for `dt-bag-analyze`

Create a program that summarizes the statistics of data in a bag file.

```
$ dt-bag-analyze bag file
```

Compute, for each topic in the bag:

- The total number of messages.
- The minimum, maximum, average, and median interval between successive messages, represented in seconds.

Print out the statistics using the YAML format. Example output:

```
$ dt-bag-analyze bag file
'topic name':
    num_messages: value
    period:
        min: value
        max: value
        average: value
        median: value
```

## 2.4. Useful APIs

### 1) Read a ROS bag

---

A bag is a file format in ROS for storing ROS message data. The package `rosbag` defines the class `Bag` that provides all the methods needed to serialize messages to and from a single file on disk using the bag format.

### 2) Time in ROS

---

In ROS the time is stored as an object of type `rostime.Time`. An object `t`, instance of `rostime.Time`, represents a time instant as the number of seconds since epoch (stored in `t.secs`) and the number of nanoseconds since `t.secs` (stored in `t.nsecs`). The utility function `t.to_sec()` returns the time (in seconds) as a floating number.

## 2.5. Test that it works

Download the ROS bag [`example\_rosbag\_H3.bag`](#). Run your program on it and compare the results:

```
$ dt-bag-analyze example_rosbag.bag
/tesla/camera_node/camera_info:
num_messages: 653
period:
  min: 0.01
  max: 0.05
  average: 0.03
  median: 0.03

/tesla/line_detector_node/segment_list:
num_messages: 198
period:
  min: 0.08
  max: 0.17
  average: 0.11
  median: 0.1

/tesla/wheels_driver_node/wheels_cmd:
num_messages: 74
period:
  min: 0.02
  max: 4.16
  average: 0.26
  median: 0.11
```

## UNIT E-3

### Exercise: Bag in, bag out



Assigned to: Andrea Daniele

#### 3.1. Skills learned

- Processing the contents of a bag to produce another bag.

#### 3.2. Instructions

Implement the program `dt-bag-decimate` as specified below.

#### 3.3. Specification of `dt-bag-decimate`

The program `dt-bag-decimate` takes as argument a bag filename, an integer value greater than zero, and an output bag file:

```
$ dt-bag-decimate "input bag" n "output bag"
```

The output bag contains the same topics as the input bag, however, only 1 in  $n$  messages from each topic are written. (If  $n$  is 1, the output is the same as the input.)

### 3.4. Useful new APIs

#### 1) Create a new Bag

---

In ROS, a new bag can be created by specifying the mode `w` (i.e., write) while instantiating the class `rosbag.Bag`.

For example:

```
from rosbag import Bag
new_bag = Bag('/output_bag.bag', mode='w')
```

Visit the documentation page for the class [rosbag.Bag](#) for further information.

#### 2) Write message to a Bag

---

A ROS bag instantiated in `write` mode accepts messages through the function [write\(\)](#).

### 3.5. Check that it works

To check that the program works, you can compute the statistics of the data using the program `dt-bag-analyze` that you have created in [Unit E-2 - Exercise: Simple data analysis from a bag](#).

You should see that the statistics have changed.

## UNIT E-4

### Exercise: Bag thumbnails

Assigned to: Andrea Daniele

#### 4.1. Skills learned

- Reading images from images topic in a bag file.

#### 4.2. Instructions

Write a program `dt-bag-thumbnails` as specified below.

#### 4.3. Specification for `dt-bag-thumbnails`

The program `dt-bag-thumbnails` creates thumbnails for some image stream topic in a bag file.

The syntax is:

```
$ dt-bag-thumbnails bag topic output_dir
```

This should create the files:

```
output_dir/00000.jpg  
output_dir/00001.jpg  
output_dir/00002.jpg  
output_dir/00003.jpg  
output_dir/00004.jpg  
...
```

where the progressive number is an incremental counter for the frames.

## 4.4. Test data

If you don't have a ROS bag to work on, you can download the test bag [example\\_ros-bag\\_H5.bag](#). You should be able to get a total of 653 frames out of it.

## 4.5. Useful APIs

### 1) Read image from a topic

---

The [duckietown\\_utils \(master\)](#) package provides the utility function `rgb_from_ros() (master)` that processes a ROS message and returns the RGB image it contains (if any).

### 2) Color space conversion

---

In OpenCV, an image can be converted from one color space (e.g., BGR) to another supported color space (e.g., RGB). OpenCV provides a list of supported conversions. A `colorConversionCode` defines a conversion between two different color spaces. An exhaustive list of color conversion codes can be found [here](#). The conversion from a color space to another is done with the function `cv.cvtColor`.

## UNIT E-5

### Exercise: Instagram filters

Assigned to: Andrea Daniele

## 5.1. Skills learned

- Image pixel representation;
- Image manipulation;
- The idea that we can manipulate operations as objects, and refer to them (higher-order computation);
- The idea that we can compose operations, and sometimes the operations do commute, while sometimes they do not.

## 5.2. Instructions

Create `dt-instagram` as specified below.

### 5.3. Specification for `dt-instagram`

Write a program `dt-instagram` that applies a list of filters to an image.

The syntax to invoke the program is:

```
$ dt-instagram image in filters image out
```

where:

- `image in` is the input image;
- `filters` is a string, which is a colon-separated list of filters;
- `image out` is the output image.

The list of filters is given in [Subsection 5.3.1 - List of filters](#).

For example, the result of the command:

```
$ dt-instagram image.jpg flip-horizontal:grayscale out.jpg
```

is that `out.jpg` contains the input image, flipped and then converted to grayscale.

Because these two commute, this command gives the same output:

```
$ dt-instagram image.jpg grayscale:flip-horizontal out.jpg
```

#### 1) List of filters

Here is the list of possible values for the filters, and their effect:

- `flip-vertical`: flips the image vertically
- `flip-horizontal`: flips the image horizontally
- `grayscale`: Makes the image grayscale
- `sepia`: make the image sepia

## 5.4. Useful new APIs

### 1) User defined filters

In OpenCV it is possible to define custom filters and apply them to an image. A linear filter (e.g., sepia) is defined by a linear 9-dimensional kernel. The `sepia` filter is defined as:

```
$$ K_{\text{sepia}} = \begin{bmatrix} 0.272 & 0.534 & 0.131 \\ \text{newline} 0.349 & 0.686 & 0.168 \\ \text{newline} 0.393 & 0.769 & 0.189 \end{bmatrix} $$
```

A linear kernel describing a color filter defines a linear transformation in the color space. A transformation can be applied to an image in OpenCV by using the function `transform()`.

## UNIT E-6

### Exercise: Bag instagram

Assigned to: Andrea Daniele

## 6.1. Instructions

Create `dt-bag-instagram` as specified below.

## 6.2. Specification for `dt-bag-instagram`

Write a program `dt-bag-instagram` that applies a filter to a stream of images stored in a ROS bag.

The syntax to invoke the program is:

```
$ dt-bag-instagram [bag in] [topic] [filters] [bag out]
```

where:

- `'bag in'` is the input bag;
- `'topic'` is a string containing the topic to process;
- `'filters'` is a string, which is a colon-separated list of filters;
- `'bag out'` is the output bag.

## 6.3. Test data

If you don't have a ROS bag to work on, you can download the test bag [example\\_ros-bag\\_H5.bag](#).

## 6.4. Useful new APIs

### 1) Compress an BGR image into a sensor\_msgs/CompressedImage message

The [duckietown\\_utils \(master\)](#) package provides the utility function [d8\\_compressed\\_image\\_from\\_cv\\_image\(\) \(master\)](#) that takes a BGR image, compresses it and wraps it into a `sensor_msgs/CompressedImage` ROS message.

## 6.5. Check that it works

Play your `bag out` ROS bag file and run the following command to make sure that your program is working.

```
$ rosrun image_view image_view image:=topic _image_transport:=compressed
```

## UNIT E-7

### Exercise: Live Instagram

Assigned to: Andrea Daniele

## 7.1. Skills learned

- Live image processing

## 7.2. Instructions

You may find useful: [Minimal ROS node - pkg\\_name \(master\)](#). That tutorial is about listening to text messages and writing back text messages. Here, we apply the same principle, but to images.

Create a ROS node that takes camera images and applies a given operation, as specified in the next section, and then publishes it.

## 7.3. Specification for the node `dt_live_instagram_<robot name>.node`

Create a ROS node `dt_live_instagram_<robot name>.node` that takes a parameter called `filter`, where the filter is something from the list [Subsection 5.3.1 - List of filters](#).

You should launch your camera and joystick from ‘`~/duckietown`’ with



```
$ make demo-joystick-camera
```

Then launch your node with



```
$ roslaunch dt_live_instagram_<robot name> dt_live_instagram_<robot name>_node.launch filter:=<filter>
```

This program should do the following:

- Subscribe to the camera images, by finding a topic that is called `.../compressed`. Call the name of the topic `topic` (i.e., `topic = ...`).
- Publish to the topic `topic/filter/compressed` a stream of images (i.e., video frames) where the filter is applied to the images.

## 7.4. Check that it works

```
$ rqt_image_view
```

and look at `topic/filter/compressed`

## UNIT E-8

### Exercise: Augmented Reality

Assigned to: Jonathan Michaux and Dzenan Lapandic

## 8.1. Skills learned

- Understanding of all the steps in the image pipeline.
- Writing markers on images to aid in debugging.

## 8.2. Introduction

During the lectures, we have explained one direction of the image pipeline:

```
image -> [feature extraction] -> 2D features -> [ground projection] -> 3D world coordinates
```

In this exercise, we are going to look at the pipeline in the opposite direction.

It is often said that:

“The inverse of computer vision is computer graphics.”

The inverse pipeline looks like this:

```
3D world coordinates -> [image projection] -> 2D features -> [rendering] -> image
```

## 8.3. Instructions

- Do intrinsics/extrinsics camera calibration of your robot as per the instructions.
- Write the ROS node specified below in [Section 8.4 - Specification of dt\\_augmented\\_reality](#).

Then verify the results in the following 3 situations.

#### 1) Situation 1: Calibration pattern

---

- Put the robot in the middle of the calibration pattern.
- Run the program `dt_augmented_reality` with map file `calibration_pattern.yaml`.

(Adjust the position until you get perfect match of reality and augmented reality.)

#### 2) Situation 2: Lane

---

- Put the robot in the middle of a lane.
- Run the program `dt_augmented_reality` with map file `lane.yaml`.

(Adjust the position until you get a perfect match of reality and augmented reality.)

#### 3) Situation 3: Intersection

---

- Put the robot at a stop line at a 4-way intersection in Duckietown.
- Run the program `dt_augmented_reality` with map file `intersection_4way.yaml`.

(Adjust the position until you get a perfect match of reality and augmented reality.)

#### 4) Submission

---

Submit the images according to location-specific instructions.

## 8.4. Specification of `dt_augmented_reality`

In this assignment you will be writing a ROS package to perform the augmented reality exercise. The program will be invoked with the following syntax:

```
$ rosrun dt_augmented_reality robot name augmenter.launch map_file:=map file
robot name:=robot name local:=1
```

where `map file` is a YAML file containing the map (specified in [Section 8.5 - Specification of the map](#)).

If `robot name` is not given, it defaults to the hostname.

The program does the following:

1. It loads the intrinsic / extrinsic calibration parameters for the given robot.
2. It reads the map file.
3. It listens to the image topic `/robot name/camera_node/image/compressed`.
4. It reads each image, projects the map features onto the image, and then writes

the resulting image to the topic `![robot name]/AR/![map file basename]/image/compressed`

where `map file basename` is the basename of the file without the extension.

We provide you with ROS package template that contains the `AugmentedRealityNode`. By default, launching the `AugmentedRealityNode` should publish raw images from the camera on the new `![robot name]/AR/![map file basename]/image/compressed` topic.

In order to complete this exercise, you will have to fill in the missing details of the `AugmentedRealityNode` class by doing the following:

1. Implement a method called `process_image` that undistorts raw images.
2. Implement a method called `ground2pixel` that transforms points in ground coordinates (i.e. the robot reference frame) to pixels in the image.
3. Implement a method called `callback` that writes the augmented image to the appropriate topic.

## 8.5. Specification of the map



The map file contains a 3D polygon, defined as a list of points and a list of segments that join those points.

The format is similar to any data structure for 3D computer graphics, with a few changes:

1. Points are referred to by name.
2. It is possible to specify a reference frame for each point. (This will help make this into a general tool for debugging various types of problems).

Here is an example of the file contents, hopefully self-explanatory.

The following map file describes 3 points, and two lines.

```
points:  
    # define three named points: center, left, right  
    center: [axle, [0, 0, 0]] # [reference frame, coordinates]  
    left: [axle, [0.5, 0.1, 0]]  
    right: [axle, [0.5, -0.1, 0]]  
  
segments:  
    - points: [center, left]  
      color: [rgb, [1, 0, 0]]  
    - points: [center, right]  
      color: [rgb, [1, 0, 0]]
```

### 1) Reference frame specification

---



The reference frames are defined as follows:

- `axle`: center of the axle; coordinates are 3D.
- `camera`: camera frame; coordinates are 3D.

- `image01`: a reference frame in which 0,0 is top left, and 1,1 is bottom right of the image; coordinates are 2D.

(Other image frames will be introduced later, such as the `world` and `tile` reference frame, which need the knowledge of the location of the robot.)

## 2) Color specification

---

RGB colors are written as:

```
[rgb, [R, G, B]]
```

where the RGB values are between 0 and 1.

Moreover, we support the following strings:

- `red` is equivalent to `[rgb, [1,0,0]]`
- `green` is equivalent to `[rgb, [0,1,0]]`
- `blue` is equivalent to `[rgb, [0,0,1]]`
- `yellow` is equivalent to `[rgb, [1,1,0]]`
- `magenta` is equivalent to `[rgb, [1,0,1]]`
- `cyan` is equivalent to `[rgb, [0,1,1]]`
- `white` is equivalent to `[rgb, [1,1,1]]`
- `black` is equivalent to `[rgb, [0,0,0]]`

## 8.6. “Map” files

### 1) `hud.yaml`

---

This pattern serves as a simple test that we can draw lines in image coordinates:

```
points:  
    TL: [image01, [0, 0]]  
    TR: [image01, [0, 1]]  
    BR: [image01, [1, 1]]  
    BL: [image01, [1, 0]]  
segments:  
    - points: [TL, TR]  
      color: red  
    - points: [TR, BR]  
      color: green  
    - points: [BR, BL]  
      color: blue  
    - points: [BL, TL]  
      color: yellow
```

The expected result is to put a border around the image: red on the top, green on the right, blue on the bottom, yellow on the left.

## 2) calibration\_pattern.yaml

---

This pattern is based off the checkerboard calibration target used in estimating the intrinsic and extrinsic camera parameters:

```
points:  
    TL: [axle, [0.315, 0.093, 0]]  
    TR: [axle, [0.315, -0.093, 0]]  
    BR: [axle, [0.191, -0.093, 0]]  
    BL: [axle, [0.191, 0.093, 0]]  
  
segments:  
- points: [TL, TR]  
  color: red  
- points: [TR, BR]  
  color: green  
- points: [BR, BL]  
  color: blue  
- points: [BL, TL]  
  color: yellow
```

The expected result is to put a border around the inside corners of the checkerboard: red on the top, green on the right, blue on the bottom, yellow on the left.

## 3) lane.yaml

---

We want something like this:



Then we have:

```
points:  
    p1: [axle, [0, 0.2794, 0]]  
    q1: [axle, [D, 0.2794, 0]]  
    p2: [axle, [0, 0.2286, 0]]  
    q2: [axle, [D, 0.2286, 0]]  
    p3: [axle, [0, 0.0127, 0]]  
    q3: [axle, [D, 0.0127, 0]]  
    p4: [axle, [0, -0.0127, 0]]  
    q4: [axle, [D, -0.0127, 0]]  
    p5: [axle, [0, -0.2286, 0]]  
    q5: [axle, [D, -0.2286, 0]]  
    p6: [axle, [0, -0.2794, 0]]  
    q6: [axle, [D, -0.2794, 0]]  
  
segments:  
- points: [p1, q1]  
  color: white  
- points: [p2, q2]  
  color: white  
- points: [p3, q3]  
  color: yellow  
- points: [p4, q4]  
  color: yellow  
- points: [p5, q5]  
  color: white  
- points: [p6, q6]  
  color: white
```

#### 4) intersection\_4way.yaml

---

points:

NL1: [axle, [0.247, 0.295, 0]]  
NL2: [axle, [0.347, 0.301, 0]]  
NL3: [axle, [0.218, 0.256, 0]]  
NL4: [axle, [0.363, 0.251, 0]]  
NL5: [axle, [0.400, 0.287, 0]]  
NL6: [axle, [0.489, 0.513, 0]]  
NL7: [axle, [0.360, 0.314, 0]]  
NL8: [axle, [0.366, 0.456, 0]]  
NC1: [axle, [0.372, 0.007, 0]]  
NC2: [axle, [0.145, 0.008, 0]]  
NC3: [axle, [0.374, -0.0216, 0]]  
NC4: [axle, [0.146, -0.0180, 0]]  
NR1: [axle, [0.209, -0.234, 0]]  
NR2: [axle, [0.349, -0.237, 0]]  
NR3: [axle, [0.242, -0.276, 0]]  
NR4: [axle, [0.319, -0.274, 0]]  
NR5: [axle, [0.402, -0.283, 0]]  
NR6: [axle, [0.401, -0.479, 0]]  
NR7: [axle, [0.352, -0.415, 0]]  
NR8: [axle, [0.352, -0.303, 0]]  
CL1: [axle, [0.586, 0.261, 0]]  
CL2: [axle, [0.595, 0.632, 0]]  
CL3: [axle, [0.618, 0.251, 0]]  
CL4: [axle, [0.637, 0.662, 0]]  
CR1: [axle, [0.565, -0.253, 0]]  
CR2: [axle, [0.567, -0.607, 0]]  
CR3: [axle, [0.610, -0.262, 0]]  
CR4: [axle, [0.611, -0.641, 0]]  
FL1: [axle, [0.781, 0.718, 0]]  
FL2: [axle, [0.763, 0.253, 0]]  
FL3: [axle, [0.863, 0.192, 0]]  
FL4: [axle, [1.185, 0.172, 0]]  
FL5: [axle, [0.842, 0.718, 0]]  
FL6: [axle, [0.875, 0.271, 0]]  
FL7: [axle, [0.879, 0.234, 0]]  
FL8: [axle, [1.180, 0.209, 0]]  
FC1: [axle, [0.823, 0.0162, 0]]  
FC2: [axle, [1.172, 0.00117, 0]]  
FC3: [axle, [0.845, -0.0100, 0]]  
FC4: [axle, [1.215, -0.0181, 0]]  
FR1: [axle, [0.764, -0.695, 0]]  
FR2: [axle, [0.768, -0.263, 0]]  
FR3: [axle, [0.810, -0.202, 0]]  
FR4: [axle, [1.203, -0.196, 0]]  
FR5: [axle, [0.795, -0.702, 0]]  
FR6: [axle, [0.803, -0.291, 0]]  
FR7: [axle, [0.832, -0.240, 0]]  
FR8: [axle, [1.210, -0.245, 0]]

segments:

- points: [NL1, NL2]  
color: white
- points: [NL3, NL4]  
color: white

## 8.7. Suggestions

Start by using the file `hud.yaml`. To visualize it, you do not need the calibration data. It will be helpful to make sure that you can do the easy parts of the exercise: loading the map, and drawing the lines.

## 8.8. Useful APIs

### 1) Loading a map file:

---

To load a map file, use the function `load_map` provided in `duckietown_utils`:

```
from duckietown_utils import load_map  
  
map_data = load_map(map_filename)
```

(Note that `map` is a reserved symbol name in Python.)

### 2) Reading the calibration data for a robot

---

To load the *intrinsic* calibration parameters, use the function `load_camera_intrinsics` provided in `duckietown_utils`:

```
from duckietown_utils import load_camera_intrinsics  
  
intrinsics = load_camera_intrinsics(robot_name)
```

To load the *extrinsic* calibration parameters (i.e. ground projection), use the function `load_homography` provided in `duckietown_utils`:

```
from duckietown_utils import load_homography  
  
H = load_homography(robot_name)
```

### 3) Path name manipulation

---

From a file name like `"/path/to/map1.yaml"`, you can obtain the basename without extension `.yaml` by using the function `get_base_name` provided in `duckietown_utils`:

```
from duckietown_utils import get_base_name  
  
filename = "/path/to/map1.yaml"  
map_name = get_base_name(filename) # = "map1"
```

### 4) Undistorting an image

---

To remove the distortion from an image, use the function `rectify` provided in `duckietown_utils`:

```
from duckietown_utils import rectify  
  
rectified_image = rectify(image, intrinsics)
```

## 5) Drawing primitives

---

To draw the line segments specified in a map file, use the `render_segments` method defined in the `Augmenter` class:

```
class Augmenter():  
  
    # ...  
  
    def ground2pixel(self):  
        '''Method that transforms ground points  
        to pixel coordinates'''  
        # Your code goes here.  
        return "????"  
  
  
  
    image_with_drawn_segments = augmenter.render_segments(image)
```

In order for `render_segments` to draw segments on an image, you must first implement the method `ground2pixel`.

## UNIT E-9

### Exercise: Lane Filtering - Extended Kalman Filter

Assigned to: Liam Paull

#### 9.1. Skills learned

- Understanding of basic filtering concepts
- Understanding of an Extended Kalman Filter

#### 9.2. Introduction

During the lectures, we have discussed general filtering techniques, and specifically the **histogram filtering** approach that we are using to estimate our location within a lane in Duckietown.

This is a 2-dimensional filter over  $d$  and  $\theta$ , the lateral displacement in the lane

and the robot heading relative to the direction of the lane.

In this exercise, we will replace the histogram filter with an Extended Kalman Filter.

### 9.3. Instructions

Create a ROS node and package that takes as input the list of line segments detected by the line detector, and outputs an estimate of the robot position within the lane to be used by the lane controller. You should be able to run:



```
$ source DUCKIETOWN_ROOT/environment.sh  
$ source DUCKIETOWN_ROOT/set_vehicle.name.sh  
$ roslaunch dt_filtering ROBOT_NAME lane_following.launch
```

and then follow the instructions in [Unit B-8 - Checkoff: Navigation](#) for trying the lane following demo.

You should definitely look at the existing histogram filter for inspiration.

You may find [this](#) a useful resource to get started.

#### 1) Workflow Tip

While you are working on your node and it is crashing, you need not kill and relaunch the entire stack (or even launch on your robot). You should build a workflow whereby you can quickly launch only the node you are developing from your laptop.

### 9.4. Submission

Submit the code using location-specific instructions

UNIT E-10

## Exercise: Lane Filtering - Particle Filter

Assigned to: Jonathan Michaux, Liam Paull, and Miguel de la Iglesia

#### 10.1. Skills learned

- Understanding of basic filtering concepts
- Understanding of a particle filter

#### 10.2. Introduction

During the lectures, we have discussed general filtering techniques, and specifically the

histogram filtering approach that we are using to estimate our location within a lane in Duckietown.

This is a 2-dimensional filter over  $d$  and  $\theta$ , the lateral displacement in the lane and the robot heading relative to the direction of the lane.

In this exercise, we will replace the histogram filter with a particle filter.

### 10.3. Instructions

Create a ROS node and package that takes as input the list of line segments detected by the line detector, and outputs an estimate of the robot position within the lane to be used by the lane controller. You should be able to run:



```
$ source DUCKIETOWN_ROOT/environment.sh  
$ source DUCKIETOWN_ROOT/set_vehicle.name.sh  
$ roslaunch dt_filtering_ROBOT_NAME lane_following.launch
```

and then follow the instructions in [Unit B-8 - Checkoff: Navigation](#) for trying the lane following demo.

You should definitely look at the existing histogram filter for inspiration.

You may find [this](#) a useful resource to get started.

#### 1) Workflow Tip

---

While you are working on your node and it is crashing, you need not kill and relaunch the entire stack (or even launch on your robot). You should build a workflow whereby you can quickly launch only the node you are developing from your laptop.

### 10.4. Submission

Submit the code using location-specific instructions

## PART F

### Theory Background

#### UNIT F-1

### Probability basics

In this chapter we give a brief review of some basic probabilistic concepts. For a more in-depth treatment of the subject we refer the interested reader to a textbook such as

[\[19\].](#)



## 1.1. Random Variables

The key underlying concept in probabilistic theory is that of an *event*, which is the output of a random trial. Examples of an event include the result of a coin flip turning up HEADS or the result of rolling a die turning up the number “4”.

**Definition 2. (Random Variable)** A (either discrete or continuous) variable that can take on any value that corresponds to the feasible output of a random trial.

For example, we could model the event of flipping a fair coin with the random variable  $X$ . We write the probability that  $X$  takes HEADS as  $p(X=\text{HEADS})$ . The set of all possible values for the variable  $X$  is its *domain*,  $\{X\}$ . In this case,  $\{X\} = \{\text{HEADS}, \text{TAILS}\}$ . Since  $X$  can only take one of two values, it is a *binary* random variable. In the case of a die roll,  $\{X\} = \{1, 2, 3, 4, 5, 6\}$ , and we refer to this as a *discrete* random variable. If the output is real value or a subset of the real numbers, e.g.,  $\{X\} = \text{reals}$ , then we refer to  $X$  as a *continuous* random variable.

Consider once again the coin tossing event. If the coin is fair, we have  $p(X=\text{HEADS})=p(X=\text{TAILS})=0.5$ . Here, the function  $p(x)$  is called the *probability mass function* or pmf. The pmf is shown in [Figure 1.1](#).



Figure 1.1. The pmf for a fair coin toss

Here are some very important properties of  $p(x)$ : -  $0 \leq p(x) \leq 1$  -  $\sum_{x \in \{X\}} = 1$

In the case of a continuous random variable, we will call this function  $f(x)$  and call it a *probability density function*, or pdf.

In the case of continuous RVs, technically the  $p(X=x)$  for any value  $x$  is zero since  $\{X\}$  is infinite. To deal with this, we also define another important function, the *cumulative density function*, which is given by  $F(x) \triangleq p(X \leq x)$ , and now we can define  $f(x) \triangleq \frac{d}{dx}F(x)$ . A pdf and corresponding cdf are shown in [Figure 1.2](#) (This happens to be a Gaussian distribution, defined

more precisely in [Subsection 1.1.8 - The Gaussian Distribution](#)).



Figure 1.2. The continuous pdf and cdf

## 1) Joint Probabilities

If we have two different RVs representing two different events  $X$  and  $Y$ , then we represent the probability of two distinct events  $x \in \text{set}(X)$  and  $y \in \text{mathcal}(Y)$  both happening, which we will denote as following:  $p(X=x \wedge Y=y) = p(x,y)$ . The function  $p(x,y)$  is called *joint distribution*.

## 2) Conditional Probabilities

Again, considering that we have two RVs,  $X$  and  $Y$ , imagine these two events are linked in some way. For example,  $X$  is the numerical output of a die roll and  $Y$  is the binary even-odd output of the same die roll. Clearly these two events are linked since they are both uniquely determined by the same underlying event (the rolling of the die). In this case, we say that the RVs are *dependent* on one another. In the event that we know one of events, this gives us some information about the other. We denote this using the following *conditional distribution*  $p(X=x \mid \text{GIVEN} \mid Y=y) \triangleq p(x|y)$ .

### Check before you continue

Write down the conditional pmf for the scenario just described assuming an oracle tells you that the die roll is even. In other words, what is  $p(x|\text{EVEN})$ ?

(Warning: if you think this is very easy that's good, but don't get over-confident.)

The joint and conditional distributions are related by the following (which could be considered a definition of the joint distribution):

$$\begin{aligned} p(x,y) &= p(x|y)p(y) \end{aligned} \quad \text{label{eq:joint}} \quad \text{tag{1}}$$

and similarly, the following could be considered a definition of the conditional distribution:

$$\begin{aligned} p(x|y) = \frac{p(x,y)}{p(y)} & ; \text{ if } p(y) > 0 \\ \end{aligned} \quad \text{label:condition}\tag{2}$$

In other words, the conditional and joint distributions are inextricably linked (you can't really talk about one without the other).

If two variables are *independent*, then the following relation holds:  
 $p(x,y) = p(x)p(y)$ .

### 3) Bayes' Rule

Upon closer inspection of `\eqref{eq:joint}`, we can see that the choice of which variable to condition upon is completely arbitrary. We can write:

$$p(y|x)p(x) = p(x,y) = p(x|y)p(y)$$

and then after rearranging things we arrive at one of the most important formulas for mobile robotics, Bayes' rule:

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} \quad \text{label:bayes}\tag{3}$$

Exactly why this formula is so important will be covered in more detail in later sections (TODO), but we will give an initial intuition here. [TODO]

Consider that the variable  $X$  represents something that we are trying to estimate but cannot observe directly, and that the variable  $Y$  represents a physical measurement that relates to  $X$ . We want to estimate the distribution over  $X$  given the measurement  $Y$ ,  $p(x|y)$ , which is called the *posterior* distribution. Bayes' rule lets us to do this. For every possible state, you take the probability that this measurement could have been generated,  $p(y|x)$ , which is called the *measurement likelihood*, you multiply it by the probability of that state being the true state,  $p(x)$ , which is called the *prior*, and you normalize over the probability of obtaining that measurement from any state,  $p(y)$ , which is called the *evidence*.

#### Check before you continue

From Wikipedia: Suppose a drug test has a 99% true positive rate and a 99% true negative rate, and that we know that exactly 0.5% of people are using the drug. Given that a person's test gives a positive result, what is the probability that this person is actually a user of the drug.

Answer:  $\approx 33.2\%$ . This answer should surprise you. It highlights the power of the *prior*.

### 4) Marginal Distribution

If we already have a joint distribution  $p(x,y)$  and we wish to recover the single variable distribution  $p(x)$ , we must *marginalize* over the variable  $Y$ . This involves summing (for discrete RVs) or integrating (for continuous RVs) over all values of the variable we wish to marginalize:

$$\begin{aligned} p(x) &= \sum_{\mathcal{Y}} p(x,y) \\ f(x) &= \int p(x,y) dy \end{aligned}$$

This can be thought of as projecting a higher dimensional distribution onto a lower dimensional subspace. For example, consider [Figure 1.3](#), which shows some data plotted on a 2D scatter plot, and then the marginal histogram plots along each dimension of the data.



Figure 1.3. A 2D joint data and 2 marginal 1D histogram plots

Marginalization is an important operation since it allows us to reduce the size of our state space in a principled way.

## 5) Conditional Independence

Two RVs,  $X$  and  $Y$  may be correlated, we may be able to encapsulate the dependence through a third random variable  $Z$ . Therefore, if we know  $Z$



Figure 1.4. A graphical representation of the conditional independence of  $X$  and  $Y$  given  $Z$

+ comment

Is there a discussion of graphical models anywhere? Doing a good job of sufficiently describing graphical models and the dependency relations that they express requires careful thought. Without it, we should refer readers to a graphical models text (e.g., Koller and Friedman, even if it is dense)

## 6) Moments

The  $n$ th moment of an RV,  $X$ , is given by  $E[X^n]$  where  $E[ ]$  is the expectation operator with:

$$E[f(X)] = \sum_{x \in \text{set}(X)} x f(x)$$
 in the discrete case and 
$$E[f(X)] = \int x f(x) dx$$
 in the continuous case.

The 1st moment is the *mean*,  $\mu_X = E[X]$ .

The  $n$ th central moment of an RV,  $X$  is given by  $E[(X - \mu_X)^n]$ . The second central moment is called the *covariance*,  $\sigma^2_X = E[(X - \mu_X)^2]$ .

## 7) Entropy

**Definition 3.** The *entropy* of an RV is a scalar measure of the uncertainty about the value the RV.

A common measure of entropy is the *Shannon entropy*, whose value is given by

$$\begin{aligned} H(X) &= -E[\log_2 p(x)] \end{aligned} \quad \text{\label{eq:shannon}} \quad \text{\tag{4}}$$

This measure originates from communication theory and literally represents how many bits are required to transmit a distribution through a communication channel. For many more details related to information theory we recommend [\[20\]](#).

As an example, we can easily write out the Shannon entropy associated with a binary RV (e.g. flipping a coin) as a function of the probability that the coin turns up heads (call this  $p$ ):

$$\begin{aligned} H(X) &= -p \log_2 p - (1-p) \log_2 (1-p) \end{aligned} \quad \text{\label{eq:binary_entropy}} \quad \text{\tag{5}}$$



Figure 1.5. The Shannon entropy of a binary RV \$X\$

Notice that our highest entropy (uncertainty) about the outcome of the coin flip is when it is a fair coin (equal probability of heads and tails). The entropy decays to 0 as we approach  $p=0$  and  $p=1$  since in these two cases we have no uncertainty about the outcome of the flip. It should also be clear why the function is symmetrical around the  $p=0.5$  value.

## 8) The Gaussian Distribution

---

In mobile robotics we use the Gaussian, or normal, distribution a lot.

+ comment

The banana distribution is the official distribution in robotics! - AC

+ comment

The banana distribution is Gaussian! <http://www.roboticsproceedings.org/rss08/p34.pdf> - LP

The 1-D Gaussian distribution pdf is given by:

$$\begin{aligned} \text{The 1-D Gaussian distribution pdf is given by:} \\ \text{The 1-D Gaussian distribution pdf is given by:} \end{aligned}$$

where  $\mu$  is called the *mean* of the distribution, and  $\sigma$  is called the *standard deviation*. A plot of the 1D Gaussian was previously shown in [Figure 1.2](#).

We will rarely deal with the univariate case and much more often deal with the multi-variate Gaussian:

$$\begin{aligned} \text{\begin{equation}} \mathcal{N}(\text{\state}\mid\text{\bm{\mu}},\text{\bm{\Sigma}}) = \frac{1}{(2\pi)^{D/2}}\text{\bm{\Sigma}}^{1/2}\exp[-\frac{1}{2}(\text{\state}-\text{\bm{\mu}})^T\text{\bm{\Sigma}}^{-1}(\text{\state}-\text{\bm{\mu}})] \end{aligned}$$

The value from the exponent:  $(\text{\state}-\text{\bm{\mu}})^T\text{\bm{\Sigma}}^{-1}(\text{\state}-\text{\bm{\mu}})$  is sometimes written  $\|\text{\state}-\text{\bm{\mu}}\|_{\text{\bm{\Sigma}}}$  and is referred to as the *Mahalanobis distance* or *energy norm*.

Mathematically, the Gaussian distribution has some nice properties as we will see. But is this the only reason to use this as a distribution. In other words, is the assumption of Gaussianicity a good one?

There are two very good reasons to think that the Gaussian distribution is the “right” one to use in a given situation.

1. The *central limit theorem* says that, in the limit, if we sum an increasing number of independent random variables, the distribution approaches Gaussian
2. It can be proven (TODO:ref) that the Gaussian distribution has the maximum entropy subject to a given value for the first and second moments. In other words, for a given mean and variance, it makes the *least* assumptions about the other moments.

Exercise: derive the formula for Gaussian entropy

## UNIT F-2

### Linearity and Vectors [draft]



This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## PART G

### Introduction to autonomy

## UNIT G-1

### Autonomous Vehicles [draft]



This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

# UNIT G-2

## Autonomy overview

Assigned to: Liam

This unit introduces some basic concepts ubiquitous in autonomous vehicle navigation.

### 2.1. Basic Building Blocks of Autonomy

The minimal basic backbone processing pipeline for autonomous vehicle navigation is shown in [Figure 2.1](#).



Figure 2.1. The basic building blocks of any autonomous vehicle

For an autonomous vehicle to function, it must achieve some level of performance for all of these components. The level of performance required depends on the *task* and the *required performance*. In the remainder of this section, we will discuss some of the most basic options. In [the next section](#) we will briefly introduce some of the more advanced options that are used in state-of-the-art autonomous vehicles.

#### 1) Sensors



Figure 2.2. Some common sensors used for autonomous navigation

**Definition 4. (Sensor)** A *sensor* is a device that or mechanism that is capable of generating a measurement of some external physical quantity

In general, sensors have two major types. *Passive* sensors generate measurements without affecting the environment that they are measuring. Examples include inertial sensors, odometers, GPS receivers, and cameras. *Active* sensors emit some form of energy into the environment in order to make a measurement. Examples of this type of sensor include Light Detection And Ranging (LiDAR), Radio Detection And Ranging (RaDAR), and Sound Navigation and Ranging (SoNAR). All of these sensors emit energy (from different spectra) into the environment and then detect some property of the energy that is reflected from the environment (e.g., the time of flight or the phase shift of the signal)

## 2) Raw Data Processing

---

The raw data that is input from a sensor needs to be processed in order to become useful and even understandable to a human.

First, **calibration** is usually required to convert units, for example from a voltage to a physical quantity. As a simple example consider a thermometer, which measures temperature via an expanding liquid (usually mercury). The calibration is the known mapping from amount of expansion of liquid to temperature. In this case it is a linear mapping and is used to put the markings on the thermometer that make it useful as a sensor.

We will distinguish between two fundamentally types of calibrations.

**Definition 5. (Intrinsic Calibration)** An *intrinsic calibration* is required to determine sensor-specific parameters that are internal to a specific sensor.

**Definition 6. (Extrinsic Calibration)** An *extrinsic calibration* is required to determine

the external configuration of the sensor with respect to some reference frame.

### Check before you continue

For more information about reference frames check out [Reference frames \(master\)](#)

Calibration is very important consideration in robotics. In the field, the most advanced algorithms will fail if sensors are not properly calibrated.

Once we have properly calibrated data in some meaningful units, we often do some preprocessing to reduce the overall size of the data. This is true particularly for sensors that generate a lot of data, like cameras. Rather than deal with every pixel value generated by the camera, we will process an image to generate feature-points of interest. In “classical” computer vision many different feature descriptors have been proposed (Harris, BRIEF, BRISK, SURF, SIFT, etc), and more recently Convolutional Neural Networks (CNNs) are being used to learn these features.

The important property of these features is that they should be as easily to associate as possible across frames. In order to achieve this, the feature descriptors should be invariant to nuisance parameters.



Figure 2.3. Top: A raw image with feature points indicated. Bottom: Lines projected onto ground plane using extrinsic calibration and ground projections

## 3) State Estimation

Now that we have used our sensors to generate a set of meaningful measurements, we need to combine these measurements together to produce an estimate of the underlying hidden *state* of the robot and possibly to environment.

**Definition 7.** (State) The state  $\text{\state}_t$  in  $\text{\statesp}$  is a *sufficient statistic* of the environment, i.e. it contains all sufficient information required for the robot to carry out its task in that environment. This can (and usually does) include the *configuration* of the robot itself.

What variables are maintained in the statespace  $\text{\statesp}$  depends on the problem at hand. For example we may just be interested in a single robot's configuration in the plane, in which case  $\text{\state}_t \equiv \text{\pose}_t$ . However, in other cases, such as simultaneous localization and mapping, we may also be tracking the map in the state space.

According to Bayesian principles, any system parameters that are not fully known and deterministic should be maintained in the state space.

In general, we do not have direct access to values in  $\text{\state}$ , instead we rely on our (noisy) sensor measurements to tell us something about them, and then we *infer* the values.



Figure 2.4. Lane Following in Duckietown. \*Top Right\*: Raw image; \*Bottom Right\*: Line detections; \*Top Left\*: Line projections and estimate of robot position within the lane (green arrow); \*Bottom Left\*: Control signals sent to wheels.

The animation in [Figure 2.4](#) shows the lane following procedure. The output of the state estimator produces the **green arrow** in the top left pane.

#### 4) Navigation and Planning

---





Figure 2.5. An example of nested control loops

In general we decompose the task of controlling an autonomous vehicle into a series of **nested control loops**.

The loops are called nested since the output of the outer loop is used as the reference input to the inner loop. An example is shown in [Figure 2.5](#).

**Recommended:** If [Figure 2.5](#) is **VERY** mysterious to you, then you may want to have a quick look in a basic feedback control textbook. For example [\[21\]](#) or [\[22\]](#).

In this case we show three loops. At the outer loop, some goal state is provided. The actual state of the robot is used as the feedback. The controller is the block labeled **Navigation and Motion Planning**. The job of this block is generate a **feasible path** from the current state to the goal state. This is executed in **configuration space** rather than the state space (although these two spaces may happen to be the same they are fundamentally conceptually different).



Figure 2.6. Navigation in Duckietown

## 5) Control

The next inner loop of the nested controller in [Figure 2.5](#) is the [Vehicle Controller](#), which takes as input the reference trajectory generated by the [Navigation and Motion Planning](#) block and the current configuration of the robot, and uses the error between the two to generate a control signal.

The most basic feedback control law (See [Feedback control \(master\)](#)) is called PID (for proportional, integral, derivative) which will be discussed in [PID Control \(master\)](#). For an excellent introduction to this control policy see [Figure 2.7](#).

Figure 2.7. Controlling Self Driving Cars

We will also investigate some more advanced non-linear control policies such as [Model Predictive Control \(master\)](#), which is an optimization based technique.

## 6) Actuation

---

The very innermost control loop deals with actually tracking the correct voltage to be sent to the motors. This is generally executed as close to the hardware level as possible. For example we have a [Stepper Motor HAT](#) [See the parts list](#).

## 7) Infrastructure and Prior Information

---

In general, we can make the autonomous navigation a simpler one by exploiting existing structure, infrastructure, and contextual prior knowledge.

Infrastructure example: Maps or GPS satellites

Structure example: Known color and location of lane markings

Contextual prior knowledge example: Cars tend to follow the *Rules of the Road*

## 2.2. Advanced Building Blocks of Autonomy

The basic building blocks enable static navigation in Duckietown. However, many other components are necessary for more realistic scenarios.

### 1) Object Detection

---



Figure 2.8. Advanced Autonomy: Object Detection

One key requirement is the ability to detect objects in the world such as but not limited to: signs, other robots, people, etc.

## 2) SLAM

---

The simultaneous localization and mapping (SLAM) problem involves simultaneously estimating not only the robot state but also the **map** at the same time, and is a fundamental capability for mobile robotics. In autonomous driving, generally the most common application for SLAM is actual in the map-building task. Once a map is built then it can be pre-loaded and then used for pure localization. A demonstration of this in Duckietown is shown in [Figure 2.9](#).



Figure 2.9. Localization in Duckietown

## 3) Other Advanced Topics

---

Other topics that will be covered include:

- Visual-inertial navigation (VINS)
- Fleet management and coordination
- Scene segmentation
- Deep perception
- Text recognition

# Modern Robotic Systems [draft]



This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT G-4

### Autonomy architectures [draft]



This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT G-5

### Representations [draft]



This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT G-6

### Duckiebot modeling



Obtaining a mathematical model of the Duckiebot is important in order to (a) understand its behavior and (b) design a controller to obtain desired behaviors and performances, robustly.

The Duckiebot uses an actuator (DC motor) for each wheel. By applying different torques to the wheels a Duckiebot can turn, go straight (same torque to both wheels) or stay still (no torque to both wheels). This driving configuration is referred to as *differential drive*.

In this section we will derive the model of a differential drive wheeled robot. The Duckiebot model will receive voltages as input (to the DC motors) and produce a configuration, or pose, as output. The pose describes unequivocally the position and orientation of the Duckiebot with respect to some Newtonian “world” frame.

Different methods can be followed to obtain the Duckiebot model, namely the La-

grangian or Newton-Euler, we choose to describe the latter as it arguably provides a clearer physical insight. Showing the equivalence of these formulations is an interesting exercises that the interested reader can take as a challenge. A useful resource for modeling of a Duckiebot may be found here [27].

### KNOWLEDGE AND ACTIVITY GRAPH

Requires:[k:reference\\_frames \(master\)](#) (inertial, body), [k:intro-transformations \(master\)](#) (Cartesian, polar)

Requires: [k:intro-kinematics \(master\)](#)

Requires: [k:intro-dynamics \(master\)](#)

Suggested: [k:intro-ODEs-to-LTIsys \(master\)](#)

Results: k:diff-drive-robot-model

## 6.1. Preliminaries



TODO: relabel inertial frame -> local frame;  $(\cdot)^I \rightarrow (\cdot)^L$

We first briefly recapitulate on the (reference frames)[#reference-frames] that we will use to model the Duckiebot, with the intent of introducing the notation used throughout this chapter. It is important to note that we restrict the current analysis to the plane, so all of the following in defined in  $\mathbb{R}^2$ .

To describe the behavior of a Duckiebot three reference frames will be used:

- A “*world*” frame: a right handed fixed reference system with origin in some arbitrary point  $O$ . We will indicate variables expressed in this frame with a superscript  $W$ , e.g.,  $X^W$ , unless there is no risk of ambiguity, in which case no superscript will be used.
- An “*inertial*” frame: a fixed reference system parallel to the “*world*” frame, that spans the plane on which the Duckiebot moves. We will denote its axis as  $\{X_I, Y_I\}$ , and it will have origin in point  $A = (x_A, y_A)$ , i.e., the midpoint of the robot’s wheel axle. We will indicate variables expressed in this frame with a superscript  $I$ , e.g.,  $X^I$ , unless there is no risk of ambiguity, in which case no superscript will be used.
- A *body* (or “*robot*”) frame: a local reference frame fixed with respect to the robot, centered in  $A$  as well. The  $x$  axis points in the direction of the front of the robot, and the  $y$  axis lies along the axis between the wheels, so to form a right handed reference system. We denote the robot body frame with  $\{X_R, X_L\}$ . The same superscript convention as above will be used. The wheels will have radius  $R$ .

Note: The robot is assumed to be a rigid body, symmetric, and  $X_r$  coincides with axis of symmetry. Moreover, the wheels are considered identical and at the same distance,  $L$ , from the axle midpoint  $A$ .

Moreover:

- The center of mass  $\mathbf{C}^W = (x_c, y_c)$  of the robot is on the  $x_r$  axis, at a distance  $c$  from  $A$ , i.e.,  $(\mathbf{C}^R = (c, 0))$ ;
- $X^r$  forms an *orientation angle*  $\theta$  with the local horizontal.

These notations are summarized in [Figure 6.1](#).



Figure 6.1. Relevant notations for modeling a differential drive robot

### 1) Moving between frames

---

We briefly recapitulate on a few transformations that we will use throughout this chapter.

*Translations:*

Let  $\mathbf{x}^I = [x^I, y^I]$  be a vector represented in the inertial frame and  $\mathbf{x}^W = [x^W, y^W, 1]^T$  be its augmented version. It is possible to express such vector in the world frame through a translation:

$$\begin{aligned} \text{\textbackslash begin\{align\}} & \quad \text{\textbackslash label\{eq:mod-translation-i2w\}\textbackslash tag\{11\}} & \mathbf{x}^W = & \quad \text{\textbackslash avec\{X}^W\textbackslash at\{T\}\textbackslash avec\{X}^R\}, \text{\textbackslash end\{align\}}} \end{aligned}$$

where the translation matrix  $\mathbf{T}$  is defined as:

$$\begin{aligned} \text{\textbackslash begin\{align\}} \mathbf{T} = & \left[ \begin{array}{ccc} 1 & 0 & x_A \\ 0 & 1 & y_A \\ 0 & 0 & 1 \end{array} \right] \left[ \begin{array}{c} x^I \\ y^I \\ 1 \end{array} \right]. \text{\textbackslash end\{align\}} \end{aligned}$$

In the Euclidian space, each translation preserves distances (norms), i.e., is an isom-

etry. So, for example, a velocity expressed in the inertial frame will have the same magnitude as that velocity expressed in the world frame.

*Rotations:*

Let  $\vec{v}^R = [x^R, y^R]^T$  be a vector represented in the robot frame and  $\vec{v}^R = [x^R, y^R, 1]^T$  its augmented version. It is possible to express such vector in the inertial frame through a rotation:

$$\begin{aligned} \text{\begin{aligned}} & \text{\label{eq:mod-rotation-r2i}\tag{12}} & \vec{v}^I = & \text{\begin{aligned}} \\ & \text{\mathtt{at}(R)(\theta)}\vec{v}^R, & \text{\end{aligned}} \end{aligned}$$

where  $\text{amat}(R)(\theta)$  in  $\text{SO}(2)$  is an orthogonal rotation matrix:

$$\begin{aligned} \text{\begin{aligned}} & \text{\label{eq:mod-rot-mat}\tag{13}} & \text{amat}(R)(\theta) = \left[ \begin{array}{ccc} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{array} \right]. & \text{\end{aligned}}$$

**Note:** The orthogonality condition implies that  $\text{amat}(R)^T(\theta)\text{amat}(R)(\theta) = \text{amat}(R)(\theta)\text{amat}(R)^T(\theta) = \text{amat}(I)$ , hence:  $\text{amat}(R)^T(\theta) = \text{amat}(R)^{-1}(\theta)$ , which is quite nice.

*Roto-translation:*

A corollary of [\eqref{eq:mod-translation-i2w}](#) and [\eqref{eq:mod-rotation-r2i}](#) is that the translations and rotations can be combined in a single transformation, because  $\vec{v}^W = \text{amat}(T)\vec{v}^I = \text{amat}(T)\text{amat}(R)(\theta)\vec{v}^R$ . The combined transformation matrix is given by:

$$\begin{aligned} \text{\begin{aligned}} & \text{\label{eq:mod-rototranslation-mat}\tag{14}} & \text{amat}(T) \text{amat}(R)(\theta) = \left[ \begin{array}{ccc} \cos \theta & -\sin \theta & x_A \\ \sin \theta & \cos \theta & y_A \\ 0 & 0 & 1 \end{array} \right]. & \text{\end{aligned}}$$

## 6.2. Dynamics

While kinematics studies the properties of motions of geometric (i.e., massless) points, dynamical modeling takes into account the actual material distribution of the system. Once mass comes into play, motion is the result of the equilibrium of external forces and torques with inertial reactions. While different approaches can be used to derive these equations, namely the Lagrangian or Newtonian approaches (former based on energy considerations, latter on equilibrium of generalized forces), we choose to follow the Newtonian one here for it grants, arguably, a more explicit physical intuition of the problem. Obviously both methods lead to the same results when the same hypothesis are made.

### 1) Notations

---

For starters, recalling that  $C^r = (c, 0)$  is the center of mass of the robot, we define the relevant notations:

TABLE 6.1. NOTATIONS FOR DYNAMIC MODELING OF A DIFFERENTIAL DRIVE ROBOT

$\$(v_u, v_w)$$	Longitudinal and lateral velocities of \$C\$, robot frame
$\$(a_u, a_w)$$	Longitudinal and lateral accelerations of \$C\$, robot frame
$\$(F_{uR}, F_{uL})$$	Longitudinal forces exerted on the vehicle by the right and left wheels
$\$(F_{wR}, F_{wL})$$	Lateral forces exerted on the vehicle by the right and left wheels
$\$(\tau_R, \tau_L)$$	Torques acting on right and left wheel
$\$\theta, \dot{\theta} =$ $\$\dot{\theta}$	Vehicle orientation and angular velocity
$\$M$$	Vehicle mass
$\$J$$	Vehicle yaw moment of inertia with respect to the center of mass \$C\$

[Figure 6.2](#) summarizes these notations.

Before deriving the dynamic model of the robot, it is useful to recall some elements of polar coordinates kinematics.

## 2) Polar coordinates kinematics

---

Let  $\$avec{r}(t)$  identify a point in the space from the inertial frame at distance  $\$r(t)$  from the \$A\$.

Warning: You might want to refresh [Euler formula](#) to convince yourself about the following.

$$\begin{aligned} \&\begin{aligned} \&\label{eq:mod-polar-kin-deriv}\tag{15} \&\$avec{r}(t) &= r(t) e^{\{j\}\theta(t)} \\ \&\&\$dot{r}(t) &= v_u(t) e^{\{j\}\theta(t)} + v_w(t) e^{\{j\}(\theta(t)+\frac{\pi}{2})} \\ \&\&\ddot{r}(t) &= a_u(t) e^{\{j\}\theta(t)} + a_w(t) e^{\{j\}(\theta(t)+\frac{\pi}{2})}, \end{aligned} \end{aligned}$$

with:

$$\begin{aligned} \&\begin{aligned} \&\label{eq:mod-polar-kin-coeff}\tag{16} v_u(t) &= \dot{r}(t) \\ \&\&a_u(t) &= \dot{v}_u - v_w \dot{\theta} = \ddot{r}(t) - r \dot{\theta}^2 \\ \&\&a_w(t) &= \dot{v}_w + v_u \dot{\theta} = 2 \dot{r} \dot{\theta} + r \ddot{\theta}. \end{aligned} \end{aligned}$$

Keeping `\eqref{eq:mod-polar-kin-deriv}` and `\eqref{eq:mod-polar-kin-coeff}` in mind, it is useful to note (for later use) that, letting  $\$avec{r}(t)$  identify the position of the center of mass \$C\$ in the inertial frame:

$$\begin{aligned} \&\begin{aligned} \&\label{eq:mod-C-world-pos}\tag{17} \left\{ \begin{array}{l} x_C^W(t) \\ y_C^W(t) \end{array} \right\} &= \left\{ \begin{array}{l} x_A(t) + r(t) \cos \theta(t) \\ y_A(t) + r(t) \sin \theta(t) \end{array} \right\} \end{aligned} \end{aligned}$$

and therefore:

$$\begin{aligned} \dot{x}^A I_A(t) &= x^W W_C(t) - v_u(t) \cos\theta + v_w(t) \sin\theta \\ &= y^W W_C(t) - v_u(t) \sin\theta - v_w(t) \cos\theta \end{aligned}$$

### 3) Free body diagram

The next step, and definitely the most critical, is writing the free body diagram of the problem ([Figure 6.2](#)). In this analysis the only forces acting on the robot are those applied from the wheels to the vehicle's chassis. It is important to note that the third passive wheel (omnidirectional or caster) is not being taken into account.



Figure 6.2. Free body diagram of a differential drive robot

### 4) Equilibrium of forces and moments

We derive the dynamic model by imposing the simultaneous equilibrium of forces along the longitudinal and lateral directions in the robot frame with the respective inertial forces, and of the moments around the vertical axis (coming out of the screen) passing through the center of mass of the robotic vehicle.

$$\begin{aligned} M_{u_L}(t) + M_{u_R}(t) &= F_{w_L} + F_{w_R} \\ \ddot{\theta}(t) &= \frac{L}{J} (F_{u_R} - F_{u_L}) + \frac{c}{J} (F_{w_R} + F_{w_L}) \end{aligned}$$

By substituting the [eq:mod-polar-kin-coeff](#) in [eq:mod-dyn-equilibria](#), the equilibrium equations are expressed in terms of accelerations of the center of mass in the robot frame:

$$\dot{v}_u(t) = v_w(t) \dot{\theta}(t) + \frac{F_{u_L} + F_{u_R}}{M}$$

$$\begin{aligned} \text{\label{eq:mod-dyn-equilibria2a}} \text{\tag{20}} \quad & \dot{v}_w(t) = -v_u(t) \\ \dot{\theta}(t) + \frac{F_{w_L} + F_{w_R}}{M} & \text{\label{eq:mod-dyn-equilibria2b}} \text{\tag{21}} \quad \ddot{\theta}(t) = \frac{L(J(F_{u_R} - F_{u_L}) - cJ(F_{w_R} + F_{w_L}))}{M} \text{\label{eq:mod-dyn-equilibria2c}} \text{\tag{22}} \end{aligned}$$

This is a general dynamic model (in the sense of no kinematic constraints) of a differential drive robot under the geometric assumptions listed above. It is noted that it is a coupled and nonlinear model, not exactly a best case scenario (we like linear things, because there are plenty of tools to handle them). When using the general dynamic model above, it makes sense to associate the general kinematic model as well, given by:

$$\begin{aligned} \text{\begin{aligned}} \text{\label{eq:mod-gen-kin-mod}} \text{\tag{23}} \quad & \dot{x}_A(t) = v_u(t) \cos \theta(t) - (v_w(t) - c \dot{\theta}) \sin \theta(t) \\ & \dot{y}_A(t) = v_u(t) \sin \theta(t) + (v_w(t) - c \dot{\theta}) \cos \theta(t) \end{aligned}$$

the above \eqref{eq:mod-gen-kin-mod} can be obtained by recalling on one side that translations are isometric transformations, and on the other side that:

$$\begin{aligned} \text{\begin{aligned}} \text{\label{eq:mod-xCW}} \text{\tag{24}} \quad & \left\{ \begin{array}{l} x_A(t) = x^W C(t) - c \cos \theta(t) \\ y_A(t) = y^W C(t) - c \sin \theta(t) \end{array} \right. \end{aligned}$$

$$\begin{aligned} \text{\begin{aligned}} \text{\label{eq:mod-xCI}} \text{\tag{25}} \quad & \left\{ \begin{array}{l} x_C(t) = v_u(t) \cos \theta(t) - v_w(t) \sin \theta(t) \\ y_C(t) = v_u(t) \sin \theta(t) + v_w(t) \cos \theta(t) \end{array} \right. \end{aligned}$$

**Note:** Equation \eqref{eq:mod-gen-kin-mod} can be rewritten as: \$\$ \text{\label{eq:mod-gen-kin-mod-better}} \text{\tag{2}} \quad \text{avec } \begin{bmatrix} v\_A^I \\ v\_A^R \end{bmatrix} = \text{amat}(R)(\theta) \text{ avec } \begin{bmatrix} v\_A^R \\ v\_A^I \end{bmatrix} \$\$, where: \$\$ \text{\label{eq:mod-v\_A^R}} \text{\tag{3}} \quad \text{avec } \begin{bmatrix} v\_A^R \\ v\_A^I \end{bmatrix} = [v\_u(t), v\_w(t) - c \dot{\theta}]^T. \$\$

In order to simplify the model, we proceed to impose some kinematic constraints.

## 6.3. Kinematics

In this section we derive the kinematic model of a differential drive mobile platform under the assumptions of (a) no lateral slipping and (b) pure rolling of the wheels. We refer to these two assumptions as kinematic constraints.

### 1) Differential drive robot kinematic constraints

The kinematic constraints are derived from two assumptions:

- *No lateral slipping motion*: the robot cannot move sideways, but only in the direction of motion, i.e., its lateral velocity in the robot frame is zero: \$\$ \text{\label{eq:mod-no-lat-slip-constraint-r}} \text{\tag{4}} \quad \dot{y}\_A^r = 0. \$\$

By inverting \eqref{eq:mod-rotation-r2i}, this constraint can be expressed through the inertial frame variables, yielding:

$\$ \$ \backslash label\{eq:mod-no-lat-slip-constraint-i\}\backslash tag{5} \backslash dot y_A(t) \cos \theta(t) - \backslash dot x_A(t) \sin \theta(t) = 0. \$ \$$

Imposing  $\backslash eqref\{eq:mod-no-lat-slip-constraint-i\}$  on  $\backslash eqref\{eq:mod-A-dot-polar\}$  results in:

$\$ \$ \backslash label\{eq:mod-no-lat-slip-constraint-v_w\}\backslash tag{6} v_w(t) = \backslash dot y_C^I(t) \cos \theta(t) - \backslash dot x_C^I(t) \sin \theta(t), \$ \$$

and by recalling that  $C^R = (c, 0)$ :

$\$ \$ \backslash label\{eq:mod-no-lat-slip-constraint-C\}\backslash tag{7} \backslash dot y_C^I(t) \cos \theta(t) - \backslash dot x_C^I(t) \sin \theta(t) = c \dot{\theta}(t). \$ \$$

Hence, we obtain the strongest expression of this constraint:

$\$ \$ \backslash label\{eq:mod-no-lat-slip-final\}\backslash tag{8} v_w(t) = c \dot{\theta}(t), \$ \$$

and therefore:

$\$ \$ \backslash label\{eq:mod-no-lat-slip-final-dot\}\backslash tag{9} \backslash dot v_w(t) = c \ddot{\theta}(t). \$ \$$

**Note:** a simpler way of deriving  $\backslash eqref\{eq:mod-no-lat-slip-final-dot\}$  is noticing, from  $\backslash eqref\{eq:mod-v_A^R\}$ , that  $\dot{y}_A^R = v_w(t) - c \dot{\theta}(t)$ .

- *Pure rolling:* the wheels never slips or skids ([Figure 6.3](#)). Recalling that  $R$  is the radius of the wheels (identical) and letting  $\dot{\varphi}_l$ ,  $\dot{\varphi}_r$  be the angular velocities of the left and right wheels respectively, the velocity of the ground contact point P in the robot frame is given by:



Figure 6.3. Pure rolling kinematic constraint

$$\begin{aligned} \backslash begin\{align\} \backslash label\{eq:mod-pure-rolling\}\backslash tag{26} \left\{ \begin{array}{ll} v_{P,r} &= \\ R \dot{\varphi}_r & \\ v_{P,l} &= R \dot{\varphi}_l \end{array} \right. \end{aligned}$$

Another notable consequence of this assumption is that, always in the robot frame, the full power of the motor can be translated into a propelling force for the vehicle in the longitudinal direction. Or, more simply, it allows to write:

$$\$ \$ \backslash label\{eq:mod-force-and-torque\}\backslash tag{10} F_u(\cdot) = \tau_{(.)}(t), \$ \$$$

where  $\tau_{(.)}(t)$  is the torque exerted by each motor on its wheel  $(\cdot) = \{l, r\}$ .

## 6.4. Differential drive robot kinematic model

In a differential drive robot, controlling the wheels at different speeds generates a rolling motion of rate  $\dot{\omega} = \dot{\theta}$ . In a rotating field there always is a fixed point, the *center of instantaneous curvature* (ICC), and all points at distance  $d$  from it will have a velocity given by  $\dot{\omega} d$ , and direction orthogonal to that of the line connecting the ICC and the wheels (i.e., the *axle*). Therefore, by looking at [Figure 6.1](#), we can write:

$$\begin{aligned} \text{\&= v\_l} \\ \text{\&= v\_r} \end{aligned} \quad \begin{aligned} \dot{\theta} (d-L) \\ \dot{\theta} (d+L) \end{aligned}$$

from which:

$$d = L \frac{v_r + v_l}{v_r - v_l} \quad \dot{\theta} = \frac{v_r - v_l}{2L}$$

A few observations stem from [eqref{eq:mod-kin-2}](#):

- If  $v_r = v_l$  the bot does not turn ( $\dot{\theta} = 0$ ), hence the ICC is not defined;
- If  $v_r = -v_l$ , then the robot “turns on itself”, i.e.,  $d=0$  and  $\text{ICC} \equiv A$ ;
- If  $v_r = 0$  (or  $v_l = 0$ ), the rotation happens around the right (left) wheel and  $d = 2L$  ( $d = L$ ).

**Note:** Moreover, a differential drive robot cannot move in the direction of the ICC, it is a singularity.

By recalling the *no lateral slipping motion* [eqref{eq:mod-no-lat-slip-constraint-r}](#) hypothesis and the *pure rolling* constraint [eqref{eq:mod-pure-rolling}](#), and noticing that the translational velocity of  $A$  in the robot frame is  $v_A = \dot{\theta} d = (v_r + v_l)/2$  we can write:

$$\begin{aligned} \dot{x}_A &= R(\dot{\varphi}_R + \dot{\varphi}_L)/2 \\ \dot{y}_A &= 0 \\ \dot{\theta} &= \omega = R(\dot{\varphi}_R - \dot{\varphi}_L)/(2L) \end{aligned}$$

which in more compact yields the *simplified forward kinematics* in the robot frame:

$$\begin{aligned} \dot{x}_A &= R(\dot{\varphi}_R + \dot{\varphi}_L)/2 \\ \dot{y}_A &= 0 \\ \dot{\theta} &= \omega = R(\dot{\varphi}_R - \dot{\varphi}_L)/(2L) \end{aligned}$$

Finally, by using [eqref{eq:mod-rot-mat}](#), we can recast [eqref{eq:mod-forward-kinematics-robot-frame}](#) in the inertial frame.

**Note:** The *simplified forward kinematics* model of a differential drive vehicle is given by:  $\begin{aligned} \dot{q}^I &= \dot{q}^R(\theta) \\ \dot{x}_A^r &= R(\dot{\varphi}_R + \dot{\varphi}_L)/2 \\ \dot{y}_A^r &= 0 \\ \dot{\theta} &= \omega = R(\dot{\varphi}_R - \dot{\varphi}_L)/(2L) \end{aligned}$

$$\begin{aligned} & \cos \theta & \frac{\partial}{\partial \theta} \cos \theta & \frac{\partial}{\partial \theta} \sin \theta & \frac{\partial}{\partial \theta} \sin \theta \\ & \frac{\partial}{\partial \theta} \frac{\partial}{\partial \theta} \cos \theta & \frac{\partial}{\partial \theta} \frac{\partial}{\partial \theta} \sin \theta & \frac{\partial}{\partial \theta} \frac{\partial}{\partial \theta} \sin \theta \\ & \left[ \begin{array}{c} \dot{\varphi}_R \\ \dot{\varphi}_L \end{array} \right] = \left[ \begin{array}{cc} \cos \theta & 0 \\ 0 & 1 \end{array} \right] \left[ \begin{array}{c} v_A \\ \omega \end{array} \right]. \end{aligned}$$

## 6.5. Simplified dynamic model

By implementing the kinematic constraints formulations derived above, i.e., the no lateral slipping ([\eqref{eq:mod-no-lat-slip-final-dot}](#)) and pure rolling ([\eqref{eq:mod-force-and-torque}](#)) in the general dynamic model, it is straightforward to obtain:

$$\begin{aligned} & \dot{v}_u(t) = c \dot{\theta}(t) \\ & \frac{1}{RM} (\tau_R(t) + \tau_L(t)) = c \dot{\theta}(t) \\ & \ddot{\theta}(t) = - \frac{Mc^2 + J}{LM} \dot{\theta}(t) v_u(t) + \frac{LR(Mc^2 + J)}{LM} (\tau_R(t) - \tau_L(t)) \end{aligned}$$

## 6.6. DC motor dynamic model

The equations governing the behavior of a DC motor are driven by an input *armature voltage*  $V(t)$ :

$$\begin{aligned} V(t) &= R i(t) + L \frac{di}{dt} + e(t) \\ e(t) &= K_b \dot{\varphi}(t) \\ \tau_m(t) &= K_t i(t) \\ \tau_m(t) &= N \tau(t), \end{aligned}$$

where  $(K_b, K_t)$  are the back emf and torque constants respectively and  $N$  is the gear ratio ( $N=1$  in the Duckiebot).

[Figure 6.4](#) shows a diagram of a typical DC motor.



Figure 6.4. Diagram of a DC motor

Having a relation between the applied voltage and torque, in addition to the dynamic

and kinematic models of a differential drive robot, allows us to determine all possible state variables of interest.

**Note:** torque disturbances acting on the wheels, such as the effects of friction, can be modeled as additive terms (of sign opposite to  $\tau$ ) in the DC motor equations.

## 6.7. Conclusions

In this chapter we derived a model for a differential drive robot. Although several simplifying assumption were made, e.g., rigid body motion, symmetry, pure rolling and no lateral slipping - still the model is nonlinear.

Regardless, we now have a sequence of descriptive tools that receive as input the voltage signal sent by the controller, and produce as output any of the state variables, e.g., the position, velocity and orientation of the robot with respect to a fixed inertial frame.

Several outstanding questions remain. For example, we need to determine what is the best representation for our robotic platform - polar coordinates, Cartesian with respect to an arbitrary reference point? Or maybe there is a better choice?

Finally, the above model assumes the knowledge of a number of constants that are characteristic of the robot's geometry, materials, and the DC motors. Without the knowledge of those constant the model could be completely off. Determination of these parameters in a measurement driven way, i.e., the "system identification" of the robot's plant, is subject of the *odometry* class.

## UNIT G-7

### Computer vision basics [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT G-8

### Camera geometry [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT G-9

# Camera calibration [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT G-10

### Image filtering [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## PART H

### Reference Material

## UNIT H-1

### Configuration management [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT H-2

### ROS installation and reference

| Assigned to: Liam

#### 2.1. Install ROS

This part installs ROS. You will run this twice, once on the laptop, once on the robot.

The first commands are copied from [this page](#).

Tell Ubuntu where to find ROS:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Tell Ubuntu that you trust the ROS people (they are nice folks):

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key  
421C365BD9FF1F717815A3895523BAEEB01FA116
```

Fetch the ROS repo:

```
$ sudo apt update
```

Now install the mega-package `ros-kinetic-desktop-full`.

```
$ sudo apt install ros-kinetic-desktop-full
```

There's more to install:

```
$ sudo apt install  
ros-kinetic-{tf-conversions,cv-bridge,image-transport,camera-info-manager,theora-image-transport,joy,image-
```

**Note:** Do not install packages by the name of `ros-X`, only those by the name of `ros-kinetic-X`. The packages `ros-X` are from another version of ROS.

**XXX: not done in aug20 image:**

Initialize ROS:

```
$ sudo rosdep init  
$ rosdep update
```

## 2.2. `rqt_console`

**TODO:** to write

## 2.3. `roslaunch`

**TODO:** to write

## 2.4. rviz

TODO: to write

## 2.5. rostopic

TODO: to write

1) rostopic hz

---

TODO: to write

2) rostopic echo

---

TODO: to write

## 2.6. catkin\_make

TODO: to write

## 2.7. rosrun

TODO: to write

## 2.8. rostest

TODO: to write

## 2.9. rospack

TODO: to write

## 2.10. rosparam

TODO: to write

## 2.11. rosdep

**TODO:** to write

## 2.12. `rosrdf`

**TODO:** to write

## 2.13. `rosbag`

A bag is a file format in ROS for storing ROS message data. Bags, so named because of their .bag extension, have an important role in ROS. Bags are typically created by a tool like [rosbag](#), which subscribe to one or more ROS topics, and store the serialized message data in a file as it is received. These bag files can also be played back in ROS to the same topics they were recorded from, or even remapped to new topics.

### 1) `rosbag record`

---

The command [rosbag record](#) records a bag file with the contents of specified topics.

### 2) `rosbag info`

---

The command [rosbag info](#) summarizes the contents of a bag file.

### 3) `rosbag play`

---

The command [rosbag play](#) plays back the contents of one or more bag files.

### 4) `rosbag check`

---

The command [rosbag check](#) determines whether a bag is playable in the current system, or if it can be migrated.

### 5) `rosbag fix`

---

The command [rosbag fix](#) repairs the messages in a bag file so that it can be played in the current system.

### 6) `rosbag filter`

---

The command [rosbag filter](#) converts a bag file using Python expressions.

### 7) `rosbag compress`

---

The command [rosbag compress](#) compresses one or more bag files.

## 8) rosbag decompress

---

The command [rosbag decompress](#) decompresses one or more bag files.

## 9) rosbag reindex

---

The command [rosbag reindex](#) re-indexes one or more broken bag files.

## 2.14. roscore

TODO: to write

## 2.15. Troubleshooting ROS

| **Symptom:** `computer` is not in your SSH known\_hosts file

See [this thread](#). Remove the `known_hosts` file and make sure you have followed the instructions in [Local configuration \(master\)](#).

## 2.16. Other materials about ROS.

\* [A gentle introduction to ROS](#)

### UNIT H-3

## 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.

### 3.1. 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, because the little  
RaspberryPi 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
```

### 3.2. 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 + S` 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
```

### 3.3. 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).

### 3.4. 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)
```

### 3.5. (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
```

## TODO

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

## PART I

# Operation manual - Duckiebot



In this section you will find information to obtain the necessary equipment for Duckietowns and different Duckiebot configurations.

## UNIT I-1

### Duckiebot configurations



#### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** nothing

**Results:** Knowledge of Duckiebot configuration naming conventions, their components and functionalities.

**Next:** After reviewing the configurations, you can proceed to purchasing the components, reading a description of the components, or assembling your chosen configuration.

We define different Duckiebot configurations depending on their time of use and hardware components. This is a good starting point if you are wondering what parts you should obtain to get started.

#### 1.1. Duckiebot Configurations: Fall 2017

The configurations are defined with a root: `DB17-`, indicating the “bare bones” Duckiebot used in the Fall 2017 synchronized course, and an appendix `y` which can be the union (in any order) of any or all of the elements of the optional hardware set  $\$\\asset{O} = \{ \$w, \$j, \$d, \$p, \$l, \$c \$\}$ .

The elements of  $\$\\asset{O}$  are labels identifying optional hardware that aids in the development phase and enables the Duckiebot to talk to other Duckiebots. The labels stand for:

- `w`: 5 GHz wireless adapter to facilitate streaming of images;
- `j`: wireless joypad that facilitates manual remote control;
- `d`: USB drive for additional storage space;
- `c`: a different castor wheel to *replace* the preexisting omni-directional wheel;
- `l`: includes LEDs, LED hat, bumpers and the necessary mechanical bits to set the bumpers in place. Note that the installation of the bumpers induces the *replacement* of a few `DB17` components;

**Note:** During the Fall 2017 course, three Duckietown Engineering Co. branches (Zurich, Montreal, Chicago) are using these configuration naming conventions. Moreover, all institutions release hardware to their Engineers in training in two phases. We summarize the configuration releases [below](#).

## 1.2. Configuration functionality

### 1) DB17

This is the minimal configuration for a Duckiebot. It is the configuration of choice for tight budgets or when operation of a single Duckiebot is more of interest than fleet behaviors.

- **Functions:** A `DB17` Duckiebot can navigate autonomously in a Duckietown, but cannot communicate with other Duckiebots.
- **Components:** A “bare-bones” `DB17` configuration includes:

TABLE 1.1. COMPONENTS OF THE DB17 CONFIGURATION

<u>Chassis</u>	USD 20
<u>Camera with 160-FOV Fisheye Lens</u>	USD 22
<u>Camera Mount</u>	USD 8.50
<u>300mm Camera Cable</u>	USD 2
<u>Raspberry Pi 3 - Model B</u>	USD 35
<u>Heat Sinks</u>	USD 5
<u>Power supply for Raspberry Pi</u>	USD 7.50
<u>16 GB Class 10 MicroSD Card</u>	USD 10
<u>Mirco SD card reader</u>	USD 6
<u>DC Motor HAT</u>	USD 22.50
<u>Spliced USB-A power cable</u>	USD 0
<u>2 Stacking Headers</u>	USD 2.50/piece
<u>Battery</u>	USD 20
<u>16 Nylon Standoffs (M2.5 12mm F 6mm M)</u>	USD 0.05/piece
<u>4 Nylon Hex Nuts (M2.5)</u>	USD 0.02/piece
<u>4 Nylon Screws (M2.5x10)</u>	USD 0.05/piece
<u>2 Zip Ties (300x5mm)</u>	USD 9
Total cost for <code>DB17</code> configuration	USD 173.6

- **Description of components:** [Unit I-2 - Acquiring the parts \(DB17-jwd\)](#)
- **Assembly instructions:** [Without videos, with videos](#)

### 2) DB17-w

This configuration is the same as `DB17` with the *addition* of a 5 Ghz wireless adapter.

- **Functions:** This configuration has the same functionality of `DB17`. In addition, it equips the Duckiebot with a secondary, faster, Wi-Fi connection, ideal for image streaming.
- **Components:**

TABLE 1.2. COMPONENTS OF THE DB17-W CONFIGURATION

<u>DB17</u>	USD 173.6
<u>Wireless Adapter (5 GHz)</u>	USD 20
Total cost for DB17-w configuration	USD 193.6
• Description of components: <a href="#">Unit I-2 - Acquiring the parts (DB17-jwd)</a>	
• Assembly instructions: <a href="#">Without videos, with videos</a>	

### 3) DB17-j

---

This configuration is the same as DB17 with the *addition* of a 2.4 GHz wireless joypad.

- **Functions:** This configuration has the same functionality of DB17. In addition, it equips the Duckiebot with manual remote control capabilities. It is particularly useful for getting the Duckiebot out of tight spots or letting younger ones have a drive, in addition to providing handy shortcuts to different functions in development phase.
- **Components:**

TABLE 1.3. COMPONENTS OF THE DB17-J CONFIGURATION

<u>DB17</u>	USD 173.6
<u>Joypad</u>	USD 10.50
Total cost for DB17-j configuration	USD 184.1
• Description of components: <a href="#">Unit I-2 - Acquiring the parts (DB17-jwd)</a>	
• Assembly instructions: <a href="#">Without videos, with videos</a>	

### 4) DB17-d

---

This configuration is the same as DB17 with the *addition* of a USB flash hard drive.

- **Functions:** This configuration has the same functionality of DB17. In addition, it equips the Duckiebot with an external hard drive that is convenient for storing videos (logs) as it provides both extra capacity and faster data transfer rates than the microSD card in the Raspberry Pi. Moreover, it is easy to unplug it from the Duckiebot at the end of the day and bring it over to a computer for downloading and analyzing stored data.
- **Components:**

TABLE 1.4. COMPONENTS OF THE DB17-D CONFIGURATION

<u>DB17</u>	USD 173.6
<u>Tiny 32GB USB Flash Drive</u>	USD 12.50
Total cost for DB17-d configuration	USD 186.1
• Description of components: <a href="#">Unit I-2 - Acquiring the parts (DB17-jwd)</a>	
• Assembly instructions: <a href="#">Without videos, with videos</a>	

### 5) DB17-c

---

In this configuration, the DB17 omni-directional wheel is *replaced* with a caster wheel.

- **Functions:** The caster wheel upgrade provides a smoother ride.
- **Components:**

TABLE 1.5. COMPONENTS OF THE DB17-C CONFIGURATION

<u>DB17</u>	USD 173.6
<u>Caster (DB17-c)</u>	USD 6.55/4 pieces
<u>4 Standoffs (M3 12mm F-F)</u>	USD 0.63/piece
<u>8 Screws (M3x8mm)</u>	USD 4.58/100 pieces
<u>8 Split washer lock</u>	USD 1.59/100 pieces
Total cost for DB17-c configuration	USD 178.25

**TODO:** update links of mechanical bits from M3.5 to M3.

**Note:** The omni-directional caster wheel is included in the chassis package, so replacing it does not reduce the DB17 cost.

- Description of components: [Unit J-1 - Acquiring the parts \(DB17-1c\)](#)
- Assembly instructions: [Unit J-3 - Assembling the Duckiebot \(DB17-1c\) \[draft\]](#)

## 6) DB17-1

---

In this configuration the Duckiebot is equipped with the necessary hardware for controlling and placing 5 RGB LEDs on the Duckiebot. Differently from previous configurations that add or replace a single component, DB17-1 introduces several hardware components that are all necessary for a proper use of the LEDs.

It may be convenient at times to refer to hybrid configurations including any of the DB17-jwcd in conjunction with a *subset* of the DB17-1 components. In order to disambiguate, let the partial upgrades be defined as:

- DB17-11: adds a PWM hat to DB17, in addition to a short USB angled power cable and a M-M power wire;
- DB17-12: adds a bumpers set to DB17, in addition to the mechanical bits to assemble it;
- DB17-13: adds a LED hat and 5 RGB LEDs to DB17-1112, in addition to the F-F wires to connect the LEDs to the LED board.

**Note:** introducing the PWM hat in DB17-11 induces a *replacement* of the [spliced cable](#) powering solution for the DC motor hat. Details can be found in [Unit J-3 - Assembling the Duckiebot \(DB17-1c\) \[draft\]](#).

- **Functions:** DB17-1 is the necessary configuration to enable communication between Duckiebots, hence fleet behaviors (e.g., negotiating the crossing of an intersection). Subset configurations are sometimes used in a standalone way for: (DB17-11) avoid using a sliced power cable to power the DC motor hat in DB17, and (DB17-12) for purely aesthetic reasons.
- **Components:**

TABLE 1.6. COMPONENTS OF THE DB17-L CONFIGURATION

<u>DB17</u>	USD 173.6
<u>PWM/Servo HAT</u> (DB17-11)	USD 17.50
<u>Power Cable</u> (DB17-11)	USD 7.80
	<u>Male-Male Jumper Wire (150mm)</u>
	(DB17-11)
	<u>Bumper set</u> (DB17-12)
	<u>8 M3x10 pan head screws</u> (DB17-12)
	<u>8 M3 nuts</u> (DB17-12)
	<u>Bumpers</u> (DB17-12)
<u>USD 1.95</u>	USD 10
USD 7 (custom made)	USD 28.20 for 3 pieces
USD 7 (custom made)	
USD 7 (custom made)	
USD 7 (custom made)	
<u>LEDs</u> (DB17-13)	
<u>LED HAT</u> (DB17-13)	
<u>20 Female-Female Jumper Wires (300mm)</u>	
(DB17-13)	USD 8
<u>4 4 pin female header</u> (DB17-13)	USD 0.60/piece
<u>12 pin male header</u> (DB17-13)	USD 0.48/piece
<u>2 16 pin male header</u> (DB17-13)	USD 0.61/piece
<u>3 pin male header</u> (DB17-13)	USD 0.10/piece
<u>2 pin female shunt jumper</u> (DB17-13)	USD 2/piece
<u>40 pin female header</u> (DB17-13)	USD 1.50
<u>5 200 Ohm resistors</u> (DB17-13)	USD 0.10/piece
<u>10 130 Ohm resistors</u> (DB17-13)	USD 0.10/piece
Total for DB17-1 configuration	USD 305

- Description of components: [Unit J-1 - Acquiring the parts \(DB17-1c\)](#)
- Assembly instructions: [Unit J-3 - Assembling the Duckiebot \(DB17-1c\) \[draft\]](#)

### 1.3. Branch configuration releases: Fall 2017

All branches release their hardware in two phases, namely **a** and **b**.

#### 1) Zurich

- 
- First release (DB17-Zurich-a): is a **DB17-wjd**.
  - Second release (DB17-Zurich-b): is a **DB17-wjdcl**.

#### 2) Montreal

- 
- First release (DB17-Montreal-a): is a hybrid **DB17-wjd** + PWM hat (or **DB17-wjd11**).
  - Second release (DB17-Montreal-b): is a **DB17-wjdl**.

**Note:** The Montreal branch is not implementing the **DB17-c** configuration.

#### 3) TTIC

- 
- First release (DB17-Chicago-a): is a **DB17-wjd**.
  - Second release (DB17-Chicago-b): is a **DB17-wjd1**.

**Note:** The Chicago branch is not implementing the **DB17-c** configuration.

## UNIT I-2

# Acquiring the parts (DB17-jwd)



The trip begins with acquiring the parts. Here, we provide a link to all bits and pieces that are needed to build a Duckiebot, along with their price tag. If you are wondering what is the difference between different Duckiebot configurations, read [this](#).

In general, keep in mind that:

- The links might expire, or the prices might vary.
- Shipping times and fees vary, and are not included in the prices shown below.
- Substitutions are OK for the mechanical components, and not OK for all the electronics, unless you are OK in writing some software.
- Buying the parts for more than one Duckiebot makes each one cheaper than buying only one.
- For some components, the links we provide contain more bits than actually needed.

### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Cost: USD 174 + Shipping Fees (minimal configuration DB17)

**Requires:** Time: 15 days (average shipping for cheapest choice of components)

**Results:** A kit of parts ready to be assembled in a DB17 or DB17-wjd configuration.

**Next:** After receiving these components, you are ready to do some [soldering \(master\)](#) before [assembling](#) your DB17 or DB17-wjd Duckiebot.

### 2.1. Bill of materials



TABLE 2.1. BILL OF MATERIALS

<u>Chassis</u>	USD 20
<u>Camera with 160-FOV Fisheye Lens</u>	USD 22
<u>Camera Mount</u>	USD 8.50
<u>300mm Camera Cable</u>	USD 2
<u>Raspberry Pi 3 - Model B</u>	USD 35
<u>Heat Sinks</u>	USD 5
<u>Power supply for Raspberry Pi</u>	USD 7.50
<u>16 GB Class 10 MicroSD Card</u>	USD 10
<u>Mirco SD card reader</u>	USD 6
<u>DC Motor HAT</u>	USD 22.50
<u>2 Stacking Headers</u>	USD 2.50/piece
<u>Battery</u>	USD 20
<u>16 Nylon Standoffs (M2.5 12mm F 6mm M)</u>	USD 0.05/piece
<u>4 Nylon Hex Nuts (M2.5)</u>	USD 0.02/piece
<u>4 Nylon Screws (M2.5x10)</u>	USD 0.05/piece
<u>2 Zip Ties (300x5mm)</u>	USD 9
<u>Wireless Adapter (5 GHz) (DB17-w)</u>	USD 20
<u>Joypad (DB17-j)</u>	USD 10.50
<u>Tiny 32GB USB Flash Drive</u> (DB17-d)	USD 12.50
Total for DB17 configuration	USD 173.6
Total for DB17-w configuration	USD 193.6
Total for DB17-j configuration	USD 184.1
Total for DB17-d configuration	USD 186.1
Total for DB17-wjd configuration	USD 216.6

## 2.2. Chassis

We selected the Magician Chassis as the basic chassis for the robot ([Figure 2.1](#)).

We chose it because it has a double-decker configuration, and so we can put the battery in the lower part.

The chassis pack includes 2 DC motors and wheels as well as the structural part, in addition to a screwdriver and several necessary mechanical bits (standoffs, screws and nuts).



Figure 2.1. The Magician Chassis

## 2.3. Raspberry Pi 3 - Model B

The Raspberry Pi is the central computer of the Duckiebot. Duckiebots use Model B ([Figure 2.2](#)) ( A 1.2GHz 64-bit quad-core ARMv8 CPU, 1GB RAM), a small but powerful computer.



Figure 2.2. The Raspberry Pi 3 Model B

### 1) Power Supply

---

We want a hard-wired power source (5VDC, 2.4A, Micro USB) to supply the Raspberry Pi ([Figure 2.3](#)) while not driving. This charger can double down as battery charger as well.



Figure 2.3. The Power Supply

**Note:** Students in the ETHZ-Fall 2017 course will receive a converter for US to CH plug.

### 2) Heat Sinks

---

The Raspberry Pi will heat up significantly during use. It is warmly recommended to add heat sinks, as in [Figure 2.4](#). Since we will be stacking HATs on top of the Raspberry Pi with 15 mm standoffs, the maximum height of the heat sinks should be well below 15 mm. The chip dimensions are 15x15mm and 10x10mm.

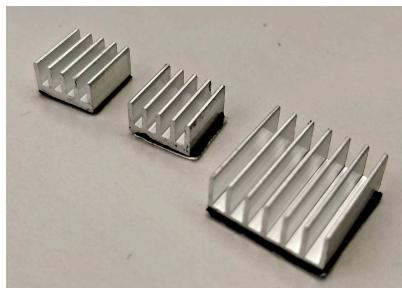


Figure 2.4. The Heat Sinks

### 3) Class 10 MicroSD Card

---

The MicroSD card ([Figure 2.5](#)) is the hard disk of the Raspberry Pi. 16 GB of capacity are sufficient for the system image.



Figure 2.5. The MicroSD card

### 4) Mirco SD card reader

---

A microSD card reader ([Figure 2.6](#)) is useful to copy the system image to a Duckiebot from a computer to the Raspberry Pi microSD card, when the computer does not have a native SD card slot.



Figure 2.6. The Mirco SD card reader

## 2.4. Camera

The Camera is the main sensor of the Duckiebot. All versions equip a 5 Mega Pixels

1080p camera with wide field of view ( $160^\circ$ ) fisheye lens ([Figure 2.7](#)).



Figure 2.7. The Camera with Fisheye Lens

### 1) Camera Mount

---

The camera mount ([Figure 2.8](#)) serves to keep the camera looking forward at the right angle to the road (looking slightly down). The front cover is not essential.



Figure 2.8. The Camera Mount

The assembled camera (without camera cable), is shown in ([Figure 2.9](#)).

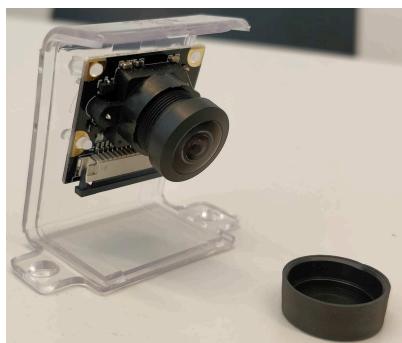


Figure 2.9. The Camera on its mount

## 2) 300mm Camera Cable

---

A longer (300 mm) camera cable ([Figure 2.10](#)) makes assembling the Duckiebot easier, allowing for more freedom in the relative positioning of camera and computational stack.

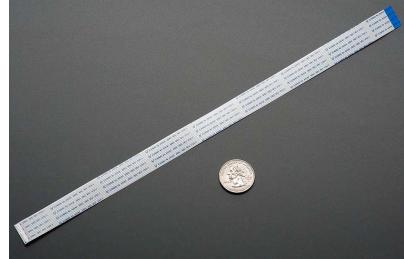


Figure 2.10. A 300 mm camera cable for the Raspberry Pi

## 2.5. DC Motor HAT

We use the DC Stepper motor HAT ([Figure 2.11](#)) to control the DC motors that drive the wheels. This item will require [soldering](#) to be functional. This HAT has dedicated PWM and H-bridge for driving the motors.

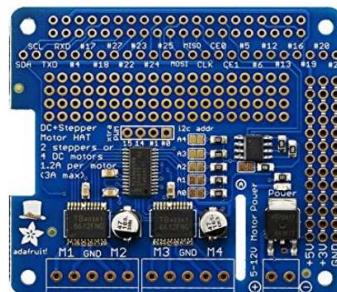


Figure 2.11. The Stepper Motor HAT

### 1) Stacking Headers

---

We use a long 20x2 GPIO stacking header ([Figure 2.12](#)) to connect the Raspberry Pi with the DC Motor HAT. This item will require [soldering \(master\)](#) to be functional.



Figure 2.12. The Stacking Headers

## 2.6. Battery

The battery ([Figure 2.13](#)) provides power to the Duckiebot.

We choose this battery because it has a good combination of size (to fit in the lower deck of the Magician Chassis), high output amperage (2.4A and 2.1A at 5V DC) over two USB outputs, a good capacity (10400 mAh) at an affordable price. The battery linked in the table above comes with two USB to microUSB cables.



Figure 2.13. The Battery

## 2.7. Standoffs, Nuts and Screws

We use non electrically conductive standoffs (M2.5 12mm F 6mm M), nuts (M2.5), and screws (M2.5x10mm) to hold the Raspberry Pi to the chassis and the HATs stacked on top of the Raspberry Pi.

The Duckiebot requires 8 standoffs, 4 nuts and 4 screws.



Figure 2.14. Standoffs, Nuts and Screws

## 2.8. Zip Tie

Two 300x5mm zip ties are needed to keep the battery at the lower deck from moving around.



Figure 2.15. The zip ties

## 2.9. Configuration DB17-w

### 1) Wireless Adapter (5 GHz)

---

The Edimax AC1200 EW-7822ULC 5 GHz wireless adapter ([Figure 2.16](#)) boosts the connectivity of the Duckiebot, especially useful in busy Duckietowns (e.g., classroom). This additional network allows easy streaming of images.



Figure 2.16. The Edimax AC1200 EW-7822ULC wifi adapter

## 2.10. Configuration DB17-j

### 1) Joypad

---

The joypad is used to manually remote control the Duckiebot. Any 2.4 GHz wireless controller (with a *tiny* USB dongle) will do.

The model linked in the table ([Figure 2.17](#)) does not include batteries.



Figure 2.17. A Wireless Joypad

Requires: 2 AA 1.5V batteries ([Figure 2.18](#)).



Figure 2.18. A Wireless Joypad

## 2.11. Configuration DB17-d

### 1) Tiny 32GB USB Flash Drive

In configuration DB17-d, the Duckiebot is equipped with an “external” hard drive ([Figure 2.19](#)). This add-on is very convenient to store logs during experiments and later port them to a workstation for analysis. It provides storage capacity and faster data transfer than the MicroSD card.



Figure 2.19. The Tiny 32GB USB Flash Drive

## UNIT I-3

# Preparing the power cable (DB17)



In configuration DB17 we will need a cable to power the DC motor HAT from the battery. The keen observer might have noticed that such a cable was not included in the [DB17 Duckiebot parts](#) chapter. Here, we create this cable by splitting open any USB-A cable, identifying and stripping the power wires, and using them to power the DC motor HAT. If you are unsure about the definitions of the different Duckiebot configurations, read [Unit I-1 - Duckiebot configurations](#).

It is important to note that these instructions are relevant only for assembling a DB17-wjdc configuration Duckiebot (or any subset of it). If you intend to build a DB17-1 configuration Duckiebot, you can skip these instructions.

### KNOWLEDGE AND ACTIVITY GRAPH

- | **Requires:** One male USB-A to anything cable.
- | **Requires:** A pair of scissors.
- | **Requires:** A multimeter (only if you are not purchasing the [suggested components](#))
- | **Requires:** Time: 5 minutes
- | **Results:** One male USB-A to wires power cable

### 3.1. Video tutorial

The following video shows how to prepare the USB power cable for the configuration DB17.



Figure 3.1

### 3.2. Step-by-step guide

- 1) Step 1: Find a cable



To begin with, find a male USB-A to anything cable.

If you have purchased the suggested components listed in [Unit I-2 - Acquiring the parts \(DB17-jwd\)](#), you can use the longer USB cable contained inside the battery package ([Figure 3.2](#)), which will be used as an example in these instructions.



Figure 3.2. The two USB cables in the suggested battery pack.

Put the shorter cable back in the box, and open the longer cable ([Figure 3.3](#))



Figure 3.3. Take the longer cable, and put the shorter on back in the box.

## 2) Step 2: Cut the cable

---

### Check before you continue

Make sure the USB cable is *unplugged* from any power source before proceeding.

Take the scissors and cut it ([Figure 3.4](#)) at the desired length from the USB-A port.

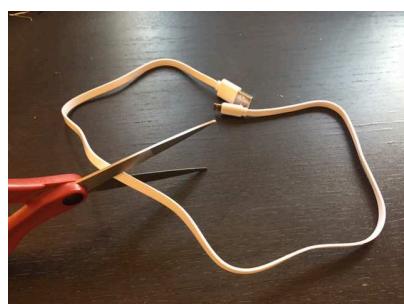


Figure 3.4. Cut the USB cable using the scissors.

The cut will look like in [Figure 3.5](#).



Figure 3.5. A cut USB cable.

### 3) Step 3: Strip the cable

---

Paying attention not to get hurt, strip the external white plastic. A way to do so without damaging the wires is shown in [Figure 3.6](#).



Figure 3.6. Stripping the external layer of the USB cable.

After removing the external plastic, you will see four wires: black, green, white and red ([Figure 3.7](#)).

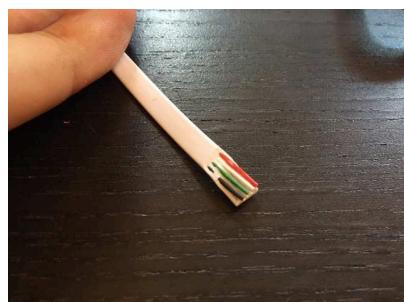


Figure 3.7. Under the hood of a USB-A cable.

Once the bottom part of the external cable is removed, you will have isolated the four wires ([Figure 3.8](#)).

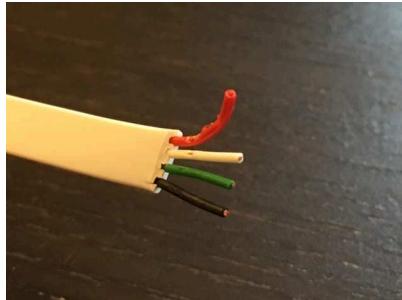


Figure 3.8. The four wires inside a USB-A cable.

#### 4) Step 4: Strip the wires

---

##### **Check before you continue**

Make sure the USB cable is *unplugged* from any power source before proceeding.

Once you have isolated the wires, strip them, and use the scissors to cut off the data wires (green and white, central positions) (Figure 3.9).



Figure 3.9. Strip the power wires and cut the data wires.

If you are not using the suggested cable, or want to verify which are the data and power wires, continue reading.

#### 5) Step 5: Find the power wires

---

If you are using the USB-A cable from the suggested battery pack, black and red are the power wires and green and white are instead for data.

If you are using a different USB cable, or are curious to verify that black and red actually are the power cables, take a multimeter and continue reading.

Plug the USB port inside a power source, e.g., the Duckiebot's battery. You can use some scotch tape to keep the cable from moving while probing the different pairs of wires with a multimeter. The voltage across the pair of power cables will be roughly twice the voltage between a power and data cable. The pair of data cables will have no voltage differential across them. If you are using the suggested Duckiebot battery as power

source, you will measure around 5V across the power cables ([Figure 3.10](#)).

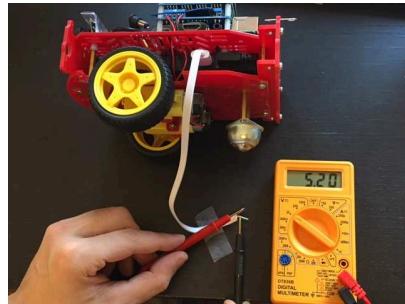


Figure 3.10. Finding which two wires are for power.

## 6) Step 6: Test correct operation

You are now ready to secure the power wires to the DC motor HAT power pins. To do so though, you need to have soldered the boards first. If you have not done so yet, read [Soldering boards \(DB17\) \(master\)](#).

If you have soldered the boards already, you may test correct functionality of the newly crafted cable. Connect the battery with the DC motor HAT by making sure you plug the black wire in the pin labeled with a minus: - and the red wire to the plus: + ([Figure 3.11](#)).

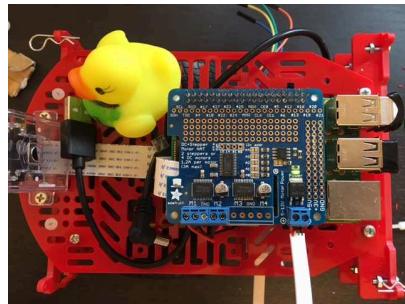


Figure 3.11. Connect the power wires to the DC motor HAT

## UNIT I-4

# Assembling the Duckiebot (DB17-jwd)

**Point of contact:** Shiying Li

Once you have received the parts and soldered the necessary components, it is time to assemble them in a Duckiebot. Here, we provide the assembly instructions for configurations [DB17-wjd](#).

## KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Duckiebot DB17-wjd parts. The acquisition process is explained in [Unit I-2 - Acquiring the parts \(DB17-jwd\)](#).

**Requires:** Having soldered the DB17-wjd parts. The soldering process is explained in [Soldering boards \(DB17\) \(master\)](#).

**Requires:** Having prepared the power cable. The power cable preparation is explained in [Unit I-3 - Preparing the power cable \(DB17\)](#). Note: Not necessary if you intend to build a DB17-1 configuration.

**Requires:** Having installed the image on the MicroSD card. The instructions on how to reproduce the Duckiebot system image are in [Reproducing the image \(master\)](#).

**Requires:** Time: about 40 minutes.

**Results:** An assembled Duckiebot in configuration DB17-wjd.

**Note:** The [FAQ](#) section at the bottom of this page may already answer some of your comments, questions or doubts.

**Note:** While assembling the Duckiebot, try to make as symmetric (along the longitudinal axis) as you can. It will help going forward.

## 4.1. Chassis

Open the Magician chassis package ([Figure 4.1](#)) and take out the following components:

- Chassis-bottom (1x), Chassis-up (1x);
- DC Motors (2x), motor holders (4x);
- Wheels (2x), steel omni-directional wheel (1x);
- All spacers and screws;
- Screwdriver.

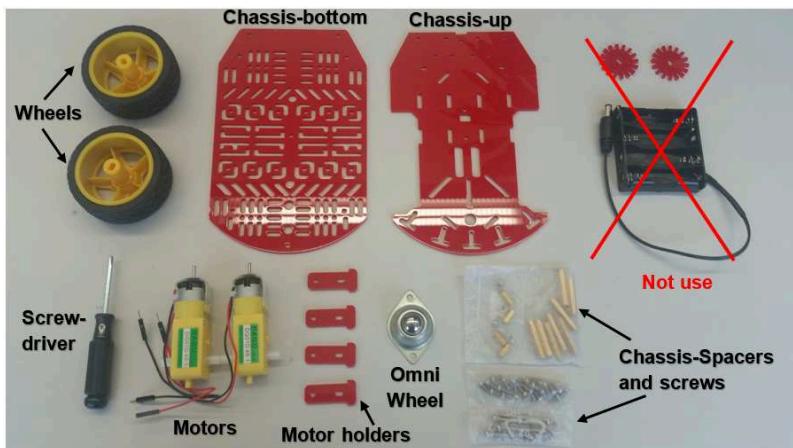


Figure 4.1. Components in Duckiebot package.

**Note:** You won't need the battery holder and speed board holder (on the right side in [Figure 4.1](#)).

### 1) Bottom

Insert the motor holders on the chassis-bottom and put the motors as shown in the figure below (with the longest screws (M3x30) and M3 nuts).

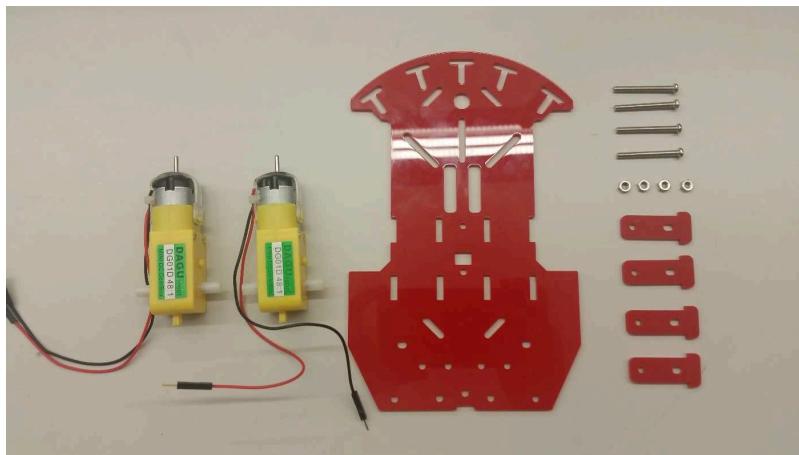


Figure 4.2. Components for mounting the motor

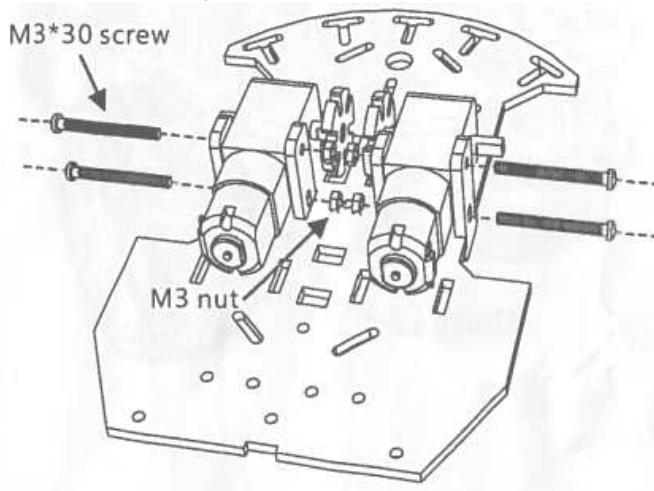


Figure 4.3. The scratch of assembling the motor



Figure 4.4. Assembled motor

**Note:** Orient the motors so that their wires are inwards, i.e., towards the center of the chassis-bottom. The black wires should be closer to the chassis-bottom to make wiring easier down the line.

**Note:** if your Magician Chassis package has unsoldered motor wires, you will have to solder them first. Check these instructions. In this case, your wires will not have the male pin headers on one end. Do not worry, you can still plug them in the stepper motor hat power terminals.

**TODO:** make instructions for soldering motor wires

## 2) Wheels

---

Plug in the wheels to the motor as follows (no screws needed):

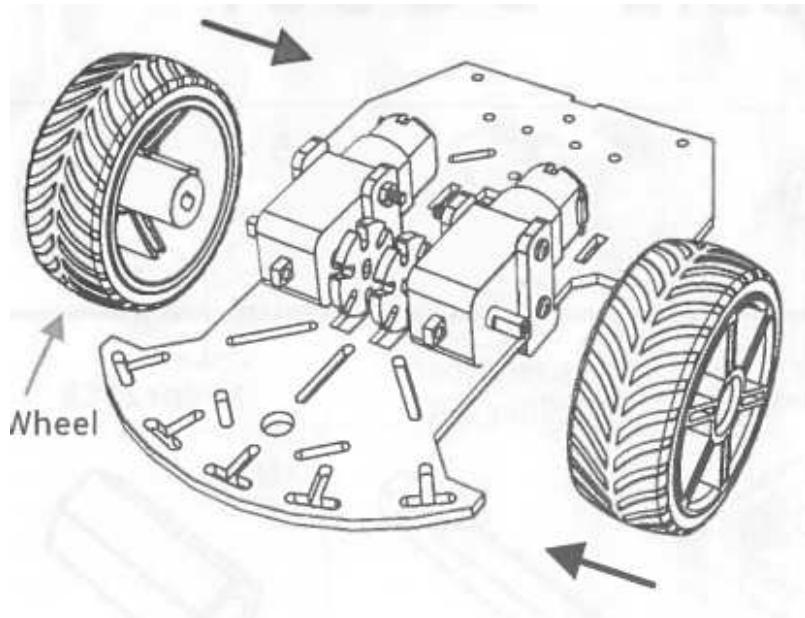


Figure 4.6. Wheel assembly schematics

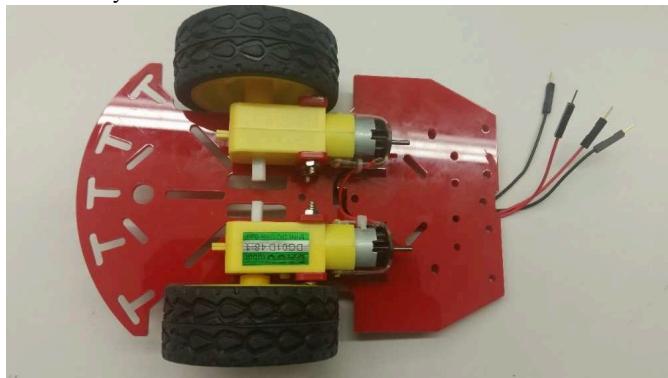


Figure 4.7. Assembled wheels

Figure 4.5. Wheel assembly instructions

### 3) Omni-directional wheel

The Duckiebot is driven by controlling the wheels attached to the DC motors. Still, it requires a “passive” omni-directional wheel (the *caster wheel*) on the back.

The Magician chassis package contains a steel omni-directional wheel, and the related standoffs and screws to secure it to the chassis-bottom part.

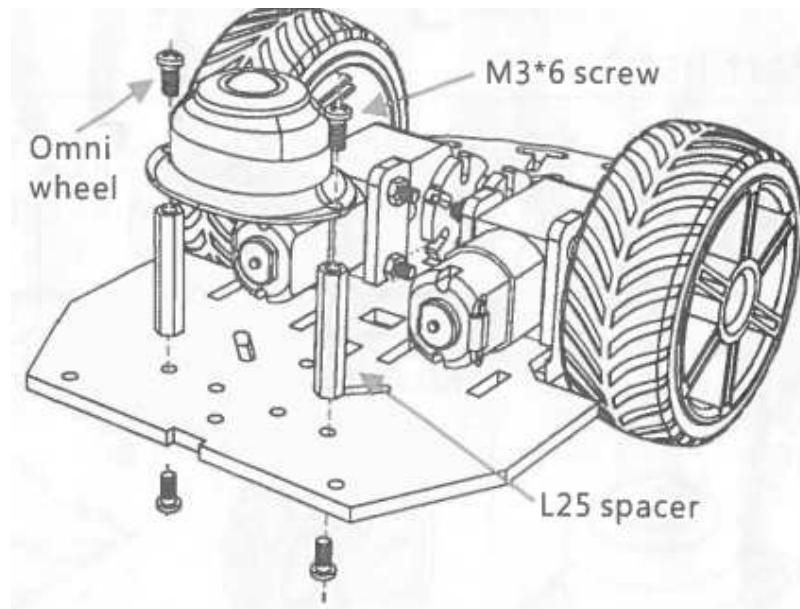


Figure 4.8. The omni-directional wheel schematics

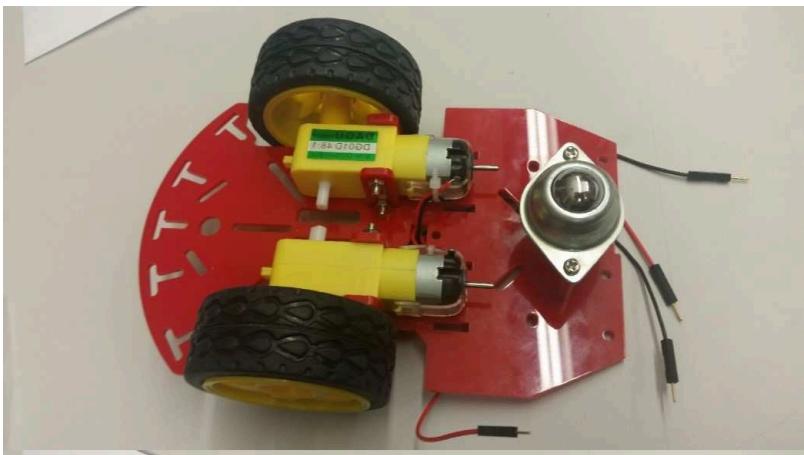


Figure 4.9. Assembled omni-directional wheel

#### 4) Caster wheel

---

As alternative to omnidirection wheel, caster wheel has less friction. If you have purchased caster wheel, read this section.

To assemble the caster wheel, the following materials are needed:

- caster wheel (1x)
- Metal standoffs (M3x12mm F-F, 6mm diameter) (4x)
- Metal screws (M3x8mm) (8x)
- Split/Spring lock washers (M3) (8x)
- Flat lock washers (M3) (8x)



Figure 4.10. Component-List for assembling the caster wheels

##### *Prepare the Screws with Washers:*

The lock washers belongs to screw-head side [Figure 4.11](#), i.e. the split lock washer and the flat lock washers stays always near the screw head. The split lock washer stays near the screw head. First split lock washer, then flat lock washer.

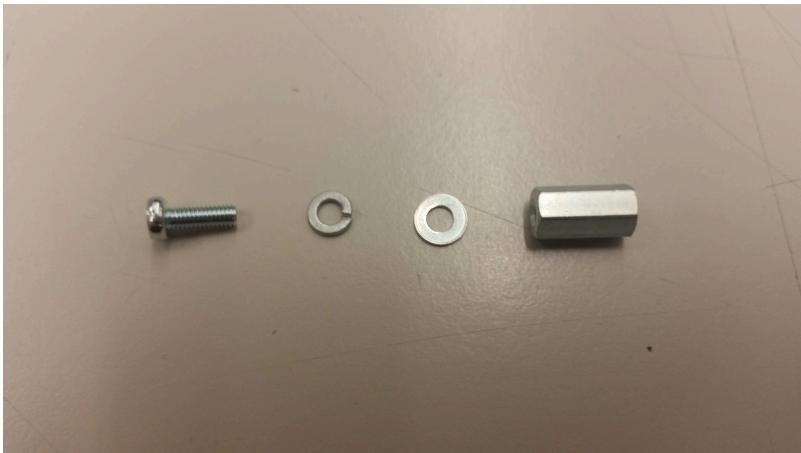


Figure 4.11. Insert the locker washers into metal screws from left to right



Figure 4.12. The metal screws with the lock washers

*Assembly the metal standoffs on the caster wheels:*

Fasten the screws with washers on the caster wheels from the bottom up and screw the metal standoffs from top to bottom. The caster before mounting looks like in [Figure 4.13.](#)



Figure 4.13. The assembled caster before mounting it under the chassis-bottom

*Assembly the caster wheels under the chassis bottom:*

Assembly the prepared caster wheels in the front side of duckiebot under the chassis bottom. Fasten the screws with washers from top to bottom.

- ✓ In order to get all the screws properly into the metal standoffs, let all the screws stay loose within the right positions before all the screws are inserted into the stand-offs [Figure 4.15](#).



Figure 4.14. Assembly the caster wheels under chassis-bottom

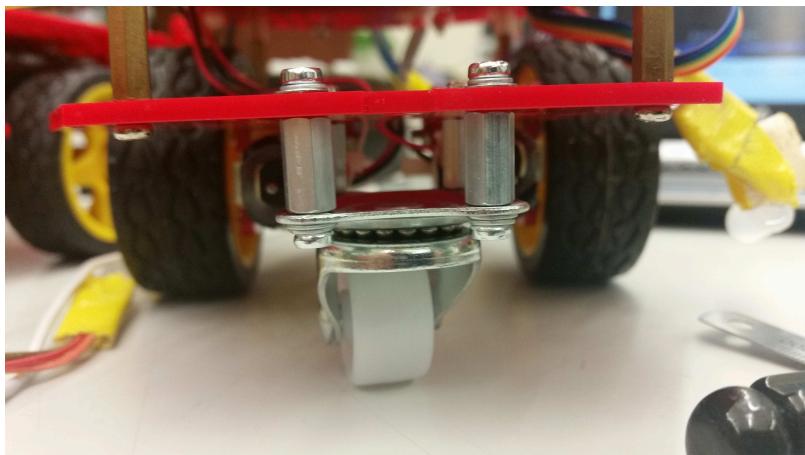


Figure 4.15. Assembled caster wheels (sideview)

## 5) Mounting the standoffs

---

Put the car upright (omni wheel pointing towards the table) and arrange wires so that they go through the center rectangle. Put 4 spacers with 4 of M3x6 screws on exact position of each corner as below [Figure 4.17](#).



Figure 4.16. Metal spacers and M3x6mm screws

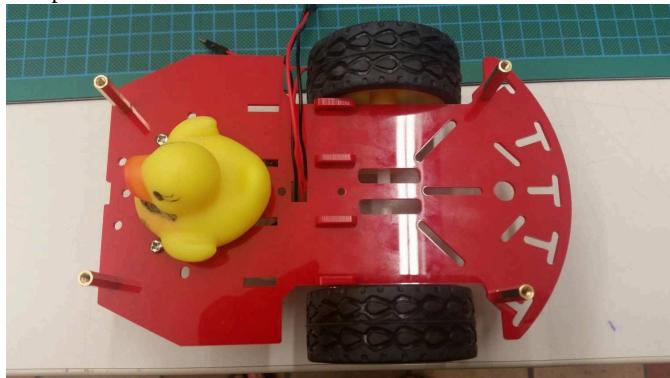


Figure 4.17. The spacers on each corner of the chassis-bottom

The bottom part of the Duckiebot's chassis is now ready. The next step is to assemble the Raspberry Pi on chassis-top part.

## 4.2. Assembling the Raspberry Pi, camera, and HATs

### 1) Raspberry Pi

---

Before attaching anything to the Raspberry Pi you should add the heat sinks to it. There are 2 small sinks and a big one. The big one best fits onto the processor (the big “Broadcom”-labeled chip in the center of the top of the Raspberry Pi). One of the small ones can be attached to the small chip that is right next to the Broadcom chip. The third heat sink is optional and can be attached to the chip on the underside of the Raspberry Pi. Note that the chip on the underside is bigger than the heat sink. Just mount the heat sink in the center and make sure all of them are attached tightly.

When this is done fasten the nylon standoffs on the Raspberry Pi, and secure them on the top of the chassis-up part by tightening the nuts on the opposite side of the chassis-up.

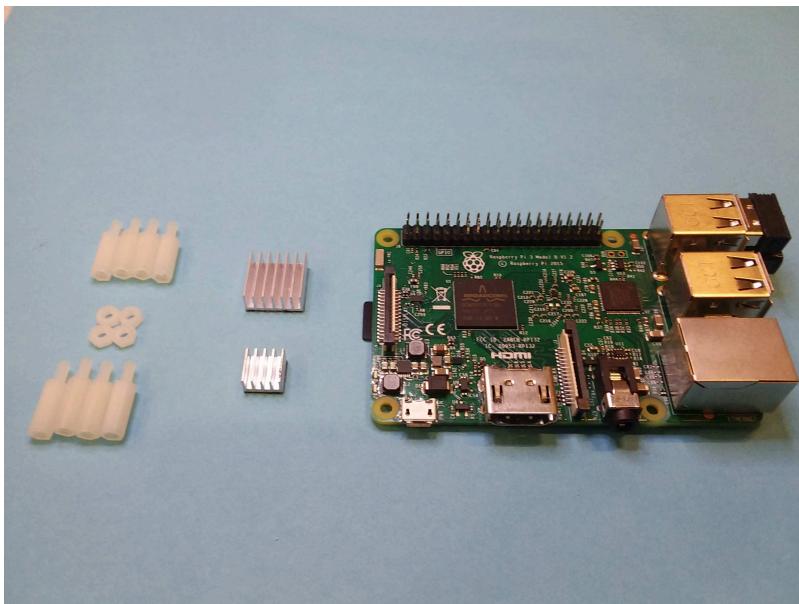


Figure 4.18. Components for Raspberry Pi3

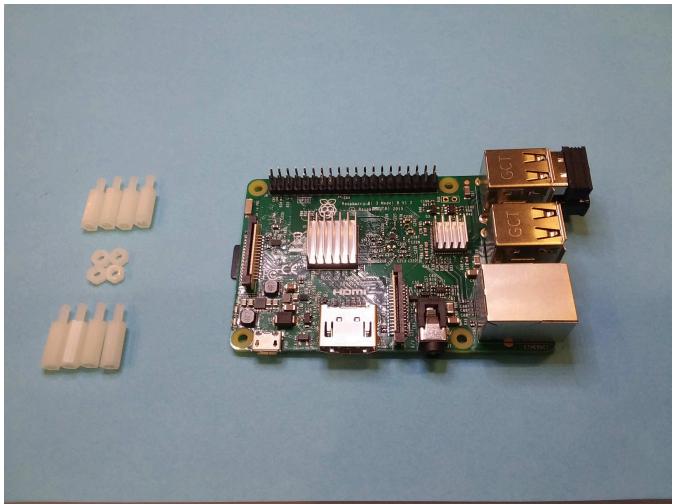


Figure 4.19. Heat sink on Raspberry Pi3

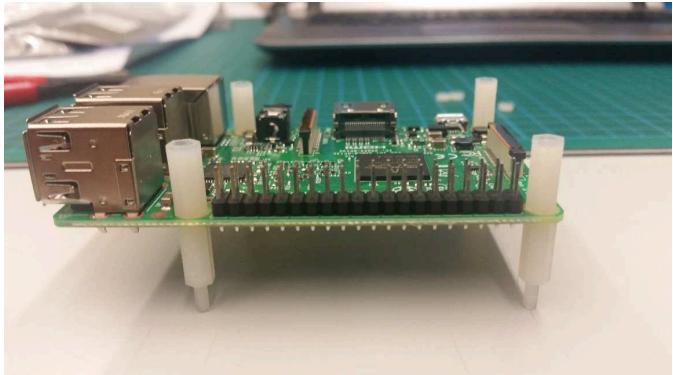


Figure 4.20. Nylon standoffs for Raspberry Pi3



Figure 4.21. Attach the nylon huts for the standoffs (bottom view)



Figure 4.22. Assembled Raspberry Pi3 (top view)

## 2) Micro SD card

---

**Requires:** Having the Duckiebot image copied in the micro SD card.

Take the micro SD card from the Duckiebox and insert its slot on the Raspberry Pi. The SD card slot is just under the display port, on the short side of the PI, on the flip side of where the header pins are.



Figure 4.23. The micro SD card and mirco SD card readers



Figure 4.24. Inserted SD card

### 3) Camera

---

**Note:** If you have camera cables of different lengths available, keep in mind that both are going to work. We suggest to use the longer one, and wrap the extra length under the Raspberry Pi stack.

*The Raspberry Pi end:*

First, identify the camera cable port on the Pi (between HDMI and power ports) and remove the orange plastic protection (it will be there if the Pi is new) from it. Then, grab the long camera cable (300 mm) and insert in the camera port. To do so, you will need to gently pull up on the black connector (it will slide up) to allow the cable to insert the port. Slide the connector back down to lock the cable in place, making sure it “clicks”.

**TODO:** insert image with long cable



Figure 4.25. Camera port on the Raspberry Pi and camera cable

**Note:** Make sure the camera cable is inserted in the right direction! The metal pins of the cable should be in contact with the metal terminals in the camera port of the PI.



Figure 4.26. Camera with long cable

#### *The camera end:*

If you have the long camera cable, the first thing to do is removing the shorter cable that comes with the camera package. Make sure to slide up the black connectors of the camera-camera cable port in order to unblock the cable.

Take the rear part of the camera mount and use it hold the camera in place. Note that the camera is just press-fitted to the camera mount, no screws/nuts are needed.

In case you have not purchased the long camera cable, do not worry! It is still very possible to get a working configuration, but you will have little wiggling space and assembly will be a little harder.

Place the camera on the mount and fasten the camera mount on the chassis-up using M3x10 flathead screws and M3 nuts from the Duckiebox.

Protip: make sure that the camera mount is: (a) geometrically centered on the chassis-up; (b) fastened as forward as it can go; (c) it is tightly fastened. We aim at having a standardized position for the camera and to minimize the wiggling during movement.



Figure 4.27. Raspberry Pi and camera with short cable

**Note:** If you only have a short camera cable, make sure that the cable is oriented in this direction (text on cable towards the CPU). Otherwise you will have to disassemble the whole thing later. On the long cable the writing is on the other side.

#### 4) Extending the intra-decks standoffs

In order to fit the battery, we will need to extend the Magician chassis standoffs with the provided nylon standoff spacers. Grab 4 of them, and secure them to one end of the long metal standoffs provided in the Magician chassis package.

Secure the extended standoff to the 4 corners of the chassis-bottom. The nylon standoffs should smoothly screw in the metal ones. If you feel resistance, don't force it or the nylon screw might break in the metal standoff. In that case, unscrew the nylon spacer and try again.



Figure 4.28. 4 nylon M3x5 extended standoffs and 4 M3x6 metal screws from Magician chassis package

## 5) Fasten the Battery with zip ties

---

Put the battery between the upper and lower decks of the chassis. It is strongly recommended to secure the battery from moving using zip ties.



Figure 4.29. Secure the battery to the chassis-top through the provided zipties. One can do the trick, two are better.

**Note:** [Figure 4.29](#) can be taken as an example of how to arrange the long camera cable as well.

## 6) Assemble chassis-bottom and chassis-up

---

Arrange the motor wires through the chassis-up, which will be connected to Stepper Motor HAT later.



Figure 4.30. The motor wires go through the center of chassis-up



Figure 4.31. Side view of metal screws and the extended standoffs

**Note:** Use the provided metal screws from chassis package for fastening the chassis up above the nylon standoffs instead of the provided M3 nylon screws.

#### 7) Place the DC Motor hat on top of the Raspberry Pi

---

Make sure the GPIO stacking header is carefully aligned with the underlying GPIO pins before applying pressure.

**Note:** In case with short camera cable, ensure that you doesn't break the cable while mounting the HAT on the Raspberry Pi. In case with long camera cable,



Figure 4.32. Assembled DC motor hat with short camera cable

**TODO:** insert pic with long camera cable

#### 8) Connect the motor's wires to the terminal

---

We are using M1 and M2 terminals on the DC motor hat. The left (in robot frame) motor is connected to M1 and the right motor is connected to M2. If you have followed Part A correctly, the wiring order will look like as following pictures:

- Left Motor: Red
- Left Motor: Black
- Right Motor: Black
- Right Motor: Red

## 9) Connect the power cables

---

You are now ready to secure the prepared power wires in [Unit I-3 - Preparing the power cable \(DB17\)](#) to the DC motor HAT power pins.

Connect the battery (not the Raspberry Pi) with the DC motor HAT by making sure you plug the black wire in the pin labeled with a minus: - and the red wire to the plus: + ([Figure 3.11](#)).

Fix all the cables on the Duckiebot so that it can run on the way without barrier.



Figure 4.33. Insert the prepared power wire to DC motor HAT power pins.

**Note:** If you have a DB17-Montreal-a or DB17-Chicago-a release, neglect this step and follow the pertinent instructions in [Unit J-3 - Assembling the Duckiebot \(DB17-1c \[draft\]\)](#) regarding the assembly of the PWM hat, its powering through the short angled USB cable, and the power transfer step using a M-M wire.

## 10) Joypad

---

With each joypad ([Figure 4.34](#)) comes a joypad dongle ([Figure 4.35](#)). Don't lose it!



Figure 4.34. All components in the Joypad package

Insert the joypad dongle into one of the USB port of the Raspberry Pi.



Figure 4.35. The dongle on the Raspberry Pi

Insert 2 AA batteries on the back side of the joypad [Figure 4.36.](#)



Figure 4.36. Joypad and 2 AA batteries

## 4.3. FAQ

*Q: If we have the bumpers, at what point should we add them?*

**Answer:** You shouldn't have the bumpers at this point. The function of bumpers is to keep the LEDs in place, i.e., they belong to DB17-1 configuration. These instructions cover the DB17-jwd configurations. You will find the bumper assembly instructions in [Unit J-3 - Assembling the Duckiebot \(DB17-1c\) \[draft\]](#).

*Q: Yeah but I still have the bumpers and am reading this page. So?*

**Answer:** The bumpers can be added after the Duckiebot assembly is complete.

*Q: I found it hard to mount the camera (the holes weren't lining up).*

**Answer:** Sometimes in life you have to push a little to make things happen. (But don't push too much or things will break!)

*Q: The long camera cable is a bit annoying - I folded it and shoved it in between two hats.*

**Answer:** The shorter cable is even more annoying. We suggest wrapping the long camera cable between the chassis and the Raspberry Pi. With some strategic planning, you can use the zip ties that keep the battery in place to hold the camera cable in place as well ([see figure below-to add](#))

**TODO:** add pretty cable handling pic

*Q: I found that the screwdriver that comes with the chassis kit is too fat to screw in the wires on the hat.*

**Answer:** It is possible you got one of the fatter screwdrivers. You will need to figure it out yourself (or ask a TA for help).

*Q: I need something to cut the end of the zip tie with.*

**Answer:** Scissors typically work out for these kind of jobs (and no, they're not provided in a Fall 2017 Duckiebox).

## UNIT I-5

### Assembling the Duckiebot (DB17-wjd TTIC)



**Point of contact:** Andrea F. Daniele

Once you have received the parts and soldered the necessary components, it is time to assemble them in a Duckiebot. Here, we provide the assembly instructions for the configuration DB17-wjd (TTIC only).

#### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Duckiebot DB17-wjd parts. The acquisition process is explained in [Unit I-2 - Acquiring the parts \(DB17-jwd\)](#).

**Requires:** Having soldered the DB17-wjd parts. The soldering process is explained in [Soldering boards \(DB17\) \(master\)](#).

**Requires:** Having prepared the power cable. The power cable preparation is explained in [Unit I-3 - Preparing the power cable \(DB17\)](#). Note: Not necessary if you intend to build a DB17-1 configuration.

**Requires:** Time: about 30 minutes.

**Results:** An assembled Duckiebot in configuration DB17-wjd.

**Note:** The [FAQ](#) section at the bottom of this page may already answer some of your comments, questions or doubts.

This section is comprised of 14 parts. Each part builds upon some of the previous parts, so make sure to follow them in the following order.

- [Part I: Motors](#)
- [Part II: Wheels](#)
- [Part III: Omni-directional wheel](#)
- [Part IV: Chassis standoffs](#)
- [Part V: Camera kit](#)
- [Part VI: Heat sinks](#)
- [Part VII: Raspberry Pi 3](#)
- [Part VIII: Top plate](#)
- [Part IX: USB Power cable](#)
- [Part X: DC Stepper Motor HAT](#)
- [Part XI: Battery](#)
- [Part XII: Upgrade to DB17-w](#)
- [Part XIII: Upgrade to DB17-j](#)
- [Part XIV: Upgrade to DB17-d](#)

## 5.1. Motors



Open the Magician Chassis package ([Figure 4.1](#)) and take out the following components:

- Chassis-bottom (1x)
- DC Motors (2x)
- Motor holders (4x)
- M3x30 screw (4x)
- M3 nuts (4x)

[Figure 5.1](#) shows the components needed to complete this part of the tutorial.



Figure 5.1. Components needed to mount the motors.

## 1) Video tutorial

---

The following video shows how to attach the motors to the bottom plate of the chassis.



Figure 5.2

## 2) Step-by-step guide

---

### *Step 1:*

Pass the motor holders through the openings in the bottom plate of the chassis as shown in [Figure 5.3](#).



Figure 5.3. The sketch of how to mount the motor holders.

*Step 2:*

Put the motors between the holders as shown in [Figure 5.4](#).



Figure 5.4. The sketch of how to mount the motors.

**Note:** Orient the motors so that their wires are inwards (i.e., towards the center of the plate).

*Step 3:*

Use 4 M3x30 screws and 4 M3 nuts to secure the motors to the motor holders. Tighten the screws to secure the holders to the bottom plate of the chassis as shown in [Figure 5.5](#).



Figure 5.5. The sketch of how to secure the motors to the bottom plate.

### 3) Check the outcome

---

[Figure 5.6](#) shows how the motors should be attached to the bottom plate of the chassis.



Figure 5.6. The motors are attached to the bottom plate of the chassis.

## 5.2. Wheels

From the Magician Chassis package take the following components:

- Wheels (2x)

[Figure 5.7](#) shows the components needed to complete this part of the tutorial.



Figure 5.7. The wheels.

### 1) Video tutorial

---

The following video shows how to attach the wheels to the motors.



Figure 5.8

### 2) Check the outcome

---

[Figure 5.9](#) shows how the wheels should be attached to the motors.



Figure 5.9. The wheels are attached to the motors.

### 5.3. Omni-directional wheel

The Duckiebot is driven by controlling the wheels attached to the DC motors. Still, it requires a *passive* support on the back. In this configuration an omni-directional wheel is attached to the bottom plate of the chassis to provide such support.

From the Magician Chassis package take the following components:

- Steel omni-directional wheel (1x)
- Long metal spacers (2x)
- M3x6 screws (4x)

[Figure 5.10](#) shows the components needed to complete this part of the tutorial.



Figure 5.10. The omni-directional wheel with \*2\* long spacers and \*4\* M3x6 screws.

### 1) Video tutorial

---

The following video shows how to attach the omni-directional wheel to the bottom plate of the chassis.



Figure 5.11

### 2) Step-by-step guide

---

#### Step 1:

Secure the long spacers to the plate using 2 M3x6 screws and the omni-directional wheel to the spacers using also 2 M3x6 screws as shown in [Figure 5.12](#).



Figure 5.12. The sketch of how to mount the omni-directional wheel.

### 3) Check the outcome

---

[Figure 5.13](#) shows how the omni-directional wheel should be attached to the plate.



Figure 5.13. The omni-directional wheel is attached to the plate.

## 5.4. Chassis standoffs

From the Magician Chassis package take the following components:

- Long metal spacers/standoffs (4x)
- M3x6 screws (4x)

From the Duckiebot kit take the following components:

- M3x5 nylon spacers/standoffs (4x)

[Figure 5.14](#) shows the components needed to complete this part of the tutorial.



Figure 5.14. The standoffs to mount on the bottom plate.

### 1) Video tutorial

The following video shows how to attach the standoffs to the bottom plate of the chassis.



Figure 5.15

## 2) Step-by-step guide

---

### Step 1:

Secure the long metal spacers to the bottom plate using 4 M3x6 screws as shown in [Figure 5.16](#).



Figure 5.16. The sketch of how to mount the standoffs on the plate.

### Step 2:

Attach the 4 nylon standoffs on top of the metal ones.

### 3) Check the outcome

[Figure 5.17](#) shows how the standoffs should be attached to the plate.



Figure 5.17. The standoffs attached to the plate.

## 5.5. Camera kit

From the Magician Chassis package take the following components:

- M3x10 flathead screws (2x)
- M3 nuts (2x)

From the Duckiebot kit take the following components:

- Camera Module (1x)
- (Optional) 300mm Camera cable (1x)
- Camera mount (1x)

**Note:** If you have camera cables of different lengths available, keep in mind that both are going to work. We suggest to use the longer one, and wrap the extra length under the Raspberry Pi stack.

[Figure 5.18](#) shows the components needed to complete this part of the tutorial.



Figure 5.18. The parts needed to fix the camera on the top plate.

### 1) Video tutorial

---

The following video shows how to secure the camera to the top plate of the chassis.



Figure 5.19

### 2) Step-by-step guide

---

#### *Step 1 (Optional):*

If you do not have the 300mm Camera cable you can jump to *Step 3*.

If you do have the long camera cable, the first thing to do is removing the shorter cable that comes attached to the camera module. Make sure to slide up the black connectors of the camera port on the camera module in order to unblock the cable.

#### *Step 2:*

Connect the camera cable to the camera module as shown in [Figure 5.20](#).



Figure 5.20. How to connect the camera cable to the camera module.

**Step 3:**

Attach the camera module to the camera mount as shown in [Figure 5.21](#).



Figure 5.21. How to attach the camera to the camera mount.

**Note:** The camera is just press-fitted to the camera mount, no screws/nuts are needed.

**Step 4:**

Secure the camera mount to the top plate by using the 2 M3x10 flathead screws and the nuts as shown in [Figure 5.22](#).



Figure 5.22. How to attach the camera mount to the top plate.

### 3) Check the outcome

[Figure 5.23](#) shows how the camera should be attached to the plate.



Figure 5.23. The camera attached to the plate.

## 5.6. Heat sinks

From the Duckiebot kit take the following components:

- Raspberry Pi 3 (1x)
- Heat sinks (2x)
- Camera mount (1x)

[Figure 5.24](#) shows the components needed to complete this part of the tutorial.



Figure 5.24. The heat sinks and the Raspberry Pi 3.

## 1) Video tutorial

---

The following video shows how to install the heat sinks on the Raspberry Pi 3.



Figure 5.25

## 2) Step-by-step guide

---

*Step 1:*

Remove the protection layer from the heat sinks.

*Step 2:*

Install the big heat sink on the big “Broadcom”-labeled integrated circuit (IC).

*Step 3:*

Install the small heat sink on the small “SMSC”-labeled integrated circuit (IC).

## 3) Check the outcome

---

[Figure 5.26](#) shows how the heat sinks should be installed on the Raspberry Pi 3.

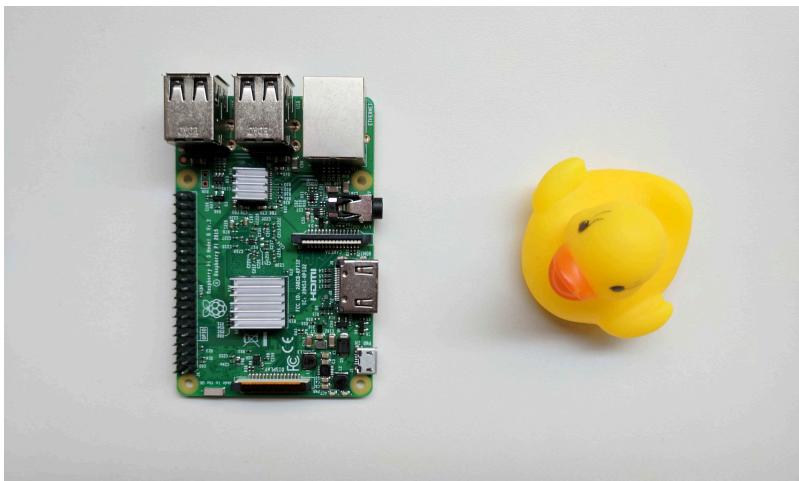


Figure 5.26. The heat sinks installed on the Raspberry Pi 3.

## 5.7. Raspberry Pi 3



From the Magician Chassis package take the following components:

- Top plate (with camera attached) (1x)

From the Duckiebot kit take the following components:

- Raspberry Pi 3 (with heat sinks) (1x)
- M2.5x12 nylon spacers/standoffs (8x)
- M2.5 nylon hex nuts (4x)

[Figure 5.27](#) shows the components needed to complete this part of the tutorial.



Figure 5.27. The parts needed to mount the Raspberry Pi 3 on the top plate.

### 1) Video tutorial

---



The following video shows how to mount the Raspberry Pi 3 on the top plate of the chassis.



Figure 5.28

## 2) Step-by-step guide

---

### Step 1:

Mount 8 M2.5x12 nylon standoffs on the Raspberry Pi 3 as shown in [Figure 5.29](#).



Figure 5.29. How to mount the nylon standoffs on the Raspberry Pi 3.

### Step 2:

Use the M2.5 nylon hex nuts to secure the Raspberry Pi 3 to the top plate as shown in [Figure 5.30](#).



Figure 5.30. How to mount the Raspberry Pi 3 on the top plate.

## 3) Check the outcome

---

[Figure 5.31](#) shows how the Raspberry Pi 3 should be mounted on the top plate of the chassis.



Figure 5.31. The Raspberry Pi 3 mounted on the top plate.

## 5.8. Top plate

From the Magician Chassis package take the following components:

- Bottom plate (with motors, wheels and standoffs attached) (1x)
- Top plate (with camera and Raspberry Pi 3 attached) (1x)
- M3x6 screws (4x)

[Figure 5.32](#) shows the components needed to complete this part of the tutorial.



Figure 5.32. The parts needed to secure the top plate to the bottom plate.

### 1) Video tutorial

---

The following video shows how to secure the top plate on top of the bottom plate.



Figure 5.33

## 2) Step-by-step guide

### Step 1:

Pass the motor wires through the openings in the top plate.

### Step 2:

Use 4 M3x6 screws to secure the top plate to the nylon standoffs (mounted on the bottom plate in [Section 5.4 - Chassis standoffs](#)) as shown in [Figure 5.34](#).



Figure 5.34. How to secure the top plate to the bottom plate.

### 3) Check the outcome

---

[Figure 5.35](#) shows how the top plate should be mounted on the bottom plate.



Figure 5.35. The chassis completed.

## 5.9. USB Power cable

The power cable preparation is explained in [Unit I-3 - Preparing the power cable \(DB17\)](#).

## 5.10. DC Stepper Motor HAT

From the Duckiebot kit take the following components:

- USB power cable (prepared in [Unit I-3 - Preparing the power cable \(DB17\)](#)) (1x)
- DC Stepper Motor HAT (1x)
- M2.5x10 Nylon screws (or M2.5x12 nylon standoffs) (4x)

[Figure 5.36](#) shows the components needed to complete this part of the tutorial.



Figure 5.36. The parts needed to add the DC Stepper Motor HAT to the Duckiebot.

## 1) Video tutorial

---

The following video shows how to connect the DC Stepper Motor HAT to the Raspberry Pi 3.



Figure 5.37

## 2) Step-by-step guide

---

### Step 1:

Connect the wires of the USB power cable to the terminal block on the DC Stepper Motor HAT labeled as “5-12V Motor Power” as shown in [Figure 5.38](#). The black wire goes to the negative terminal block (labeled with a minus: -) and the red wire goes to the positive terminal block (labeled with a plus: +).



Figure 5.38. How to connect the USB power cable to the DC Stepper Motor HAT.

**Step 2:**

Pass the free end of the camera cable through the opening in the DC Stepper Motor HAT as shown in [Figure 5.39](#).



Figure 5.39. How to pass the camera cable through the opening in the DC Stepper Motor HAT.

**Step 3:**

Connect the free end of the camera cable to the CAMERA port on the Raspberry Pi 3 as shown in [Figure 5.40](#).



Figure 5.40. How to connect the camera cable to the CAMERA port on the Raspberry Pi 3.

To do so, you will need to gently pull up on the black connector (it will slide up) to allow the cable to insert the port. Slide the connector back down to lock the cable in place, making sure it “clicks”.

**Note:** Make sure the camera cable is inserted in the right direction! The metal pins of the cable must be in contact with the metal terminals in the camera port of the PI. Please be aware that different camera cables have the text on different sides and with different orientation, **do not** use it as a landmark.

#### Step 4:

Attach the DC Stepper Motor HAT to the GPIO header on the Raspberry Pi 3. Make sure that the GPIO stacking header of the Motor HAT is carefully aligned with the underlying GPIO pins before applying pressure.

**Note:** In case you are using a short camera cable, ensure that the camera cable does not stand between the GPIO pins and the the GPIO header socket before applying pressure.

#### Step 5:

Secure the DC Stepper Motor HAT using 4 M2.5x10 nylon screws.

**Note:** If you are planning on upgrading your Duckiebot to the configuration DB17-1, you can use 4 M2.5x12 nylon standoffs instead.

#### Step 6:

Connect the motor wires to the terminal block on the DC Stepper Motor HAT as shown in [Figure 5.41](#).



Figure 5.41. How to connect the motor wires to the terminal block on the DC Stepper Motor HAT.

While looking at the Duckiebot from the back, identify the wires for left and right motor. Connect the left motor wires to the terminals labeled as M1 and the right motor wires to the terminals labeled as M2. This will ensure that the pre-existing software that we will later install on the Duckiebot will send the commands to the correct motors.

### 3) Check the outcome

---

[Figure 5.42](#) shows how the DC Stepper Motor HAT should be connected to the Raspberry Pi 3.



Figure 5.42. The DC Stepper Motor HAT connected to the Raspberry Pi 3.

## 5.11. Battery

From the Duckiebot kit take the following components:

- Battery (1x)
- Zip tie (1x)
- Short micro USB cable (1x)

[Figure 5.43](#) shows the components needed to complete this part of the tutorial.



Figure 5.43. The parts needed to add the battery to the Duckiebot.

### 1) Video tutorial

The following video shows how to add the battery to the Duckiebot and turn it on.



Figure 5.44

### 2) Step-by-step guide

#### *Step 1:*

Pass the zip tie through the opening in the top plate.

#### *Step 2:*

Slide the battery between the two plates. Make sure it is above the zip tie.

#### *Step 3:*

Push the free end of the zip tie through the opening in the top plate.

#### *Step 4:*

Tighten the zip tie to secure the battery.

*Step 5:*

Connect the short micro USB cable to the Raspberry Pi 3.

*Step 6:*

Connect the short micro USB cable to the battery.

*Step 7:*

Connect the USB power cable to the battery.

*Step 8:*

Make sure that the LEDs on the Raspberry Pi 3 and the DC Stepper Motor HAT are on.

---

3) Check the outcome

[Figure 5.45](#) shows how the battery should be installed on the Duckiebot.



---

Figure 5.45. The configuration 'DB17' completed.

## 5.12. Upgrade to DB17-w

This upgrade equips the Duckiebot with a secondary, faster, Wi-Fi connection, ideal for image streaming. The new configuration is called [DB17-w](#).

[Figure 5.46](#) shows the components needed to complete this upgrade.



Figure 5.46. The parts needed to upgrade the Duckiebot to the configuration DB17-w.

### 1) Instructions

- Insert the USB WiFi dongle into one of the USB ports of the Raspberry Pi.



Figure 5.47. Upgrade to DB17-w completed.

## 5.13. Upgrade to DB17-j

This upgrade equips the Duckiebot with manual remote control capabilities. It is par-

ticularly useful for getting the Duckiebot out of tight spots or letting younger ones have a drive, in addition to providing handy shortcuts to different functions in development phase. The new configuration is called DB17-j.

[Figure 5.48](#) shows the components needed to complete this upgrade.



Figure 5.48. The parts needed to upgrade the Duckiebot to the configuration DB17-j.

**Note:** The joystick comes with a USB receiver (as shown in [Figure 5.48](#)).

### 1) Instructions

- 
- Insert the USB receiver into one of the USB ports of the Raspberry Pi.
  - Insert 2 AA batteries on the back side of the joystick.
  - Turn on the joystick by pressing the `HOME` button. Make sure that the LED above the `SELECT` button is steady.



Figure 5.49. Upgrade to DB17-j completed.

**TODO:** explain how to test the joystick with `jstest`

## 5.14. Upgrade to DB17-d

This upgrade equips the Duckiebot with an external hard drive that is convenient for storing videos (logs) as it provides both extra capacity and faster data transfer rates than the microSD card in the Raspberry Pi 3. Moreover, it is easy to unplug it from the Duckiebot at the end of the day and bring it over to a computer for downloading and analyzing stored data. The new configuration is called `DB17-d`.

[Figure 5.50](#) shows the components needed to complete this upgrade.



Figure 5.50. The parts needed to upgrade the Duckiebot to the configuration DB17-d.

### 1) Instructions

- Insert the USB drive into one of the USB ports of the Raspberry Pi.



Figure 5.51. Upgrade to DB17-d completed.

- Mount your USB drive as explained in [Mounting USB drives \(master\)](#).

## 5.15. FAQ



**Q:** If we have the bumpers, at what point should we add them?

**Answer:** You shouldn't have the bumpers at this point. The function of the bumpers is to keep the LEDs in place, i.e., they belong to DB17-1 configuration. These instructions cover the DB17-wjd configurations. You will find the bumper assembly instructions in [Unit J-3 - Assembling the Duckiebot \(DB17-1c\) \[draft\]](#).

**Q:** Yeah but I still have the bumpers and am reading this page. So?

**Answer:** The bumpers can be added after the Duckiebot assembly is complete.

**Q:** I found it hard to mount the camera (the holes weren't lining up).

**Answer:** Sometimes in life you have to push a little to make things happen. (But don't push too much or things will break!)

**Q:** The long camera cable is a bit annoying - I folded it and shoved it in between two hats.

**Answer:** The shorter cable is even more annoying. We suggest wrapping the long camera cable between the chassis and the Raspberry Pi. With some strategic planning, you can use the zip ties that keep the battery in place to hold the camera cable in place as well ([see figure below-to add](#))

**TODO:** add pretty cable handling pic

**Q:** I found that the screwdriver that comes with the chassis kit is too fat to screw in the wires on the hat.

**Answer:** It is possible you got one of the fatter screwdrivers. You will need to figure it out yourself (or ask a TA for help).

**Q:** I need something to cut the end of the zip tie with.

**Answer:** Scissors typically work out for these kind of jobs (and no, they're not provided in a Fall 2017 Duckiebox).

## UNIT I-6

### Installing Ubuntu on laptops



Assigned to: Andrea

Before you prepare the Duckiebot, you need to have a laptop with Ubuntu installed.

#### KNOWLEDGE AND ACTIVITY GRAPH

Requires: A laptop with free disk space.

Requires: Internet connection to download the Ubuntu image.

**Requires:** About 30 minutes.

**Results:** A laptop ready to be used for Duckietown.

## 6.1. Install Ubuntu

Install Ubuntu 16.04.3.

→ For instructions, see for example [this online tutorial](#).

**On the choice of username:** During the installation, create a user for yourself with a username different from `ubuntu`, which is the default. Otherwise, you may get confused later.

## 6.2. Install useful software

Use `etckeeper` to keep track of the configuration in `/etc`:

 \$ sudo apt install etckeeper

Install `ssh` to login remotely and the server:

 \$ sudo apt install ssh

Use `byobu`:

 \$ sudo apt install byobu

Use `vim`:

 \$ sudo apt install vim

Use `htop` to monitor CPU usage:

 \$ sudo apt install htop

Additional utilities for `git`:

 \$ sudo apt install git git-extras

Other utilities:

 \$ sudo apt install avahi-utils ecryptfs-utils

## 6.3. Install ROS

Install ROS on your laptop.

- The procedure is given in [Section 2.1 - Install ROS](#).

## 6.4. Other suggested software

### 1) Redshift

---

This is Flux for Linux. It is an accessibility/lab safety issue: bright screens damage eyes and perturb sleep [\[6\]](#).

Install redshift and run it.

 \$ sudo apt install redshift-gtk

Set to “autostart” from the icon (on the panel - near wifi/lan).

## 6.5. Passwordless sudo

Set up passwordless `sudo`.

- This procedure is described in [Passwordless sudo \(master\)](#).

+ comment

Huh I don't know - this is great for usability, but horrible for security. If you step away from your laptop for a second and don't lock the screen, a nasty person could `sudo rm -rf / . -FG`

## 6.6. SSH and Git setup

### 1) Basic SSH config

---

Do the basic SSH config.

- The procedure is documented in [Local configuration \(master\)](#).

### 2) Create key pair for `username`

---

Next, create a private/public key pair for the user; call it `username@robot name`.

- The procedure is documented in [Creating an SSH keypair \(master\)](#).

### 3) Add **username**'s public key to Github

---

Add the public key to your Github account.

- The procedure is documented in [Add a public key to Github \(master\)](#).

If the step is done correctly, this command should succeed:



```
$ ssh -T git@github.com
```

### 4) Local Git setup

---

Set up Git locally.

- The procedure is described in [Setting up global configurations for Git \(master\)](#).

## 6.7. Installation of the duckuments system

Optional but very encouraged: install the duckuments system. This will allow you to have a local copy of the documentation and easily submit questions and changes.

- The procedure is documented in [Section 1.3 - Installing the documentation system](#).

## UNIT I-7

## Duckiebot Initialization

Assigned to: Andrea

### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** An SD card of dimensions at least 16 GB.

**Requires:** A computer with an internet connection, an SD card reader, and 16 GB of free space.

**Requires:** An assembled Duckiebot in configuration DB17. This is the result of [Unit I-4 - Assembling the Duckiebot \(DB17-jwd\)](#).

**Results:** A Duckiebot that is configured correctly, that you can connect to with your laptop and hopefully also has internet access

### 7.1. Acquire and burn the image

On the laptop, download the compressed image at this URL:

<https://www.dropbox.com/s/ckpqpp0cav3aucb/duckiebot-RPI3-AD-2017-09-12.img.xz?dl=1>

The size is 1.7 GB.

You can use:

```
$ wget -O duckiebot-RPI3-AD-2017-09-12.img.xz URL above
```

+ comment

The original was:

```
$ curl -o duckiebot-RPI3-AD-2017-09-12.img.xz URL above
```

It looks like that `curl` cannot be used with Dropbox links because it does not follow redirects.

To make sure that the image is downloaded correctly, compute its hash using the program `sha256sum`:

```
$ sha256sum duckiebot-RPI3-AD-2017-09-12.img.xz  
7136f9049b230de68e8b2d6df29ece844a3f830cc96014aaa92c6d3f247b6130  
duckiebot-RPI3-AD-2017-09-12.img.xz
```

Compare the hash that you obtain with the hash above. If they are different, there was some problem in downloading the image.

Uncompress the file:

```
$ xz -d -k --verbose duckiebot-RPI3-AD-2017-09-12.img.xz
```

This will create a file of 11 GB in size.

Next, burn the image on disk.

- The procedure of how to burn an image is explained in [How to burn an image to an SD card \(master\)](#).

## 7.2. Turn on the Duckiebot

Put the SD Card in the Duckiebot.

Turn on the Duckiebot by connecting the power cable to the battery.

**TODO:** Add figure

+ comment

In general, for the battery: if it's off, a single click on the power button will turn the battery on. When it's on, a single click will show you the charge indicator (4 white lights = full), and holding the button for 3s will turn off the battery. Shutting down the Duckiebot is not recommended because it may cause corruption of the SD card.

### 7.3. Connect the Duckiebot to a network

You can login to the Duckiebot in two ways:

1. Through an Ethernet cable.
2. Through a `duckietown` WiFi network.

In the worst case, you can use an HDMI monitor and a USB keyboard.

#### 1) Option 1: Ethernet cable

---

Connect the Duckiebot and your laptop to the same network switch.

Allow 30 s - 1 minute for the DHCP to work.

#### 2) Option 2: Duckietown network

---

The Duckiebot connects automatically to a 2.4 GHz network called “`duckietown`” and password “`quackquack`”.

Connect your laptop to the same wireless network.

### 7.4. Ping the Duckiebot

To test that the Duckiebot is connected, try to ping it.

The hostname of a freshly-installed duckiebot is `duckiebot-not-configured`:

 \$ ping `duckiebot-not-configured.local`

You should see output similar to the following:

```
PING duckiebot-not-configured.local (X.X.X.X): 56 data bytes  
64 bytes from X.X.X.X: icmp_seq=0 ttl=64 time=2.164 ms  
64 bytes from X.X.X.X: icmp_seq=1 ttl=64 time=2.303 ms  
...
```

### 7.5. SSH to the Duckiebot

Next, try to log in using SSH, with account `ubuntu`:

 \$ ssh ubuntu@duckiebot-not-configured.local

The password is `ubuntu`.

By default, the robot boots into Byobu.

Please see [Byobu \(master\)](#) for an introduction to Byobu.

+ doubt

Not sure it's a good idea to boot into Byobu. -??

## 7.6. Setup network

→ [Unit I-8 - Networking aka the hardest part](#)

## 7.7. Update the system

Next, we need to update to bring the system up to date.

Use these commands



```
$ sudo apt update  
$ sudo apt dist-upgrade
```

## 7.8. Give a name to the Duckiebot

It is now time to give a name to the Duckiebot.

These are the criteria:

- It should be a simple alphabetic string (no numbers or other characters like “`-`”, “`_`”, etc.).
- It will always appear lowercase.
- It cannot be a generic name like “`duckiebot`”, “`robot`” or similar.

From here on, we will refer to this string as “`robot name`”. Every time you see `robot name`, you should substitute the name that you chose.

## 7.9. Change the hostname

We will put the robot name in configuration files.

**Note:** Files in `/etc` are only writable by `root`, so you need to use `sudo` to edit them.  
For example:



```
$ sudo vi filename
```

Edit the file

```
/etc/hostname
```

and put “`robot name`” instead of `duckiebot-not-configured`.

Also edit the file

```
/etc/hosts
```

and put “`robot name`” where `duckiebot-not-configured` appears.

The first two lines of `/etc/hosts` should be:

```
127.0.0.1 localhost  
127.0.1.1 robot name
```

**Note:** there is a command `hostname` that promises to change the hostname. However, the change given by that command does not persist across reboots. You need to edit the files above for the changes to persist.

**Note:** Never add other hostnames in `/etc/hosts`. It is a tempting fix when DNS does not work, but it will cause other problems subsequently.

Then reboot the Raspberry Pi using the command

```
$ sudo reboot
```

After reboot, log in again, and run the command `hostname` to check that the change has persisted:

```
$ hostname  
robot name
```

## 7.10. Expand your filesystem

If your SD card is larger than the image, you’ll want to expand the filesystem on your robot so that you can use all of the space available. Achieve this with:



```
$ sudo raspi-config --expand-rootfs
```

and then reboot



```
$ sudo shutdown -r now
```

once rebooted you can test whether this was successful by doing

 \$ df -lh

the output should give you something like:

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/root	15G	6.3G	8.2G	44%	/
devtmpfs	303M	0	303M	0%	/dev
tmpfs	431M	0	431M	0%	/dev/shm
tmpfs	431M	12M	420M	3%	/run
tmpfs	5.0M	4.0K	5.0M	1%	/run/lock
tmpfs	431M	0	431M	0%	/sys/fs/cgroup
/dev/mmcblk0p1	63M	21M	43M	34%	/boot
tmpfs	87M	0	87M	0%	/run/user/1000

You should see that the Size of your `/dev/root` Filesystem is “close” to the size of your SD card.

## 7.11. Create your user



You must not use the `ubuntu` user for development. Instead, you need to create a new user.

Choose a user name, which we will refer to as `username`.

To create a new user:

 \$ sudo useradd -m `username`

Make the user an administrator by adding it to the group `sudo`:

 \$ sudo adduser `username` sudo

Make the user a member of the groups `input`, `video`, and `i2c`

 \$ sudo adduser `username` input  
\$ sudo adduser `username` video  
\$ sudo adduser `username` i2c

Set the shell `bash`:

 \$ sudo chsh -s /bin/bash `username`

To set a password, use:



```
$ sudo passwd username
```

At this point, you should be able to login to the new user from the laptop using the password:



```
$ ssh username@robot name
```

Next, you should repeat some steps that we already described.

+ comment

What steps?? -LP

### 1) Basic SSH config

---

Do the basic SSH config.

- The procedure is documented in [Local configuration \(master\)](#).

### 2) Create key pair for **username**

---

Next, create a private/public key pair for the user; call it **username@robot name**.

- The procedure is documented in [Creating an SSH keypair \(master\)](#).

### 3) Add SSH alias

---

Once you have your SSH key pair on both your laptop and your Duckiebot, as well as your new user- and hostname set up on your Duckiebot, then you should set up an SSH alias as described in [Section 13.1 - SSH aliases](#). This allows your to log in for example with



```
$ ssh abc
```

instead of



```
$ ssh username@robot name
```

where you can chose **abc** to be any alias / shortcut.

### 4) Add **username**'s public key to Github

---

Add the public key to your Github account.

- The procedure is documented in [Add a public key to Github \(master\)](#).

If the step is done correctly, the following command should succeed and give you a welcome message:



```
$ ssh -T git@github.com
Hi username! You've successfully authenticated, but GitHub does not provide shell
access.
```

## 5) Local Git configuration

---

- This procedure is in [Setting up global configurations for Git \(master\)](#).

## 6) Set up the laptop-Duckiebot connection

---

Make sure that you can login passwordlessly to your user from the laptop.

- The procedure is explained in [How to login without a password \(master\)](#). In this case, we have: `local` = laptop, `local-user` = your local user on the laptop, `remote` = `robot name`, `remote-user` = `username`.

If the step is done correctly, you should be able to login from the laptop to the robot, without typing a password:



```
$ ssh username@robot name
```

## 7) Some advice on the importance of passwordless access

---

In general, if you find yourself:

- typing an IP
- typing a password
- typing `ssh` more than once
- using a screen / USB keyboard

it means you should learn more about Linux and networks, and you are setting yourself up for failure.

Yes, you “can do without”, but with an additional 30 seconds of your time. The 30 seconds you are not saving every time are the difference between being productive roboticists and going crazy.

Really, it is impossible to do robotics when you have to think about IPs and passwords...

### 7.12. Other customizations

If you know what you are doing, you are welcome to install and use additional shells, but please keep Bash as be the default shell. This is important for ROS installation.

For the record, our favorite shell is ZSH with [oh-my-zsh](#).

## 7.13. Hardware check: camera

Check that the camera is connected using this command:



```
$ vcgencmd get_camera  
supported=1 detected=1
```

If you see `detected=0`, it means that the hardware connection is not working.

You can test the camera right away using a command-line utility called `raspistill`.

Use the `raspistill` command to capture the file `out.jpg`:



```
$ raspistill -t 1 -o out.jpg
```

Then download `out.jpg` to your computer using `scp` for inspection.

- For instructions on how to use `scp`, see [Download a file with SCP \(mas-ter\)](#).

## 7.14. Final touches: duckie logo

In order to show that your Duckiebot is ready for the task of driving around happy little duckies, the robot has to fly the Duckietown flag. When you are still logged in to the Duckiebot you can download and install the banner like this:

Download the ANSI art file from Github:



```
$ wget --no-check-certificate -O duckie.art "https://raw.githubusercontent.com/  
duckietown/Software/master/misc/duckie.art"
```

(optional) If you want, you can preview the logo by just outputting it onto the command line:



```
$ cat duckie.art
```

Next up create a new empty text file in your favorite editor and add the code for showing your duckie pride:

Let's say I use `nano`, I open a new file:



```
$ nano 20-duckie
```

And in there I add the following code (which by itself just prints the duckie logo):

```
#!/bin/sh
printf "\n$(cat /etc/update-motd.d/duckie.art)\n"
```

Then save and close the file. Finally you have to make this file executable...

 \$ chmod +x 20-duckie

...and copy both the duckie logo and the script into a specific directory `/etc/update-motd.d` to make it appear when you login via SSH. `motd` stands for “message of the day”. This is a mechanism for system administrators to show users news and messages when they login. Every executable script in this directory which has a filename a la `NN-some name` will get exected when a user logs in, where `NN` is a two digit number that indicates the order.

```
sudo cp duckie.art /etc/update-motd.d
sudo cp 20-duckie /etc/update-motd.d
```

Finally log out of SSH via `exit` and log back in to see duckie goodness.

## 1) Troubleshooting

---

**Symptom:** `detected=0`

**Resolution:** If you see `detected=0`, it is likely that the camera is not connected correctly.

If you see an error that starts like this:

```
mmal: Cannot read camera info, keeping the defaults for OV5647
...
mmal: Camera is not detected. Please check carefully the camera module is installed
correctly.
```

then, just like it says: “Please check carefully the camera module is installed correctly.”

**Symptom:** random `wget`, `curl`, `git`, and `apt` calls fail with SSL errors.

**Resolution:** That’s probably actually an issue with your system time. Type the command `timedatectl` into a terminal, hit enter and see if the time is off. If it is, you might want to follow the instructions from [this article](#), or entirely [uninstall your NTP service and manually grab the time on reboot](#). It’s a bit dirty, but works surprisingly well.

**Symptom:** Cannot find `/etc` folder for configuring the Wi-Fi. I only see `Desktop`, `Downloads` when starting up the Duckiebot.

**Resolution:** If a directory name starts with `/`, it’s not supposed to be in the home directory, but rather at the root of the filesystem. You are currently in `/home/ubuntu`. Type

`ls /` to see the folders at the root, including `/etc.

## UNIT I-8

# Networking aka the hardest part

### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** A Duckiebot in configuration `DB17-C0+w`

**Requires:** Either a router that you have control over that has internet access, or your credentials for connecting to an existing wireless network

**Requires:** Patience (channel your inner Yoda)

**Results:** A Duckiebot that you can connect to and that is connected to the internet

**Note:** this page is primarily for folks operating with the “two-network” configuration, `C0+w`. For a one adapter setup you will can skip directly to [Section 8.2 - Setting up wireless network configuration](#), but you will have to connect to a network that you can ssh through.

The basic idea is that we are going to use the “Edimax” thumbdrive adapter to create a dedicated wireless network that you can always connect to with your laptop. Then we are going to use the built-in Broadcom chip on the Pi to connect to the internet, and then the network will be bridged.

### 8.1. (For DB17-w) Configure the robot-generated network

This part should work every time with very low uncertainty.

The Duckiebot in configuration `C0+w` can create a WiFi network.

It is a 5 GHz network; this means that you need to have a 5 GHz WiFi adapter in your laptop.

First, make sure that the Edimax is correctly installed. Using `iwconfig`, you should see four interfaces:



```
$ iwconfig  
wlan0 AABCCDDEEFFGG unassociated Nickname:"rtl8822bu"  
...  
lo      no wireless extensions.  
enxb827eb1f81a4  no wireless extensions.  
wlan1    IEEE 802.11bgn  ESSID:"duckietown"  
...
```

Make note of the name `wlan0 AABCCDDEEFFGG`.

Look up the MAC address using the command:

```
$ ifconfig wlan0 AABCCDDEEFFGG  
wlan0 AABCCDDEEFFGG Link encap:Ethernet HWaddr AA:BB:CC:DD:EE:FF:GG
```

Then, edit the connection file

```
/etc/NetworkManager/system-connections/create-5ghz-network
```

Make the following changes:

- Where it says `interface-name=...`, put “`wlan0 AABCCDDEEFFGG`”.
- Where it says `mac-address=...`, put “`AA:BB:CC:DD:EE:FF:GG`”.
- Where it says `ssid=duckiebot-not-configured`, put “`ssid=robot name`”.

![Newly upgraded]

To ensure nobody piggybacks on our connection, which poses a security risk especially in a public environment, we will protect access to the 5 GHz WiFi through a password. To set a password you will need to log in the Duckiebot with the default “ubuntu” user-name and password and change your system files. In the `/etc/NetworkManager/system-connections/create-5ghz-network`, add:

```
[wifi-security]  
key-mgmt=wpa-psk  
psk=YOUR_OWN_WIFI_PASSWORD_NO_QUOTATION_MARKS_NEEDED  
auth-alg=open
```

and then reboot.

At this point you should see a new network being created named “`robot name`”, protected by the password you just set.

+ comment

Make sure the password contains min. 8 character or combined with numbers. If no networks shows up after the configuration and no feedback from system, please check the content of the file again. The program, which activates the Edimax wifi adapter is very sensitive to its content.

?? | Adding a password to your 5GHz connection is a mandatory policy in the Zurich branch.

## 8.2. Setting up wireless network configuration

You are connected to the Duckiebot via WiFi, but the Duckiebot also needs to connect to the internet in order to get updates and install some software. This part is a little bit more of a “black art” since we cannot predict every possible network configurations. Below are some settings that have been verified to work in different situations:

### 1) Option 1: duckietown WiFi

Check with your phone or laptop if there is a WiFi in reach with the name of `duckietown`. If there is, you are all set. The default configuration for the Duckiebot is to have one WiFi adapter connect to this network and the other broadcast the access point which you are currently connected to.

### 2) Option 2.a): eduroam WiFi (Non-UdeM/McGill instructions)

If there should be no `duckietown` network in reach then you have to manually add a network configuration file for the network that you'd like to connect to. Most universities around the world should have to `eduroam` network available. You can use it for connecting your Duckiebot.

Save the following block as new file in `/etc/NetworkManager/system-connections/eduroam`:

```

[connection]
id=eduroam
uuid=38ea363b-2db3-4849-a9a4-c2aa3236ae29
type=wifi
permissions=user:oem:;
secondaries=

[wifi]
mac-address=the MAC address of your internal wifi adapter, wlan0
mac-address-blacklist=
mac-address-randomization=@
mode=infrastructure
seen-bssids=
ssid=eduroam

[wifi-security]
auth-alg=open
group=
key-mgmt=wpa-eap
pairwise=
proto=

[802-1x]
altsubject-matches=
eap=tls;
identity=your eduroam username@your eduroam domain
password=your eduroam password
phase2-altsubject-matches=
phase2-auth=pap

[ipv4]
dns-search=
method=auto

[ipv6]
addr-gen-mode=stable-privacy
dns-search=
method=auto

```

Set the permissions on the new file to 0600.

```
sudo chmod 0600 /etc/NetworkManager/system-connections/eduroam
```

### 3) Option 2.b): eduroam WiFi (UdeM/McGill instructions)

Save the following block as new file in `/etc/NetworkManager/system-connections/  
eduroam-USERNAME`: where USERNAME is the your logged-in username in the duck-  
iebot.

```

[connection]
id=eduroam
uid=38ea363b-2db3-4849-a9a4-c2aa3236ae29
type=wifi
permissions=user:USERNAME:;
secondaries=

[wifi]
mac-address=the MAC address of your internal wifi adapter, wlan0
mac-address-blacklist=
mac-address-randomization=0
mode=infrastructure
seen-bssids=
ssid=eduroam

[wifi-security]
auth-alg=open
group=
key-mgmt=wpa-eap
pairwise=
proto=

[802-1x]
altsubject-matches=
eap=peap;
identity=DGTIC UNIP
password=DGTIC PWD
phase2-altsubject-matches=
phase2-auth=mschapv2

[ipv4]
dns-search=
method=auto

[ipv6]
addr-gen-mode=stable-privacy
dns-search=
method=auto

```

Set the permissions on the new file to 0600.

```
sudo chmod 0600 /etc/NetworkManager/system-connections/eduroam-USERNAME
```

4) Option 3 (For Université de Montréal students only): Use UdeM avec cryptage

**TODO:** someone replicate please - LP

**Note:** you can use the `autoconnect-priority=XX` inside the `[connection]` block to establish

a priority. If you want to connect to one network preferentially if two are available then give it a higher priority.

Save the following block as new file in `/etc/NetworkManager/system-connections/secure`:

```
[connection]
id=secure
uuid=e9cef1bd-f6fb-4c5b-93cf-cca837ec35f2
type=wifi
permissions=
secondaries=
timestamp=1502254646
autoconnect-priority=100

[wifi]
mac-address-blacklist=
mac-address-randomization=0
mode=infrastructure
ssid=UdeM avec cryptage
security=wifi-security

[wifi-security]
key-mgmt=wpa-eap

[802-1x]
eap=peap;
identity=DGTIC UNIP
phase2-auth=mschapv2
password=DGTIC PWD

[ipv4]
dns-search=
method=auto

[ipv6]
addr-gen-mode=stable-privacy
dns-search=
ip6-privacy=0
method=auto
```

Set the permissions on the new file to 0600.

```
sudo chmod 600 /etc/NetworkManager/system-connections/secure
```

## 5) Option 4: custom WiFi

First run the following to see what networks are available:

 \$ nmcli dev wifi list

You should see the network that you are trying to connect (**SSID**) to and you should know the password. To connect to it run:



```
$ sudo nmcli dev wifi con SSID password PASSWORD
```

## 6) Option 5: ETH Wifi



The following instructions will lead you to connect your PI to the “eth” wifi network.

First, run the following on duckiebot



```
$ iwconfig  
...  
  
lo      no wireless extensions.  
  
enxbxxxxxxxxx  no wireless extensions.  
  
...
```

Make note of the name `enxbxxxxxxxxx`. `xxxxxxxxx` should be a string that has 11 characters that is formed by numbers and lower case letters.

Second, edit the file `/etc/network/interfaces` which requires `sudo` so that it looks like the following, and make sure the `enxbxxxxxxxxx` matches.

Pay special attention on the line “pre-up wpa\_supplicant -B -D wext -i wlan0 -c /etc/wpa\_supplicant/wpa\_supplicant.conf”. This is expected to be exactly one line instead of two but due to formatting issue it is shown as two lines.

Also, make sure every characters match exactly with the provided ones. TAs will not help you to do spelling error check.

```
# interfaces(5) file used by ifup(8) and ifdown(8) Include files from /etc/network/
interfaces.d:
source-directory /etc/network/interfaces.d

# The loopback network interface
auto lo
auto enxbxxxxxxxxx

# the wired network setting
iface enxbxxxxxxxxx inet dhcp

# the wireless network setting
auto wlan0
allow-hotplug wlan0
iface wlan0 inet dhcp
    pre-up wpa_supplicant -B -D wext -i wlan0 -c /etc/wpa_supplicant/wpa_supplicant.conf
    post-down killall -q wpa_supplicant
```

Third, edit the file `/etc/wpa_supplicant/wpa_supplicant.conf` which requires `sudo` so that it looks like the following, and make sure you substitute [identity] and [password] content with your eth account information:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1

network={
    ssid="eth"
    key_mgmt=WPA-EAP
    group=CCMP TKIP
    pairwise=CCMP TKIP
    eap=PEAP
    proto=RSN
    identity="your user name goes here"
    password="your password goes here"
    phase1="peaplabel=0"
    phase2="auth=MSCHAPV2"
    priority=1
}
```

Fourth, reboot your PI.



```
$ sudo reboot
```

Then everything shall be fine. The PI will connect to “eth” automatically everytime it starts.

Note that, if something went wrong, your Duckiebot tries to connect to the network for 5.5mins at startup while it's blocking SSH connection to it completely (“Connec-

tion refused” error when connecting). If this is the case, please wait those 5.5mins until your Duckiebot lets you connect again and recheck your settings.

**TODO:** Find a solution to this since it occurs very often

## UNIT I-9

# Software setup and RC remote control

### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Laptop configured, according to [Unit I-6 - Installing Ubuntu on laptops](#).

**Requires:** You have configured the Duckiebot. The procedure is documented in [Unit I-7 - Duckiebot Initialization](#).

**Requires:** You have created a Github account and configured public keys, both for the laptop and for the Duckiebot. The procedure is documented in [Setup Github access \(master\)](#).

**Results:** You can run the joystick demo.

## 9.1. Clone the Duckietown repository

Clone the repository in the directory `~/duckietown`:



```
$ git clone git@github.com:duckietown/Software.git ~/duckietown
```

For the above to succeed you should have a Github account already set up.

It should not ask for a password.

**Note:** you must not clone the repository using the URL starting with `https`. Later steps will fail.

### 1) Troubleshooting

**Symptom:** It asks for a password.

**Resolution:** You missed some of the steps described in [Setup Github access \(master\)](#).

**Symptom:** Other weird errors.

**Resolution:** Probably the time is not set up correctly. Use `ntpdate` as above:

```
$ sudo ntpdate -u us.pool.ntp.org
```

Or see the hints in the troubleshooting section on the previous page.

## 9.2. Update the system

The software used for the Duckiebots changes every day, this means that also the dependencies change. In order to check whether your system meets all the requirements for running the software and install all the missing packages (if any), we can run the following script:



```
$ cd ~/duckietown  
$ /bin/bash ./dependencies_for_duckiebot.sh
```

This command will install only the packages that are not already installed in your system.

## 9.3. Set up the ROS environment on the Duckiebot

All the following commands should be run in the `~/duckietown` directory:



```
$ cd ~/duckietown
```

Now we are ready to make the workspace. First you need to source the baseline ROS environment:



```
$ source /opt/ros/kinetic/setup.bash
```

Then, build the workspace using:



```
$ catkin_make -C catkin_ws/
```

\* For more information about `catkin_make`, see [Section 2.6 - catkin\\_make](#).

**Note:** there is a known bug, for which it fails the first time on the Raspberry Pi. Try again; it will work.

+ comment

I got no error on first execution on the Raspberry Pi

## 9.4. Clone the duckiefleet repository

Clone the relevant `duckiefleet` repository into `~/duckiefleet`.

See [Duckiefleet directory DUCKIEFLEET\\_ROOT \(master\)](#) to find the right `duckiefleet` repository.

In `~/.bashrc` set `DUCKIEFLEET_ROOT` to point to the directory:

```
export DUCKIEFLEET_ROOT=~/duckiefleet
```

Also, make sure that you execute `~/.bashrc` in the current shell by running the command:

```
source ~/.bashrc
```

## 9.5. Add your vehicle data to the robot database

Next, you need to add your robot to the vehicles database. This is not optional and required in order to launch any ROS scripts.

You have already a copy of the vehicles database in the folder `robots` of `DUCKIEFLEET_ROOT`.

Copy the file `emma.robot.yaml` to `robotname.robot.yaml`, where `robotname` is your robot's hostname. Then edit the copied file to represent your Duckiebot.

- For information about the format, see [The “scuderia” \(vehicle database\) \(master\)](#).

Generate the machines file.

- The procedure is listed here: [The machines file \(master\)](#).

Finally, push your robot configuration to the duckiefleet repo.

## 9.6. Test that the joystick is detected

Plug the joystick receiver in one of the USB port on the Raspberry Pi.

To make sure that the joystick is detected, run:

```
raspberrypi ~ $ ls /dev/input/
```

and check if there is a device called `js0` on the list.

### Check before you continue

Make sure that your user is in the group `input` and `i2c`:

```
raspberrypi ~ $ groups  
username sudo input i2c
```

If `input` and `i2c` are not in the list, you missed a step. Ohi oh! You are not following the instructions carefully!

- Consult again [Section 7.11 - Create your user.](#)

To test whether or not the joystick itself is working properly, run:

 `$ jstest /dev/input/js0`

Move the joysticks and push the buttons. You should see the data displayed change according to your actions.

## 9.7. Run the joystick demo

SSH into the Raspberry Pi and run the following from the `duckietown` directory:

 `$ cd ~/duckietown`  
`$ source environment.sh`

The `environment.sh` setups the ROS environment at the terminal (so you can use commands like `rosrun` and `roslaunch`).

Now make sure the motor shield is connected.

Run the command:

 `$ roslaunch duckietown joystick.launch veh:=robot name`

If there is no “red” output in the command line then pushing the left joystick knob controls throttle - right controls steering.

This is the expected result of the commands:

left joystick up	forward
left joystick down	backward
right joystick left	turn left (positive yaw)
right joystick right	turn right (negative yaw)

It is possible you will have to unplug and replug the joystick or just push lots of buttons on your joystick until it wakes up. Also make sure that the mode switch on the top of your joystick is set to “X”, not “D”.

XXX Is all of the above valid with the new joystick?

Close the program using `Ctrl-C`.

## 1) Troubleshooting

---

| **Symptom:** The robot moves weirdly (e.g. forward instead of backward).

**Resolution:** The cables are not correctly inserted. Please refer to the assembly guide for pictures of the correct connections. Try swapping cables until you obtain the expected behavior.

**Resolution:** Check that the joystick has the switch set to the position “x”. And the mode light should be off.

| **Symptom:** The left joystick does not work.

**Resolution:** If the green light on the right to the “mode” button is on, click the “mode” button to turn the light off. The “mode” button toggles between left joystick or the cross on the left.

| **Symptom:** The robot does not move at all.

**Resolution:** The cables are disconnected.

**Resolution:** The program assumes that the joystick is at `/dev/input/js0`. In doubt, see [Section 9.6 - Test that the joystick is detected](#).

## 9.8. The proper shutdown procedure for the Raspberry Pi

Generally speaking, you can terminate any `roslaunch` command with `Ctrl-C`.

To completely shutdown the robot, issue the following command:



```
$ sudo shutdown -h now
```

Then wait 30 seconds.

**Warning:** If you disconnect the power before shutting down properly using `shutdown`, the system might get corrupted.

Then, disconnect the power cable, at the **battery end**.

As an alternative you can use the `poweroff` command:



```
$ sudo poweroff
```

**Warning:** If you disconnect frequently the cable at the Raspberry Pi’s end, you might damage the port.

**Requires:** You have configured the Duckiebot. The procedure is documented in [Unit I-7 - Duckiebot Initialization](#).

**Requires:** You know the basics of ROS (launch files, `roslaunch`, topics, `rostopic`).

**TODO:** put reference

**Results:** You know that the camera works under ROS.

### 10.1. Check the camera hardware

It might be useful to do a quick camera hardware check.

- The procedure is documented in [Section 7.13 - Hardware check: camera](#).

### 10.2. Create two windows

On the laptop, create two Byobu windows.

- A quick reference about Byobu commands is in [Byobu \(master\)](#).

You will use the two windows as follows:

- In the first window, you will launch the nodes that control the camera.
- In the second window, you will launch programs to monitor the data flow.

**Note:** You could also use multiple *terminals* instead of one terminal with multiple Byobu windows. However, using Byobu is the best practice to learn.

### 10.3. First window: launch the camera nodes

In the first window, we will launch the nodes that control the camera. All the following commands should be run in the `~/duckietown` directory:

 \$ `cd ~/duckietown`

Activate ROS:

 \$ `source environment.sh`

Run the launch file called `camera.launch`:

 \$ `roslaunch duckietown camera.launch veh:=robot name`

At this point, you should see the red LED on the camera light up continuously.

In the terminal you should not see any red message, but only happy messages like the following:

```
...
[INFO] [1502539383.948237]: [/robot name/camera_node] Initialized.
[INFO] [1502539383.951123]: [/robot name/camera_node] Start capturing.
[INFO] [1502539384.040615]: [/robot name/camera_node] Published the first image.
```

- \* For more information about `roslaunch` and “launch files”, see [Section 2.3 - roslaunch](#).

## 10.4. Second window: view published topics

Switch to the second window. All the following commands should be run in the `~/duckietown` directory:

 \$ `cd ~/duckietown`

Activate the ROS environment:

 \$ `source environment.sh`

### 1) List topics

---

You can see a list of published topics with the command:

 \$ `rostopic list`

- \* For more information about `rostopic`, see [Section 2.5 - rostopic](#).

You should see the following topics:

```
/>/robot name/camera_node/camera_info
/>/robot name/camera_node/image/compressed
/>/robot name/camera_node/image/raw
/>/rosout
/>/rosout_agg
```

### 2) Show topics frequency

---

You can use `rostopic hz` to see the statistics about the publishing frequency:

 \$ `rostopic hz /robot name/camera_node/image/compressed`

On a Raspberry Pi 3, you should see a number close to 30 Hz:

```
average rate: 30.016
min: 0.026s max: 0.045s std dev: 0.00190s window: 841
```

### 3) Show topics data

You can view the messages in real time with the command `rostopic echo`:

 `$ rostopic echo /robot_name/camera_node/image/compressed`

You should see a large sequence of numbers being printed to your terminal.

That's the "image" — as seen by a machine.

If you are Neo, then this already makes sense. If you are not Neo, in [Unit I-12 - RC+camera remotely](#), you will learn how to visualize the image stream on the laptop using `rviz`.

use `Ctrl-C` to stop `rostopic`.

**TODO:** Physically focus the camera.

## UNIT I-11 RC control launched remotely

Assigned to: Andrea

### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** You can run the joystick demo from the Raspberry Pi. The procedure is documented in [Unit I-9 - Software setup and RC remote control](#).

**Results:** You can run the joystick demo from your laptop.

### 11.1. Two ways to launch a program

ROS nodes can be launched in two ways:

1. “local launch”: log in to the Raspberry Pi using SSH and run the program from there.
2. “remote launch”: run the program directly from a laptop.

Which is better when is a long discussion that will be done later. Here we set up the “remote launch”.

**TODO:** draw diagrams

## 11.2. Download and setup Software repository on the laptop

As you did on the Duckiebot, you should clone the `Software` repository in the `~/duckietown` directory.

- The procedure is documented in [Section 9.1 - Clone the Duckietown repository](#).

Then, you should build the repository.

- This procedure is documented in [Section 9.3 - Set up the ROS environment on the Duckiebot](#).

## 11.3. Rebuild the machines files

In a previous step you have created a robot configuration file and pushed it to the duckiefleet repo. Now you have to pull duckiefleet on the laptop and rebuild the machines configuration file there.

- The procedure is documented in [Section 9.5 - Add your vehicle data to the robot database](#).

## 11.4. Start the demo

Now you are ready to launch the joystick demo remotely.

### Check before you continue

Make sure that you can login with SSH **without a password**. From the laptop, run:

 `$ ssh username@robot name.local`

If this doesn't work, you missed some previous steps.

Run this *on the laptop*:

 `$ source environment.sh  
$ rosrun duckietown joystick.launch veh:=robot name`

You should be able to drive the vehicle with joystick just like the last example. Note that remotely launching nodes from your laptop doesn't mean that the nodes are running on your laptop. They are still running on the Raspberry Pi in this case.

\* For more information about `roslaunch`, see [Section 2.3 - roslaunch](#).

## 11.5. Watch the program output using `rqt_console`

Also, you might have noticed that the terminal where you launch the launch file is not printing all the printouts like the previous example. This is one of the limitations of remote launch.

Don't worry though, we can still see the printouts using `rqt_console`.

On the laptop, open a new terminal window, and run:

```
💻 $ export ROS_MASTER_URI=http://robot_name.local:11311/  
$ rqt_console
```

You should see a nice interface listing all the printouts in real time, completed with filters that can help you find that message you are looking for in a sea of messages. If `rqt_console` does not show any message, check out the *Troubleshooting* section below.

You can use `Ctrl-C` at the terminal where `roslaunch` was executed to stop all the nodes launched by the launch file.

\* For more information about `rqt_console`, see [Section 2.2 - rqt\\_console](#).

## 11.6. Troubleshooting

| **Symptom:** `rqt_console` does not show any message.

**Resolution:** Open `rqt_console`. Go to the Setup window (top-right corner). Change the “Rosout Topic” field from `/rosout_agg` to `/rosout`. Confirm.

| **Symptom:** `roslaunch` fails with an error similar to the following:

```
remote[robot_name.local-0]: failed to launch on robot_name:
```

```
Unable to establish ssh connection to [username@robot_name.local:22]:  
Server u'robot_name.local' not found in known_hosts.
```

**Resolution:** You have not followed the instructions that told you to add the `HostKeyAlgorithms` option. Delete `~/.ssh/known_hosts` and fix your configuration.

- The procedure is documented in [Local configuration \(master\)](#).

## UNIT I-12

### RC+camera remotely

| Assigned to: Andrea

**Requires:** You can run the joystick demo remotely. The procedure is documented in [Unit I-11 - RC control launched remotely](#).

**Requires:** You can read the camera data from ROS. The procedure is documented in [Unit I-10 - Reading from the camera](#).

**Requires:** You know how to get around in Byobu. You can find the Byobu tutorial in [Byobu \(master\)](#).

**Results:** You can run the joystick demo from your laptop and see the camera image on the laptop.

## 12.1. Assumptions

We are assuming that the joystick demo in [Unit I-11 - RC control launched remotely](#) worked.

We are assuming that the procedure in [Unit I-10 - Reading from the camera](#) succeeded.

We also assume that you terminated all instances of `roslaunch` with `Ctrl-C`, so that currently there is nothing running in any window.

## 12.2. Terminal setup

On the laptop, this time create four Byobu windows.

→ A quick reference about Byobu commands is in [Byobu \(master\)](#).

You will use the four windows as follows:

- In the first window, you will run the joystick demo, as before.
- In the second window, you will launch the nodes that control the camera.
- In the third window, you will launch programs to monitor the data flow.
- In the fourth window, you will use `rviz` to see the camera image.

**TODO:** Add figures

## 12.3. First window: launch the joystick demo

In the first window, launch the joystick remotely using the same procedure in [Section 11.4 - Start the demo](#).



```
$ source environment.sh
$ rosrun duckietown joystick.launch veh:=robot name
```

You should be able to drive the robot with the joystick at this point.

## 12.4. Second window: launch the camera nodes

In the second window, we will launch the nodes that control the camera.

The launch file is called `camera.launch`:

```
💻 $ source environment.sh  
$ roslaunch duckietown camera.launch veh:=robot name
```

You should see the red led on the camera light up.

+ comment

It is recommended to launch the joystick and the camera from onboard the robot after sshing in - LP

## 12.5. Third window: view data flow

Open a third terminal on the laptop.

You can see a list of topics currently on the `ROS_MASTER` with the commands:

```
💻 $ source environment.sh  
$ export ROS_MASTER_URI=http://robot name.local:11311/  
$ rostopic list
```

You should see the following:

```
/diagnostics  
/robot name/camera_node/camera_info  
/robot name/camera_node/image/compressed  
/robot name/camera_node/image/raw  
/robot name/joy  
/robot name/wheels_driver_node/wheels_cmd  
/rosout  
/rosout_agg
```

## 12.6. Fourth window: visualize the image using `rviz`

Launch `rviz` by using these commands:

```
💻 $ source environment.sh  
$ source set_ros_master.sh robot name  
$ rviz
```

\* For more information about `rviz`, see [Section 2.4 - rviz](#).

In the `rviz` interface, click “Add” on the lower left, then the “By topic” tag, then select the “Image” topic by the name

```
/robot name/camera_node/image/compressed
```

Then click “ok”. You should be able to see a live stream of the image from the camera.

## 12.7. Proper shutdown procedure

To stop the nodes: You can stop the node by pressing `Ctrl-C` on the terminal where `roslaunch` was executed. In this case, you can use `Ctrl-C` in the terminal where you launched the `camera.launch`.

You should see the red light on the camera turn off in a few seconds.

Note that the `joystick.launch` is still up and running, so you can still drive the vehicle with the joystick.

## UNIT I-13

### Interlude: Ergonomics



Assigned to: Andrea

So far, we have been spelling out all commands for you, to make sure that you understand what is going on.

Now, we will tell you about some shortcuts that you can use to save some time.

**Note:** in the future you will have to debug problems, and these problems might be harder to understand if you rely blindly on the shortcuts.

#### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Time: 5 minutes.

**Results:** You will know about some useful shortcuts.

## 13.1. SSH aliases



Instead of using

```
$ ssh username@robot name.local
```

You can set up SSH so that you can use:

```
$ ssh my-robot
```

To do this, create a host section in `~/.ssh/config` on your laptop with the following contents:

```
Host my-robot
User username
Hostname robot name.local
```

Here, you can choose any other string in place of “`my-robot`”.

Note that you **cannot** do

```
$ ping my-robot
```

You haven’t created another hostname, just an alias for SSH.

However, you can use the alias with all the tools that rely on SSH, including `rsync` and `scp`.

### 13.2. `set_ros_master.sh`

Instead of using:

```
$ export ROS_MASTER_URI=http://robot name.local:11311/
```

You can use the “`set_ros_master.sh`” script in the repo:

```
$ source set_ros_master.sh robot name
```

Note that you need to use `source`; without that, it will not work.

## UNIT I-14

### Camera calibration

#### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** You can see the camera image on the laptop. The procedure is documented in [Unit I-12 - RC+camera remotely](#).

**Requires:** You have all the repositories (described in [Unit A-7 - Git usage guide for Fall 2017](#)) cloned properly and you have your environment variables set properly.

**Results:** Calibration for the robot camera.

## 14.1. Intrinsic calibration

### 1) Setup

Download and print a PDF of the calibration checkerboard ([A4 intrinsic](#), [A3 extrinsic](#), [US Letter](#)). Fix the checkerboard to a planar surface.



Figure 14.1

**Note:** the squares must have side equal to 0.031 m = 3.1 cm.

### 2) Calibration

Make sure your Duckiebot is on, and both your laptop and Duckiebot are connected to the duckietown network.

*Step 1:*

Open two terminals on the laptop.

*Step 2:*

In the first terminal, log in into your robot using SSH and launch the camera process:



```
$ cd duckietown root
$ source environment.sh
$ roslaunch duckietown camera.launch veh:=robot name raw:=true
```

*Step 3:*

In the second laptop terminal run the camera calibration:

```
 $ cd duckietown root
$ source environment.sh
$ source set_ros_master.sh robot name
$ roslaunch duckietown intrinsic_calibration.launch veh:=robot name
```

You should see a display screen open on the laptop (Figure 14.2).



Figure 14.2

Position the checkerboard in front of the camera until you see colored lines overlaying the checkerboard. You will only see the colored lines if the entire checkerboard is within the field of view of the camera.

You should also see colored bars in the sidebar of the display window. These bars indicate the current range of the checkerboard in the camera's field of view:

- X bar: the observed horizontal range (left - right)
- Y bar: the observed vertical range (top - bottom)
- Size bar: the observed range in the checkerboard size (forward - backward from the camera direction)
- Skew bar: the relative tilt between the checkerboard and the camera direction

Also, make sure to focus the image by rotating the mechanical focus ring on the lens of the camera.

+ comment

Do not change the focus during or after the calibration, otherwise your calibration is no longer valid. I'd also suggest to not to use the lens cover anymore; removing the lens cover changes the focus. -MK

Now move the checkerboard right/left, up/down, and tilt the checkerboard through various angles of relative to the image plane. After each movement, make sure to pause long enough for the checkerboard to become highlighted. Once you have collected enough data, all four indicator bars will turn green. Press the “CALIBRATE” button in the sidebar.

Calibration may take a few moments. Note that the screen may dim. Don’t worry, the calibration is working.



Figure 14.3

### 3) Save the calibration results

---

If you are satisfied with the calibration, you can save the results by pressing the “COMMIT” button in the side bar. (You never need to click the “SAVE” button.)



Figure 14.4

This will automatically save the calibration results on your Duckiebot:

```
duckiefleet root/calibrations/camera_intrinsic/robot name.yaml
```

*Step 4:*

Now let's push the `robot name.yaml` file to the git repository. You can stop the `camera.launch` terminal with `Ctrl-C` or open a new terminal in Byobu with `F2`.

Update your local git repository:

```
raspberrypi ~ % $ cd duckiefleet root  
raspberrypi ~ % $ git pull  
raspberrypi ~ % $ git status
```

You should see that your new calibration file is uncommitted. You need to commit the file to your branch.

```
raspberrypi ~ % $ git checkout -b Github username-devel  
raspberrypi ~ % $ git add calibrations/camera_intrinsic/robot name.yaml  
raspberrypi ~ % $ git commit -m "add robot name intrinsic calibration file"  
raspberrypi ~ % $ git push origin Github username-devel
```

Before moving on to the extrinsic calibration, make sure to kill all running processes by pressing `Ctrl-C` in each of the terminal windows.

## 14.2. Extrinsic calibration

### 1) Setup

Arrange the Duckiebot and checkerboard according to [Figure 14.5](#). Note that the axis of the wheels should be aligned with the y-axis ([Figure 14.5](#)).



Figure 14.5

[Figure 14.6](#) shows a view of the calibration checkerboard from the Duckiebot. To ensure proper calibration there should be no clutter in the background and two A4 papers should be aligned next to each other.



Figure 14.6

## 2) Calibration procedure

### Step 1:

Log in into your robot using SSH and launch the camera:

```
raspberrypi ~ % $ cd duckietown root  
$ source environment.sh  
$ roslaunch duckietown camera.launch veh:=robot name raw:=true
```

### Step 2:

Run the `ground_projection_node.py` node on your laptop:

```
laptop ~ % $ cd duckietown root  
$ source environment.sh  
$ source set_ros_master.sh robot name  
$ roslaunch ground_projection ground_projection.launch veh:=robot name local:=true
```

### Step 3:

Check that everything is working properly. In a new terminal (with environment sourced and ros\_master set to point to your robot as above)

```
laptop ~ % $ rostopic list
```

You should see new ros topics:

```
/robot name/camera_node/camera_info  
/robot name/camera_node/framerate_high_switch  
/robot name/camera_node/image/compressed  
/robot name/camera_node/image/raw  
/robot name/camera_node/raw_camera_info
```

The ground\_projection node has two services. They are not used during operation. They just provide a command line interface to trigger the extrinsic calibration (and for debugging).



```
$ rosservice list
```

You should see something like this:

```
...  
/robot name/ground_projection/estimate_homography  
/robot name/ground_projection/get_ground_coordinate  
...
```

If you want to check whether your camera output is similar to the one at the [Figure 14.6](#) you can start `rqt_image_view`:



```
$ rosrun rqt_image_view rqt_image_view
```

In the `rqt_image_view` interface, click on the drop-down list and choose the image topic:

```
/robot name/camera_node/image/compressed
```

*Step 4:*

Now you can estimate the homography by executing the following command (in a new terminal):



```
$ rosservice call /robot name/ground_projection/estimate_homography
```

This will do the extrinsic calibration and automatically save the file to your laptop:

```
duckiefleet root/calibrations/camera_extrinsic, robot name.yaml
```

As before, add this file to your local Git repository on your laptop, push the changes to your branch and do a pull request to master. Finally, you will want to update the local repository on your Duckiebot.

# Updated camera calibration and validation

Here is an updated, more practical extrinsic calibration and validation procedure.

## 15.1. Check out the experimental branch

Check out the branch `andrea-better-camera-calib`.

## 15.2. Place the robot on the pattern

Arrange the Duckiebot and checkerboard according to [Figure 15.1](#). Note that the axis of the wheels should be aligned with the y-axis ([Figure 15.1](#)).



Figure 15.1

[Figure 15.2](#) shows a view of the calibration checkerboard from the Duckiebot. To ensure proper calibration there should be no clutter in the background and two A4 papers should be aligned next to each other.



Figure 15.2

### 15.3. Extrinsic calibration procedure

Run the following on the Duckiebot:



```
$ rosrun complete_image_pipeline calibrate_extrinsics
```

That's it!

No laptop is required.

You can also look at the output files produced, to make sure it looks reasonable. It should look like [Figure 15.3](#).



Figure 15.3

Note the difference between the two types of rectification:

1. In `bgr_rectified` the rectified frame coordinates are chosen so that the frame is filled entirely. Note the image is stretched - the April tags are not square. This is the rectification used in the lane localization pipeline. It doesn't matter that the image is stretched, because the homography learned will account for that deformation.
2. In `rectified_full_ratio_auto` the image is not stretched. The camera matrix is preserved. This means that the aspect ratio is the same. In particular note the April tags are square. If you do something with April tags, you need this rectification.

## 15.4. Camera validation by simulation



You can run the following command to make sure that the camera calibration is reasonable:



```
$ rosrun complete_image_pipeline validate_calibration
```

What this does is simulating what the robot should see, if the models were correct ([Figure 15.4](#)).



Figure 15.4. Result of validate\_calibration.

Then it also tries to localize on the simulated data ([Figure 15.5](#)). It usually achieves impressive calibration results!

Simulations are doomed to succeed.



Figure 15.5. Output of validate\_calibration: localization in simulated environment.

## 15.5. Camera validation by running one-shot localization

Place the robot in a lane.

Run the following command:



```
$ rosrun complete_image_pipeline single_image_pipeline
```

What this does is taking one snapshot and performing localization on that single image. The output will be useful to check that everything is ok.

### 1) Example of correct results

[Figure 15.6](#) is an example in which the calibration was correct, and the robot localizes perfectly.



Figure 15.6. Output when camera is properly calibrated.

## 2) Example of failure

This is an example in which the calibration is incorrect.

Look at the output in the bottom left: clearly the perspective is distorted, and there is no way for the robot to localize given the perspective points.



Figure 15.7. Output when camera not properly calibrated.

## 15.6. The importance of validation

Validation is useful because otherwise it is hard to detect wrong calibrations.

For example, in 2017, a bug in the calibration made about 5 percent of the calibrations useless ([Figure 15.8](#)), and people didn't notice for weeks (!).



Figure 15.8. In 2017, a bug in the calibration made about 5 percent of the calibrations useless.

## UNIT I-16

### Taking and verifying a log

#### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** [Unit I-10 - Reading from the camera](#)

**Requires:** [Unit I-9 - Software setup and RC remote control](#)

**Results:** A verified log.

#### 16.1. Preparation

**Note:** it is recommended that you log to your USB and not to your SD card.

- To mount your USB see [Mounting USB drives \(master\)](#).

#### 16.2. Run something on the Duckiebot

For example, if you want to drive the robot around and collect image data you could

run:

 \$ make demo-joystick-camera

But anything could do.

### 16.3. View images on the laptop

Run on the laptop:

 \$ cd *Duckietown root*  
\$ source environment.sh  
\$ source set\_ros\_master.sh *robot name*  
\$ rqt\_image\_view

and verify that indeed your camera is streaming imagery.

### 16.4. Record the log

#### 1) Option: Full Logging

---

To log everything that is being published, on the Duckiebot in a new terminal (See [Byobu \(master\)](#)):

 \$ make log-full

where here we are assuming that you are logging to the USB and have followed [Mounting USB drives \(master\)](#).

#### 2) Option: Log Minimal

---

To log only the imagery, camera\_info, the control commands and a few other essential things, on the Duckiebot in a new terminal (See [Byobu \(master\)](#)):

 \$ make log-minimal

where here we are assuming that you are logging to the USB and have followed [Mounting USB drives \(master\)](#).

### 16.5. Verify a log

On the Duckiebot run:



```
$ rosbag info FULL_PATH_TO_BAG --freq
```

Then:

- verify that the “duration” of the log seems “reasonable” - it’s about as long as you ran the log command for
- verify that the “size” of the log seems “reasonable” - the log size should grow at about 220MB/min
- verify in the output that your camera was publishing very close to 30.0Hz and verify that your joysick was publishing at a rate between 3Hz and 6Hz.

**TODO:** More complex log verification methods.

## UNIT I-17

# Troubleshooting



### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** The Raspberry Pi of the Duckiebot is connected to the battery.

**Requires:** The Stepper Motor HAT is connected to the battery.

**Requires:** You have a problem!

### 17.1. The Raspberry Pi does not turn ON

**Symptom:** The red LED on the Raspberry Pi is OFF

**Resolution:** Press the button on the side of the battery ([Figure 17.1](#)).



Figure 17.1. The power button on the RAVPower Battery.

### 17.2. I cannot access my Duckiebot via SSH

**Symptom:** When I run `ssh robot_name.local` I get the error `ssh: Could not resolve hostname robot_name.local`.

**Resolution:** Make sure that your Duckiebot is ON. Connect it to a monitor, a mouse and a keyboard. Run the command



```
$ sudo service avahi-daemon status
```

You should get something like the following

```
• avahi-daemon.service - Avahi mDNS/DNS-SD Stack
  Loaded: loaded (/lib/systemd/system/avahi-daemon.service; enabled; vendor preset: enabled)
  Active: active (running) since Sun 2017-10-22 00:07:53 CEST; 1 day 3h ago
    Main PID: 699 (avahi-daemon)
   Status: "avahi-daemon 0.6.32-rc starting up."
     CGroup: /system.slice/avahi-daemon.service
             ├─699 avahi-daemon: running [robot name in avahi].local
             └─727 avahi-daemon: chroot helpe
```

Avahi is the module that in Ubuntu implements the mDNS responder. The mDNS responder is responsible for advertising the hostname of the Duckiebot on the network so that everybody else within the same network can run the command `ping robot name.local` and reach your Duckiebot. Focus on the line containing the hostname published by the `avahi-daemon` on the network (i.e., the line that contains `robot name in avahi.local`). If `robot name in avahi` matches the `robot name`, go to the next Resolution point. If `robot name in avahi` has the form `robot name-xx`, where `xx` can be any number, modify the file `/etc/avahi/avahi-daemon.conf` as shown below.

Identify the line

```
use-ipv6=yes
```

and change it to

```
use-ipv6=no
```

Identify the line

```
#publish-aaaa-on-ipv4=yes
```

and change it to

```
publish-aaaa-on-ipv4=no
```

Restart Avahi by running the command

 \$ sudo service avahi-daemon restart

## 17.3. The Duckiebot does not move



**Symptom:** I can SSH into my Duckiebot and run the joystick demo but the joystick does not move the wheels.

**Resolution:** Press the button on the side of the battery ([Figure 17.1](#)).

**Resolution:** Check that the red indicator on the joystick stopped blinking.



(a) Bad joystick status

(b) Bad joystick status

Figure 17.2

**Symptom:** The joystick is connected (as shown in [Figure 17.2b - Bad joystick status](#)) but the Duckiebot still does not move.

**Resolution:** Make sure that the controller is connected to the Duckiebot and that the OS receives the data from it. Run



```
$ jstest /dev/input/js0
```

If you receive the error

```
jstest: No such file or directory
```

it means that the USB receiver is not connected to the Raspberry Pi or is broken. If the command above shows something like the following

```
Driver version is 2.1.0.  
Joystick (ShanWan PC/PS3/Android) has 8 axes (X, Y, Z, Rz, Gas, Brake, Hat0X, Hat0Y)  
and 15 buttons (BtnX, BtnY, BtnZ, BtnTL, BtnTR, BtnTL2, BtnTR2, BtnSelect, BtnStart,  
BtnMode, BtnThumbL, BtnThumbR, ?, ?, ?).  
Testing ... (interrupt to exit)  
Axes: 0: 0 1: 0 2: 0 3: 0 4:-32767 5:-32767 6: 0 7: 0  
Buttons: 0:off 1:off 2:off 3:off 4:off 5:off 6:off 7:off 8:off 9:off 10:off  
11:off 12:off 13:off 14:off
```

it means that the USB receiver is connected to the Raspberry Pi. Leave the terminal above open and use the joystick to command the Duckiebot. If you observe that the numbers shown in the terminal change according to the commands sent through the joystick than the problem is in ROS. Make sure that the joystick demo is launched. Restart the Duckiebot if needed and try again.

If the numbers do not change while using the joystick then follow this guide at the next Resolution point.

**Resolution:** The controller might be connected to another Duckiebot nearby. Turn

off the controller, go to a room with no other Duckiebots around and turn the controller back on. Retry.

## PART J

# Duckiebot - DB17-1c configurations

This section contains the acquisition, assembly and setup instructions for the DB17-1c configurations. These instructions are separate from the rest as they approximately match the b releases of the Fall 2017 Duckietown Engineering Co. branches.

To understand how configurations and releases are defined, refer to: [Unit I-1 - Duckiebot configurations](#).

### UNIT J-1

## Acquiring the parts (DB17-1c)

Upgrading your DB17 (DB17-wjd) configuration to DB17-1c (DB17-wjd1c) starts here, with purchasing the necessary components. We provide a link to all bits and pieces that are needed to build a DB17-1c Duckiebot, along with their price tag. If you are wondering what is the difference between different Duckiebot configurations, read [Unit I-1 - Duckiebot configurations](#).

In general, keep in mind that:

- The links might expire, or the prices might vary.
- Shipping times and fees vary, and are not included in the prices shown below.
- Buying the parts for more than one Duckiebot makes each one cheaper than buying only one.
- A few components in this configuration are custom designed, and might be trickier to obtain.

### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** A Duckiebot in DB17-wjd configuration.

**Requires:** Cost: USD 77 + Bumpers manufacturing solution

**Requires:** Time: 21 Days (LED board manufacturing and shipping time)

**Results:** A kit of parts ready to be assembled in a DB17-1c configuration Duckiebot.

**Next:** After receiving these components, you are ready to do some [soldering](#) before [assembling](#) your DB17-1c Duckiebot.

### 1.1. Bill of materials

TABLE 1.1. BILL OF MATERIALS

<u>LEDs</u> (DB17-1)	USD 10
<u>LED HAT</u> (DB17-1)	USD 28.20 for 3 pieces
<u>Power Cable</u> (DB17-1)	USD 7.80
<u>20 Female-Female Jumper Wires (300mm)</u> (DB17-1)	USD 8
<u>Male-Male Jumper Wire (150mm)</u> (DB17-1)	USD 1.95
<u>PWM/Servo HAT</u> (DB17-1)	USD 17.50
<u>Bumpers</u>	TBD (custom made)
<u>40 pin female header</u> (DB17-1)	USD 1.50
<u>5 4 pin female header</u> (DB17-1)	USD 0.60/piece
<u>2 16 pin male header</u> (DB17-1)	USD 0.61/piece
<u>12 pin male header</u> (DB17-1)	USD 0.48/piece
<u>3 pin male header</u> (DB17-1)	USD 0.10/piece
<u>2 pin female shunt jumper</u> (DB17-1)	USD 2/piece
<u>5 200 Ohm resistors</u> (DB17-1)	USD 0.10/piece
<u>10 130 Ohm resistors</u> (DB17-1)	USD 0.10/piece
<u>Caster</u> (DB17-c)	USD 6.55/4 pieces
<u>4 Standoffs (M3.5 12mm F-F)</u> (DB17-c)	USD 0.63/piece
<u>8 Screws (M3.5x8mm)</u> (DB17-c)	USD 4.58/100 pieces
<u>8 Split washer lock</u> (DB17-c)	USD 1.59/100 pieces
Total for DB17-wjd configuration	USD 212
Total for DB17-1c components	USD 77 + Bumpers
Total for DB17-wjd1c configuration	USD 299+Bumpers

## 1.2. LEDs

The Duckiebot is equipped with 5 RGB LEDs ([Figure 1.1](#)). LEDs can be used to signal to other Duckiebots, or just make *fancy* patterns.

The pack of LEDs linked in the table above holds 10 LEDs, enough for two Duckiebots.



Figure 1.1. The RGB LEDs

---

### 1) LED HAT

The LED HAT ([Figure 1.2](#)) provides an interface for our RGB LEDs and the computational stack. This board is a daughterboard for the Adafruit 16-Channel PWM/Servo HAT, and enables connection with additional gadgets such as [ADS1015 12 Bit 4 Channel ADC](#), [Monochrome 128x32 I2C OLED graphic display](#), and [Adafruit 9-DOF IMU](#).

[Breakout - L3GD20H+LSM303](#). This item will require [soldering](#).

This board is custom degined and can only be ordered in minimum runs of 3 pieces. The price scales down quickly with quantity, and lead times may be significant, so it is better to buy these boards in bulk.



Figure 1.2. The LED HAT

## 2) PWM/Servo HAT

---

The PWM/Servo HAT ([Figure 1.3](#)) mates to the LED HAT and provides the signals to control the LEDs, without taking computational resources away from the Raspberry Pi itself. This item will require [soldering](#).



Figure 1.3. The PWM-Servo HAT

## 3) Power Cable

---

To power the PWM/Servo HAT from the battery, we use a short (30cm) angled male USB-A to 5.5/2.1mm DC power jack cable ([Figure 1.4](#)).



Figure 1.4. The 30cm angled USB to 5.5/2.1mm power jack cable.

## 4) Male-Male Jumper Wires

---

The Duckiebot needs one male-male jumper wire ([Figure 1.5](#)) to power the DC Stepper Motor HAT from the PWM/Servo HAT.



Figure 1.5. Premier Male-Male Jumper Wires

### 5) Female-Female Jumper Wires

---

20 Female-Female Jumper Wires ([Figure 1.6](#)) are necessary to connect 5 LEDs to the LED HAT.



Figure 1.6. Premier Female-Female Jumper Wires

### 1.3. Bumpers

These bumpers are designed to keep the LEDs in place and are therefore used only in configuration DB17-1. They are custom designed parts, so they must be produced and cannot be bought. We used laser cutting facilities.



Figure 1.7. The Bumpers

### 1.4. Headers, resistors and jumper

Upgrading DB17 to DB17-1 requires several electrical bits: 5 of 4 pin female header, 2 of 16 pin male headers, 1 of 12 pin male header, 1 of 3 pin male header, 1 of 2 pin female

shunt jumper, 5 of 200 Ohm resistors and finally 10 of 130 Ohm resistors.

These items require [soldering](#).



Figure 1.8. The Headers



Figure 1.9. The Resistors

## 1.5. Caster (DB17-c)

The caster ([Figure 1.10](#)) is an [DB17-c](#) component that substitutes the steel omnidirectional wheel that comes in the Magician Chassis package. Although the caster is not essential, it provides smoother operations and overall enhanced Duckiebot performance.



Figure 1.10. The caster wheel

To assemble the caster at the right height we will need to purchase:

- 4 standoffs (M3 12mm F-F) ([Figure 1.11a - Standoffs for caster wheel.](#)),
- 8 screws (M3x8mm) ([Figure 1.11b - Screws for caster wheel.](#)), and
- 8 split lock washers ([Figure 1.11c - Split lock washers for caster wheel.](#)).



(a) Standoffs for caster wheel.



(b) Screws for caster wheel.



(c) Split lock washers for caster wheel.

Figure 1.11. Mechanical bits to assemble the caster wheel.

**TODO:** missing figures, update caster bits figures

## UNIT J-2

### Soldering boards (DB17-1) [draft]



This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT J-3

### Assembling the Duckiebot (DB17-1c) [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT J-4

### Bumper Assembly [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT J-5

### DB17-1 setup [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## PART K

### Appendix

## UNIT K-1

### Bibliography

[2] [Liam Paull](#), [Jacopo Tani](#), Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Daniel Hoehener, Shih-Yuan Liu, Michael Novitzky, Igor Franzoni Okuya-

- ma, Jason Pazis, Guy Rosman, Valerio Varricchio, Hsueh-Cheng Wang, Dmitry Yershov, Hang Zhao, Michael Benjamin, [Christopher Carr](#), [Maria Zuber](#), [Sertac Karaman](#), [Emilio Frazzoli](#), [Domitilla Del Vecchio](#), [Daniela Rus](#), [Jonathan How](#), [John Leonard](#), and Andrea Censi. Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In *IEEE International Conference on Robotics and Automation (ICRA)*. Singapore, May 2017.  [pdf](#)
- [3] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
- [1] [Jacopo Tani](#), [Liam Paull](#), [Maria Zuber](#), [Daniela Rus](#), [Jonathan How](#), [John Leonard](#), and Andrea Censi. Duckietown: an innovative way to teach autonomy. In *EduRobotics 2016*. Athens, Greece, December 2016.  [pdf](#)
- [6] Tosini, G., Ferguson, I., Tsubota, K. *Effects of blue light on the circadian system and eye physiology*. *Molecular Vision*, 22, 61–72, 2016 ([online](#)).
- [7] Albert Einstein. Zur Elektrodynamik bewegter Körper. (German) On the electrodynamics of moving bodies. *Annalen der Physik*, 322(10):891–921, 1905. [DOI](#)
- [15] Ivan Savov. Linear algebra explained in four pages. [https://minireference.com/static/tutorials/linear\\_algebra\\_in\\_4\\_pages.pdf](https://minireference.com/static/tutorials/linear_algebra_in_4_pages.pdf), 2017. Online; accessed 23 August 2017.
- [14] Kaare Brandt Petersen and Michael Syskind Pedersen. The matrix cookbook. [.pdf](#)
- [17] Matrix (mathematics). Matrix (mathematics) — Wikipedia, the free encyclopedia, 2017. Online; accessed 2-September-2017. [www:](#)
- [18] Peter D. Lax. *Functional Analysis*. Wiley Interscience, New York, NY, USA, 2002.
- [19] Athanasios Papoulis and S. Unnikrishna Pillai. *Probability, Random Variables and Stochastic Processes*. McGraw Hill, fourth edition, 2002.
- [20] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.
- [21] H. Choset, W. Burgard, S. Hutchinson, G. Kantor, L. E. Kavraki, K. Lynch, and S. Thrun. *Principles of robot motion: Theory, algorithms, and implementation*. MIT Press, June 2005.
- [22] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006.
- [23] M. Miskowicz. *Event-Based Control and Signal Processing*. Embedded Systems. CRC Press, 2015. [http](#)
- [24] Karl J. Åström. *Event Based Control*, pages 127–147. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. [DOI](#) [http](#)
- [25] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
- [26] Rajit Manohar and K Mani Chandy. Delta-dataflow networks for event stream processing. pages 1–6, June 2010.
- [27] Romano M DeSantis. Modeling and path-tracking control of a mobile wheeled robot with a differential drive. *Robotica*, 13(4):401–410, 1995.
- [28] Richard Szeliski. *Computer Vision: Algorithms an Applications*. Springer, 2010. [http](#)
- [29] David Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 2011.
- [30] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition, 2004.

- [31] Jacopo Tani. ETHZ AMOD 2017 lecture: Modeling of a differential drive vehicle, 2017.
- [32] Jacopo Tani. ETHZ AMOD 2017 lecture: Odometry calibration, 2017.
- [34] Censi Tani. Project pitches. Slides, 10 2017.
- [35] Davide Scaramuzza. Design and realization of a stereoscopic vision system for robotics, with applications to tracking of moving objects and self-localization. Master thesis, department of electronic and information engineering, University of Perugia, Perugia, Italy, 2005. [.pdf](#)
- [36] Gang Yi Jiang. Lane and obstacle detection based on fast inverse perspective mapping algorithm. In *IEEE Xplore, Conference: Systems, Man, and Cybernetics, 2000 IEEE International Conference on, Volume: 4*, Febuary 2000. [.http](#)
- [37] Alessandra Fascioli Massimo Bertozzi, Alberto Broggi. Stereo inverse perspective mapping: Theory and applications. *Image and Vision Computing 16 (1998)*, 1997. [.pdf](#)
- [38] Kesheng Wu. Optimizing connected component labeling algorithms, 2005. [.http](#)
- [39] Richard Fitzpatrick. Moment of inertia tensor, 03 2011. [.html](#)
- [40] Google offers raspberry pi owners this new ai vision kit to spot cats, people, emotions, 12 2017. [.http](#)
- [41] Davide Scaramuzza. Lecture: Vision algorithms for mobile robotics, 2017. [.html](#)
- [43] Davide Scaramuzza. Towards robust and safe autonomous drones, September 2015. [.http](#)
- [44] Rasmussen. Presentation on theme: "computer vision : Cisc 4/689". [.http](#)
- [45] Inverse perspective mapping, September 2012. [.http](#)
- [47] VIKAS GUPTA. Color spaces in opencv (c++ / python), May 2017. [.http](#)
- [48] Convert from hsv to rgb color space, 2018. [.http](#)
- [50] Michel Alves. About perception and hue histograms in hsv space, July 2015. [.http](#)
- [55] CHENG. Slides from the lecture "computer graphics".
- [51] Hsl and hsv, 12 2017. [.http](#)

## UNIT K-2

### Last modified

- Mon, Mar 12: [Unit F-15 - Lane control with supervised learning \(master\)](#) (tanij)
- Mon, Mar 12: [Unit F-12 - Coordination \(master\)](#) (tanij)
- Mon, Mar 12: [Unit F-3 - Anti-instagram \(Lane following\) \(master\)](#) (tanij)
- Mon, Mar 12: [Unit L-36 - Supervised Learning: final report \[draft\]](#) (tanij)
- Mon, Mar 12: [Unit F-2 - Demo system ID \(master\)](#) (tanij)
- Mon, Mar 12: [Unit F-5 - Indefinite Navigation \(master\)](#) (tanij)
- Mon, Mar 12: [Unit F-6 - Intersection Navigation \(master\)](#) (tanij)
- Mon, Mar 12: [Unit F-7 - Implicit Coordination \(master\)](#) (tanij)
- Mon, Mar 12: [Unit F-8 - Explicit Coordination \(master\)](#) (tanij)
- Mon, Mar 12: [Unit F-9 - Demo Saviors \(master\)](#) (tanij)
- Mon, Mar 12: [Unit L-44 - Anti Instagram ReadMe](#) (tanij)
- Mon, Mar 12: [Unit L-43 - Anti-Instagram: final report](#) (tanij)
- Sun, Mar 11: [Unit L-31 - Fleet Planning: Final Report](#) (tanij)
- Sun, Mar 11: [Unit L-32 - Transferred Lane following](#) (tanij)

- Sun, Mar 11: [Unit L-33 - Transfer Learning in Robotics](#) (tanij)
- Sun, Mar 11: [Unit L-42 - Anti-Instagram: intermediate report \[draft\]](#) (tanij)
- Sun, Mar 11: [Unit L-49 - Transfer learning: preliminary report](#) (tanij)
- Sun, Mar 11: [Unit L-35 - Supervised learning: intermediate report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-34 - Supervised learning: preliminary report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-9 - System identification: final report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-30 - Fleet Planning: Intermediate Report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-18 - Navigators: final report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-17 - Navigators: intermediate report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-15 - The Saviors: Final Report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-13 - The Saviors: preliminary report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-23 - Explicit Coordination: intermediate report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-48 - Demo instructions Fleet Communications \[draft\]](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-47 - Distributed Estimation: final report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-46 - Distributed Estimation: intermediate report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit F-11 - Fleet planning Demo \(master\)](#) (tanij)
- Sun, Mar 11: [Unit L-24 - Explicit coordination: final report \[draft\]](#) (tanij)
- Sun, Mar 11: [Unit F-10 - Parking demo instructions \(master\)](#) (Jacopo Tani)
- Sun, Mar 11: [Unit O-5 - Package easy\\_logs \(master\)](#) (Andrea Censi)
- Sun, Mar 11: [Unit L-21 - Parking: final report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-27 - Implicit Coordination: final report \[draft\]](#) (Jacopo Tani)
- Sun, Mar 11: [Unit L-5 - The Heroes - System Architecture: final report](#) (Jacopo Tani)
- Sun, Mar 11: [Unit F-4 - Lane following \(master\)](#) (tanij)
- Sun, Mar 11: [Unit L-4 - The Heroes quests: preliminary report](#) (Jacopo Tani)
- Wed, Mar 07: [Unit H-3 - How to install PyTorch on the Duckiebot](#) (Andrea F. Daniele)
- Sun, Mar 04: [Unit L-12 - The Controllers: final report](#) (tanij)
- Sun, Mar 04: [Unit L-41 - Anti-Instagram: preliminary design](#) (tanij)
- Sun, Mar 04: [Unit I-2 - Acquiring the parts \(DB17-jwd\)](#) (tanij)
- Sun, Mar 04: [Unit G-25 - Clustering methods \(master\)](#) (tanij)
- Sun, Mar 04: [Unit H-3 - Anti-Instagram \(master\)](#) (tanij)
- Fri, Mar 02: [Unit J-23 - Atom \(master\)](#) (Julien Kindle)
- Wed, Feb 28: [Unit L-45 - PDD - Distributed Estimation](#) (tanij)
- Wed, Feb 28: [Unit L-16 - Navigators: preliminary report](#) (tanij)
- Wed, Feb 28: [Unit L-7 - System Identification: preliminary report](#) (tanij)
- Wed, Feb 28: [Unit L-8 - System Identification: intermediate Report](#) (tanij)
- Wed, Feb 28: [Unit L-19 - Parking: preliminary report](#) (tanij)
- Tue, Feb 27: [Unit L-10 - The Controllers: preliminary report](#) (tanij)
- Tue, Feb 27: [Unit L-22 - Explicit Coordination: preliminary report](#) (tanij)
- Tue, Feb 27: [Unit L-25 - Implicit Coordination: preliminary report](#) (tanij)
- Tue, Feb 27: [Unit L-26 - Implicit Coordination: intermediate report](#) (tanij)
- Tue, Feb 27: [Unit F-14 - Follow Leader \(master\)](#) (Kornel Eggerschwiler)
- Sun, Feb 25: [Unit L-29 - Fleet Planning: Preliminary Report](#) (tanij)
- Sun, Feb 25: [Unit G-27 - Histogram Equalization \(master\)](#) (czuidema)
- Sun, Feb 25: [Unit G-26 - Convex optimization \(master\)](#) (tanij)
- Wed, Feb 21: [Unit L-20 - Parking: intermediate report](#) (BrettRStephens)
- Tue, Feb 20: [Unit I-9 - Software setup and RC remote control](#) (Florian Golemo)
- Fri, Feb 16: [Unit C-16 - Wheel calibration \(master\)](#) (Manfred Diaz)
- Tue, Feb 13: [Unit I-8 - Networking aka the hardest part](#) (Julien Kindle)

- Fri, Feb 09: [Unit L-3 - Group name: final report \[draft\]](#) (Jacopo Tani)
- Fri, Jan 19: [Unit I-14 - Camera calibration](#) (Manfred Diaz)
- Sun, Jan 14: [Unit Q-2 - Package complete\\_image\\_pipeline \(master\)](#) (Andrea Censi)
- Mon, Jan 01: [Section 3.1 - Package information \(master\)](#) (sonja)
- Sun, Dec 31: [Unit O-6 - Package easy\\_node \(master\)](#) (Andrea Censi)
- Sat, Dec 30: [Unit D-2 - Basic Markduck guide](#) (Andrea Censi)
- Sat, Dec 30: [Unit D-2 - Basic Markduck guide](#) (Andrea Censi)
- Sat, Dec 30: [Unit D-2 - Duckietown Appearance Specification \(master\)](#) (Andrea Censi)
- Sat, Dec 30: [Unit R-3 - Package fsm \(master\)](#) (sonja)
- Fri, Dec 29: [Unit K-6 - Minimal ROS node - pkg\\_name \(master\)](#) (Theodore Koutros)
- Mon, Dec 18: [Unit D-1 - Contributing to the documentation](#) (Jacopo Tani)

Sat, Dec 16: [\(master\)](#) + syntax error

This link text is empty:

```
<a class='number_name link-to-master' href='http://purl.org/dth/booktitle'></a>
```

Note that the syntax for links in Markdown is

[link text](URL)

For the internal links (where URL starts with "#"), then the documentation system can fill in the title automatically, leading to the format:

[](#other-section)

However, this does not work for external sites, such as:

[](http://purl.org/dth/booktitle)

So, you need to provide some text, such as:

- [\[this useful website\]\(http://purl.org/dth/booktitle\)](#) (Andrea Censi)

- Sat, Dec 16: [Unit F-13 - Parallel Autonomy \(master\)](#) (Jacopo Tani)
- Mon, Dec 11: [Unit L-39 - Visual Odometry Project](#) (tanij)
- Mon, Dec 11: [Unit L-40 - Deep Visual Odometry ROS Package](#) (tanij)
- Thu, Dec 07: [Unit L-11 - The Controllers: Intermediate Report](#) (lapandic)
- Wed, Dec 06: [Unit L-37 - PDD Neural Slam](#) (lapandic)
- Wed, Dec 06: [Unit J-33 - Movidius Neural Compute Stick Install \(master\)](#) (Michael Noukhovitch)
- Wed, Dec 06: [Unit L-14 - The Saviors: intermediate report](#) (tanij)
- Tue, Dec 05: [Part X - Packages - Deep Learning \(master\)](#) (liampaull)
- Tue, Dec 05: [Section 11.7 - Sep 28: some announcements](#) (ValentinaCavinato)
- Fri, Dec 01: [Unit C-7 - Reproducing the image \(master\)](#) (Elias Zoller)
- Tue, Nov 28: [Unit I-1 - Duckiebot configurations](#) (Andrea F. Daniele)
- Tue, Nov 28: [Unit L-2 - Group name: intermediate report](#) (Jacopo Tani)
- Mon, Nov 27: [Unit I-5 - Assembling the Duckiebot \(DB17-wid TTIC\)](#) (Andrea F. Daniele)
- Mon, Nov 27: [Unit C-3 - Duckietown classes \[draft\]](#) (Andrea F Daniele)
- Mon, Nov 27: [Part I - Operation manual - Duckiebot](#) (Andrea F. Daniele)
- Mon, Nov 27: [Unit C-3 - Soldering boards \(DB17\) \(master\)](#) (Andrea F. Daniele)
- Mon, Nov 27: [Unit I-3 - Preparing the power cable \(DB17\)](#) (Andrea F. Daniele)
- Mon, Nov 27: [Unit I-4 - Assembling the Duckiebot \(DB17-jwd\)](#) (Andrea F. Daniele)
- Mon, Nov 27: [Part D - Operation manual - Duckietown \(master\)](#) (Andrea F. Daniele)

- Mon, Nov 27: [Part J - Duckiebot - DB17-1c configurations](#) (Andrea F. Daniele)
- Mon, Nov 27: [Unit J-1 - Acquiring the parts \(DB17-1c\)](#) (Andrea F. Daniele)
- Mon, Nov 27: [Unit J-2 - Soldering boards \(DB17-1\) \[draft\]](#) (Andrea F. Daniele)
- Mon, Nov 27: [Unit J-3 - Assembling the Duckiebot \(DB17-1c\) \[draft\]](#) (Andrea F. Daniele)
- Mon, Nov 27: [Unit J-5 - DB17-1 setup \[draft\]](#) (Andrea F. Daniele)
- Fri, Nov 24: [Unit J-32 - How to install Caffe and Tensorflow on the Duckiebot \(master\)](#) (Nick Wang)
- Thu, Nov 23: [Unit L-1 - Group name: preliminary design document](#) (Andrea Censi)

## PART L

# Fall 2017 projects



Welcome to the Fall 2017 projects.

## 0.1. Instructions for using the template

1. Make a copy of the template `10_templates` folder and paste it inside `/atoms_85_fall2017_projects`.
2. Rename the folder to the next available integer followed by the short group name. E.g.: `10_templates` becomes `11_first_group_name` for the first group, then `12_second_group_name` for the second, and so forth.
3. Edit the `10-preliminary-design-document-template` file name by substituting `template` with `group-name`
4. Open the preliminary design document and personalize the template to your group.

**Note:** All groups have got their unique ID number and folders are renamed according to the following table. You are allowed and encouraged to use short names. Please merge from the master. New pull requests conflicting to this table will be rejected.

## 0.2. Group names and ID numbers

ID	Group name	Short name
11	The Heroes	heroes
12	The Architects	smart-city
13	The Identifiers	sysid
14	The Controllers	controllers
15	The Saviors	saviors
16	The Navigators	navigators
17	Parking	parking
18	The Coordinators	explicit-coord
19	Formations and implicit coordination	implicit-coord
20	Distributed estimation	distributed-est
21	Fleet-level planning	fleet-planning
22	Transfer-learning	transfer-learning
23	Supervised learning	super-learning

ID	Group name	Short name
24	Neural-slam	neural-slam
25	Visual-odometry	visual-odometry
26	Single-slam	single-slam
27	Anti-instagram	anti-instagram

## UNIT L-1

### Group name: preliminary design document

#### 1.1. Part 1: Mission and scope

##### 1) Mission statement

---

Instructions: What is the overarching mission of this team? You should write in one sentence.

Instructions: What is the need that is being addressed? Do not focus on technical specifics yet.

##### 2) Motto

---

Instructions: Your rallying cry into battle. Traditionally, Duckietown uses Latin mottos. If you don't speak Latin, please contact Jacopo Tani to have your motto translated into latin.

QUIDQUID LATINE DICTUM SIT, ALTUM VIDETUR

(Anything that is said in Latin sounds important)

##### 3) Project scope

---

Instructions: Are you going to rewrite Duckietown from scratch? Probably not. You need to decide what are the boundaries in which you want to move.

*What is in scope:*

Instructions: What do you consider in scope? (e.g. having a different calibration pattern)

*What is out of scope:*

Instructions: What do you consider out of scope? (e.g. hardware modifications)

*Stakeholders:*

Instructions: What other pieces of Duckietown interact with your piece?

Instructions: List here the teams.

## 1.2. Part 2: Definition of the problem

### 1) Problem statement

---

Time to define the particular problem that you choose to solve.

Suppose that we need to free our prince/princess from a dragon. So the mission is clear:

Mission = we must recover the prince/princess.

Now, are we going to battle the dragon, or use diplomacy?

If the first, then the problem statement becomes:

Problem statement = We need to slain a dragon.

Otherwise:

Problem statement = We need to convince the dragon to give us the prince/princess.

Suppose we choose to slain the dragon.

### 2) Assumptions

---

At this point, you might need to make some assumptions before proceeding.

- Does the dragon breath fire?
- What color is the dragon? Does the color matter?
- How big is this dragon, exactly?

### 3) Approach

---

All right. We are going to kill the dragon. How? Are we going to battle the dragon? Are we trying to poison him? Are we going to hire an army of mercenaries to kill the dragon for us?

### 4) Functionality-resources trade-offs

---

The space of possible implementations / battle plans is infinite. We need to understand what will be the trade-offs.

### 5) Functionality provided

---

How do you measure the functionality (what this module provides)? What are the “metrics”?

*example*

numbers of dragons killed per hour

Note that this is already tricky. In fact, the above is not a good metric. Maybe we kill the dragon with an explosion, and also the prince/princess is killed. A better one might be:

*example*

numbers of royals freed per hour

*example*

probability of freeing a royal per attempt

It works better if you can choose the quantities so that functionality is something that you maximize to maximize. (so that you can “maximize performance”, and “minimize resources”).

## 6) Resources required / dependencies / costs

---

How do you measure the resources (what this module requires)?

*example*

numbers of knights to hire

*example*

total salary for the mercenaries.

*example*

liters of poison to buy.

*example*

average duration of the battle.

It works better if you think of these resources as something to minimize.

## 7) Performance measurement

---

How would you measure the performance/resources above? If you don't know how to measure it, it is not a good quantity to choose.

*example*

we dress up Brian as a Dragon and see how long it takes to kill him.

### 1.3. Part 3: Preliminary design

#### 1) Modules

---

Can we decompose the problem?

Can you break up the solution in modules?

Note here we talk about logical modules, not the physical architecture (ROS nodes).

#### 2) Interfaces

---

For each module, what is the input, and what is the output?

How is the data represented?

Note we are not talking about ROS messages vs services vs UDP vs TCP etc.

#### 3) Preliminary plan of deliverables

---

What needs to be designed?

What needs to be implemented?

What already exists and needs to be revised?

#### 4) Specifications

---

Do you need to revise the Duckietown specification?

#### 5) Software modules

---

Here, be specific about the software: is it a ROS node, a Python library, a cloud service, a batch script?

#### 6) Infrastructure modules

---

Some of the modules have been designated as infrastructure

### 1.4. Part 4: Project planning

Now, make a plan for the next phase.

## 1) Data collection

---

What data do you need to collect?

## 2) Data annotation

---

Do you have data that needs to be annotated? What would the annotations be?

*Relevant Duckietown resources to investigate:*

List here Duckietown packages, slides, previous projects that are relevant to your quest

*Other relevant resources to investigate:*

List papers, open source code, software libraries, that could be relevant in your quest.

## 3) Risk analysis

---

What could go wrong?

How to mitigate the risks?

## UNIT L-2

### Group name: intermediate report

*It's time to commit on what you are building, and to make sure that it fits with everything else.*

This consists of 3 parts:

- Part 1: System interfaces: Does your piece fit with everything else? You will have to convince both system architect and software architect and they must sign-off on this.
- Part 2: Demo and evaluation plan: Do you have a credible plan for evaluating what you are building? You will have to convince the VPs of Safety and they must sign-off on this.
- Part 3: Data collection, annotation, and analysis: Do you have a credible plan for collecting, annotating and analyzing the data? You will have to convince the data czars and they must sign-off on this.

TABLE 2.1. INTERMEDIATE REPORT SUPERVISORS

System Architects	Sonja Brits, Andrea Censi
Software Architects	Breandan Considine, Liam Paull
Vice President of Safety	Miguel de la Iglesia, Jacopo Tani
Data Czars	Manfred Diaz, Jonathan Aresenault

## 2.1. Part 1: System interfaces

Please note that for this part it is necessary for the system architect and software architect to check off before you submit it. Also note that they are busy people, so it's up to you to coordinate to make sure you get this part right and in time.

### 1) Logical architecture

---

- Please describe in detail what the desired functionality will be. What will happen when we click “start”?
- Please describe for each quantity, what are reasonable target values. (The system architect will verify that these need to be coherent with others.)
- Please describe any assumption you might have about the other modules, that must be verified for you to provide the functionality above.

### 2) Software architecture

---

- Please describe the list of nodes that you are developing or modifying.
- For each node, list the published and subscribed topics.
- For each subscribed topic, describe the assumption about the latency introduced by the previous modules.
- For each published topic, describe the maximum latency that you will introduce.

## 2.2. Part 2: Demo and evaluation plan

*Please note that for this part it is necessary for the VPs for Safety to check off before you submit it. Also note that they are busy people, so it's up to you to coordinate to make sure you get this part right and in time.*

### 1) Demo plan

---

The demo is a short activity that is used to show the desired functionality, and in particular the difference between how it worked before (or not worked) and how it works now after you have done your development.

It should take a few minutes maximum for setup and running the demo.

- How do you envision the demo?
- What hardware components do you need?

### 2) Plan for formal performance evaluation

---

- How do you envision the performance evaluation? Is it experiments? Log analysis?

In contrast with the demo, the formal performance evaluation can take more than a few minutes.

Ideally it should be possible to do this without human intervention, or with minimal human intervention, for both running the demo and checking the results.

## 2.3. Part 3: Data collection, annotation, and analysis

*Please note that for this part it is necessary for the Data Czars to check off before you submit it. Also note that they are busy people, so it's up to you to coordinate to make sure you get this part right and in time.*

### 1) Collection

---

- How much data do you need?
- How are the logs to be taken? (Manually, autonomously, etc.)

Describe any other special arrangements.

- Do you need extra help in collecting the data from the other teams?

### 2) Annotation

---

- Do you need to annotate the data?
- At this point, you should have you tried using [thehive.ai](#) to do it. Did you?
- Are you sure they can do the annotations that you want?

### 3) Analysis

---

- Do you need to write some software to analyze the annotations?
- Are you planning for it?

## UNIT L-3

### Group name: final report [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT L-4

### The Heroes quests: preliminary report

The “Heroes” team is a special task force with the responsibility to make sure that “everything works” and create a smooth experience for the rest of the teams, in terms of developing own projects, integration with other teams and documentation. Apart from that, each of the heroes will also have their own individual quest...

#### 4.1. Motto

E PLURIBUS UNUM

(From many, unity)

## 4.2. Overview

### 1) Responsibility

---

The system architect is ultimately responsible:

- The logical architecture
  - Decomposition in modules
  - Who knows what
  - Who says what to whom
- Definition of performance metrics
- Definition of contracts (bounds on metrics)

### 2) Philosophy

---

- Look over Duckie teams and remind them of the greater goals of Duckietown.
- Improve Duckietown by using a higher dimensional view.

### 3) Quests

---

The system architect project can be split into two quests:

1. Ensure that the development of the system goes smoothly (wooden spoon)
2. Develop a framework/tool to formally describe (and later optimize) the system (bronze, silver and gold)

The two quests and their respective descriptions will be explained separately in this document.

## 4.3. Quest 1

With many teams working on many different parts of the system, chaos is inevitable (without divine intervention). Quest 1 is to minimise the chaos by acting as system-level watchdog; spotting and addressing interface, contract and dependency issues between the teams.

## 4.4. Quest 1, Part 1: Mission and scope

### 1) Mission statement

---

Ensure that the development and integration of the projects into the system goes smoothly and that the resulting system makes sense, and is useful for future duckierations (duckie + generations).

## 2) Project scope

---

*What is in scope:*

- Documentation regarding system architecture (including ownership of functional diagram of system)
- Assisting in contract negotiation
- Giving advice to teams regarding functional layout and interfaces

*What is out of scope:*

- Lower level architecture such as message formats, coding conventions etc (see Software Architect project)
- Documentation framework itself (see Knowledge Czarina project)

*Stakeholders:*

- The Duckietown masters
- All other project teams that are part of the system

## 4.5. Quest 1, Part 2: Definition of the problem

### 1) Problem statement

---

Ensure that all teams know what their goal is and how it fits into the bigger picture.

### 2) Assumptions

---

The current system makes at least kind-of sense. The current system will be used as a base, onto which improvements or functionality will be added by the projects.

### 3) Approach

---

- Become one with the goals of Duckietown
  - In order to make Duckietown a better place, one has to keep in mind what “better” means in Duckie terms.
- Be familiar with the current system architecture and track changes
  - This can include having to update the functional diagram, for instance.
  - This also means identifying which teams affect which modules in the diagram
- Keep in close contact with teams
  - All teams will designate a contact person who can contact me whenever they change their project boundaries or have doubts/ need advice on their project’s boundaries/negotiating with other
    - The meetings of some of the other teams will be attended (especially early meetings). Some teams’ meetings have been prioritized since many parts of the system are dependant on their work, namely:
      - Anti-instagram
      - Controllers
      - Navigators
      - Explicit coordination

- Offer nudges in a different direction if needed
- Acting as middleman/helper to facilitate negotiation of contracts between groups
- Monitor status of projects to find possible problems

#### 4) Functionality-resources trade-offs

---

*Functionality provided:*

- System integration of project modules

*Resources required / dependencies / costs:*

- Biggest resource: Time
- Finding out how to maximise usefulness while being efficient with time

*Performance measurement:*

- Approval of Duckietown masters
- Number of miscommunications about contracts between teams (measured in what-the-ducks per second)
- How many things didn't go wrong
  - Some jobs are of the type where no one notices you until something doesn't work. You should be the silent angel fixing all the problems that no one even noticed existed.

### 4.6. Quest 1, Part 3: Preliminary design

#### 1) Preliminary plan of deliverables

---

- Functional diagram of the system
  - The system functional diagram will be the main tool to visualise the system decomposition, and show the relationships between the different teams.
- Documentation of system architecture

### 4.7. Quest 1, Part 4: Project planning

#### 1) First steps for next phase

---

- Familiarisation with existing system architecture
- Going to group meetings
- Identifying potential problems

*Relevant Duckietown resources to investigate:*

- The functional diagram of the system
- Other teams' preliminary design reports

#### 2) Risk analysis

---

**Challenges:**

- Maintaining the balance between project level scope and Duckietown level scope. For instance, teams are focused on completing their project, and might forget the greater vision of Duckietown. This might mean having to convince teams to do slightly more work, for it to be more useful to Duckietown. After all, what's the point of doing a project if it does not contribute to Duckietown?

- Balancing priorities of quest 1 and 2. Quest 1 is crucial, and takes priority over quest 2. Therefore it will be challenging to find time (main resource) to work on quest 2.

## 4.8. Quest 2

Where there is a system, there is a want (nay, need) for optimisation. Describing a system's performance and resource requirements in a quantifiable way is a critical part of being able to benchmark modules and optimise the system.

The different levels of quest 2 are defined as follows:

- **Bronze standard:**
  - Define a formal, qualitative language to describe constraints/requirements between modules.
- **Silver standard:**
  - Each module has table of performance. A tool is developed that can give a qualitative answer to the question: With given configuration  $x$ , is the cost/requirements possible with available resources?  $f(x)$  smaller equal to  $R_{max}$ ?
- **Gold standard:**
  - Optimization of the system is possible to find the best implementation (most functionality and performance), given the available resources. Given  $f(x)$  and  $R_{max}$ , find the/an optimal configuration  $x$ .

## 4.9. Quest 2, Part 1: Mission and scope

### 1) Mission statement

---

Formalise the description of the system characteristics, so that eventually the system performance can be optimised for some given resources.

#### *Stakeholders:*

- The Duckietown masters, for they have started me on this quest.
- The future users of Duckietowns, as the Duckietown experience can be optimised per user, given their available resources.

## 4.10. Quest 2, Part 2: Definition of the problem

### 1) Problem statement

---

- Find a way to describe all the module requirements, specifications, etc in a formal, quantifiable language.
- Find a way to calculate the requirements and specifications of a whole system or subsystem, based on the requirements and specifications of the individual modules of the system.
- Find a way to calculate the optimal system configuration, based on the desired requirements and specifications of the system.

## 2) Approach

---

- Research on the topic of formal description of a system
- Find/develop a suitable language to describe module characteristics
- Require groups to compile a description of their respective modules' characteristics
- Find/develop functions to do mathematics on the language description of modules

## 3) Functionality-resources trade-offs

---

*Functionality provided:*

- Being able to quantifiably describe the system
- Being able to compare and benchmark different implementations of parts of the system

*Resources required / dependencies / costs:*

- Time is the main resource that needs to be divided between tasks.
- The current system's characteristics will need to be measured/evaluated somehow to create a database of all the modules' specifications.

*Performance measurement:*

- How effectively is the system described, and can the tool provide the answers we seek.
- Which level of functionality was provided? (bronze, silver, gold)

## 4.11. Quest 2, Part 3: Preliminary design

### 1) Preliminary plan of deliverables

---

- Documentation on the result of the project
- A description of the current system's characteristics (bronze)
- A program/tool that can give a qualitative answer (yes/no) to the question: Are these resources sufficient for this system configuration? (silver)
- A program/tool that will give an optimised system configuration, based on the given available resources (gold)

## 4.12. Quest 2, Part 4: Project planning

### 1) First steps for next phase

---

- Look into yaml description of modules
- Research into existing methods of system description
- Graph based databases?
- Perhaps graph theory can be useful later if the (suspiciously graph-looking) system can be described suitably.

*Data collection:*

- Module descriptions can be collected from the respective groups (by asking nicely)

*Relevant Duckietown resources to investigate:*

- `devel-heroes-formal-description` branch on Software repository
  - How the current system's characteristics are defined
  - What are the dependencies/interfaces between modules
  - Which projects affect which modules
  - Which values/parameters are needed

## 2) Risk analysis

- Quest 1 takes priority over quest 2, since it is more crucial to the functioning of the system. This means that quest 2 may suffer if quest 1 takes more time than expected.
  - Quest 2 has a research/experimental aspect, which makes it both interesting and challenging.
  - There is a chance that it might not be solved, as it is not a trivial problem.

UNIT L-5

The Heroes - System Architecture: final report

**TODO:** links to relevant files

System Architecture refers to the high-level conceptual model that defines the structure and behaviour of a system. There are different ways to get an insight into the architecture of a system, for example functional decomposition diagrams, package composition, or a Finite State Machine diagram.

### 5.1. The final result

The System Architecture project helped to ensure that the Fall 2018 projects integrate into the existing system. The role of the System Architect was to identify and solve problems that arose during the project development and integration, as well as influencing the high-level design of the system.

In Duckietown, the Finite State Machine (FSM) diagram plays an important role in determining how the higher-level system behaves in different scenarios. The FSM defines which states the system can be in, and which functionalities must be active in which states. During the project, the need for development on the FSM arose. Below is the resulting updated Finite State Machine (FSM) diagram.

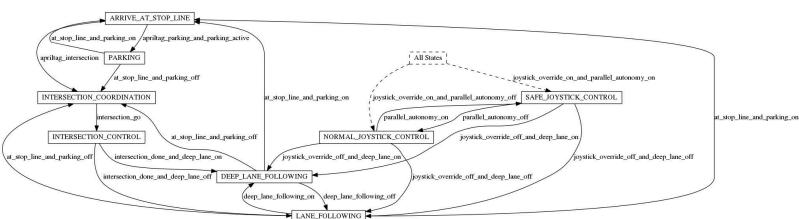


Figure 5.1. The Finite State Machine

## 5.2. Mission and Scope

The System Architecture project was not a clearly defined work package, but rather a responsibility to ensure smooth development and integration of the new projects into the existing system.

### 1) Motivation

---

With many teams working on many different parts of the system, chaos is inevitable (without divine intervention). The role of the System Architect was to ensure that development and integration of new projects goes smoothly, by addressing interface, contract and dependency issues between the teams.

During the project, the need arose for an updated version of the FSM.

### 2) Existing solution

---

Duckietown already had an existing system architecture. As mentioned before, the FSM is closely related to the system architecture, since it defined the high-level behaviour of the system. There was an existing infrastructure for the FSM, which could be modified to include the newly developed functionality. The infrastructure consisted of the `fsm` ROS package, the configuration of the FSM in the form of `.yaml` files, as well as a tool to visualise the FSM structure.

*ROS fsm package:*

The `fsm` package consists of two nodes, namely the `fsm_node` and the `logic_gate_node`. The `fsm_node` is in charge of determining the current state and computing state transitions, and the `logic_gate_node` acts as a helper node to the `fsm_node`. For more information, see the README of the `fsm` package, found at [20-indefinite-navigation/fsm/README.md](#).

+ comment

JT: README for fsm in indefinite navigation seems out of place

*Configuration of the FSM:*

While the `fsm` package handles the computation of state transitions, the FSM states and transitions can be configured using the supplied `.yaml` files. The `fsm` package then reads the configuration in order to know which states and transitions are available in the system. This allows for separation of the computation and configuration of the FSM.

*FSM visualisation tool:*

There exists a tool to parse and visualise the FSM configuration (in the form of an FSM diagram), found at [00-infrastructure/ros\\_diagram/parse\\_fsm.py](#). The tool parses the `.yaml` configuration file and outputs a `.dot` format graph, which can be converted to `.png`.

### 3) Opportunity

---

During Fall 2018, many new projects were being developed by multiple teams. This created the need for someone to ensure harmony between the projects and the system during the process.

Duckietown was being expanded with new functionalities such as parking and deep learning lane following. The previous FSM was not equipped to deal with the new features, therefore it had to be further developed.

### 5.3. Definition of the problem

Ensuring that development of the new projects integrate into the existing system with as little as possible chaos. This implies ensuring that all teams understand their package's objective and their impact on the bigger system.

### 5.4. Contribution / Added functionality

The contribution of this project was in the form of both organisational activities, as well as software development. System integration of project modules. The approach can be summarised as follows:

- Become one with the goals of Duckietown
- Be familiar with the current system architecture
- Track changes and identify how they influence the system.
- Keep close communication with project teams.
- Act as middleman/helper to facilitate negotiation of contracts in and between groups.
- Monitor status of projects to find possible problems.

#### 1) Mediation of project interface negotiations

It was important to define how the new projects will interact with the old system, as well as with the other projects. Familiarisation with the existing system architecture was the first step. During the beginning of the project development, the System Architect attended some of the individual groups' meetings to get a better overview of what they plan, as well as how they will affect and be affected by other teams. This process helped to spot problems early on, so that teams can be certain of what is expected of them from other teams and vice versa.

#### 2) Development of updated FSM

A dedicated System Architecture meeting was held in class to resolve any further conflicts and to conclude discussion on the interfaces between projects. More information on the `fsm` package at ...

#### 3) Documentation of FSM package

The previous FSM did not have a README, therefore the documentation was improved greatly.

## 5.5. Formal performance evaluation / Results

*Be rigorous!*

- For each of the tasks you defined in your 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?

## 5.6. Future avenues of development

The existing framework for the FSM made it relatively easy to update it to include new functionalities (once you’ve decided on the system architecture). The FSM is configured using .yaml files, which are then loaded into the `fsm_node`.

Development of the updated FSM was done in response to a need before demo day, and while it has been tested on its own, it has not been tested thoroughly with all other parts of the system yet.

# UNIT L-6

## PDD - Smart City

### 6.1. Part 1: Mission and scope

#### 1) Mission statement

---

Make Duckietown a smarter city.

#### 2) Motto

---

OMNES VIAE ANATUM URBEM DUCUNT  
(All roads lead to Duckietown)

#### 3) Project scope

---

*What is in scope:*

- Manufacturing process of tiles
  - Consider different ways to implement the lines on the road
  - Spray tiles for lines instead of tape
- Power Grid
  - Add power to each tile

- Establish power grid layout
- Establish power grid design and implementation

*What is out of scope:*

- Communication protocol (what to do with the data)
- Appearance specifications redefinition (cit. “we need an Italian architect for this”)
- Traffic controller HUB
- City wide power hub

*Stakeholders:*

Team	Reference Person
Intersection Navigation	Nicolas Lanzetti (ETHZ)
Parking	Samuel Nyffenegger (ETHZ)
Traffic controller HUB	no teams actively working on this project in Fall-2017
System Architect	Sonja (ETHZ)
Software Architect	Breandean (UdM)
Knowledge	Tzarina

## 6.2. Part 2: Definition of the problem

### 1) Problem statement

---

We have to design a traffic lights system that integrates seamlessly and efficiently with the tiles currently used to build Duckietown. The development of a traffic lights system has to be considered as part of a bigger plan aimed to make Duckietown a smart city. A smart Duckietown has the capability of delivering wireless connectivity everywhere (Duckietown Wireless Network - DWN) in the town and power to each tile. A tile that can provide power is called a hot tile. The power grid that provides power to all the tiles is called Duckietown Power Grid (DPG). A simple use case for this infrastructure would then be the traffic lights system. Traffic lights at each intersection are powered and controlled by a Raspberry Pi with a Duckiebot-like LED Hat and 3 (or 4) LEDs. A Raspberry Pi responsible for the traffic lights at an intersection draws power from a hot tile and connects to the DWN.

### 2) Terminology

---

- DWN: Duckietown Wireless Network
- DPG: Duckietown Power Grid
- PD: Powered Device

### 3) Assumptions

---

- 2 traffic light per current city design (1 at 4-way the other at 3-way intersection)
- Appearance specifications are given and must be respected
- There exists a Duckietown Wireless Network (DWN) that any wifi enabled device placed within the town can connect to (e.g. traffic lights).
- There is access to power source close to Duckietown that can power the power grid

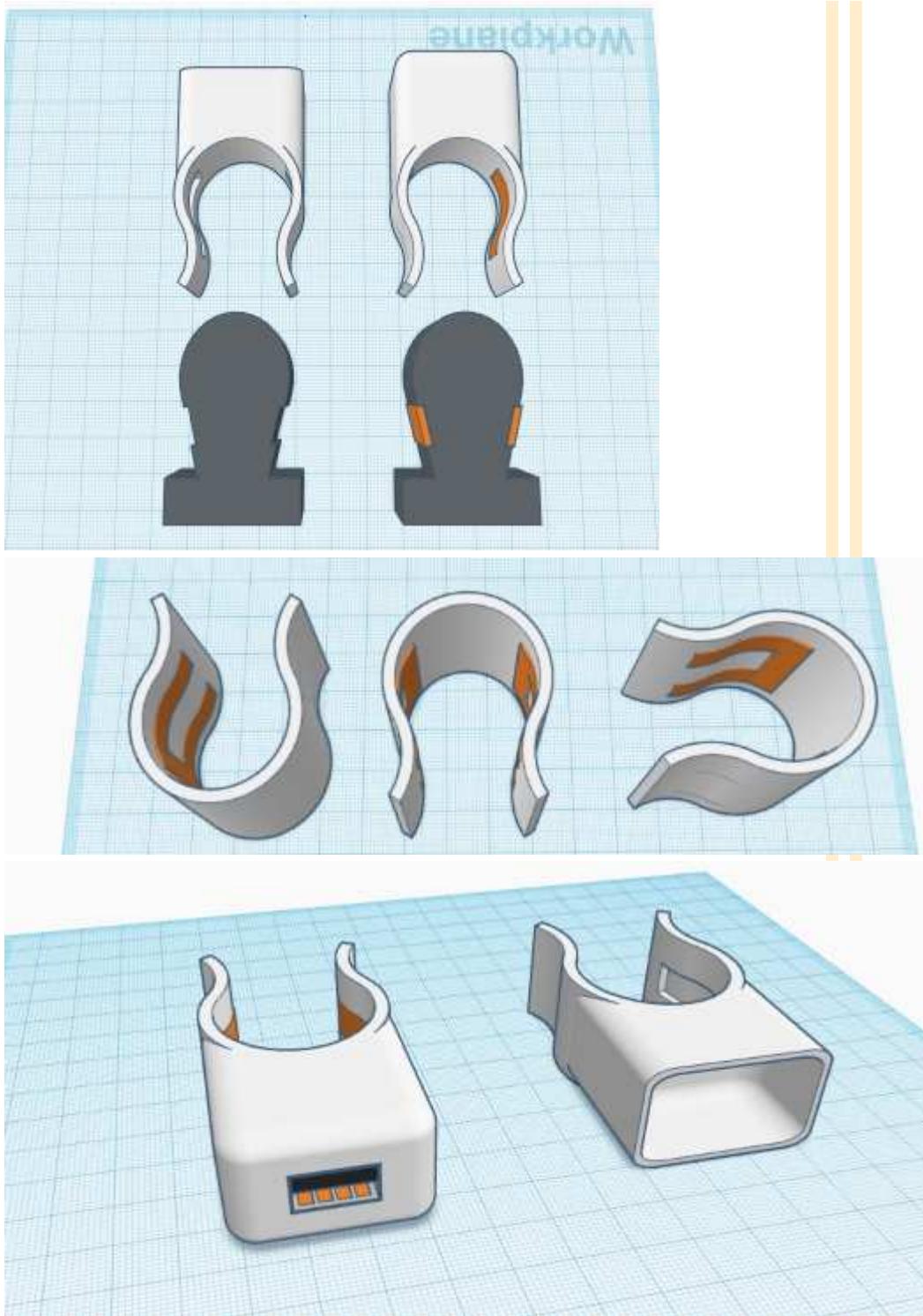
### 4) Approach

---

To create a smarter Duckietown and provide data and power to the tiles, we will wireless networks (e.g., WiFi, Bluetooth, etc.) for data communication, and we will implement a power grid to provide power to the various devices and PDs throughout Duckietown. Since we are using common implementation of wireless networks, the rest of this design document will focus on the specifications of the power grid.

### Power Grid Implementation Ideas:

- Idea 1: Attach a 2-row breadboard along the edges of each tile, between the white tape and the teeth of the tile. PDs are connected to the power grid simply by inserting the two wires (+ and -) into the relative holes.
  - Problems: The primary problem under this approach is that breadboards are rated for only ~1amp, which is not nearly enough power needed for the power grid (for example, a single Raspberry Pi can use more power than that. This problem essentially eliminates the feasibility of this idea for the DPG).
- Idea 2: Attach a plastic rail to the edge of each tile, between the white tape and the teeth of the tile. The rail would carry two conductive strips (copper strips), one on each side (see image below).
  - Prototype: The image below shows a possible design of the plastic rail along with a compatible plug. The black part of the 3D model above constitutes the rail (sectional view) while the white part is the plug. The system is designed so that the plug, once pressed onto the rail, remains attached. The white box on the plug would contain one of the step-down converters (<http://a.co/fAIAbuw>) described above. This would solve the problem of having a weak 5V power grid by running 24V through the grid and stepping it down to 5V only when, and exactly where, we need it. There would be limit neither to the number of plugs nor to the position where we can attach them (even better than a breadboard in this sense). We can then design simple connectors for straight and curved tiles to make everything modular. Since the most common PD in Duckietown is a Raspberry Pi, we can design a USB plug (shown below) to make things even easier.
  - Enhancement 1: We can modify the plug by adding an extrusion to one side and carving its negative into the rail. This would prevent us from attaching the plug in the wrong direction, thus violating the positive/negative polarity of the conductors.
  - Enhancement 2: Since the plastic material used by 3D printers is usually inflexible we can change the plug such that the plastic does not follow the design of the rail (i.e., it would look like a U flipped upside-down) and have a curved copper strip that stretches when the plug is pushed onto the rail and loosens when the plug sits completely on the rail. Basically, it follows the same concept used in the classic cigarette lighter plug present in a vehicle.
  - Enhancement 3: We can use plastic T-slotted extrusion elements and design a plug that works with them. Problem: The primary problem with this approach is its difficult, especially given the project's short timeframe. However, we could focus on designing and building a prototype that works that could then be mass produced and implemented for the whole Duckietown sometime in the future.



- Idea 3: Have the connectors between tiles also serve as the output location for power to the tile. Use audio cable or RCA cable for the power rails and connect

them at the corners of the tile using a 3 way connector, such as those shown below. This approach solves the issue regarding gendering the connectors and providing power nodes to the city. Cheap and easy to mass produce.

- o Problem: This may require modification of the tiles, such as removing one of the interlocking teeth to allow for the connector.



## 5) Functionality provided

---

The actual voltage and amperage available at each tile/power terminal will depend on the power grid approach we choose. Regardless of the implementation, the primary functionality provided by the power grid is access to power for at each tile in the Duckietown.

## 6) Resources required / dependencies / costs

---

The resources for this project are the parts to build the traffic lights and the power grid. Since, the specific parts and associated costs for the power grid are highly dependent on the implementation approach we decide on, we are unable to obtain specific details at this time. However, for all of the approaches, we will need enough parts to build a power grid that provides power for all of the tiles in the Duckietown.

## 7) Performance measurement

---

Power Grid:

- Maximum number of powered devices per tile
- Ease of assembly/disassembly
- Ease of manufacturing
- How robust is the power grid under normal usage

System:

- Maximum image frame-rate traffic lights can sustain over the utilities network (network bandwidth)

## 6.3. Part 3: Preliminary design

---

### 1) Modules

---

- Laying the power cables
- Connecting the power cables
- Output power for tile

### 2) Interfaces

---

Input: 12/24 V, Output: 12/24 V between tiles, 5 V on tile

### 3) Preliminary plan of deliverables

---

Power grid and integration into the individual tiles must be designed and implemented. While the traffic lights exist, there needs to be a revised method of providing power.

### 4) Specifications

---

May have to modify Duckietown tiles.

### 5) Software modules

---

None, this is a hardware project.

### 6) Infrastructure modules

---

All modules are infrastructure.

## 6.4. Part 4: Project planning

### 1) First Steps for the next phase

---

Decide connector option and wire routing.

### 2) Data collection

---

Stability of power grid. How many traffic lights can be supported per voltage source.

### 3) Data annotation

---

None.

### 4) Relevant Duckietown resources to investigate

---

Specification of traffic light.

### 5) Other relevant resources to investigate

---

None.

### 6) Risk analysis

---

What could go wrong?

- Wire gauge too low to accommodate power load, causing shorts and possibly melting tiles or starting small fires.
- Live wires are exposed and come into human contact.

How to mitigate the risks?

- Appropriately fuse the tiles and use appropriate wires for power load.
- Insulate everything well and keep open contacts small and covered.

## UNIT L-7

# System Identification: preliminary report

## 7.1. Part 1: Mission and scope

### 1) Mission statement

---

Estimate better models to make localization and control more efficient and robust to different configurations of the robot.

## 2) Motto

---

NOSCE TE IPSUM

(Know thyself)

## 3) Project scope

---

*What is in scope:*

- hardware specifications for calibration
- choose which model to “identify”
- Identify kinematic parameters (mapping of commands to actuators)
- Include model of the caster wheel

*What is out of scope:*

- Additional onboard or external sensors
- Measuring the latency of the system
- Camera calibration
- State estimation using motion blur

*Stakeholders:*

- Control
- localization

## 7.2. Part 2: Definition of the problem

### 1) Problem statement

---

Every Duckiebot is different in configuration.

Mission = we need to make control robust to different configuration

Problem statement = we need to identify kinematic model to make control robust enough

### 2) Assumptions

---

- Robot will move only in horizontal plane
- No lateral slipping of robot
- The body fixed longitudinal velocity and angular velocity will be provided as well as the timestamp of each measurement

### 3) Approach

---

- Simplified kinematic model will be used to estimate parameters for each duckiebot :
  - Semi axis length l

- Mapping : voltage → velocity

*Kinematic model:*

We make use of the no lateral slipping motion hypothesis and the pure rolling constrain as shown in the [#duckiebot-modeling](#), to write the following equation:

$$\begin{aligned} \text{\label{eq:mod-kin-3}\tag{1}} & \left\{ \begin{array}{l} \dot{x}_A^R = R (\dot{\varphi}_R + \dot{\varphi}_L)/2 \\ \dot{y}_A^R = 0 \\ \dot{\theta} = \omega = R(\dot{\varphi}_R - \dot{\varphi}_L)/(2L) \end{array} \right. , \end{aligned}$$

Further we make the assumption that for steady state that there is a linear relationship between the input voltage and the velocity of the wheel:

$$\begin{aligned} \text{\label{eq:mod-kin-4}\tag{2}} & v_r = R \dot{\varphi}_l = c_r V_r \\ & v_l = R \dot{\varphi}_r = c_l V_l \end{aligned}$$

This lets us rewrite equation [\eqref{eq:mod-kin-3}](#):

$$\begin{aligned} \text{\label{eq:mod-kin-5}\tag{3}} & \left\{ \begin{array}{l} \dot{x}_A^R = (c_r V_r + c_l V_l)/2 \\ \dot{y}_A^R = 0 \\ \dot{\theta} = \omega = (c_r V_r + c_l V_l)/(2L) \end{array} \right. , \end{aligned}$$

Using the assumption that we can measure  $v_A$  we can determine  $c_r$  by setting the voltage  $V_l=0$ . The same procedure can be done to get  $c_l$ .

Using the assumption that we can measure  $\dot{\theta}$  we will then get the semiaxis length  $L$ .

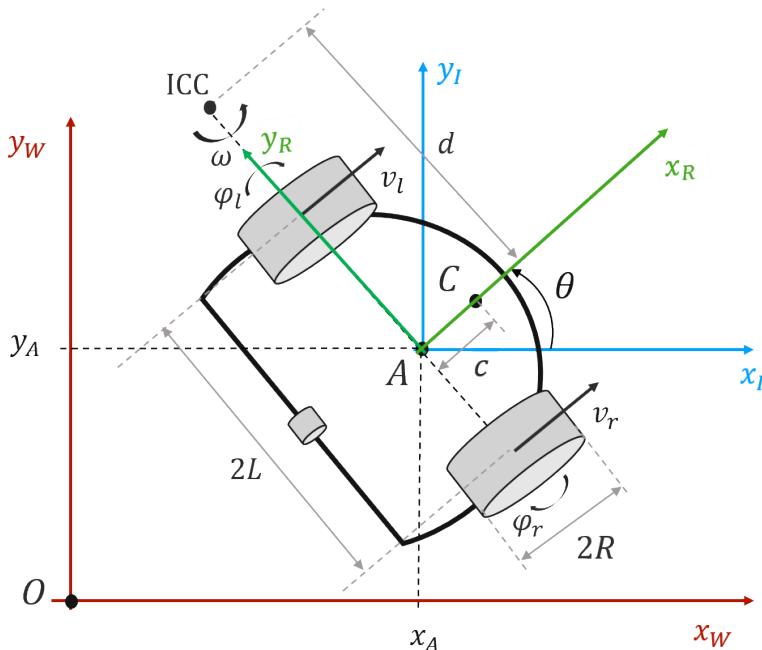


Figure 7.1. Relevant notations for modeling a differential drive robot

#### 4) Functionality provided

---

- A model with calibrated parameters for each duckiebot
- A calibration protocol that creates a map of:
  - input voltage → output longitudinal and angular velocity
- Semi axis length

#### 5) Resources required / dependencies / costs

---

- Good state estimation, independent of a model
- potential approaches for state estimate
  - lane filter
  - april tags
  - camera calibration sheet
- Accurate line-detection
- Accurate april tags

#### 6) Performance measurement

---

Run lane follower with old version and new version with kinematic model. Drive on the track for one minute and count the number of times the bot touches the side or center line.

#### Metrics

- Robustness to different duckies
  - Control can handle different duckiebot configurations based on our models
- Robustness to different wheels
  - Omnidirectional wheel, caster wheel
- Robustness to initial pose
  - Run lane following using 5 different initial poses
- Repeatability
  - Run lane following 5 times and compare

We will use the performance measurement setup of the devel-control group

### 7.3. Part 3: Preliminary design

#### 1) Modules and interfaces

---

##### Parameter estimation

- Input :
  - State estimation
  - Specific voltage to each wheel
- Output :
  - semi axis length and wheel radii

Mapping voltage → velocity

- Input :

- Specific velocity to each wheel
- Output :
  - Voltage

## 2) Specifications

---

Duckiebots with different hardware configurations for testing

## 3) Software modules

---

- Parameter estimation:
  - runs calibration protocol
- Velocity translation: (Node)
  - get velocity as input and translate it to voltage as output

## 4) Infrastructure modules

---

None

## 7.4. Part 4: Project planning

Date	Task Name	Target Deliverables
17/11/17	Kick-Off	Preliminary Design Document
24/11/17	Play around	Identify current problems
01/12/17	First estimation	find paramers of robot
08/12/17	Validation	Performance measure
15/12/17	Caster wheel	Performance measure of new implementation
22/12/17	Buffer	
29/12/17	Documentation	Duckuments
05/01/18		End of Project

## 1) Data collection

---

What data do you need to collect?

## 2) Data annotation

---

Performances of the current implementation

*Relevant Duckietown resources to investigate:*

- Current State Estimation
- Calibration files

*Other relevant resources to investigate:*

[Handbook of robotics](#)

the above contains a number of interesting sections of relevance to the work of this group:

- exact modeling of caster wheel and the kinematic constraints it introduces (pg. 395)

- different system identification procedures: parametric or nonparametric (Chapter 14); in particular, a note on Observability (pg. 337)
- we want to maximize performance of control + localization. Control uses unicycle model in Frenet frame (pg. 803 of handbook of robotics)
- We need to identify wheel radii ( $r_1, r_2$ : assume equal at start =  $r$ ), semi-axle length  $L$ , and motors steady state parameters (mapping between voltage and angular rate, i.e. mapping between voltage and velocity once (a) wheel radius is known and no slipping hypothesis is made).
- Adaptive control (pg. 147): another approach is implementing an adaptive controller. It is meant to work with plant uncertainty.

## Caster wheel literature

### 3) Risk analysis

---

What could go wrong?

- It could happen that we identify a model which is not useful for control.
- Perfect model will be useless if control is not improved

Mitigation strategy:

- Early testing with control group

## UNIT L-8

# System Identification: intermediate Report

## 8.1. Part 1: System interfaces

### 1) Logical architecture

---

#### Desired functionality

The desired functionality is a node that takes the desired linear and angular velocity as input and maps it to the input voltages for the motors.

#### Approach

In a first step we measure manually the longitudinal and angular velocity for given input voltage. The aim is to use these measurements to find the  $c_r$ ,  $c_l$  and  $L$  (or  $c$ ,  $trim$ ,  $L$ ) of our model that was introduced in the preliminary design document.

$$\begin{aligned} \text{\backslash begin\{align\}} \text{\backslash label\{eq:mod-kin-5\}} \text{\backslash tag\{1\}} & \text{\backslash left\{ \backslash begin\{array\}\{l\} } \dot{x}_A^R &= (c_r \\ V_r + c_l V_l)/2 & \text{\backslash dot } y_A^R &= 0 \\ & \text{\backslash dot } \theta &= \omega = (c_r V_r - c_l V_l)/(2L) \text{\backslash end\{array\}} \text{\backslash right., \backslash end\{align\}} \end{aligned}$$

This “linear” velocity to voltage function can be used for testing by the controller group. If we manage the first step, we will move on to a second step. Here we will aim to get a “non-linear” velocity to voltage map. There will be a calibration procedure that creates a custom velocity to voltage map for each Duckiebot that should

be independent for different hardware configurations. The current idea for the calibration procedure is to drive the Duckiebot with different voltage commands to the motor while a checkerboard is in its viewfield. The logs can then be used to extract a pose estimation of the Duckiebot compared to the checkerboard. The recorded voltage and the pose data will then be fitted in a nonlinear manner, maybe using the aca-do optimization toolbox.

## Assumptions

### *Ground truth estimation*

- Repeatability (currently investigating)
- Can get ground truth also while driving (currently investigating)
- Minimum distance from which each square is visible is sufficient to do calibration trajectories with the Duckiebot (currently investigating)
- The calibration procedure gives very accurate ( $\leq 5 \text{ mm}$ ) pose estimates to do a decent mapping

### *Performance*

- Our assumption is that once calibrated the Duckiebot will be able to repeat the same behavior, and the kinematics do not change.

## 2) Software architecture

---

### Calibration procedure Node

To achieve the desired functionality, we will create a mapping from velocity states to input motor voltages that will be used by the control. The procedure that will be followed is summarized as follows:

- Drive the Duckiebot with different open loop voltage commands to the motor on a 2x2 tile while a checkerboard is in its view field. The voltage commands should be chosen to allow sufficient exploration of the velocity states to input voltages mapping. Take a log of the motion for offline processing in the next step.
- Process the logs of the images to extract a pose estimation of the Duckiebot using the checkerboard as a reference.
- Fit The recorded voltage and the pose data in a nonlinear manner potentially using the Acado optimization toolbox.

### *Input*

- Intrinsic camera calibration
- Extrinsic camera calibration
- Rosbag of raw images and control commands of the Duckiebot driving in front of the checkerboard, the rosbag is recorded by running the calibration node

### *Output*

- Control commands during calibration procedure (topic: car\_cmd, same as control)
- calibration.yaml file containing velocity to voltage map

### *Assumed Latency*

Offline. The calibration procedure is done on the computer

### **Position estimate**

In order to identify a system model, we need the best possible state estimation. This shall be achieved by calculating the camera extrinsics from the checkerboard for each frame. The picture below shows our first experiments with the setup.

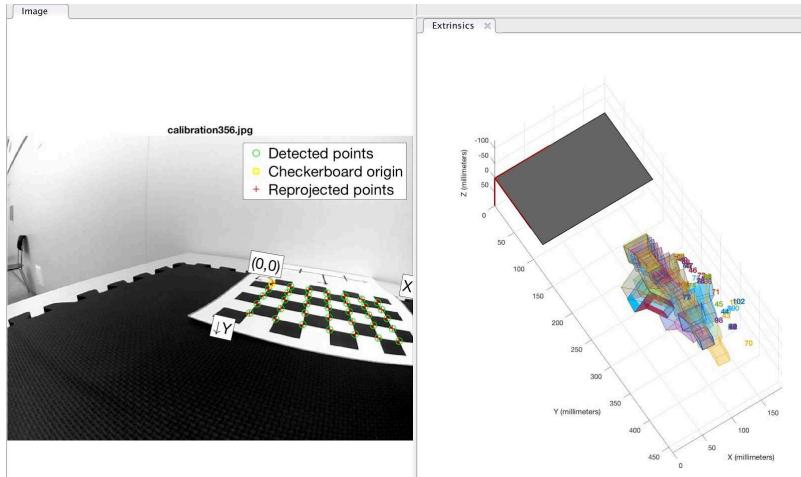


Figure 8.1. Setup for state estimation

**Inverse\_kinematics\_node** We will edit the existing inverse\_kinematics\_node. It will get the desired velocity as input and find the corresponding motor speeds using the parameters from the calibration.yaml file generated by the Calibration procedure. For desired velocities that exceed the system's capabilities, the maximum possible velocity will be returned.

### **Subscribed message**

car\_cmd

### **Published message**

wheels\_cmd

### **Assumed Latency**

Negligible, will run a very lightweight callback directly once it receives a car\_cmd message

## **8.2. Part 2: Demo and evaluation**

### **1) Demo plan**

The main goal is to demonstrate improved calibration. For this purpose, we will first run the lane following module with a Duckiebot that has the default velocity to volt-

age mapping on the same test track, see picture. Afterwards, we will run the improved calibration procedure that will create the custom velocity to voltage mapping. We will then run the lane following module again and see how lane following task is improved. As the actual calibration procedure will take sometime, we will do it beforehand.

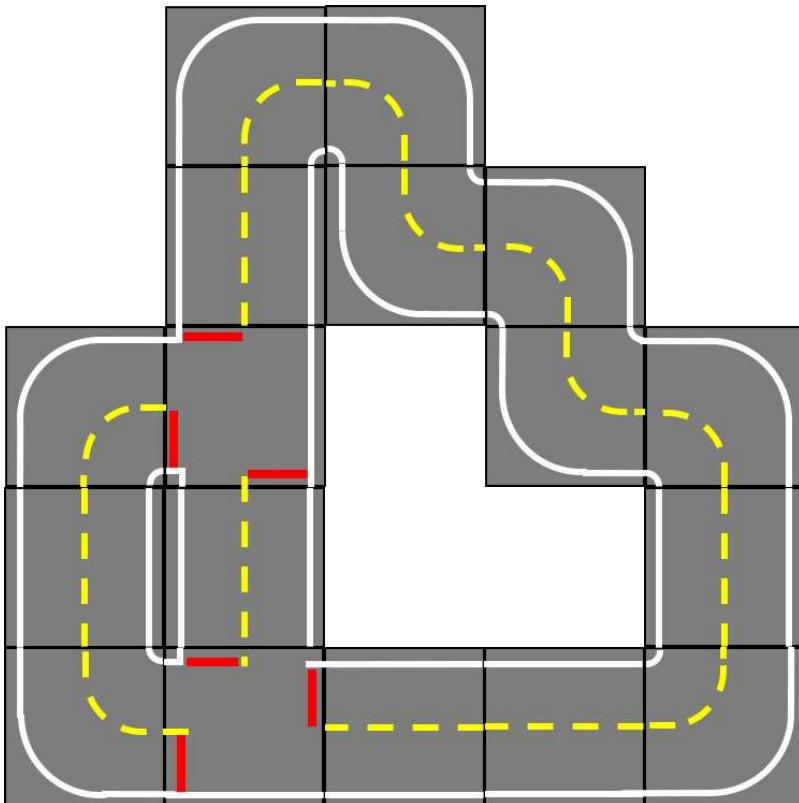


Figure 8.2. Duckietown test track

### 8.3. Plan for formal performance evaluation

We will run all of the tests 3x times uncalibrated, and 3x calibrated for the following Duckiebot configuration:

- Normal Duckiebot
- Duckiebot with different right and left wheel diameters and compare the improvements

#### Offset in straight line :

We will let the Duckiebot drive straight in open loop and measure its offset after X tiles of straight lane in Duckietown. The performance metric will be the absolute position offset of the expected to the actual terminal position after the run, measured with a ruler.

#### Circle test :

We will drive the Duckiebot with a constant velocity  $v_a$  and constant angular velocity  $\dot{\omega}$  in open loop on a Duckietown corner tile. We will compare the actual path with the desired path. This is done both clockwise and counterclockwise. The performance metric will be the absolute position offset of the expected to the actual terminal position after the run, measured with a ruler.

#### Integration test :

We want to test how the improved calibration affects the line following mode. Compare different behaviors in line-following mode.

#### Material needed to do calibration

- 4 Black tiles
- Normal Camera Calibration Checkerboard that can be attached vertically

#### Material needed to do performance test

- Duckietown with straights and corners
- Ruler to measure offset of terminal position

## 8.4. Part 3: Data collection, annotation, and analysis

### 1) Collection

- How much data do you need?

At least one log file (rosbag) of the Duckiebot being driven with various voltage inputs and has a camera calibration checkerboard in view at all times. The checkerboard will be used to estimate position from images. We'll write the code for pose estimation (eg. assemble existing libraries).

- How are the logs to be taken? (Manually, autonomously, etc.)

Initially logs will be taken manually, but later will be taken automatically by a calibration node. The Duckiebot, should be on a Duckietown tile and have the checkerboard in view.

- Do you need extra help in collecting the data from the other teams?

We do not need data from other teams and therefore do not need help.

### 2) Annotation

- Do you need to annotate the data?

No, we need to extract pose from images, which is done by geometry from a checkerboard, not annotation

- At this point, you should have tried using thehive.ai to do it. Did you?

No, because we do not need any annotations.

## 8.5. Analysis

- Do you need to write some software to analyze the annotations?

We don't need data annotation since we can do all the benchmarking by our own. However we are writing the software to estimate the model based on data collection.

We already did a basic analysis of the system by running control commands and measuring the distance or angle by hand. The duration and magnitude of the control commands were extracted from a rosbag. This lets us generate a map from control to velocities, thus basically we have a first guess of the model parameters (average C, C/2L). The results can be found in the two plots below. The estimated speed of the Duckiebot in lane following mode is 0.27 m/s and the yaw rate at maximum control input 3.7 rad/s. The estimate control gains are 0.67 m/s ( $\pm 15\%$ ) per  $v_{cmd}$  and 0.45 rad/s ( $\pm 30\%$ ) per  $\omega_{cmd}$ .

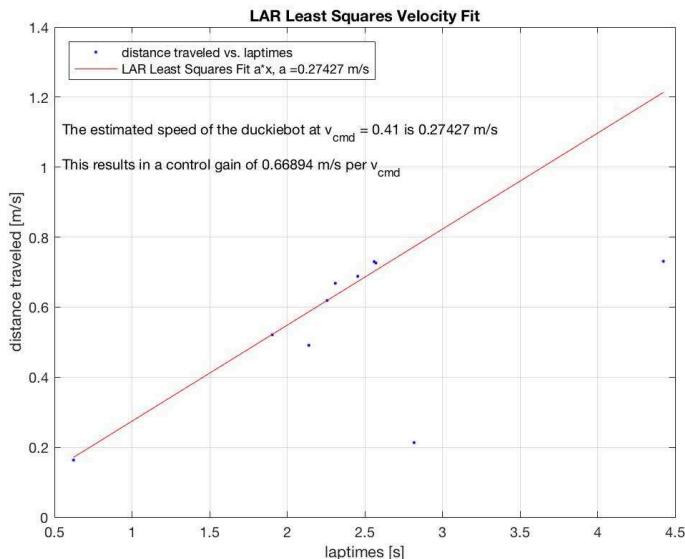


Figure 8.3. Estimated linear velocity

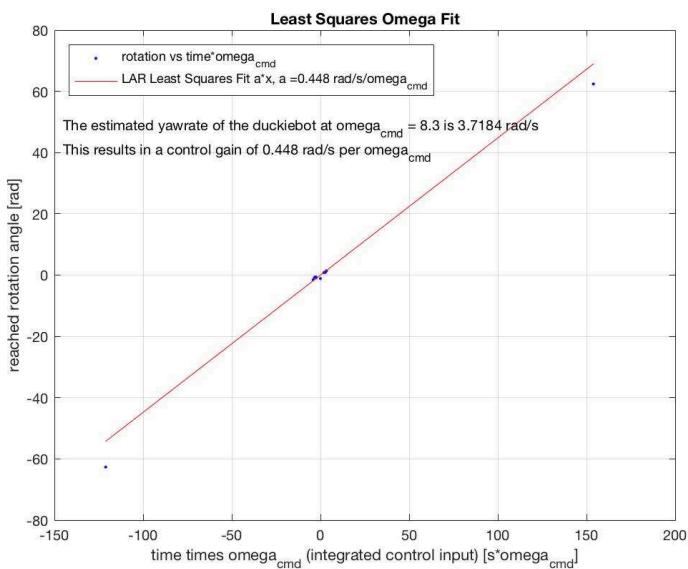


Figure 8.4. Estimated yaw rate

## UNIT L-9

### System identification: final report

**TODO:** JT: switch intermediate and first videos

#### 9.1. The final result



Figure 9.1. Demo of the calibration procedure

To reproduce the results see the [operation manual \(master\)](#) which includes detailed instructions for a demo.

#### 9.2. Mission and Scope

## 1) Motivation

The mission is to make the controller more robust to different configurations of the robot. The approach chosen to do this is obtaining a mathematical model of the Duckiebot in order to understand its behavior. The mathematical model can then be used to design a controller to obtain robust desired behaviors and performances.

The Duckiebot is in a differential-drive configuration. It actuates each wheel with a separate DC Motor. By applying the same torque on both wheels one can go straight, and by applying different torques the Duckiebot turns. A schematic overview of the model can be seen in Figure [Figure 9.2 \[31\]](#).

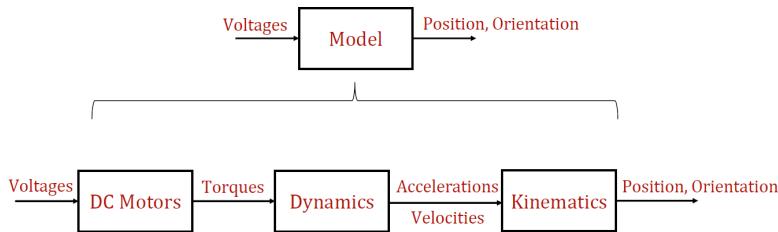


Figure 9.2. Model of differential drive robot

A mathematical model for a differential drive robot will be derived. This model can be used to provide a simple method of maintaining an individual's position or velocity estimate in the absence of computationally expensive position updates from external sources such as the mounted camera.

The derived model describes the expected output of the pose (e.g. position, velocity) w.r.t. a fixed inertial frame for a certain voltage input. The model makes several assumptions, such as rigid body motion, symmetry, pure rolling and no lateral slipping. Most important of all, the model assumes the knowledge of certain constants that characterize the DC motors as well as the robot's geometry.

However, there will never be two duckiebots that show exactly the same behavior. This can be very problematic. You might have noticed that your vehicle doesn't really go in a straight line when you command it to. For example, when the same voltage is supplied to each motor, the Duckiebot will not go straight as might be expected. Also, the vehicle might not go at the velocity you are commanding it to drive at.

Therefore, these constants need to be identified individually for each single robot. The determination process to do so is called system identification. This can be done by odometry calibration : we determine the model parameter by finding the parameters that fit best some measurements of the position we can get.

Hence, when these kinematic parameters are defined, we are able to reconstruct the robot's velocity from the given voltage input.

Increasing the accuracy of the Duckiebot's odometry will result in reduced opera-

tion cost as the robot requires fewer absolute positioning updates with the camera. When the duckiebot is crossing an intersection forward kinematics is used. Therefore, the performance of safe crossing is closely related to having well calibrated odometry parameters.

## 2) Existing solution

### *Forward Kinematics:*

The existing mathematical model was the following :

$$\begin{aligned} V_l &= (g+t)(v_A - \omega L) \quad \text{label: eq:V_l} \\ V_r &= (g-t)(v_A + \omega L) \quad \text{label: eq:V_r} \end{aligned}$$

TABLE 9.1. NOTATIONS FOR THE EXISTING MODEL OF THE DIFFERENTIAL DRIVE ROBOT

$V_{l,r}$	Voltage to left/right motors
$g$	Gain
$t$	Trim
$v_A$	Linear velocity of Duckiebot in bodyframe
$\omega$	Angular velocity
$L$	Half of distance between the two wheels

Note that if the gain  $g = 1.0$  and trim  $t = 0.0$ , the wheel's voltages are exactly the same as the linear velocity + or - angular velocity times half the baseline length  $V_{l,r}=v_a \pm \omega L$ . With gain  $g > 1.0$  the vehicle goes faster given the same velocity command, and for gain  $g < 1.0$  it would go slower. With trim  $t > 0$ , the right wheel will turn slightly more than the left wheel given the same velocity command; with trim  $t < 0$ , the left wheel will turn slightly more than the right wheel.

The parameters  $g$  and  $t$  were to be set manually during the wheels calibration procedure.

### *Calibration Procedure:*

The current implementation of the calibration procedure can be found in the [#wheel-calibration \(master\)](#) section.

Hereby, the Duckiebot is placed on a line (e.g. tape). Afterwards the joystick demo is launched with the following command:

```
duckiebot: $ roslaunch duckietown_demos joystick.launch veh:=${VEHICLE_NAME}
```

Now the human operator commands the Duckiebot to go straight for around 2m.

Observe the Duckiebot from the point where it started moving and annotate on which side of the tape the Duckiebot drifted ([Figure 9.3](#)).



Figure 9.3. Left/Right drift

If the Duckiebot drifted to the left side of the tape, decrease the value of \$t\$, for example:

```
duckiebot: $ rosservice call /${VEHICLE_NAME}/inverse_kinematics_node/set_trim -- 0.01
```

Or Changing the trim in a negative way, e.g. to -0.01:

```
duckiebot: $ rosservice call /${VEHICLE_NAME}/inverse_kinematics_node/set_trim -- -0.01
```

This procedure is repeated until there is less than around \$10 cm\$ drift for two meters distance. The speed of the duckiebot can be adjusted by setting the gain:

```
duckiebot: $ rosservice call /${VEHICLE_NAME}/inverse_kinematics_node/set_gain -- 1.1
```

The parameters of the Duckiebot are saved in the file

```
duckietown/config/baseline/calibration/kinematics/{VEHICLE_NAME}.yaml
```

### 3) Opportunity

*Current shortcomings:*

- Human in the loop
  - The car is not able to calibrate itself without human input
  - The procedure is laborious and can be long
- Lack of precision
  - The calibration is only done for a straight line
  - The speed of the Duckiebot is not known

*Possible approaches:*

A crucial step should be to take the human out of the loop. This means that the car will calibrate itself, without any human input.

There were several possible approaches discussed to overcome the shortcomings of the current calibration:

- Localization based calibration
  - E.g. determine relative pose w.r.t. Chessboard from successive images
- Closed loop calibration
  - Modify the trim while Duckiebot is following a loop until satisfactory
- Motion blur based calibration
  - Reconstruct dynamics from blurred images

Because we needed to have very precise measurements of the Duckiebot's position, the localization based calibration has been chosen. To simplify the calibration procedures, we decided also to use the same chessboard as for the camera calibration. But since the computational power needed for detecting the chessboard was big, we had to do the chessboard detection on the laptop.

We also kept a kinematic model, without including any dynamic and made some assumptions about the physics of the Duckiebot: the wheels do not slip and the velocity of the wheels is proportional to the voltage applied. Hence, if the results do not meet our expectations or if the Duckiebot's configuration is changed, the model can also be changed or it can be made more complex.

#### 4) Preliminaries

---

- Differential-drive model [#duckiebot-modeling](#)
- Pinhole-camera model [#camera-geometry](#)

### 9.3. Definition of the problem

The approach we chose to improve the behaviour of the Duckiebots was to derive a model with some parameters, and to identify these parameters for each Duckiebot independently. Hence, we first construct a theoretical model and then we try to fit the model to the measurements of the position we get from the camera and the chessboard.

#### 1) Kinematic model

---

The Duckiebot was modeled as a symmetric rigid body, according to the following figure.

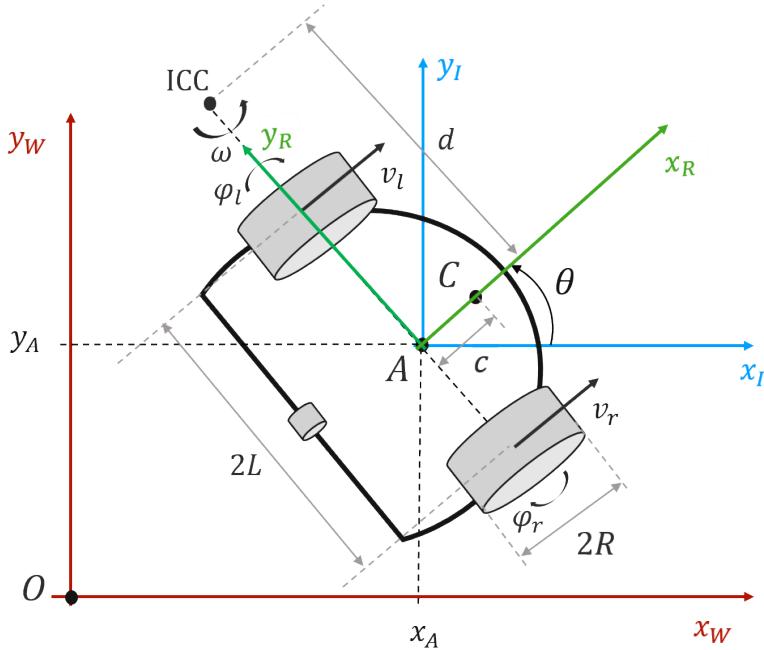


Figure 9.4. Schematics of differential drive robot [](#bib:Modeling)

Considering only the kinematics, we get the following equations for the linear and angular velocity  $v_A$  and  $\dot{\theta}$  of the Duckiebot :

$$\begin{aligned} v_A &= \frac{v_r + v_l}{2} \quad \text{label{vA}\tag{3}} \\ \dot{\theta} &= \frac{v_r - v_l}{2L} \quad \text{label{theta}\tag{4}} \end{aligned}$$

With the assumption that the velocity of the wheels is proportional to the voltage applied on each wheel  $V_l$  and  $V_r$  and that there is no slipping, we can write the following :

$$\begin{aligned} v_l &= c_l \cdot V_l \quad \text{label{vl}\tag{5}} \\ v_r &= c_r \cdot V_r \quad \text{label{vr}\tag{6}} \end{aligned}$$

Thus the above equations can be rewritten as :

$$\begin{aligned} v_A &= \frac{c_r \cdot V_r + c_l \cdot V_l}{2} \quad \text{label{vA2}\tag{7}} \\ \dot{\theta} &= \frac{c_r \cdot V_r - c_l \cdot V_l}{2L} \quad \text{label{theta2}\tag{8}} \end{aligned}$$

With,  $c_r$ ,  $c_l$  some constants to define for each duckiebot.

Alternatively, we can define  $c = c_r$  and  $c_l = c + \Delta c$  and we get :

$$\begin{aligned} v_A &= \frac{c \cdot (V_r + V_l) + \Delta c \cdot V_l}{2} \quad \text{label{vA3}\tag{9}} \\ \dot{\theta} &= \frac{c \cdot (V_r - V_l) - \Delta c \cdot V_l}{2L} \quad \text{label{theta3}\tag{10}} \end{aligned}$$

We get a kinematic model, that shows the relation between the linear and angular

velocity of the Duckiebot and the voltage applied to each wheel. To have our model totally defined, we only need to calculate three parameters, namely  $\$c$ ,  $\$Delta c$  and  $\$L$ . These three parameters will be calculated with odometry calibration.

## 2) Odometry formulation

The general problem definition for the odometry is to find the most likely calibration parameters given the duckiebot model [#duckiebot-modeling](#) and a set of discrete measurement from which the output can be estimated. [32] The model of the system [32] with the notations explained in Table [Table 9.2](#) can be described as :

$$\begin{aligned} \dot{x} &= f(p; x, u) \quad \text{label{eq: model1}\tag{11}} \\ y &= g(x) \quad \text{label{eq: model2}\tag{12}} \\ M &= \{ m_k = m(t_k), t_1 \dots t_n \} \quad \text{label{eq: measurements}\tag{13}} \\ \hat{y}_k &= h(m_k), k=1, \dots, n \quad \text{label{eq: output estimates}\tag{14}} \end{aligned}$$

TABLE 9.2. NOTATIONS FOR ODOMETRY CALIBRATION A DIFFERENTIAL DRIVING ROBOT

	Calibration Parameters
$p$	Model
$f(\dot{p})$	Pose
$g(\dot{p})$	Set of discrete measurements
$M$	Measurements (not necessarily evenly space in time)
$\hat{y}$	Set of output estimates

The model  $f(\dot{p})$  can be a kinematic model, constrained dynamic model or more general dynamic model. The pose  $g(\dot{p})$  can be the robot pose or the sensor pose. The measurements  $M$  can be from “internal” sensors e.g. wheel encoders, IMUs etc. or from “external” sensors such as Lidar, Infrared or camera.

For our project, our set of measurements was obtained thanks to the camera : we put the Duckiebot in front of a chessboard, and then we were able to derive the position of the Duckiebot at every image relative to the chessboard  $\big( \hat{x}_i, \hat{y}_i \big)$ .

At the same time, from our kinematic model, we could estimate the position of the Duckiebot  $(x_i, y_i)$  recursively with the formula :

$$\begin{aligned} x_{k+1} &= x_k + v_A \cdot \cos(\theta) \quad \text{label{eq:1}\tag{15}} \\ y_{k+1} &= y_k + v_A \cdot \sin(\theta) \quad \text{label{eq:2}\tag{16}} \end{aligned}$$

Because  $V_A = \frac{c_r \cdot V_r + c_l \cdot V_l}{2}$  we can express every position  $(x_i, y_i)$  of the Duckiebot with help of the parameters:

$$\begin{aligned} x_{k+1} &= x_k + \frac{c_r \cdot V_r + c_l \cdot V_l}{2} \cdot \cos(\theta) \quad \text{label{eq:xk}\tag{17}} \\ y_{k+1} &= y_k + \frac{c_r \cdot V_r + c_l \cdot V_l}{2} \cdot \sin(\theta) \quad \text{label{eq:yk}\tag{18}} \end{aligned}$$

By minimizing the position of the Duckiebot  $(x_i, y_i)$  and its theoretical position given by our model  $(x_i, y_i)$  at every time  $t_i$ , we can estimate the parameters

$\$c_r$ ,  $c_l$  and  $L$ .

We used the L2-norm :

$$\begin{aligned} \begin{aligned} & \begin{pmatrix} c_l^* \\ c_r^* \\ L^* \end{pmatrix} = \\ & \underset{\{c_l, c_r, L\}}{\operatorname{argmin}} \begin{pmatrix} x_1 - \hat{x}_1 \\ y_1 - \hat{y}_1 \\ \vdots \\ x_n - \hat{x}_n \\ y_n - \hat{y}_n \end{pmatrix}^T \begin{pmatrix} x_1 - \hat{x}_1 \\ y_1 - \hat{y}_1 \\ \vdots \\ x_n - \hat{x}_n \\ y_n - \hat{y}_n \end{pmatrix} \end{aligned} \end{aligned}$$

### 3) Dealing with uncertainty

Because our model does not take into account the dynamics of the system, and many assumptions as been made, the results we will get won't perfectly match with the reality. Assuming that the states estimation  $v_{A_i}$  and  $\dot{\theta}_i$  is accurate enough and a Gaussian distribution of the noise, we can quantify this noise by estimating its variance :

$$\begin{aligned} \sigma_v^2 &= \frac{1}{n} \sum_{k=1}^n (v_{A_i} - \tilde{v}_{A_i})^2 \quad \text{label{eq:sigmap}} \\ &= \frac{1}{n} \sum_{k=1}^n (\dot{\theta}_i - \tilde{\dot{\theta}}_i)^2 \quad \text{label{eq:sigmatilde}} \end{aligned}$$

with

$$\begin{aligned} \tilde{v}_{A_i} &= \frac{c_r^* V_{r_i} + c_l^* V_{l_i}}{2} \quad \text{label{eq:vtilde}} \\ &= \frac{c_r^* V_{r_i} + c_l^* V_{l_i}}{2L^*} \quad \text{label{eq:thetatilde}} \end{aligned}$$

## 9.4. Contribution / Added functionality

The calibration procedure consists of two parts:

- Recording Rosbag for different Duckiebot maneuvers in front of a chessboard
- Offline processing of rosbag to find odometry parameters with fit

To reproduce the results see the [operation manual \(master\)](#) which includes detailed instructions for a demo.

### 1) Recording rosbag log of Duckiebot maneuvers

For recording the Rosbag, the Duckiebot has to be placed in front of the chessboard at a distance of slightly more than 1 meter in front of the chessboard (~2 duckie tiles), as shown in the image. The heading has to be set iteratively to maximize the time the Duckiebot sees the chessboard.



Figure 9.5. The calibration setup

You then have to run the calibration procedure

 `$ rosrun calibration commands.launch veh:=robot_name`

The program will publish at a frequency of 30 Hz in the topic `robot_name/wheels_driver_node/wheels_cmd` the following commands :

- A ramp (the same increasing voltage command to the right and left wheels), of the form

$$V_l = V_r = V_{fin} \cdot \text{cfrac}(N/N_{step})$$

- No command for 10 seconds (so you can replace your Duckiebot at 1 meter of the chessboard)
- A sinusoid (a cosinus voltage command in opposite phase between the left and the right wheels) of the form

$$V_l = k_1 + k_2 \cdot \cos(\omega \cdot t)$$

$$V_r = k_1 - k_2 \cdot \cos(\omega \cdot t)$$

TABLE 9.3. NOTATIONS FOR THE VOLTAGE COMMANDS SENT

$\$V\_l, V\_r\$$	The voltages applied to the left and right wheel
$\$V\_{fin}\$$	The ramp's final voltage applied
$\$N\_{step}\$$	The number of steps of the ramp
$\$NS\$$	The number of the current step (that goes from $\$0\$$ to $\$N\_{step}\$$ at a frequency of 30 Hz)
$\$k\_1, k\_2\$$	The gains for the sinusoid command
$\$\\omega\$$	The angular velocity of the sinusoid command
$\$t\$$	The time of the voltage signal

All these parameters can be modified if the chessboard does not stay in the field of view of the camera long enough during the calibration procedure.

When the program will exit, you will have a rosbag named `robot_name_calibration.bag` in your USB drive containing the commands published and the images.

You will then have to copy on your computer the rosbag that has been taken during the maneuvers and run the calibration process with the following command :



```
$ roslaunch calibration calibration.launch veh:=robot name path:=/absolute/path/to/the/rosbag/folder/
```

(path example: path:=/home/user\_name/syisid/)

Once the command has finished, the parameters of your Duckiebot are stored in the folder

`DUCKIEFLEET ROOT/calibrations/kinematics/robot name.yaml`

## 2) Pose estimation w.r.t. chessboard

### Step 1 - Find 2D image points of chessboard in picture:

To find pattern in chess board, we use the function, `findChessboardCorners` It gives us the image points locations where two black squares touch each other in chess boards)

We also need to pass what kind of pattern we are looking, like 8x8 grid, 5x5 grid etc. For duckietown, we use 7x5 grid. (Normally a chess board has 8x8 squares and 7x7 internal corners). It returns the corner points and retval which will be True if pattern is obtained. These corners will be placed in an order (from left-to-right, top-to-bottom).

In order that the chessboard detection works reliable, one need an assymetric pattern. (e.g. [here](#)). However, the standard chessboard provided by Duckietown includes an ambiguity, i.e. if the chessboard looks the same if chessboard image is turned 180 degrees. This can result in the issue that the `findChessboardCorners` functions from right-to-left, bottom-to-top. Therefore, the origin of chessboard coordinate frame switches from the upper-left to the lower-right corners for certain

images. A fix is implemented to overcome this inconsistency in order to make it work for the default Duckietown chessboard. However it is suggested to use an asymmetric chessboard for future use, or a [ChArUco Board](#).

If the chessboard is successfully found in the image, the corners position are refined with the [cv2.cornerSubPix\(\)](#) function.

The chessboard is then projected on to the image for debugging purposes [Figure 9.6](#).

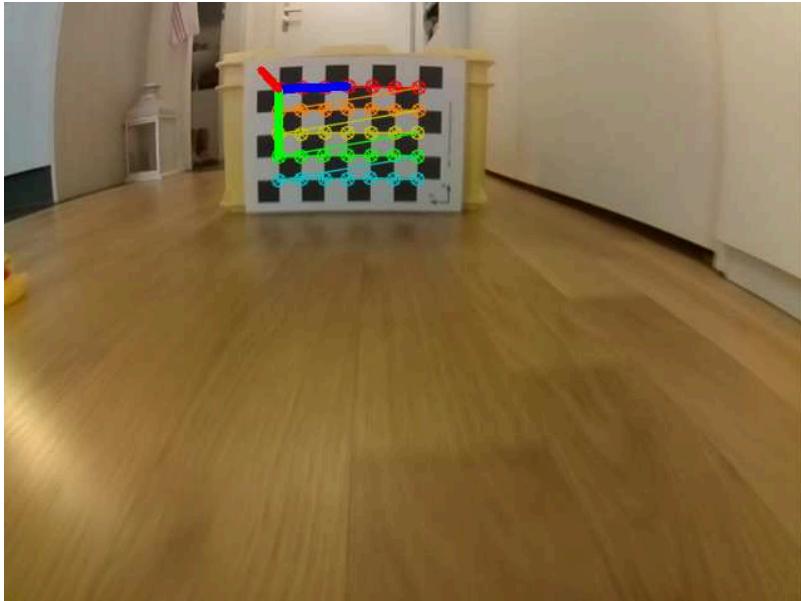


Figure 9.6. Projected chessboard

*Step 2 - Find the rotation and translation vectors of chessboard w.r.t camera:*

The next steps is to find the relation between the found image points and the objects points, the so called extrinsic parameters. Extrinsic parameters corresponds to rotation and translation vectors which translates a coordinates of a 3D point to a coordinate system. 3D points are called object points and 2D image points are called image points.

The chessboard was kept stationary at XY plane, (so  $Z=0$  always) and camera was moved accordingly. This consideration helps us to find only  $X$ , $Y$  values. Now for  $X$ , $Y$  values, we can simply pass the points as  $(0,0)$ ,  $(1,0)$ ,  $(2,0)$ , ... which denotes the location of points. In this case, the results we get will be in the scale of size of chess board square. But if we know the square size, (Duckie chessboard  $32\text{ mm}$ ), and we can pass in our case the values as  $(0,0),(32,0),(64,0),...,$  we get the results in mm.

For the pose estimation we need to know the extrinsic and intrinsic parameters of the camera. They can be loaded with the implemented `load_camera_info()` duckietown function. Intrinsic parameters are specific to a camera. It includes information like focal length ( $f_x$ , $f_y$ ), optical centers ( $c_x$ , $c_y$ ) etc. It is also

called camera matrix. It is expressed as a 3x3 matrix:

$$\begin{aligned} \text{camera} &= \left[ \begin{matrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{matrix} \right] \end{aligned}$$

We are using the function `cv.solvePnP` to calculate the rotation and translation. [solvePnP](#) finds an object pose from 3D-2D point correspondences using the RANSAC scheme. It uses the object points, the chessboard corners and the camera matrix as an input and gives the translation  $t_{\text{CamChess}}$  and rotation  $R^{\text{CamChess}}$

*Step 3 - Find the rotation and translation vectors of duckiebot w.r.t chessboard.*

We are interested to find the vehicle pose with respect to world frame. This is done by using the homography relation:

$$T^{\text{I}}_{\text{cam}} = T^{\text{I}}_{\text{chess}} T^{\text{chess}}_{\text{cam}}$$

where we use the following equations:

$$\begin{aligned} T^{\text{world}}_{\text{veh}} &= R^{\text{I}}_{\text{veh}} t_{\text{I}} \\ &\quad \text{label{eq:world}\tag{23}} \\ &\quad &= R^{\text{chess}}_{\text{cam}} t_{\text{ChessCam}} \\ &\quad \text{label{eq:world1}\tag{24}} \\ &\quad &= (R^{\text{cam}}_{\text{chess}})^T t_{\text{CamChess}} \end{aligned}$$

The yaw  $\theta$  can then be extracted from  $R^{\text{I}}_{\text{veh}}$  with the implemented `rot2euler()` function. The resulting translation can be extracted from  $t_{\text{I}}$

### 3) Fit

Finally the kinematic calibration parameters are found by an optimization. With the recorded commands and the kinematic model of the duckiebot velocities are predicted. A simple forward Euler integration gives us the position x, y and heading. This is compared to the measured positions and headings during all the times the checkerboard was detected. A nonlinear optimizer is used with a least squares objective on the error between the prediction and the measurement. The results are the kinematic calibration parameters gain, baseline and trim which best explain the driven trajectories during the calibration runs.

In order to keep track of missing measurements, the optimization is resampled to a constant sampling time which is equivalent to the frame rate. Keeping track of the timestamps of the control input and the recorded images allows us to “bridge the gap” if in some or even many pictures the checkerboard was not detected. The position prediction is done at every timestep but it is only compared to the measurement at the timesteps where measurements actually exist.

## 9.5. Formal performance evaluation / Results

## 1) Performance evaluation

---

### Offset in straight line :

We will let the Duckiebot drive straight in open loop and measure its offset after X tiles of straight lane in Duckietown. The performance metric will be the absolute position offset of the expected to the actual terminal position after the run, measured with a ruler.

### Circle test :

We will drive the Duckiebot with a constant velocity  $v_a$  and constant angular velocity  $\dot{\omega}$  in open loop on a Duckietown corner tile. We will compare the actual path with the desired path. This is done both clock and counterclockwise. The performance metric will be the absolute position offset of the expected to the actual terminal position after the run, measured with a ruler.

### Overall User friendliness

This is a quantitative test. The overall calibration procedure will have to be as simple and short as possible for the user.

## 2) Results

---

The two first tests have been made thanks to the following command line :

 `$ roslaunch calibration test.launch`

During this validation test, the Duckiebot should first drive straight for 1m (in 5s) then turn a full circle to the left (in 8s) and then a full circle to the right (in 8s). Both circles have a diameter of 50 cm.

In general, the Duckiebot is able to go straight relatively precisely : on average the Duckiebot stops its 1 meter run with a precision of more or less 6 cm and drifts for approximately 4 cm.

For the circles, the results are less precise : when the Duckiebot closes the circle, it has a deviation of around 10 cm. This could be due to the limited length and amplitude of the sine steer maneuver, which often results in the prediction fitting it better if it assumes less yaw movement, leading to a larger predicted baseline.

This large error also suggest that our kinematic model is not sufficient enough to capture precisely the movement of the Duckiebot. One should build a model that perhaps takes into account some dynamic or develop a more complex model of the wheels. In this project, we have made the assumption that they don't slip which appear to be not the case especially when the Duckiebot turns.

An other source of error could come from our measurements : when the Duckiebot runs the sinusoidal maneuvers, its position gets more noisy and hence the

position estimation becomes less precise. Other methods of localization should be tried.

During the calibration, if it is clearly visible in the plots that the predicted movement does not match the measurements or if the duckiebot almost doesn't turn during the sine steer experiment, the amplitude of the sine command can be changed in the launch file for the calibration runs. Otherwise, the baseline can be adjusted manually in order to achieve approximately a full circle during the test.

Nevertheless, the drift when the Duckiebot goes straight is tolerable, and for this movement our model and the measurements taken are sufficient.

**Overall User friendliness :**

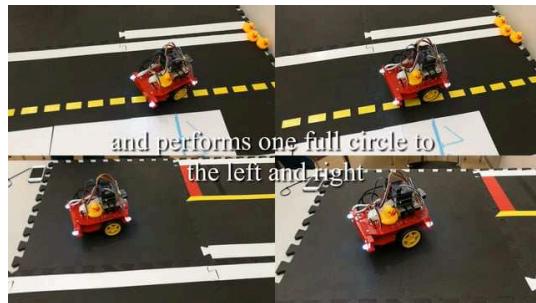


Figure 9.7. Demo of the calibration procedure

Thanks to the odometry calibration, the user has only to place its Duckiebot in front of the chessboard and type a command. But because of computational power restrictions, he has then to transfer the Rosbag from the Duckiebot to its computer before launching the calibration and then sending the calibration file to its Duckiebot again. These manipulations are not straightforward and should be improved in the future.

## 9.6. Future avenues of development

- Dynamic model of the Duckiebot
  - Since the kinematic model seem to be insufficient for the rotation, a dynamic model should be developed.
- Caster wheel identification
  - The initial aim was to include the kinematics of the caster wheel, however due to time constraint, we sticked to the roller wheel.
- Position estimation based on april tags
  - Because of noisy position measurements with the chessboard, some other methods could be used, as the april tags. It could even be possible to put several april tags on a track, so that we do not have to replace the Duckietown at the beginning of the track after each movement.
- Simultaneous odometry and camera calibration
  - To take even more the human out of the loop, a automatic camera and odometry calibration could be implemented

# The Controllers: preliminary report

## 10.1. Part 1: Mission and scope

### 1) Mission statement

Make lane following more robust to model assumptions and Duckietown geometric specification violations and provide control for a different reference control.

### 2) Motto

IMPERIUM ET POTESTAS EST  
(With control comes power)

### 3) Project scope

*What is in scope:*

- Control Duckiebot on straight lane segments and curved lane segments.
- Robustness to geometry (width of lane, width of lines)
- Detection and stopping at red (stop) lines
- Providing control for a given reference  $d$  for avoidance and intersections (but for intersections, we additionally need the pose estimation and a curvature from the navigators team)

*What is out of scope:*

- Pose estimation and curvature on Intersections (plus navigation / coordination)
- Model of Duckiebot and uncertainty quantification of parameters (System Identification)
- Object avoidance involving going to the left lane
- Extraction and classification of edges from images (anti-instagram)
- Any hardware design
- Controller for Custom maneuvers (e.g. Parking, Special intersection control)
- Robustness to non existing line

*Stakeholders:*

**System Architect**

She helps us to interact with other groups. We talk with her if we change our project.

**Software Architect**

They give us Software guidelines to follow. They give a message definition.

**Knowledge Tzarina Duckiebook**

**Anti-Instagram** They provide classified edges (differentiation of centerline, outer lines and stop lines)

Direction of the edges (background to line vs. line to background)

**Intersection Coordination (Navigators)** They tell where to stop at red line. We give a message once stopped. They give pose estimation and curvature (constant) to navigate on intersection. We provide controller for straight line or standard curves.

**Parking** They tell where to stop at red line. We give a message once stopped

**System Identification** They provide model of Duckiebot.

**Obstacle Avoidance Pipelines (Saviors)** They provide reference  $d$ .

**SLAM** They might want to know some information from our pose estimation (e.g. lane width or theta).

## 10.2. Part 2: Definition of the problem

### 1) Problem statement

---

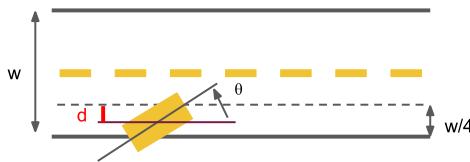
We must keep the Duckiebots at a given distance  $d$  from center of the lane, on straight and curved roads, under bounded variations of the city geometric specifications.

Geometric specifications:

- Nominal lane width
- Tape width (white, yellow, red)
- Spacing between yellow dashed line
- Curvature of the curves

Given at every time step a reference  $d$  (the reference  $\theta$  gets chosen in a way to match the reference  $d$ ):  
$$\begin{aligned} q_r(t) = & [x_r(t), y_r(t), \theta_r(t)]^T \\ & \rightarrow [d_r(t), \theta_r(t)] \end{aligned}$$

The following image shows the definition of the parameters  $d$  and  $\theta$ .



( $d$ : distance perpendicular to the lane.  $\theta$  is defined in the center of the lane (see definition in duckumentation))

( $\theta$ : angle between the lane and the robot body frame. )

and given a **model of the system**,

define a control action:

- the heading and velocity of the center between the wheels of robot, leading to a sequence of motor commands

at every time step, such that the pose (estimate of the pose) converges to the target.

#### **Performance:**

- Steady state within a tile
- Never leaves the lane
- Small steady state error

#### **Robustness to slight changes in:**

- Model parameters
- Width of the lane
- Width of the lines
- Curvature of the road

#### **2) Assumptions**

- 
- There is a reason for the caster wheel
  - A system model of the Duckiebot is provided
  - The system model parameters for every duckiebot are given
  - A set of classified lane edges are given with a certain frequency, latency, resolution and a maximum number of false positives and maximum number of misclassifications.

- Anti-instagram people do low level vision -> extract lines in image space
- Only small deviation from the specified geometric values
- Surface properties of Duckietown tiles are similar in each Duckietown
- Bounded initial pose, when driving straight no wheel of the Duckiebot should touch any line within 25cm

### 3) Approach

---

- Benchmark actual system → identify bottlenecks (in estimation and control)
- Identify bottlenecks by modifying different parts of the system and adding them each at a time and have a look at how much is the improvement → Make new branches for each of those modifications
- We want to improve the pose estimation by applying a particle filter. To measure the impact of the improved pose estimation, we will design a test procedure.
- One possible test procedure: Set a calibrated duckiebot to many known points on straight lanes and curved lanes. Save the information of these actual poses (measured by hand) together with the images taken by the Duckiebot's camera in the respective poses. On this data, different anti-instagram methods and different pose estimations can be run and evaluated directly, without any physical duckiebot nor Duckietown.
- We want to improve the parameters of the current controller by tuning it experimentally.
  - We want to increase the frequency of the controller update.
  - We want to handle actuator saturation, if we adding an I part to the controller.

### 4) Functionality provided

---

Drive on straight lane and curves without large deviations from the center of the lane.

### 5) Resources required / dependencies / costs

---

Hardware resources:

- Tapes for lanes
- Tiles to make different straight lanes and curved lanes
- Timer
- Functional Duckiebot

Dependencies: see assumptions

We assume to have image space line segments extracted and classified from images.

- frequency
- latency
- accuracy (resolution)
- maximum false positives
- maximum misclassification error (confusion matrix)

### 6) Performance measurement

---

Drive on the track for one minute and count the number of times the bot touches the

side or center line. Repeat this 5 times.

**Metrics** *Error from the reference distance d when driving straight. - Mean and variance of 5 experiments* Estimate of lane width. - Estimate lane width and compare to measurement *Estimate road curvature. - Estimate curvature and compare to measured radius of curve* Speed - is a control variable. *Robustness to initial pose. - Run lane following using 5 different initial poses* Transient error after curved section (e.g. dies in one tile length). - 5 experiments of measuring the error d when driving straight after a curved segment  $\rightarrow$  Did the transient error die? *Robustness to the curvature. - Run curve following on 5 lanes made of different combinations of curve tiles (left-left-right, left-right-left, ... )* Robustness to lane specifications - Run lane following on 5 lanes with different lane width when driving straight

## 10.3. Part 3: Preliminary design

### 1) Modules

---

Estimation of Position:

- Input: segments detected by Anti-Instagram-Filter
- Output:
- Distance from center of lane,
- Heading angle,
- Curve or straight lane
- Curvature

Controller:

- Input: state (Distance from center of lane, heading angle, other) by an Estimator
- Output: Control Output to motors

### 2) Interfaces

---

- Anti-instagram: Labelled (centerline, outer line, stop line) edges with color and direction in image plane. The messages is defined already.
- Odometry calibration: get the model parameters. Yaml file
- Obstacle avoidance: get reference distance d from center of lane.. Zero assumed otherwise.
- Estimator: state vector at regular intervals.
- Control: Motor voltages at regular intervals

### 3) Specifications

---

We do not need to change the Duckietown specifications.

### 4) Software modules

---

- Estimator: NODE. There is a markovian approach. A particle filter should be implemented
- Controller: NODE. there is a P controller. A feedforward should be implemented. Pure-pursuit, or simple FeedForward.

- Outlier rejection: NODE (or part of estimator). there is nothing. A system has to be designed to detect edges that clearly don't belong to the line.
- Automated testing: NODE. There is nothing. A system should be implemented to test estimation from recorded data. It should be easy to update this data.

## 10.4. Part 4: Project planning

### 1) Timeline

Date	Task Name	Target Deliverables
17/11/	Kick-Off	Preliminary Design Document
17		
24/11/	Get familiar with state of the art	Benchmark state of the art, Identify
17		bottlenecks
01/12/	Theoretical derivation of Controller	
17	and Estimator	
08/12/	Implementation of Controller and Esti-	
17	mator	
15/12/	Benchmark new implementation	Performance measure of new imple-
17		mentation
22/12/	Buffer	
17		
29/12/	Documentation	Duckuments
17		
05/01/		End of Project
18		

### 2) Data collection

Take rosbag logs every time.

Rosbag:

- Image
- Edges from Anti-Instagram
- Motor control values

### 3) Data annotation

Curvature of road.

*Relevant Duckietown resources to investigate:*

Vision odometry Lane detection Anti instagram

*Other relevant resources to investigate:*

[Particle Filter coded in python and useful intro to the subject](#)

### 4) Risk analysis

Risk	Likelihood	Impact	Risk response	Actions required
			Strategy	
Cannot estimate curvature	2	5	mitigate	Start early with testing thresholds for curvature identification
Cannot define distance to curve	2	4	mitigate	Try various methods to identify the distance to curve
Duckiebot leaves the lane after curve (current situation)	2	5	mitigate	
Cannot handle the inputs given by other teams	4	4	mitigate	Get more information from Sonja, Talk to other teams, Clear comments in the code for easier problem detection

## UNIT L-11

### The Controllers: Intermediate Report

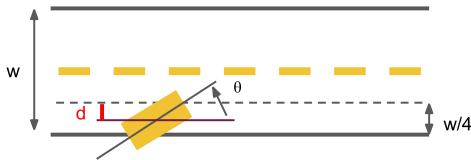


TABLE 11.1. INTERMEDIATE REPORT SUPERVISORS

System Architects	Sonja Brits, Andrea Censi
Software Architects	Breandan Considine, Liam Paull
Vice President of Safety	Miguel de la Iglesia, Jacopo Tani
Data Czars	Manfred Diaz, Jonathan Aresenault

TABLE 11.2. CONVENTIONS USED IN THE FOLLOWING DOCUMENT

Variable	Description
\$d_{ref}\$	Reference distance from center of lane
\$d_{act}\$	Actual distance from center of lane
\$d_{est}\$	Estimated distance from center of lane
\$\theta_{act}\$	Actual angle between robot and center of lane
\$\theta_{est}\$	Estimated angle between robot and center of lane
\$c_{act}\$	Actual curvature of lane
\$c_{est}\$	Estimated curvature of lane
\$c_{ref}\$	Reference curvature of the path to follow
\$v_{ref}\$	Reference velocity



## 11.1. Part 1: System interfaces

### 1) Logical architecture

---

#### Desired functionality

We assume that the Duckiebot is placed on the right lane of the road within a defined boundary (as described in our preliminary design document) for the initial pose. By starting the lane following module it should begin to follow the lane, whether it is straight or curved, until we stop the lane following module. The module consists of two parts, a pose-estimator part and a lane-controller part. We will briefly describe these two modules:

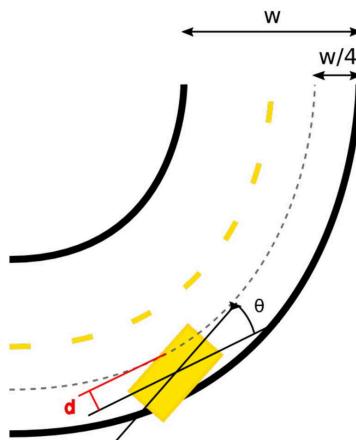


Figure 11.1. Pose of Duckiebot in a curve element.

- **Pose-estimator:** Estimates distance  $d_{\text{est}}$  from the center of the lane and the angle  $\theta_{\text{est}}$  with respect to the center of the lane as near as possible to the

actual values  $d_{act}$  and  $\theta_{act}$ . In a curve,  $\theta_{act}$  is the angle between the centerline of the Duckiebot and the tangent to the centerline of the lane at the corresponding position (where the origin of the robot frame (center of the wheel axis) is closest to the centerline of the lane). Additionally, the estimator estimates the curvature  $c_{est}$  of the lane. Further if possible, the estimator will approximate the lane width and the width of the side lines to be robust with respect to the geometric specifications of Duckietown. (The curvature  $c_{est}$  was not estimated in the previous estimator.)

- **Lane-controller:** Given the pose ( $d_{est}$ ,  $\theta_{est}$ ) and curvature ( $c_{est}$ ) estimation and a reference  $d_{ref}$ , the lane-controller will control the Duckiebot along this reference. The controller only accepts  $d_{ref}$  which allow the Duckiebot to stay in the right lane. In other cases, the Duckiebot will be stopped to avoid accidents and a corresponding flag `flag_obstacle_emergency_stop` is set. The lane-controller further strictly limits the velocity of the Duckiebot, for values see next section. The lane-controller takes the velocity at all time from implicit coordination except when another team demands a lower value. In case no velocity is received, we set a default constant velocity by ourselves.

The following diagram shows the input the controller node needs to control for other teams.

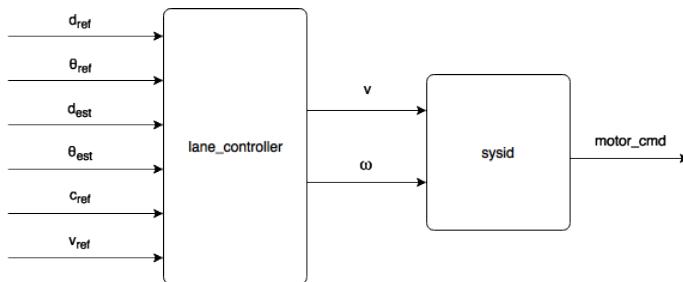


Figure 11.2. Diagram showing values needed by the controller if used by other teams.

Special events:

- **Detection of red line:** After the stop line has been detected by the `stop_line_filter_node`, it sends an `at_stop_line` message (flag `at_stop_line`), the controller will continually slow down the velocity of the Duckiebot and stop between 16 to 10 cm from the center of the red line and with an angle  $\theta_{act}$  of  $\pm 10^\circ$  (requirements given by Explicit Coordination Team). Furthermore the Duckiebot will stop in the center of the lane within a range of  $\pm 5$  cm.

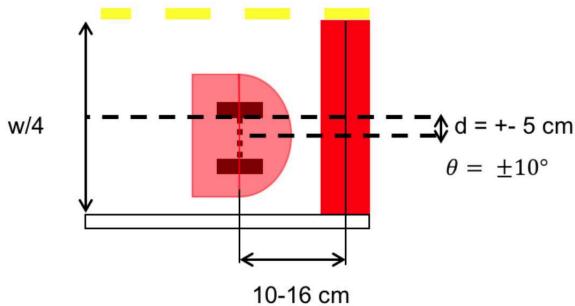


Figure 11.3. Pose and distance range in front of red line.

- **Intersection:** When the Parking team set the flag `flag_at_intersection true` (because the Duckiebot is at a stop line and there is no april tag for a parking lot), we will stop the pose\_estimator of the lane-following module and listen to data provided by the Navigators. It consists of the curvature `$c_{ref}$` the Duckiebot needs to follow, as well as a reference `$d_{ref}$` (which should be zero in case the Duckiebot needs to drive on the path with given curvature), the estimates of our distance `$d_{est}$` and angle `$\theta_{est}$` with respect to the path and a desired velocity `$v_{ref}$`. Everything on the intersection except of using our standard lane following controller is out of our scope. The pose\_estimator of the lane-following module will start again after the intersection, triggered when the flag `flag_at_intersection` is turned `false`.
- **Obstacle avoidance:** Once `flag_obstacle_detected` is set `true` by the Savior Team, they will continuously send us references `$d_{ref}$` to lead us around the obstacle, as well as a desired velocity `$v_{ref}$`. For better control performance, the velocity can be set lower than the usual velocity. The Saviors will start to send references when the Duckiebot still has a distance to the obstacle of at least 20-30cm to make sure the controller is able to react enough in advance. Out of scope is the controlled obstacle avoidance involving leaving the right lane. This case is determined by a stop flag (`flag_obstacle_emergency_stop`) sent from the Saviors module and the Duckiebot will stop. Since there is also a stop flag received from the `stop_line_filter_node`, we will introduce priorities for the several flags received to decide how the Duckiebot should behave.
- **Parking:** At a stop line, if a parking-lot april tag is detected (by the Parking team), the parking flag `flag_at_parking_lot` will be set `true` (otherwise the `flag_at_parking_lot` would be set `false` and the `flag_at_intersection` would be set `true`). After this the Parking team will take over responsibility. They will first calculate a path using RRT (Rapidly-Exploring Random Tree). In case, they set the `flag_parking_stop` to `false`, the lane controller will take over the control along this precalculated path. For this, the parking team needs to send the curvature `$c_{ref}$` the Duckiebot needs to follow, as well as a reference `$d_{ref}$` (which should be zero in case the Duckiebot needs to drive on the path with given curvature), the estimates of our distance `$d_{est}$` and angle `$\theta_{est}$` with respect to the path and a desired velocity `$v_{ref}$`. In case we need to stop, the Parking team will set the `flag_parking_stop` `true` again. Driving backwards is not in our scope. After leaving the parking lot, we will take over estimation and control again, unless directly after the parking lot is an intersection, which would be handled by the Navigators (after Explicit and/or Implicit Coordination).
- **Implicit Coordination:** If Implicit Coordination is running, they send to us the desired reference velocity `$v_{ref}$` at all time and set the flag `flag_implicit_coordination`.

- **Fleet-level Planning:** As part of simulating pick-up / drop-off of customers, the Fleet-level Planning team will want to stop the Duckiebot at a certain distance from the center of the lane  $d_{\text{ref}}$ . Therefore they give us the desired  $d_{\text{ref}}$  and a declining reference velocity  $v_{\text{ref}}$  until the desired full stop.

## Target values

The Duckiebot should run at a reduced velocity of 0.2 m/s for optimal controllability. The reason for the limited velocity is the low image update frequency which limits our pose estimation update, hence a lower velocity enhances the performance of our lane-follower module. Since not every Duckiebot has the same gain set, we will pass the desired velocity to team System Identification. Their module will convert the demanded velocity to the according input voltages for the motors.

Our goal is to control the deviation from the middle of the lane  $d_{\text{act}}$  smaller than  $\pm 2\text{cm}$ . The estimator should estimate our pose with  $d_{\text{est}}$  and  $\theta_{\text{est}}$  with an accuracy of  $\pm 1\text{cm}$ . Further, the Duckiebot will stop in the center of the lane within a range of  $\pm 2 \text{ cm}$ , although a larger range of  $\pm 5 \text{ cm}$  is given by the explicit coordination team.

Whenever we detect a red line, we will slow down the Duckiebot and stop between 16 to 10 cm from the center of the red line and with an angle  $\theta_{\text{act}}$  of  $\pm 10^\circ$ , see caption 4.

In case the flag\_obstacle\_detected is activated by the Saviors, they will provide us with the input described above to avoid the obstacle without leaving the lane. Otherwise they activate the flag\_obstacle\_emergency\_stop and we will need to stop the Duckiebot within 5 cm from the position at which the flag\_obstacle\_emergency\_stop is received (requirement by Saviors).

## Assumptions

We assume the following modules will behave in the described manner:

- **Savior:** They will detect obstacles on the road and are able to generate  $d_{\text{ref}}$  that allows to avoid the object without leaving the lane or touching any object (safety for the Duckies!). They will set the flag\_obstacle\_detected 20-30cm in front of the obstacle, so we have enough time to avoid the obstacle. They are able to decide if avoidance of an obstacle is feasible or they have to set the flag\_obstacle\_emergency\_stop.
- **Anti-instagram:** They can compensate the colors for different ambient light conditions and allow for good edge detection robust to changing light conditions. In future, they could optionally help by introducing an area of interest of the image to process. Irrelevant image data could be filtered out to speed up the image processing pipeline. If Anti-Instagram could publish the area of interest as a node, we could use it in the estimation part to remove all visual clutter in the line segments.
- **Line detection:** Relevant line segments of the side, center and stop line are detected without introducing an exceeding amount of false detections and passed on in a SegmentList, classified including direction (from background do line or vice versa).
- **Navigators:** They are able to generate a path along the intersection and deliver accurate pose estimates which allow for proper working of the lane controller.
- **Parking:** They are able to generate a path from the parking entrance to a free parking space and deliver accurate pose estimates which allow for proper working of

the lane controller. They take care of any backward driving without using the lane controller.

- **Implicit Coordination:** They are able to generate a reference velocity that will not result in a crash with another Duckiebot. They will provide enough distance to the Duckiebot in front, to allow the pose\_estimator of the lane-following module and the stop\_line\_filter\_node to work properly.
- **Fleet-level Planning:** They provide a profile of reference distance from the center of the lane  $d_{ref}$  and velocity  $v_{ref}$ , such that they stop at their desired location to pick-up / drop-off their customer.

## 2) Software architecture

---

### *Lane Filter Node*

The Lane Filter Node will, in addition to the existing fields, also estimate the curvature.

In the following table, published topics are listed:

TABLE 11.3. PUBLISHED TOPICS BY LANE FILTER NODE

Topic	Max Latency
lane_pose	15 ms
belief_img	2 ms
entropy	negligible
in_lane	negligible
switch	negligible

In the following table, subscribed topics are listed:

TABLE 11.4. PUBLISHED TOPICS BY LANE FILTER NODE

Topic	Max Latency
segment_list	25 ms
velocity	egligible
car_cmd (not yet)	negligible

Total latency from image taken, processed through anti-instagram, up until setting the motor control command is on average 140 ms.

### *Lane Controller Node*

In the following table, published topics are listed:

TABLE 11.5. PUBLISHED TOPICS BY LANE CONTROLLER NODE

Topic	Type	Max Latency
car_cmd	duckietown_msgs/Twist2DStamped	negligible

In the following table, subscribed topics are listed:

TABLE 11.6. SUBSCRIBED TOPICS BY LANE CONTROLLER NODE

Topic	Type	Max Latency
lane_pose	duckietown_msgs/ LanePose	15 ms
lane_pose_intersection_navigation	duckietown_msgs/ControlMessage_	20 ms
lane_pose_obstacle_avoidance	duckietown_msgs/ControlMessage_	20 ms
lane_pose_parking	duckietown_msgs/ControlMessage_	25 ms
stop_line_reading	duckietown_msgs/StopLineReading	negligible
implicit_coordination_velocity	duckietown_msgs/ControlVelocity	negligible
Flags defined in table below	BoolStamped	negligible

#### Flags received by other nodes

These following flags are received from other modules. While one of these flags is *true*, the Duckiebot will behave according to the descriptions in the system architecture section.

TABLE 11.7. FLAGS RECEIVED BY OTHER MODULES

flag_at_stop_line	<i>True</i> when the distance to stop line is below a predefined distance.
flag_stop_line_deteced	<i>True</i> when number of detected red segments are above a threshold
flag_at_intersection	<i>True</i> when at intersection. This flag is passed to us by the Parking team.
flag_obstacle_detected	<i>True</i> when obstacle is in the lane. This flag is passed to us by the Saviors.
flag_obstacle_emergency_stop	<i>True</i> when it is not possible to avoid the obstacle without leaving the right lane.
flag_at_parking_lot	<i>True</i> when stopping at an intersection and the april tag for the parking lot is detected by the Parking team.
flag_parking_stop	Per default = <i>true</i> . If <i>false</i> , the lane-follower will move along the given trajectory on the parking lot.
flag_implicit_coordination	<i>True</i> when implicit coordination is running, in this case we listen to the velocity published by them.

#### Structure of the received messages with type *duckietown\_msgs/LanePose*

The following table defines the structure of the pose messages the Lane Controller Node receives.

TABLE 11.8. STRUCTURE OF POSE MESSAGE TO BE USED.

Field	Abstract Data Type	SI Units	Description
header	Header	-	Header
d	float32	\$m\$	Estimated lateral offset
sigma_d	float32	\$m\$	Variance of lateral offset
phi	float32	\$rad\$	Estimated Heading error
sigma_phi	float32	\$rad\$	Variance of heading error
c	float32	\$1/m\$	Reference curvature
status	int32	-	Status of Duckiebot 0 if normal, 1 if error is encountered
in_lane	bool	-	In lane status

Structure of the received messages with type *duckietown\_msgs/ControlMessage*

The following table defines the structure of the control messages the Lane Controller Node receives from all the teams who want to send commands to our controller.

TABLE 11.9. STRUCTURE OF CONTROL MESSAGE TO BE USED.

Field	Abstract Data Type	SI Units	Description
header	Header	-	Header
d_est	float32	\$m\$	Estimated lateral offset
d_ref	float32	\$m\$	Reference lateral offset
phi_est	float32	\$rad\$	Estimated Heading error
phi_ref	float32	\$rad\$	Reference heading
c_ref	float32	\$1/m\$	Reference curvature
v_ref	float32	\$m/s\$	Reference Velocity

Structure of the received messages with type *\_duckietown\_msgs/ControlVelocity*

The following table defines the structure of the control messages that mostly the implicit coordination group will send us to control the velocity.

TABLE 11.10. STRUCTURE OF VELOCITY MESSAGE TO BE USED.

Field	Abstract Data Type	SI Units	Description
header	Header	-	Header
v_ref	float32	\$m/s\$	Reference Velocity

Information to be provided from each team

The following table defines the information we need from each team that uses the lane controller.

TABLE 11.11. INFORMATION NEEDED FROM EACH TEAM.

Team	Information
Saviors	$d_{ref}$ , $v_{ref}$
Navigators	$d_{est}$ , $d_{ref}$ , $\theta_{est}$ , $c_{ref}$ , $v_{ref}$
Parking team	$d_{est}$ , $d_{ref}$ , $\theta_{est}$ , $c_{ref}$ , $v_{ref}$
Implicit Coordination	$v_{ref}$
Fleet-level Planning	$d_{ref}$ , $v_{ref}$

### Stop Line Filter Node

In the following table, published topics are listed:

TABLE 11.12. PUBLISHED TOPICS BY STOP LINE FILTER NODE

Topic	Max Latency
stop_line_reading	negligible
flag_at_stop_line	negligible

In the following table, subscribed topics are listed:

TABLE 11.13. PUBLISHED TOPICS BY STOP LINE FILTER NODE

Topic	Max Latency
segment_list	25 ms
lane_pose	15 ms

## 11.2. Part 2: Demo and evaluation plan

### 1) Demo plan

---

The main goal is to demonstrate the improved curve driving. For this purpose, we will run two Duckiebots with different versions of Estimator and Controller running on the same test track, see picture. With the old lane following module, the Duckiebot corrected its position when it was not parallel to the white lines and the correction caused an overshoot. The new module allows the Duckiebot to detect an upcoming curve early and the controller will be adjusted to the curve. Our module also improves the execution of other tasks such as stopping at an intersection or in front of a Duckie. Further, the Duckiebot will drive with an offset of at most 2 cm. While running in an endless loop, we can also show that the steady state error has been minimized.

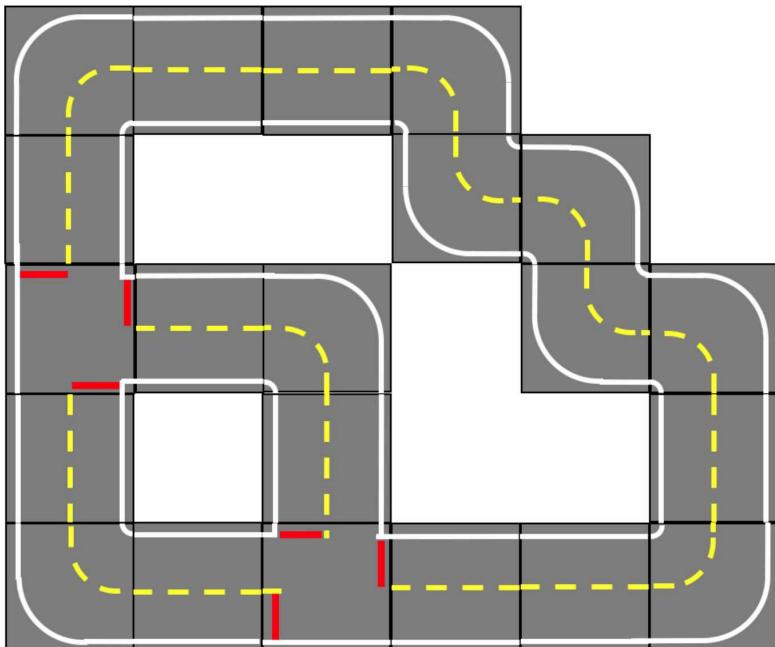


Figure 11.4. Possible map for lane following demo.

#### *What hardware components do you need?*

For our demo we want to build a small Duckietown test track, see picture. Thus we need about 21 tiles, DUCKIEtape and the usual Duckietown decoration.

#### 2) Plan for formal performance evaluation

We will run all of the tests 5 times.

- **Stopping in front of red line:** in the demo mode, we will let the Duckiebot drive to a red line and measure the distance between the center of the stop line to the wheel axis of the Duckiebot after it stopped.
- **Pose Estimation:** We will manually drive the Duckiebot along a pre-taped route in the Duckietown, of which the  $d$  is equal to zero, and collect multiple sets of data with the old pose estimator running and then with the new pose estimator running. From the bag data, we can analyse the estimation deviation from both estimators.
- **Offset minimization in straight lanes:** In the demo mode, we will let the Duckiebot drive down a straight lane. At the end of the straight lane, we will fix two laser pointers pointing to a wall and count how many times we can't see the light point on the wall while driving. In the case, a light dot disappears, the Duckiebot has left the target range. The distance between the laser pointers will be the width of a Duckiebot plus 4 cm.
- **Performance of the controller on curvy roads:** For curvy roads we will check the visual performance of the line following by counting how many times the Duckiebot touches a line on the S-curve section of the Zurich Duckietown. Additionally we want to compare the control motor commands in the curve section with the commands of the old controller and verify their smoothness.
- **Performance of the controller on lanes with dynamic width:** If we altered the

controller to be more robust for non nominal appearance, we eventually check if the Duckiebot is robust to changes in lane specifications, such as narrower lanes or different width of lane tapes. We will let the Duckiebot drive on modified tiles and check the performance of estimation and lane following.

### 11.3. Part 3: Data collection, annotation, and analysis

#### 1) Collection

---

*How much data do you need?*

For every future step we need fixed logs and logs from driving. Baseline has been set and logs have been taken with the current implementation of the code to evaluate current performance.

*How are the logs to be taken? (Manually, autonomously, etc.)*

- **Manually:** Static logs with different values for  $d_{act}$  and  $\theta_{act}$  have been taken. These can be used for a unit test of the estimator.

Bot	Institution	Timestamp	Lane position	d: distance [cm]	phi: angle [deg]
yaf	ETHZ	2017-11-24-17-36-01	straight	0	0
yaf	ETHZ	2017-11-24-17-45-12	straight	1	0
yaf	ETHZ	2017-11-24-17-40-10	straight	5	0
yaf	ETHZ	2017-11-24-17-40-48	straight	10	0
yaf	ETHZ	2017-11-24-17-46-22	straight	-1	0
yaf	ETHZ	2017-11-24-17-41-43	straight	-5	0
yaf	ETHZ	2017-11-24-17-42-23	straight	-10	0
yaf	ETHZ	2017-11-24-17-58-56	straight	0	5
yaf	ETHZ	2017-11-24-18-00-42	straight	0	10
yaf	ETHZ	2017-11-24-18-02-23	straight	0	30
yaf	ETHZ	2017-11-24-18-12-22	straight	0	60
yaf	ETHZ	2017-11-24-17-53-30	straight	0	-5
yaf	ETHZ	2017-11-24-17-54-40	straight	0	-10
yaf	ETHZ	2017-11-24-17-56-28	straight	0	-30
yaf	ETHZ	2017-11-24-18-06-10	straight	5	10
yaf	ETHZ	2017-11-24-18-05-08	straight	5	-10
yaf	ETHZ	2017-11-24-18-08-27	straight	-5	10
yaf	ETHZ	2017-11-24-18-07-11	straight	-5	-10
<b>yaf</b>	<b>ETHZ</b>	<b>2017-11-24-18-14-36</b>	<b>curve</b>	<b>0</b>	<b>0</b>
yaf	ETHZ	2017-11-24-18-16-35	curve	1	0
yaf	ETHZ	2017-11-24-18-17-27	curve	5	0
yaf	ETHZ	2017-11-24-18-18-17	curve	10	0
yaf	ETHZ	2017-11-24-18-22-35	curve	-1	0
yaf	ETHZ	2017-11-24-18-19-15	curve	-5	0
yaf	ETHZ	2017-11-24-18-20-42	curve	-10	0
yaf	ETHZ	2017-11-24-18-28-33	curve	0	5
yaf	ETHZ	2017-11-24-18-29-16	curve	0	10
yaf	ETHZ	2017-11-24-18-30-43	curve	0	30
yaf	ETHZ	2017-11-24-18-24-31	curve	0	-5
yaf	ETHZ	2017-11-24-18-25-24	curve	0	-10
yaf	ETHZ	2017-11-24-18-27-27	curve	0	-30

Figure 11.5. Table of static logs taken to evaluate the estimator.

- **Autonomous:** Logs should also be taken during lane-following-demo to evaluate the estimator and control performance (see performance evaluation).

*Do you need extra help in collecting the data from the other teams?*

We do not need data from other teams and therefore do not need help.

## 2) Annotation

---

*Do you need to annotate the data?*

No, because we will receive the needed edges from the Anti-Instagram group.

*At this point, you should have tried using [thehive.ai](#) to do it. Did you?*

In autonomous driving thehive.ai is mostly used to annotate images in order to detect and recognize obstacles and for semantic segmentation. As our project does not rely on these information, we do not need it.

*Are you sure they can do the annotations that you want?*

Probably they could, but so do we with our estimator. There are no obstacles or Duckies to annotate.

## 3) Analysis

---

We don't need data annotation since we can do all the benchmarking by our own. We are not involved in any obstacle detection so we do not need any obstacles annotated.

*Do you need to write some software to analyze the annotations?*

No, because we do not use the annotations of thehive.ai.

*Are you planning for it?*

No

# UNIT L-12

## The Controllers: final report

### 12.1. The final result



Figure 12.1. The Controllers Demo Video

See the [operation manual \(master\)](#) to reproduce these results.

### 12.2. Mission and Scope

IMPERIUM ET POTESTAS EST

(With control comes power)

Our Mission was to make lane following more robust to model assumptions and Duckietown geometric specification violations and provide control for a different reference.

## 1) Motivation

---

In Duckietown, Duckiebots are cruising on the streets and also Duckies are sitting on the sidewalk waiting for a Duckiebot to pick them up. To ensure a baseline safety of the Duckiebots and the Duckies, we have to make sure the Duckiebots are able to follow the lane (or a path on intersections and in parking lots) and stop in front of red lines. For instance, the Duckiebot is driving on the right lane. It should never cross the centerline to avoid any collisions with an oncoming Duckiebot.

The overall goal of our project is to stay in the lane while driving and stopping in front of a red line. Due to the tight time plan, we focused on improving the existing code and benchmarking the tasks. In order to let the Duckiebot drive to a given point, the robot has to know where it is in the lane, calculate the error and define a control action to reach the target. To retrieve the location and orientation information, a pose estimator is implemented. The estimator receives line segments from the image pipeline with information about line tape colour (white, yellow, red) ([Figure 12.2](#)) and whether the segment is on the left or right edge of the line tape. Using those information, we determine if the Duckiebot is inside or outside the lane, how far it is from the middle of the lane and at what angle it stands. The relative location to the middle of the lane and the orientation of the Duckiebot are passed on to the controller. In order to minimize the error, the controller calculates the desired velocity and heading of the Duckiebot using the inputs and controller parameters. The importance of our project in the framework “Duckietown” was obvious, as it contains the fundamental functionality of autonomous driving. Furthermore, many other projects relied on our project’s functionality such as obstacle avoidance, intersection navigation or parking of a Duckiebot. We had to ensure that our part is robust and reliable.



Figure 12.2. Image with Line Segments,  $d_{\text{err}}$  and  $\phi_{\text{err}}$  displayed.

## 2) Existing solution

---

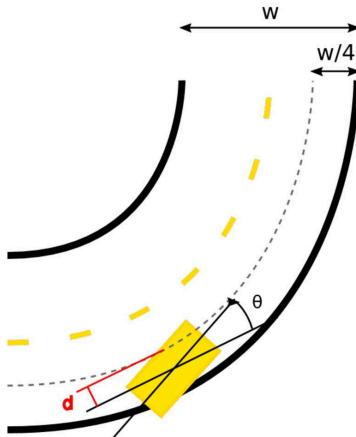


Figure 12.3. Pose of Duckiebot in a curve element.

From last year's project, the baseline implementation of a pose estimator and a controller were provided to us for further improvement. The prior pose estimator was designed to deliver the pose for a Duckiebot on straight lanes only. If the Duckiebot was in or before a curve and in the middle of the lane, the estimated pose showed an offset  $d$ , see definition of  $d$  in figure below. The existing controller worked reasonably on straight lines. Although, due to the inputs from the pose estimator to the controller, the Duckiebot overshot in the curves and crossed the left/right line during or after the curve.



Figure 12.4. Old vs. new controller

### 3) Opportunity

In the previous implementation, the lane following was not guaranteed on curved lane segments, because the Duckiebot often left the lane while driving in the curve or after the curve. Although the Duckiebot sometimes returned correctly to the right lane after leaving it and continued following the lane, robust lane following was not provided. On straight lanes, the Duckiebot frequently drove with a large static offset from the center of the lane. The previously implemented pose estimator and controller left room for improvement.

Further, the previous lane controller was not benchmarked for robustness nor for performance, therefore we defined various tests to benchmark the previous controller and our updated solution. During the project, we continuously tested our code with the entire lane following pipeline for best practice and compared our implemented solution to the existing one to record the improvement.

Our Scope was first of all to enable controlled autonomous driving of the Duckiebot on straight lane segments and curved lane segments which are in compliance with the geometry defined in [Duckietown Appearance Specifications \(master\)](#). Further, we wanted to enhance the robustness to arbitrary geometry of lane width or curvature of the lane to ensure the autonomous driving of the Duckiebot in an individual Duckietown setup. We also tackled the detection and stopping at red (stop) lines. With the previous implementation, the Duckiebot stopped rather at random points in front of the red line. We wanted to improve the implementation, to ensure a stop in the middle of the lane, in a predefined range and at a straight angle to the red line. As the Duckietown framework is a complex system involving various functionalities such as obstacle avoidance and intersection navigation, our lane following pipeline provides the basic function for those functionalities and it has to be able to interact with the modules of other teams. Hence, it was also our duty to design an interface which can receive and apply information from other modules. For example, our controller can take reference  $d$  from obstacle avoidance, intersection crossing and parking. For intersection navigation and parking, our controller needs additionally the pose estimation and a curvature from the navigators and the parking team respectively.

Out of scope was:

- Pose estimation and curvature on Intersections (plus navigation / coordination)
- Model of Duckiebot and uncertainty quantification of parameters (System

Identification)

- Object avoidance involving going to the left lane
- Extraction and classification of edges from images (anti-instagram)
- Any hardware design
- Controller for Custom maneuvers (e.g. Parking, Special intersection control)
- Robustness to non existing line

#### 4) Preliminaries (optional)

---

### 12.3. Definition of the problem

Our final objective is to keep the Duckiebots at a given distance  $d$  from the center of the lane, on straight and curved roads, under bounded variations of the city geometric specifications.

The project was on the bottom line, taking the line segments which gave information about the line colour and the segment positions to estimate the Duckiebot's pose and return a command for the motors to steer the robot to the center of the lane. After roughly analysing the existing solution, we divided the work load into two topics **pose estimation** and **controller** to enable parallel dealing with the problems in the short period of time.

In our [Preliminary Design Document](#) and [Intermediate Report](#), we have listed all variables and their definitions, as well as all system interfaces with other groups and assumptions we made. Due to limitation of time and different priorities of other teams, some integrations with other teams are not yet activated but they are already prepared in our code (some of it commented out).

#### 1) Pose Estimation

---

Starting with the image taken by a monocular camera, we assume that [Anti-Instagram](#) compensates for color changes from different ambient light conditions and the detected segments of the line edges are always in the corresponding colour (yellow, white, red) to the tape and point to the correct direction (to the Duckiebot = left edge, away from the Duckiebot = right edge), see ([Figure 12.2](#)). The detected line segments are passed on in a list to the 'Lane Filter', the Duckiebot estimates the distance  $d_{\text{est}}$  from center of the lane and heading  $\theta_{\text{est}}$  with respect to the center of the lane.

To improve curve following, our main goal was to decrease the overshoot after a turn. A main reason for the overshoot was that the previous implementation of the pose estimator was designed for straight lanes only. We planned to predict if a curve is upcoming and in which direction the turn will be. This curvature estimation could then be used as an input for a feedforward part of the controller. This would help for a smoother transition in curves.

We planned on testing two different ideas for curvature estimation. One based on the distribution of segments to different domains with different ranges. The second one based on the Discrete Fourier Transform (DFT).

Our general goal was to improve the accuracy of estimated pose in regard to reference pose and increase the computational efficiency. Also robustness regarding slight changes in width of the lane, width of the lines and curvature should be achieved.

*How the pose estimation works:*

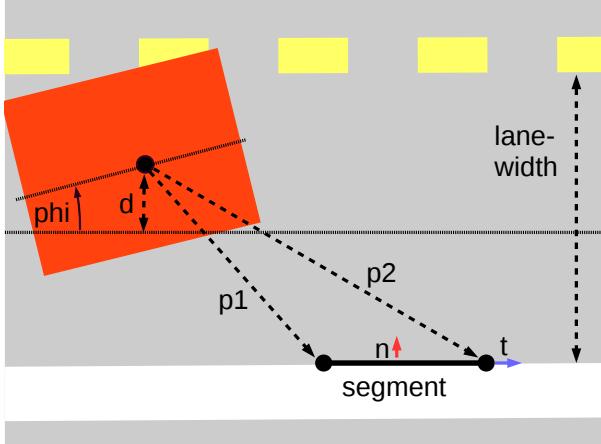


Figure 12.5. Vote generation on straight lane from one line segment.

The lane filter gets a list of detected segments by the line filter with their colors. One segment is described by two vectors pointing from the center of the Duckiebot to the start and endpoint of the segment as shown in [Figure 12.5](#) where one segment is described by the vectors  $p_1$  and  $p_2$ . From these two points we can calculate the vector  $t$  which is tangential to the segment and the vector  $n$  which is perpendicular to the segment. The vector  $t$  can be calculated using

$$t = \frac{p_2 - p_1}{\|p_2 - p_1\|}$$

and  $n$  is the unit vector perpendicular to  $t$ . In case the segment was perfectly detected, the distance of the center of the Duckiebot to the white line is then the scalar product of  $p_1$  with  $n$ . By using the width of the lane we can calculate the distance from the center of the lane.

To get the angle  $\phi$  we use the fact that the tangential vector  $t$  is scaled to length one. From the geometry in [Figure 12.6](#) we see that  $\phi$  can be calculated as

$$\phi = \arcsin(t_2)$$

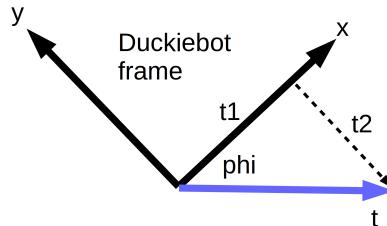


Figure 12.6. Geometry for getting the angle  $\phi$ .

This gives one vote consisting of the two coordinates shown in [Figure 12.5](#) where  $\theta$  and  $\phi$  are representing the same coordinate. For every detected segment one of those votes can be calculated. The one pose having the most votes will be selected as our estimated pose.

*Vote generation on a curve:*

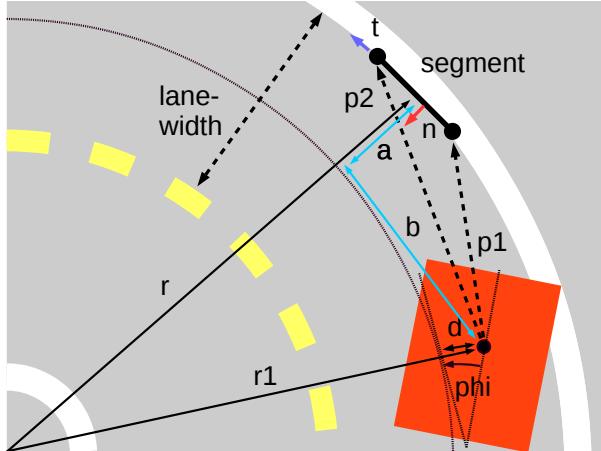


Figure 12.7. Vote generation on curved lane from one line segment.

On a curved lane the generation of votes differs from the one on a straight lane. We derived the vote generation on a curved lane, but did not implement it since a working curvature estimation is needed for this. Again we generate a vote for one given segment with the two endpoints  $p_1$  and  $p_2$  and additionally we need to know the radius  $r$  of this segment. The geometry is shown in [Figure 12.7](#). We again calculate the tangential vector  $t$  and normal vector  $n$  as shown before. We get the center point  $p$  of the segment using

$$p = \frac{p_1 + p_2}{2}$$

Now we can get the two lengths  $a$  and  $b$  by taking the scalar product of  $p$  with  $n$  and  $p$  with  $t$  respectively. Using this we can find the length of  $r_1$  by

$$\|r_1\| = \sqrt{\left(\|r\| - a\right)^2 + b^2}$$

From this we can get the length of  $d$  using

$$d = \|r_1\| - \left(r - \frac{\text{lanewidth}}{2}\right)$$

In the end,  $\phi$  can be obtained using  $d$  and the  $x$  coordinate vector of the Duckiebot coordinate system. This gives us the pose of the Duckiebot in a curve.

## 2) Controller

---

In the existing implementation, the controller has taken the output of the estimator as input and calculated the motor command, velocity  $v$  and angular velocity  $\omega$ , with help of hardcoded parameters. Running the current lane follow-

ing demo, we determined the weaknesses of the controller's performance. Since the existing controller only had a proportional part (P-part) and the  $\theta$  acted similarly to a derivative part, (D-part), we decided to implement following parts and benchmark its performance.

- Integrator for both  $\theta$  and d
  - First approach: No saturation for integrator
    - Problem: Very strong oscillation
  - Second approach: Saturation for integrator
    - Performance improved a lot. Oscillation is reduced
  - Third approach: Saturation of integrator and reset of integrator whenever zero error is reached
    - Performance improved again.
  - Fourth approach: Added true integration with correct timestep instead of simply summing up the error and neglecting the timestep
    - This is the correct approach since the time step needs to be taken into account.
- Feedforward for driving on a curved lane
  - We take the current curvature as an Input for the feedforward part
    - The feedforward part is very much dependent on the correct curvature estimation. If curvature was estimated correctly, feedforward worked well.

In some cases, the controller won't need the real time pose input of the pose estimator but a given path from other teams for example during intersection navigation. For this task, we had to define a more general 'Lane Pose' message, communicate and coordinate the integration with other teams and subscribe to the topic of their nodes (see [Subscribed topics by Lane Controller Node](#)). To decide about the input source, we have to subscribe to flags passed by the other teams and create a prioritization logic.

Regarding performance, the Duckiebot should have a small steady state error within a tile and never leave the lane given the [Duckietown appearance specifications \(master\)](#).

## 12.4. Contribution / Added functionality

### 1) Curvature Estimation

*Curvature estimation using multiple domains:*

The idea of our curvature estimation approach is to split the domain in front of the Duckiebot into multiple range areas. A version using three areas is shown in [Figure 12.8](#).

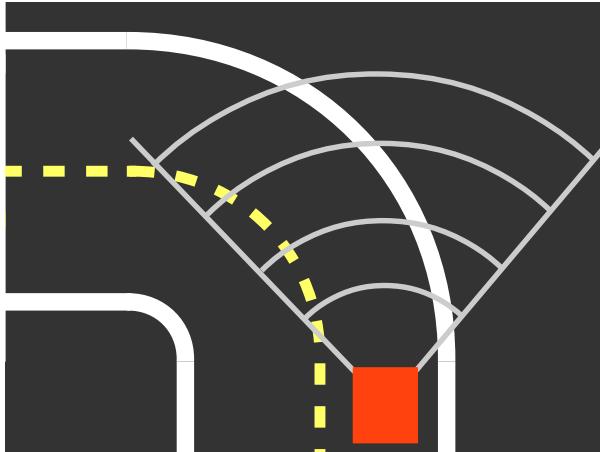


Figure 12.8. Curvature estimation using the segments in different domains.

In each of the range areas the road can roughly be assumed as a straight lane. But for every area further away from the bot this straight lane fit is tilted more towards the left. For each area, only those segments with center point inside the area are considered. Using these segments for each area, we run the standard estimation and thereafter for each area we get a  $d$  and a  $\theta$  value. Now we can compare those results with the  $d$  and  $\theta$  value from the estimation of the position.

The expected results are shown in [Figure 12.9](#) where the left most points in each graph represent the actual position estimation and the further three point represent the estimations of the three different range areas. As a leftover of the existing code where this was already the case,  $\phi$  and  $\theta$  are still used as synonyms within the code. Due to limited time, it did not make it to our highest priority at any time within our project, to merge those names. This should be done in future work.

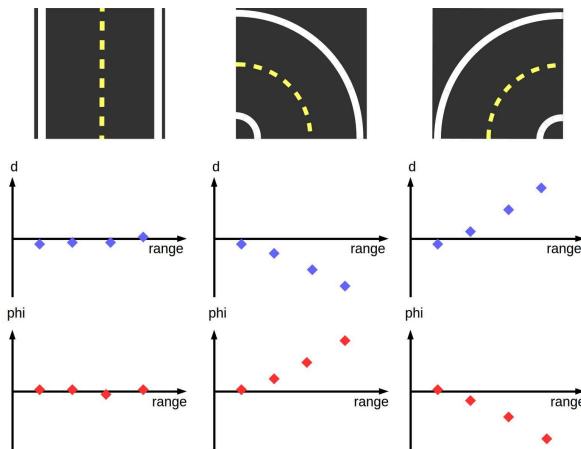


Figure 12.9. Expected results of  $d$  and  $\phi$  values for straight lane, left curve and right curve.

Unfortunately, the measurements for the higher ranges are very noisy as there are only a few line segments detected at further distance and therefore the signal to

noise ratio is very bad. To get rid of outliers, we decided to add a median filter over values of the ranges. Additionally we saved this median  $d$  and  $\phi$  values over time for the last five time steps and again took the median value of it. Then we checked if it is above or below the value of the closest range.

Another possibility would be to fit a line through the data points and decide on the lane type based on the slope of the line. Nevertheless, we decided to use the before mentioned method using the median values because we wanted to keep the computational complexity as low as possible.

By testing we found good results for the cutoffs shown in [figure](#) where  $d_{\text{median}}$  and  $\phi_{\text{median}}$  represent the median over 5 time steps of the values resulting from the range area and  $d_{\text{est}}$  and  $\phi_{\text{est}}$  represent the actual pose estimate.

TABLE 12.1. CUT OFFS FOR THE DECISION ON THE CURVATURE TYPE.

	$d_{\text{median}} - d_{\text{est}}$	$\phi_{\text{median}} - \phi_{\text{est}}$
left curve	$< -0.3$	$> 0.05$
right curve	$> 0.2$	$< -0.02$

#### *Curvature estimation using Discrete Fourier Transform:*

By generating a discrete binary image from the segments projected to the ground and applying the discrete fourier transform to this image, the curvature of the road in front of the Duckiebot can be detected. Fourier transforms of such binary images are shown in [Figure 12.10](#). By using the correct fourier features, straight lanes, right and left curves could be detected due to to their differing fourier transform.

This method has been implemented successfully but the problem was first of all choosing the right resolution for the segment images and additionally, the method introduced a delay of about 0.2 seconds. Since we want to avoid decreasing the lane following performance of the Duckiebot, we decided to dismiss this method.

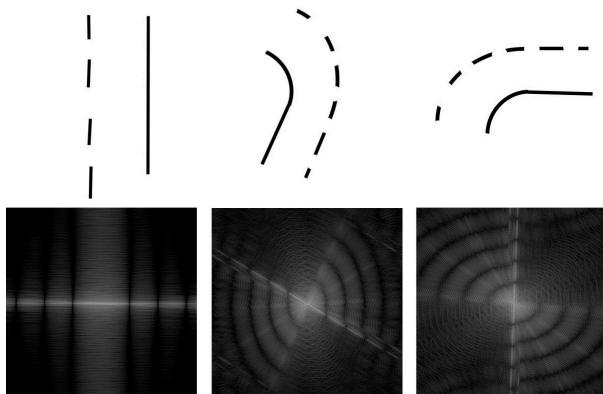


Figure 12.10. Discrete Fourier Transform (DFT) of a street image in ground frame (credits jukin-dle).

In the controller, a feedforward part was added to figuratively speaking straighten the lane and ease the work of the controller. Therefore, the feedforward part takes the reference curvature  $c_{ref}$  and reference velocity  $v_{ref}$  as inputs and returns the needed yaw rate  $\omega$ , which is then added to the output of the controller. The block diagram of the control loop is shown in [Figure 12.11](#). Since the kinematic calibration was not yet yielding the demanded values of  $v_{ref}$  and  $\omega$  in  $[m/s]$  and  $[\text{rad}/s]$ , correction factors `velocity_to_m_per_s` and `omega_to_rad_per_s` were introduced. With the new kinematic calibration, those correction factors need to be adjusted or ideally become obsolete and need to be deleted in future work.

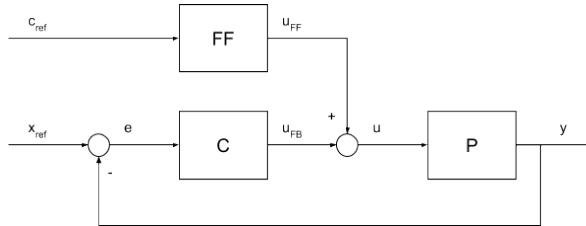


Figure 12.11. Block diagram including feedforward part (FF)

The feedforward part also enables the controller to work for other applications than just lane following. It can follow a path, for example on intersections and parking lots, if the information (including localization) is given in the format described in our [Intermediate Report](#). The respective code is written but some of it is commented out respectively not activated yet due to the limited time of the project and the coordination between teams.

From the coordinates as shown in [Figure 12.5](#), we get

$$\begin{aligned} \left[ \begin{array}{c} c \\ d \end{array} \right] &= \left[ \begin{array}{c} v \sin \theta \\ v \cos \theta \end{array} \right] \\ \left[ \begin{array}{c} c \\ d \end{array} \right] &= \left[ \begin{array}{c} \dot{v} \sin \theta + v \cos \theta \\ \dot{v} \cos \theta - v \sin \theta \end{array} \right] \end{aligned}$$

Through linearization, assuming  $\theta$  to stay small and with  $u = \omega$ , this becomes

$$\begin{aligned} \left[ \begin{array}{c} c \\ d \end{array} \right] &= \left[ \begin{array}{c} v \sin \theta \\ v \cos \theta \end{array} \right] \\ &= \left[ \begin{array}{c} 0 & v \\ 0 & 0 \end{array} \right] \times \left[ \begin{array}{c} c \\ d \end{array} \right] + \left[ \begin{array}{c} 0 \\ 1 \end{array} \right] \times u \end{aligned}$$

with

$$x = \left[ \begin{array}{c} c \\ d \end{array} \right] \quad \theta = \left[ \begin{array}{c} \theta \end{array} \right]$$

To reduce static offset, integral parts were implemented for both  $d$  and  $\theta$ . This was achieved by augmenting the system to  $\hat{x}$ , as shown below.

$$\hat{x} = \left[ \begin{array}{c} x \\ e_I \end{array} \right]$$

```
\begin{equation} \left[ \begin{array}{c} \dot{x} \\ \dot{e}_I \end{array} \right] = \begin{pmatrix} 0 & v & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \times \left[ \begin{array}{c} x \\ e_I \end{array} \right] + \left[ \begin{array}{c} 0 \\ 1 \\ 0 \\ 0 \end{array} \right] \times u + \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix} \times x_{\text{ref}} \end{equation}
```

**With**

```
\begin{equation} \left[ \begin{array}{c} e \\ e_I \end{array} \right] = \left[ \begin{array}{c} x_{\text{ref}} - x \\ \int (x_{\text{ref}} - x) dt \end{array} \right] \end{equation}
```

```
\begin{equation} x_{\text{ref}} = \left[ \begin{array}{c} d_{\text{ref}} \\ \theta_{\text{ref}} \end{array} \right] \end{equation}
```

In order to omit oscillation and guarantee the stability of the system, the poles were placed on the negative real axis. With that we found the initial values for  $\$k_p$  and  $\$k_I$  in

```
\begin{equation} u = - \left[ \begin{matrix} k_p & k_I \end{matrix} \right] \times \left[ \begin{array}{c} e \\ e_I \end{array} \right] \end{equation}
```

through

```
\begin{equation} \left[ \begin{array}{c} \dot{d} \\ \theta \end{array} \right] = \begin{pmatrix} 0 & v \\ -k_p & -k_I \end{pmatrix} \times \left[ \begin{array}{c} d \\ \theta \end{array} \right] + \left[ \begin{array}{c} 1 \\ 1 \end{array} \right] \times x_{\text{ref}} \end{equation}
```

The controllability matrix shows that the integrator of  $\theta$  is not controllable, since it has rank 3 instead of 4:

```
\begin{equation} \mathcal{C} = \left[ \begin{matrix} B & AB & A^2B & A^3B \\ \end{matrix} \right] = \begin{pmatrix} 0 & v & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -v & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} \end{equation}
```

To prevent the integral parts from diverging, an Anti Reset Windup was implemented. Therefore, whenever actuator limits were reached, the integral steps at the corresponding time step were not added to the integrator. The actuator limits were reached when the motors were sent lower values than would be necessary to reach the controller outputs, because of certain limitations within the software. The limitations include for example a limitation on the turn radius of the Duckiebot, because it should not be able to turn on the spot but to move more similarly to common passenger cars.

In curves, the integrator values accumulate rapidly and lead to an overshoot after the curve. A possible approach would be to turn off the integrator in curves, but in consequence the curvature estimation would need to be used and in addition need to be robust. Or if the feedforward part could be fully used (while also needing a robust and low-latency curvature estimation), the problem might be diminished. In the current state, the integrator is reset to zero whenever it is at or crosses the zone of zero error. In addition the integrator was also reset to zero, whenever the velocity sent to the motors was zero.

Since the integral part of theta is not controllable, it was set to zero. The resulting parameter, the proportional gains of both  $d$  and  $\theta$  plus the integrator gain of  $d$ , were tuned. First with pole placement initial values were approximated, as described above. For the final tuning, each parameter was varied until the unstable state and the approximate boundary to the stable state were found, while all the other parameters were kept in a stable state. This was repeated multiple times with ever more aggressive controller behavior until an optimum was found. The controller is optimized to run with a gain (of the kinematic calibration) of 0.6.

### 3) Benchmark

---

To benchmark the state zero at the beginning of the project and our final implementation and to compare them, we implemented a benchmark package. This package contains the benchmark code used for the Controllers project. It basically takes one or more rosbags in a specific folder and evaluates the run of the corresponding Duckiebot for  $d_{ref}$  and  $\phi_{ref}$  and plots them into a diagram.

Additionally if the rosbag does not contain any pose information, it takes the pictures and calculates the transformation and line segments itself. It also plots the values onto the pictures, so those pictures can be combined to a video.

*Output:*

The output diagram should look like the one shown in [Figure 12.12](#).

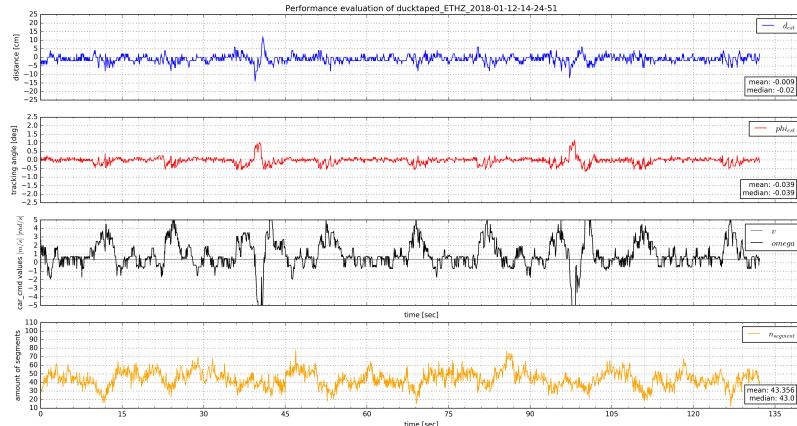


Figure 12.12. Output diagram of benchmark code.

Output data should look like the one shown in [Figure 12.13](#).

```

Benchmark results for ducktaped_ETHZ_2018-01-12-14-48-09
  dist min:      -0.145
  dist max:      0.165
  dist mean:     -0.014
  dist med:      0.005
  dist var:      0.0014
  dist std:      0.0381

  phi min:       -1.45
  phi max:       1.25
  phi mean:      0.117
  phi med:       0.05
  phi var:       0.1516
  phi std:       0.3894

Image segments statistics:   Average amount of segments:    33.946
                           Median amount of segments: 34.0
                           Average processing time per frame (on duckiebot): 0.073

Rosbag processing time: 3.983850
Average processing time per frame: 0.002665
Image preparer used on computer: prep_200_70

```

Figure 12.13. Output data of benchmark code.

An example of a processed frame is shown in [Figure 12.2](#). To run the benchmark code see the **README** file in `duckietown/Software/catkin_ws/src/10-lane-control/benchmark`.

#### 4) Logs

---

We took a huge amount of logs to benchmark the performance of the controller and estimator. These logs are available [here](#). Our Duckiebots were a313, yaf, fobot, ducktaped and tori.

### 12.5. Formal performance evaluation / Results

We evaluated the improvement of the performance with help of several tests. The evaluation procedure are defined in our [Intermediate Report](#). The main benchmark feature was the average deviation from tracking reference during a run (distance to middle lane) and the standard deviation of the same value. We also benchmarked the deviation from the heading angle as well but since the bot is mainly controlled according to the deviation of the tracking distance, it was the main feature to lead our development. Benchmarking in general occurred by letting the Duckiebot run a specific experiment and recording a rosbag. We wrote a distinct offline benchmarking application mentioned above, that analyzes the rosbag containing the recorded values and creates plots with the extracted information about tracking distance and heading angle over the run.

Furthermore, we assessed the performance of the Duckiebots in the following dimensions:

- **Estimator:**
  - Static lane pose estimation benchmark
  - Static curve pose estimation benchmark
  - Image resolution benchmark
  - Segment interpolation benchmark
  - Curvature estimation benchmark
- **Controller:**
  - Stop at red line benchmark

- Controller benchmark
- Non-conforming curve benchmark

## 1) Performance Evaluation of Estimator

### *Static lane pose estimation benchmark:*

In the static lane pose estimation, we put the Duckiebot on predefined poses and checked how well the pose estimator performs. In this section, the Duckiebot was placed on a straight lane segment with different measured distances from the middle of the lane and different measured heading angles. The results can be seen in the following graphs:



Figure 12.14. Old Lane Estimator Performance of estimating  $d$  on straight lane.

As one can see from [Figure 12.14](#), the estimated distance from the middle of the lane and the actual value correspond very good in most experiments. There is only one case where the actual deviation was \$-1 cm\$ and measured was \$4 cm\$. Note that the histogram resolution used to determine the pose is \$1 cm\$.

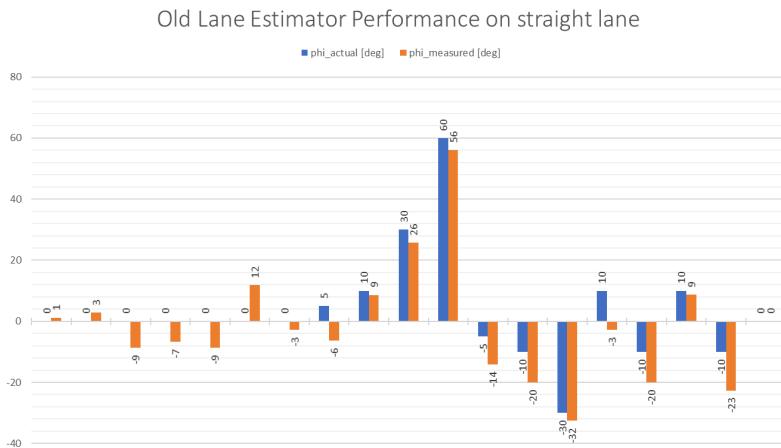


Figure 12.15. Old Lane Estimator Performance of estimating  $\phi$  on straight lane.

[Figure 12.15](#) shows a similar picture for the heading angle estimation from the segments. Deviation from actual values vary from \$1\$ to \$12^{\circ}\$, whereas the Duckiebot performed better when being rotated to the left. Note that the histogram resolution to determine the heading angle is \$3^{\circ}\$ or \$0.15 \text{ rad}\$.

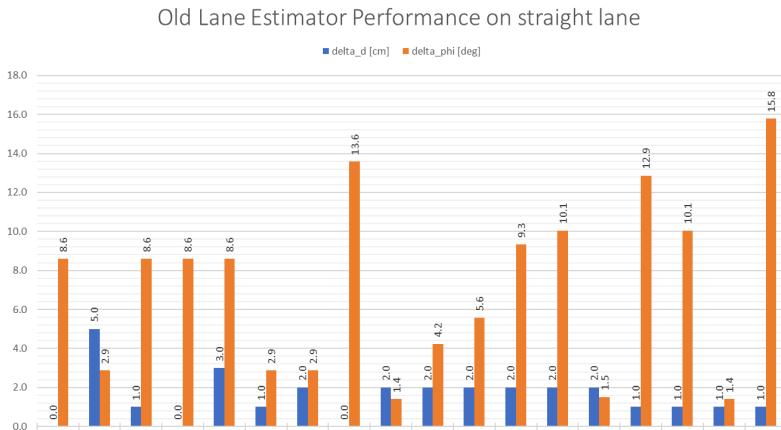


Figure 12.16. Differences of  $d_{\text{actual}}$  and  $d_{\text{measured}}$  and  $\phi_{\text{actual}}$  and  $\phi_{\text{measured}}$  respectively.

If we look at the overall deviations in all experiments shown in [Figure 12.16](#), we can see that the pose estimator performs fairly well, and it is possible to control on the deviation of the distance. The heading angle shows more error. The average deviation from the actual tracking distance in all experiments accounts to \$1.6 \text{ cm}\$ and the average deviation from the actual heading angle in all experiments is \$7.2^{\circ}\$.

#### Static curve pose estimation benchmark:

In the static curve pose estimation, we put the Duckiebot on predefined poses and checked how well the pose estimator performs. In this section the Duckiebot was placed on left curve with different distances from the middle of the lane and different heading angles. The results can be seen in the following graphs:

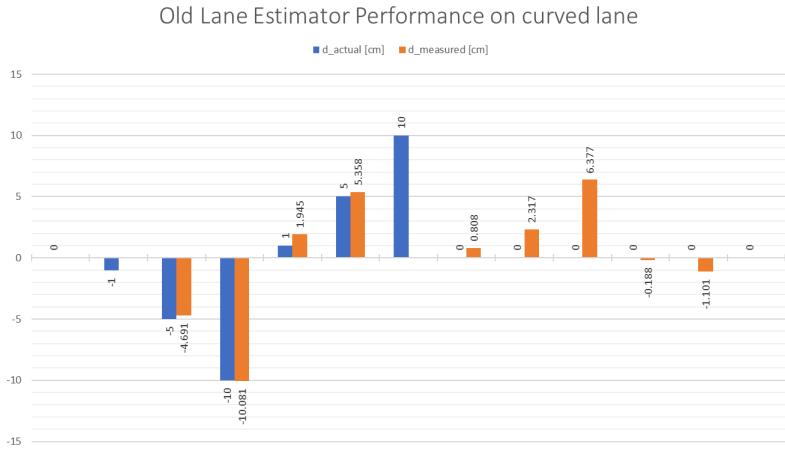


Figure 12.17. Old Lane Estimator Performance of estimating  $d$  on curved lane.

As one can see from [Figure 12.17](#), the estimated distance from the middle lane and the actual value correspond partially to the actual values. Especially for the distance of \$10 cm\$ to the right of the middle of the lane in a left curve the estimator has problems to detect the correct deviation. This is due to the low number of segments and the fact that the pose estimator is actually only constructed to estimate the pose on a straight lane. Also, there is quite some noise which leads to wrong interpretation of the distance, even when the Duckiebot is perfectly situated in the middle of the lane. For some experiments there is no pose estimation due to too much noise in the segment list. Note that the histogram resolution used to determine the pose is \$1 cm\$.

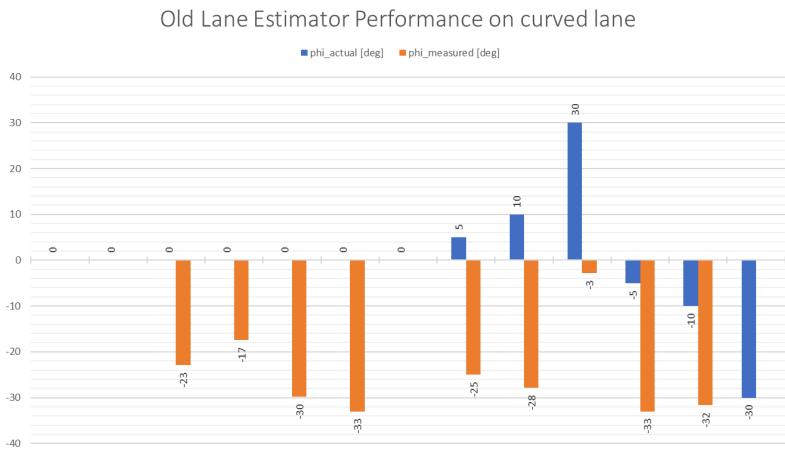


Figure 12.18. Old Lane Estimator Performance of estimating  $\phi$  on a curved lane.

Whereas the estimator is still able to estimate  $d$  quite well on a left curve, for the heading angle most of the values are completely off as can be seen in [Figure 12.18](#). This means the heading angle prediction is not reliable on curved lanes. Note that the histogram resolution to determine the heading angle is  $3^{\circ}$  or  $0.15$  rad.

Old Lane Estimator Performance on curved lane

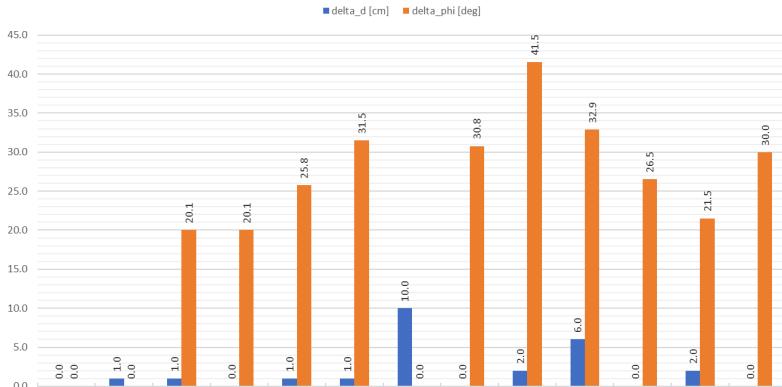


Figure 12.19. Differences of  $d_{\text{actual}}$  and  $d_{\text{measured}}$  and  $\phi_{\text{actual}}$  and  $\phi_{\text{measured}}$  respectively on a curved lane.

If we look at the overall deviations in all experiments shown in [Figure 12.19](#), we can see that the pose estimator performs ok in the determination of the distance from the middle of the lane in a curved section. The values from the heading angle are unlikely correct and therefore should not be used as control input. The average deviation from the actual tracking distance in all experiments accounts to 1.8 cm and the average deviation from the actual heading angle in all experiments is  $21.6^{\circ}$ .

#### *Image resolution benchmark:*

Since the image resolution has an impact on the number of segments being visible to the Duckiebot and the image processing latency time, we benchmarked the impact on the entire lane following performance. We tested different image resolutions, top cut off amounts and changed the histogram size to evaluate how it influences the control performance.

Image Resolution performance test [yaf]

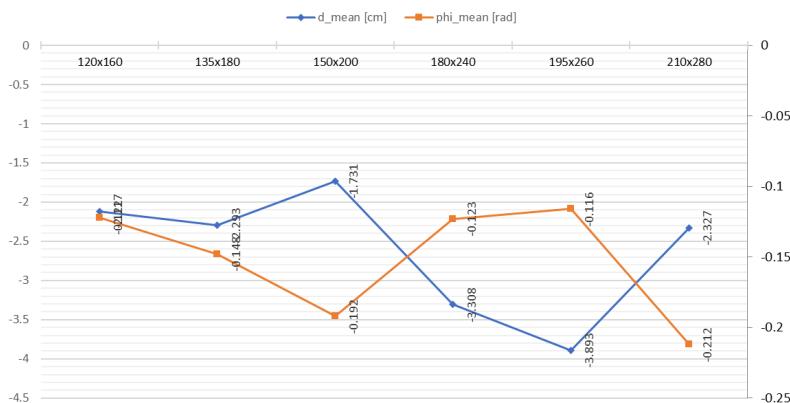


Figure 12.20. Measured  $d_{\text{mean}}$  and  $\phi_{\text{mean}}$  values for different image resolutions for Duckiebot 'yaf'.

As one can see in [Figure 12.20](#), the performance of the Duckiebot measured as the mean deviation from the reference trajectory (which is usually \$0 cm\$) is getting worse the higher the resolution. There are outliers though, since the highest resolution being tested shows better performance than the resolution just one step smaller. The best performance is achieved with slightly higher resolution at 150x200 pixels. To validate these results, we tested it on another Duckiebot as well.

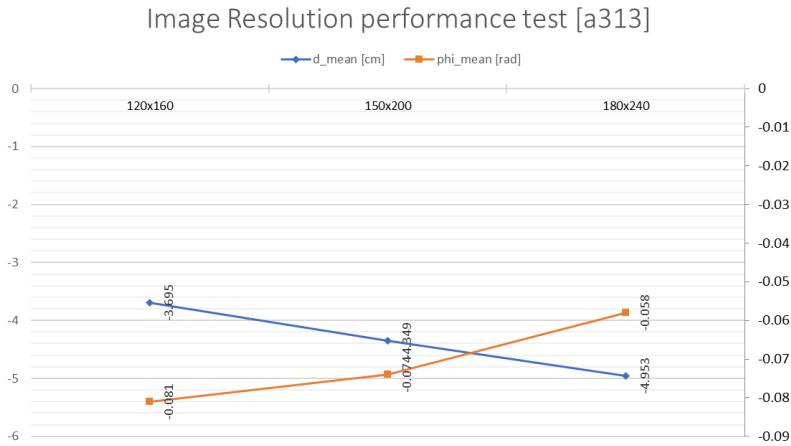


Figure 12.21. Measured  $d_{\text{mean}}$  and  $\phi_{\text{mean}}$  values for different image resolutions for Duckiebot 'a313'.

We see that the results shown in [Figure 12.20](#) and [Figure 12.21](#) are not congruent. We think that this has to do with the fact that each Duckiebot is slightly different and also has different latencies.

The worse performance for higher resolutions can be explained with the change in processing time of the images. Although there are more line segments, which means more precise information about our pose, the processing time increases, and thus this adds latency and affects the whole system performance. The Duckiebot reacts slower to offsets of its pose. [Figure 12.22](#) shows the segment processing time and number of segments for different image resolutions.

### Image Resolution performance test [a313]

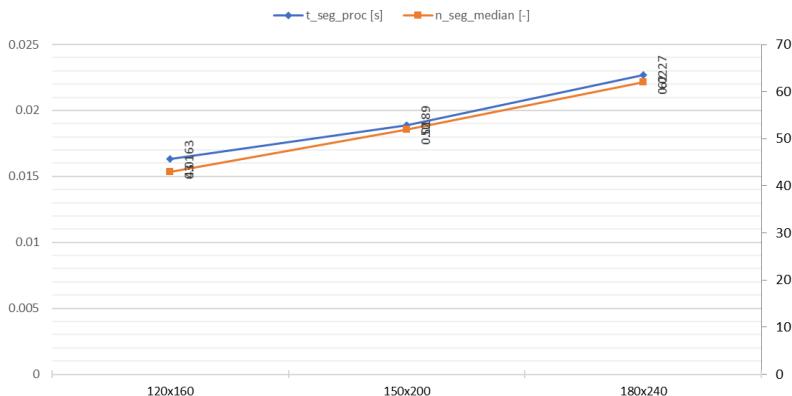


Figure 12.22. Segment processing time and median of number of segments for different image resolutions.

Increasing the top cutoff value means, that from the input image more of the top part is cut away to reduce visual clutter from the image background. At the same time this also decrease the number of pixels being processed and thus lowers the mean latency as well.

### Top cutoff performance test [a313]

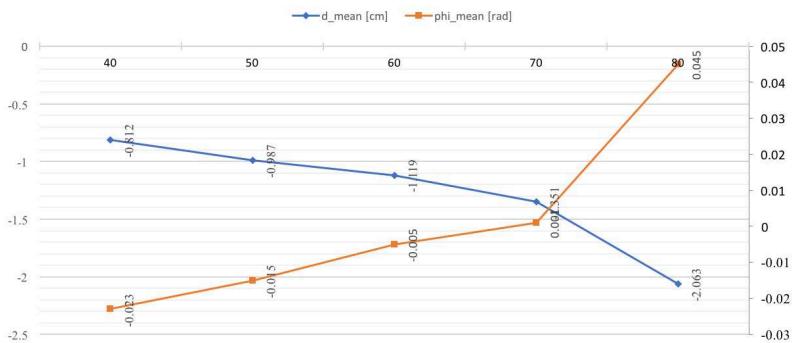


Figure 12.23. Performance of 'a313' for different top cutoffs in pixels.

We run a benchmark to evaluate the influence of the top cutoff on the performance. The test was performed with an image resolution of 120x160 pixels. The results are shown in [Figure 12.23](#). 40 pixels is the standard top cutoff values. This means the upper 40 pixels are cut away from each image. While increasing the top cutoff amount, the \$d\_{mean}\$ decreases slightly while \$\phi\_{mean}\$ increases slightly. We don't see big changes in performance until the top cutoff gets quite big. At this point the Duckiebot does not see enough to control according to the actual pose situation.

### Image Resolution performance test [a313]

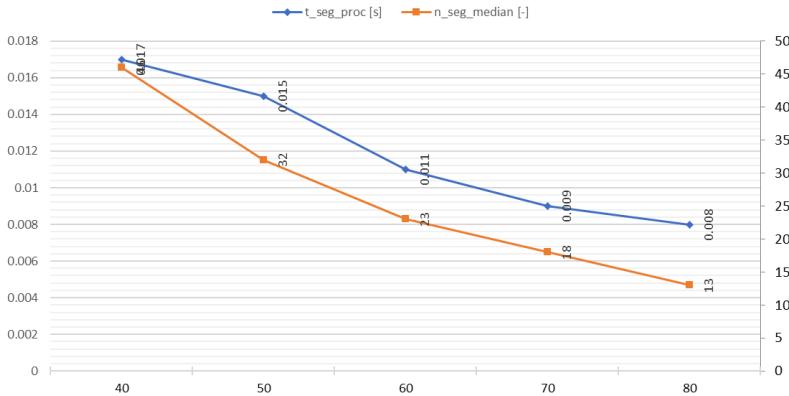


Figure 12.24. Segment process time and number of segments for different top cutoffs in pixels.

As we can see in [Figure 12.24](#), the segment process time and therefore the latency decreases proportionally to the number of segments. This graph also explains the reduced performance in [Figure 12.23](#) since with under 25 segments it is hard to get an accurate pose estimation. In this case a higher top cutoff lowers the performance and at the same time the latency. So, we might see an increase in performance if we combine higher top cutoff with higher resolution, since there the increased latency was an issue.

### Histogram Resolution performance test [a313]

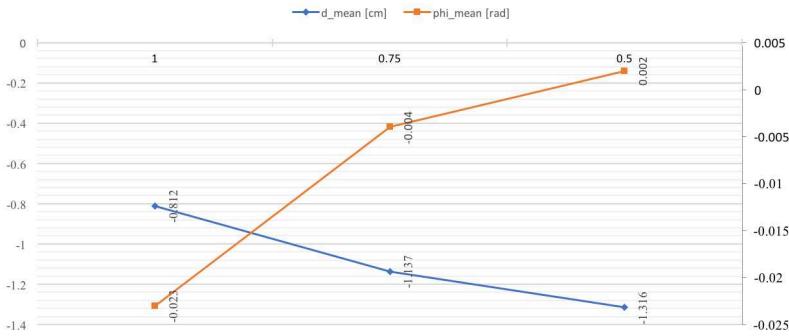


Figure 12.25. Performance for different resolutions of  $d$  in histogram in  $\text{cm}$ .

We also tested the influence of the histogram size for the generation of the votes. The results are shown in [Figure 12.25](#). Making the vote histogram cell size smaller increases the accuracy of the pose estimation. At the same time more segments are needed to get a precise estimate and reduce the influence of noise. We see that the performance is going down for a higher histogram resolution. At the same time [Figure 12.26](#) shows that the segment processing time stays more or less constant for different histogram resolutions. This actually shows, that the decrease in performance results from the missing of a distinct pose for higher resolutions.

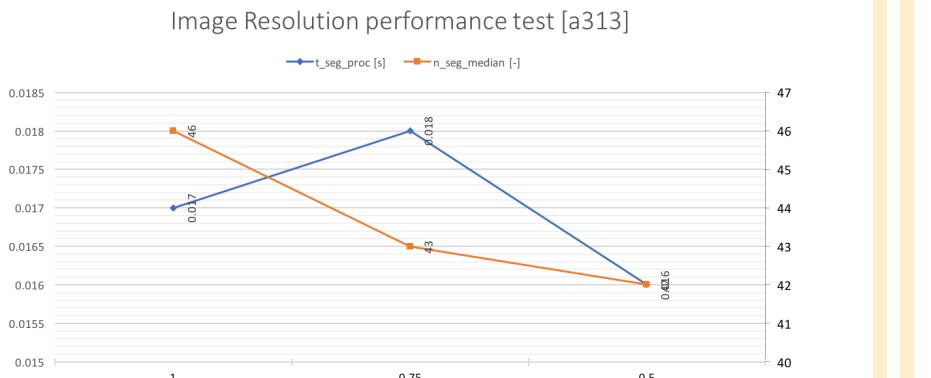


Figure 12.26. Segment processing time and number of segments for different resolutions of  $d$  in histogram in cm.

j

TABLE 12.2. COMBINING IMAGE RESOLUTION AND TOP CUTOFF.

Resolution	Top Cutoff	$t_{latency}$ [s]	$d_{mean}$ [cm]	$\phi_{mean}$ [rad]	$n_{segments}$ [-]
120x160	40	\$0.019\$	\$-1.802\$	\$-0.025\$	\$39\$
150x200	75	\$0.012\$	\$-0.011\$	\$-0.004\$	\$23\$

As we can see from [Table 12.2](#), the configuration with resolution 150x200 and top cutoff 75 can improve the lane control performance compared to the standard configuration with resolution 120x160 and top cutoff 40 without changing the lane controller itself or the pose estimator. Note that all the results from this section have been tested with the improved lane controller.

#### *Segment interpolation benchmark:*

Another approach to improve the pose estimator is to increase the amount of line segments without increasing the image resolution. Here we take each line segment and divide it into smaller pieces of which each has a vote on the belief image. Good line segments cast more votes to the same pose estimate, while bad segments (e.g. which are further away or outliers) have less weight on casting wrong results. Think of it as a filter to improve quality of the lane pose estimate.

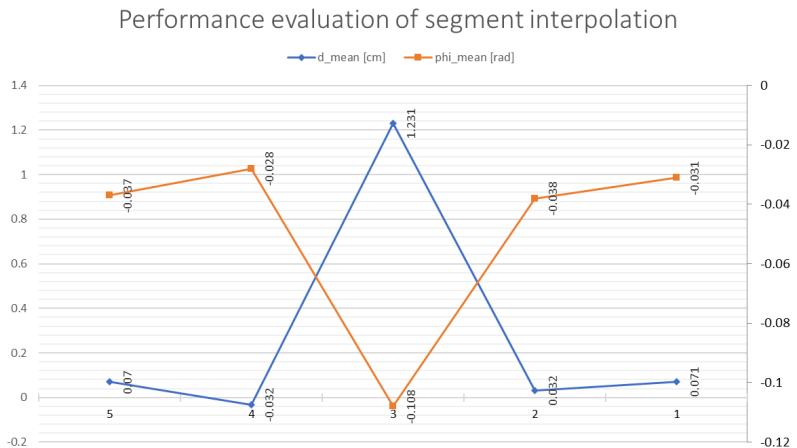


Figure 12.27. Performance for different segment interpolations.

As we can see in [Figure 12.27](#), we tested up to interpolating a line segment 5 times. There aren't any significant changes to the lane following performance except for one outlier while interpolating with 3 line segments. If we look closer, we can see that the actual performance gets worse the more we interpolate due to processing speed of the raspberry pi.

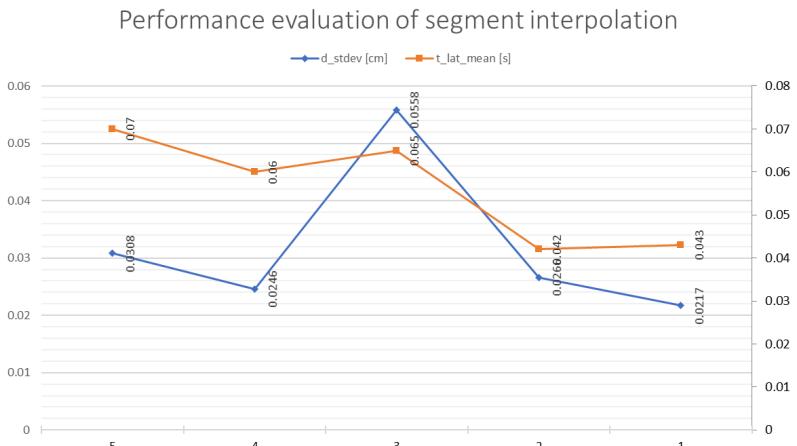


Figure 12.28. Standard deviation of  $d$  and mean latency for different segment interpolations.

[Figure 12.28](#) shows the standard deviation of  $d$  ( $d_{\text{stdev}}$ ) and how it increases the more we interpolate due to higher latency. This behavior is observable on straight lanes where the Duckiebot oscillates around the reference trajectory. We can see this in [Figure 12.29](#) for a run with 5 interpolated segments per detected segments. From 50 to 70 sec we can observe the oscillations on a straight lane due to high latencies.

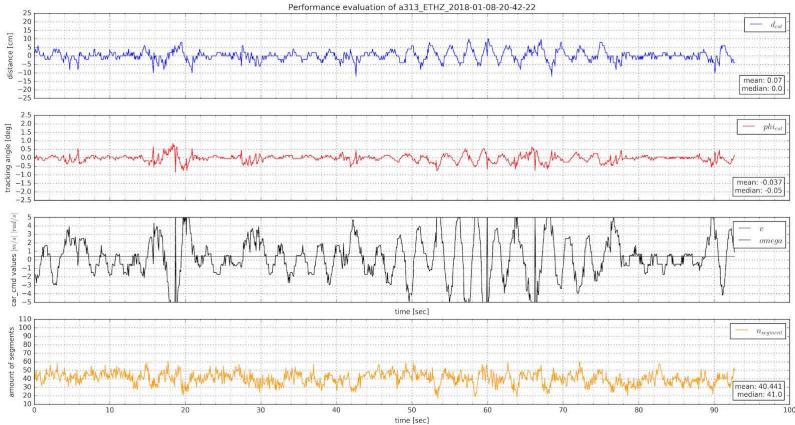


Figure 12.29. Benchmark graph for a run with 'a313' and 5 interpolated segments per detected segment.

#### *Curvature estimation benchmark:*

In this section, we want to evaluate the curvature estimation performance. What the curvature estimator basically does is dividing the input image into several circular sections with equi-radial distance to the Duckiebot. From each section it derives the pose and evaluates, how it changes in these sections. This will tell us, how the road in front of the Duckiebot looks like. Then again, this feature has an impact on the lane following performance of the Duckiebot since the processing power of the raspberry pi is limited and any added latency will slow the bot down.

Performance evaluation of curvature estimation [old / new]

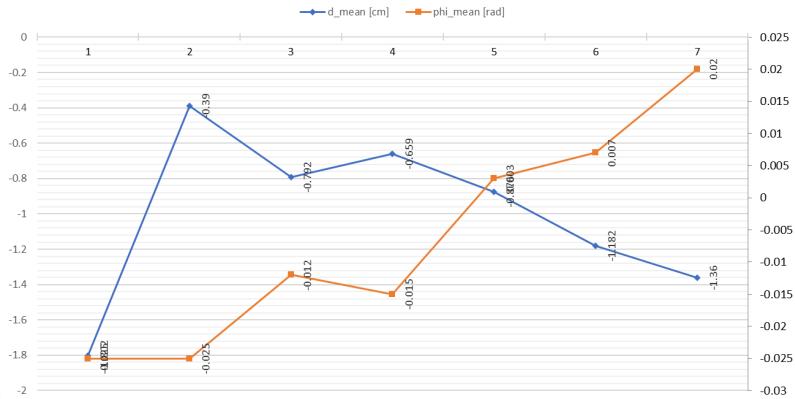


Figure 12.30. Performance for different numbers of belief images from 1 to 7 where the first image is for the actual pose estimation and the further ones are for the curvature estimation (curvature resolution).

In [Figure 12.30](#) on the horizontal we can see the number of belief images being evaluated (curvature estimation resolution). The higher the number, the better we can forecast the type of the road (left curve, right curve, straight lane). With a number of 1, there is no curvature estimation (basically the old pose estimator). We can see that the performance compared to the old pose estimator is much better. This is because the reference run with 1 belief image has been recorded before and the

calibration may have changed. Anyway, we can see a decrease in lane following performance, the higher the amount of belief images or image sections are created. This is due to higher cpu cost and increased latency. From tests we can see that a number of 4 belief images is sufficient to tell in most cases, at what kind of road type we are looking at.

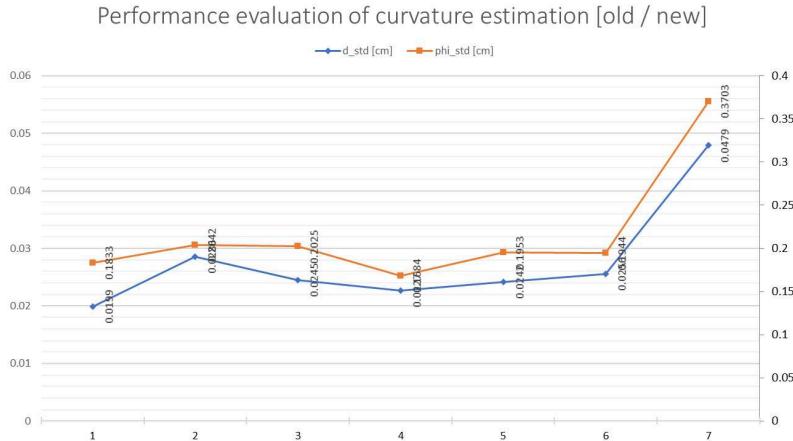


Figure 12.31. Standard deviation of  $d_{std}$  and  $\phi_{std}$  for different numbers of belief images.

A look at [Figure 12.31](#) showing the standard deviation tells us, that performance decreases with higher numbers of belief images (curvature estimation resolution).

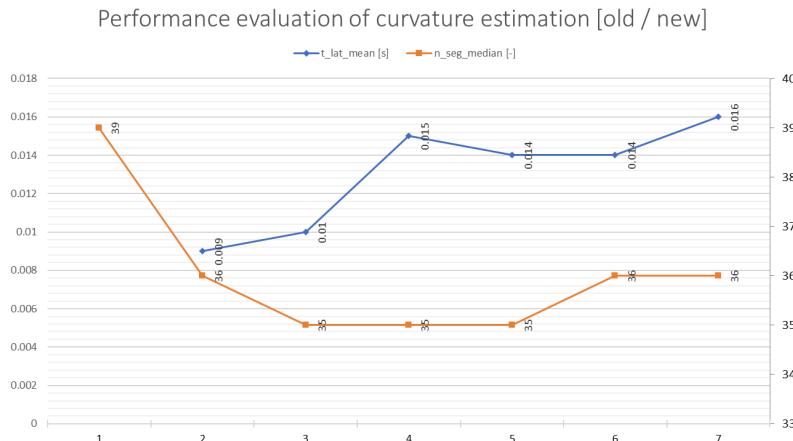


Figure 12.32. Segment processing time and number of segments for different numbers of belief images.

Same as in other sections, the main performance is heavily depending on the overall latency of the code being executed on the Duckiebot. The latency of segment processing is shown in [Figure 12.32](#).

We improved the code on curvature estimation and retook all tests to better compare how the Duckiebot behaves. In the following we will see similar graphs with 1 belief image on the old pose estimator, 4 and 7 belief images on the new curva-

ture estimator and again 1, 4 and 7 belief images on the improved estimator.

Performance evaluation of curvature estimation [improved]

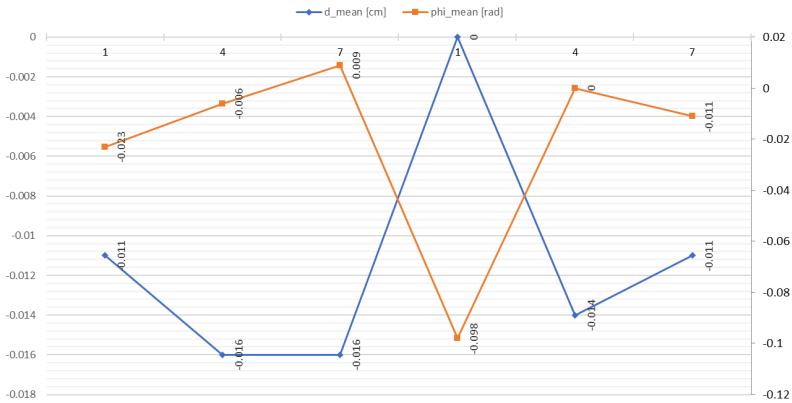


Figure 12.33. Performance of 1, 4 and 7 belief images for old curvature estimation on the left 1, 4 and 7 belief images for improved curvature estimation on the right.

It is observable in [Figure 12.33](#) that the improved curvature estimation performs slightly better in all three cases. In [Figure 12.34](#) we see that the latency for the improved curvature estimator is lower and therefore the case with just one belief image (meaning the curvature estimation is turned off) performs especially well.

Performance evaluation of curvature estimation [improved]

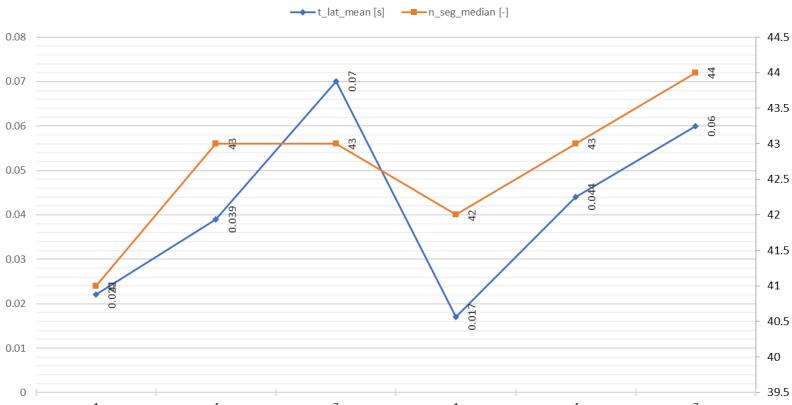


Figure 12.34. Segment processing time and number of segments of 1, 4 and 7 belief images for old curvature estimation on the left 1, 4 and 7 belief images for improved curvature estimation on the right.

## 2) Performance Evaluation of Controller

### *Stopping in front of red stop line:*

We evaluated if the Duckiebot is able to stop in front of the red stop line within the defined specifications. In order to test the stopping behavior, we tested the old controller and the new controller and measured the pose in front of the stop line. The results in [Table 12.3](#) show that we are able to improve the stopping in front of the

red line. The performance shows to be in the bound of the target values. The target stopping distance to the center of the red line should be 16 to 10 cm and the final heading angle should be in the range of  $\phi = -10^{\circ}$  to  $\phi = 10^{\circ}$ .

TABLE 12.3. STOPPING AT STOP LINE EVALUATION.

	$d_{\text{mean}}$	$\phi_{\text{mean}}$	Mean stopping distance to center of red line
Old Controller	\$5.6cm\$	$5^{\circ}$	\$17.4 cm\$
New Controller	\$-0.6cm\$	$3.6^{\circ}$	\$8.2cm\$

*Controller benchmark:*

The performance of the controller has been benchmarked under varying configurations, i.e. with the old baseline controller, the improved controller with the implemented Integrator and finally the same improved controller with addition of a correction for the static offset. The results of this benchmarking are shown in [Table 12.4](#). Notably the controller did not use the improved estimator for this benchmark, rather the baseline estimator was used. The desired state throughout the benchmark is  $d = 0.0$  and  $\phi = 0.0$ .

TABLE 12.4. RESULTS FOR CONTROLLER EVALUATION OF OLD CONTROLLER, NEW INTEGRAL PART AND OFFSET CORRECTION.

	$d_{\text{mean}}$ [cm]	$d_{\text{std}}$ [cm]	$\phi_{\text{mean}}$ [rad]	$\phi_{\text{std}}$ [rad]
Old Controller	3.16	0.45	-0.40	0.1
New Integral Part	-2.08	0.08	-0.11	0.07
Offset Correction	-0.45	0.16	-0.03	0.20

As observable in [Table 12.4](#), the lane following performance increased drastically after improving the controller. First, by implementing the Integrator into the controller, the performance improved in terms of a lower static offset as well as a lower mean heading angle. Additionally, the standard deviation of both  $d$  and  $\phi$  was lowered considerably. This means that the Duckiebot stayed much closer to the desired position in the center of the lane, even after a long time. Therefore, the performance improved greatly with help of the Integrator alone already.

Further, by correcting the remaining static offset, the static offset was cancelled out completely and the median heading angle was lowered as well. This is a very important result, as the static offset represented a vital problem.

In addition to the quantitative benchmarking above, the performance was evaluated qualitatively as well by observing the driving Duckiebot. From those observations, the performance improvements in terms of a cancelled static offset as well as a much lower median heading angle were very clearly noticed as well. By directly comparing the performance of the old and new controller qualitatively, the improvement of the controller is very clearly visible. With the new controller the

Duckiebot never touches the middle and outer lines, drives very robust, there is no static offset and no overshoot after the curves is observed.

*Non-conforming curve benchmark:*



Figure 12.35. Non conforming big curve.

We benchmarked the controller not only for the straight lanes and curves which are conforming with the Duckietown specification, rather the new improved controller was also tested on lanes with non-conforming geometries such as a very large and wide curve as shown in [Figure 12.35](#) and a very narrow and harsh S-curve as shown in [Figure 12.4](#). This benchmark was conducted in order to test the robustness of the controller to varying lane geometries. This is a very relevant test, as the geometry of the duckietown can not always be guaranteed. In addition, a controller which works good for a wide range of geometries would be desired. The results of those tests with non conforming curve geometries can be found in [Figure 12.36](#).

Figure 12.36. Results from benchmark on non-conforming curves.

As can be seen from the results in [Figure 12.36](#) for both tested non conforming curves the performance improved considerably by introducing the new controller. Both the mean distance to the center of the lane  $d$  and the mean heading angle  $\phi$  have been improved. In addition, the standard deviation of both of those metrics were reduced as well. Those results show that the performance of the new controller was improved with respect to non conforming curve geometries. Since only two non conforming geometries have been tested, this test represents far from all non conforming geometries. In future benchmarks with respect to geometry robustness, this fact should be considered and more non conforming geometries should be tested. Nevertheless, the performance on those two tested non conforming curves are very promising and point to a strong robustness with respect to altering geometries.

*Performance of the controller on curvy road:*

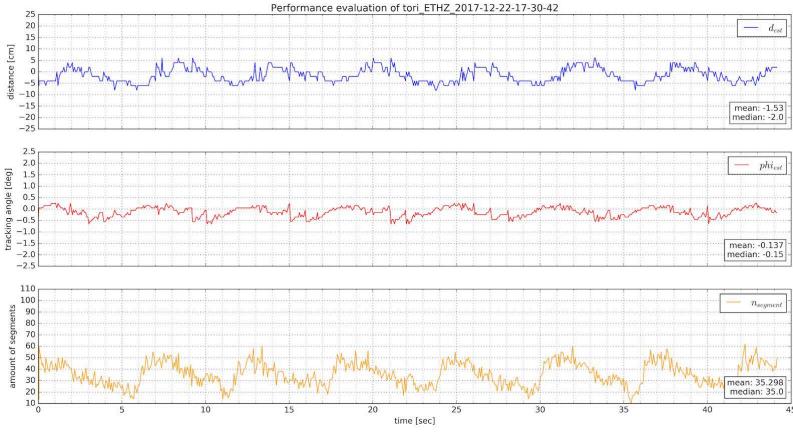


Figure 12.37. Benchmark of Duckiebot on curvy roads with the baseline controller from last year.

Benchmark of Duckiebot with baseline controller from last year's implementation. Most notably, the median lateral position of the Duckiebot  $d_{\text{est\_median}}$  is higher compared to the new implementation of the controller.

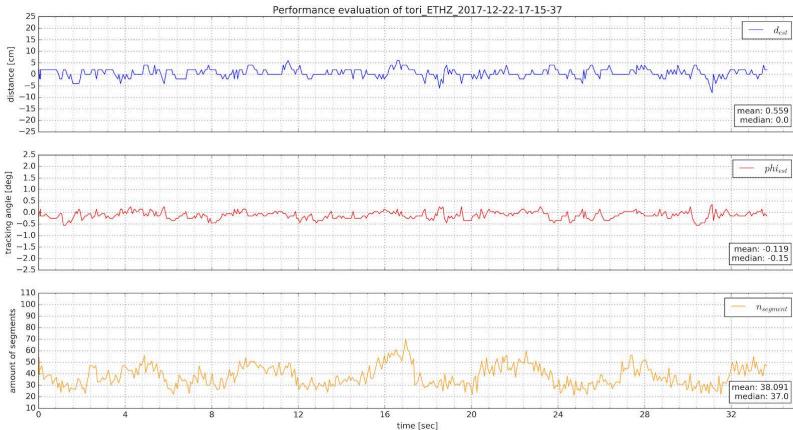


Figure 12.38. Benchmark of Duckiebot on curvy roads with the new improved controller.

Benchmark of Duckiebot with new controller implementation. Most notably, the median lateral position of the Duckiebot  $d_{\text{est\_median}}$  is lower compared to the old implementation of the controller.

#### *Performance of the controller on non conforming lanes:*

We also made some test to show that our controller is able to cope with situations that are not conforming with the Duckietown specifications. [Figure 12.39](#) shows a run with thicker white and yellow lines then specified and [Figure 12.40](#) shows a run with some white and yellow lines missing. In both videos the new controller is still able to follow the lane as expected.

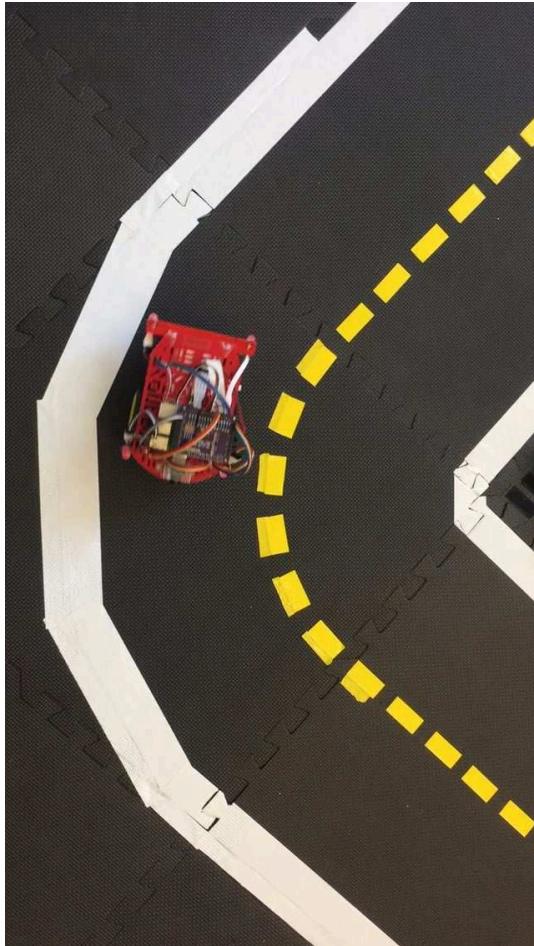


Figure 12.39. Lane following with thicker lines



Figure 12.40. Lane following with partial lines missing

### 3) Failed Implementations

---

#### Estimator

Although the 7 ranges estimation provided low mean deviation from the actual position and provided good prediction of the upcoming curve as well as its curvature direction. The 7 ranges estimation failed in the implementation of the lane follow-

ing demo due to high computation requirement and the caused time latency.

## Controller

Feedforward during lane following: As the feedforward part during lane following depends entirely on the estimation of the curve, this part failed due to bad estimation of the curves in certain situations as well as the increased latency due to the curvature estimation. Whenever a curve is not correctly detected or not precisely at the beginning of the curve, the feedforward part introduces additional instability. This is especially a problem in the notorious S curves. Therefore, the implementation of the feedforward works good if a precise estimation of the curve is available that works without introducing high latencies. Although, such a precise curvature estimation with low latency is not available at the moment. Hence, the feedforward part during lane following is not robust enough for the current curvature estimation. Nevertheless, the feedforward part is useful for other nodes to interact with the controller. In certain situations other nodes are able to use the feedforward part in order to follow paths (navigators on intersections and parking team on parking lots).

## 4) Challenges

---

- Limited computational power of Raspberry Pi.
  - Estimation of curvature introduced high latencies.
  - By increasing the resolution of the picture we would get more segments and this would make both a better pose and curvature estimation possible. Nevertheless, the latency is also increased significantly.
- Duckiebot with different wheels (slippery and non-slippery wheels)
- Some Duckiebots prove to have higher latency when running the same code. This increased latency is a problem.
- Lightning has a very big influence on the performance
  - Depending on the light condition of duckietown the number of detected segments as well as the correctness of the color is varying. Especially reflection on yellow tape makes it appear white. To tackle this issue, a polarisation filter was found to have a positive influence. This might need to be considered in future hardware updates.
  - Anti Instagram might not be as good for every light condition.

### *Effect of anti-instagram on segment detection in curves:*

In case Anti-Instagram is badly calibrated, the Duckiebot will not see enough line segments. This is especially a problem in the curve and the Duckiebot could leave the lane. An example of this failure can be seen in [Figure 12.41](#) for which we had a bad Anti-Instagram calibration. Hence, the Duckiebot sees not enough line segments and the lane following fails in the curve. To solve the problem Anti-Instagram needs to be relaunched. In the last part of the video above the X button on the joystick is pressed and the Anti-Instagram node gets relaunched. We can see in the last part of the video RVIZ that the number of detected line segments gets increased dramatically after the recalibration.



Figure 12.41. Anit Instagram failure on curve

## 5) Conclusion

---

Even though a slightly higher image resolution with higher top cutoff can improve the lane following performance slightly we stucked with the original resolution of 120x160 pixels with 40 pixels top cutoff because also other teams depend on the image resolution. We saw that our curvature estimation was able to detect the standard curves of Duckietown in many cases, at the same time it introduced a high latency which again lowered the performance. Therefore we decided to set the curvature resolution to 0 by default, which means that no curvature estimation is done. The code nevertheless is still in the lane filter to give a basis for further improvements.

Regarding the controller the test showed that the added integral part in the PID controller and the tuning of the control parameters gave a huge improvement of the lane following performance. The integrated feedforward part can not be used during the lane following, because it is depending on the curvature estimation. The feedforward part can be used by other teams (e.g. on intersections and parking lots) to integrate our controller.

The improved controller gives a clear improvement over the baseline controller as can be seen in [Figure 12.1](#) and [Figure 12.4](#)

## 12.6. Future avenues of development

As there is always more to do and the performance for both the controller and the estimator can still be further enhanced we list in this section some suggestions for next steps to take.

### 1) Estimator

---

To make curvature estimation applicable it has to be made more robust and at the same time more computationally efficient adding less delay to the system. In its current state the added delay is too high and the performance with curvature estimation switched on decreases.

### 2) Controller

---

- Integrate the inputs of other teams, [see](#).
- After doing the new kinematic calibration provided by the System Identification group:
  - The controller parameters should be adjusted according to the output of the calibration.
  - The correction factors `velocity_to_m_per_s` and `omega_to_rad_per_s` need to be adjusted or ideally become obsolete and thus need to be deleted.
- To reduce impact of time delays, e.g. a Smith Predictor could be implemented.
- For the activation of the remaining interfaces (e.g. intersection navigation and parking), the respective commented out sections of the final code needs to be activated and the integration needs to be completed in collaboration with the other teams.

### 3) General

---

- Anti-Instagram should be enhanced, in order to identify more line segments and perceive the correct color.
- Adding a polarization filter to reduce impact of reflections on color perception.
- New edge detection with higher accuracy.
- Replacing the Raspberry Pi with something more computationally powerful to ensure low latency and enable a more complex pose estimation.

## UNIT L-13

# The Saviors: preliminary report

### 13.1. Part 1: Mission and scope

#### 1) Mission statement

---

Detect obstacles, plan a route and drive around them.

#### 2) Motto

---

URBEM ANATUM TUTIOS FACENDA

(Duckietown is to be made safer)

#### 3) Project scope

---

##### What is in scope

Detecting cones and duckies of different sizes (obstacles) and plan a reasonable path or stop to avoid hitting them.

Stage 1: 1 obstacle and simply stop no crossing of lines (2 cases: drive by or stop)

Stage 2: 1 obstacle, drive by without crossing of line

Stage 3: 1 obstacle, potentially cross the line

Stage 3: Multiple obstacles, crossing line if needed

### What is out of scope

No obstacles in crossings

Obstacles on the middle line

Complicated situations with oncoming traffic

### Stakeholders

*Controllers (Lane following, adaptive curvature control)*

They ensure following our desired trajectory and we can tell them to stop or reduce the speed. They provide the heading and position relative to track (for path planning)

*Vehicle detection - tbd*

*Potentially Anti-Instagram* They provide classified edges to limit the area where we have to find obstacles.

*Organizational Help* - System Architect - Software Architect - Duckiebook

## 13.2. Part 2: Definition of the problem

### 1) Problem statement

---

Reliably detect and avoid obstacles, plan a meaningful path around them or simply stop if nothing else is possible.

Robustness to changes in:

- Obstacle size
- Obstacle color (but only slight changes in yellow/orange)
- Illumination

### 2) Assumptions

---

- Obstacles are only yellow duckies (different sizes) and orange cones.
- No duckies on the middle line.
- No obstacles on intersections.
- Heading and position relative to track given.
- Control responsible for following trajectory.
- Possibility to influence vehicle speed (slow down, stop).
- Calibrated camera

### 3) Approach

---

Stage 0: Collect enough data and annotate them

Stage 1: Develop a first obstacle detection algorithm

Stage 2: Agree on final internal and external interface

Stage 3: from now on, obstacle detection and trajectory planning can be developed in parallel

Stage 4: handle the case(s) involving: 1 obstacle, no crossing of lines (2 cases: drive by or stop) → simple logical conditions

Stage 5: handle the case(s) involving: 1 obstacle, crossing line if needed (1 case: should always be possible to drive by) → Either with grid map or obstacle coordinates

Stage 6: handle the case(s) involving: Multiple obstacles, crossing line if needed

Stage 7: verify the whole system

### 4) Functionality provided

---

- Detect Obstacles
- Plan path around them or decide to stop

### 5) Resources required / dependencies / costs

---

- Calibrated camera.
- Position estimate and position uncertainty.
- Execution of our desired control commands
- Enough computing power

### 6) Performance measurement

---

- Avoid/hit-ratio in Stages 4-6 (see Approach)
- Percentage of correctly classified obstacles on our picture dataset
- Both of the measures above in case of changing light conditions

### 7) Functionality-resources trade-offs

---

- Robust obstacle detection (many filters,...) vs. computational efficiency
- Maximizing speed (e.g. controllers might want to do that) vs. motion blur

## **13.3. Part 3: Preliminary design**

### 1) Modules

---

- Obstacle Detection in 2D space
- Reconstruct 3D obstacle coordinates and radius

- Path planning/ Decision making

## 2) Interfaces

---

### Detection 2D space

- *Input:*
- Camera image
- Current position and orientation
- Lane coordinates
- Camera intrinsics
- (Curvature of upcoming track)
- *Output:*
- 2D obstacle coordinates

### Reconstruction of 3D obstacle coordinates and radius

- *Input:*
- 2D obstacle coordinates
- Extrinsic
- *Output:*
- 3D obstacle coordinates and radius

### Avoid obstacle

- *Input:*
- 3D obstacle coordinates
- Obstacle size
- Lane information
- *Output:*
- Trajectory
- Control command

## 3) Specifications

---

No need to revise duckietown specifications

## 4) Software modules

---

- Detection and Projection Node
- Path Planning Node

## 13.4. Part 4: Project planning

### 1) Timeline

---

Date	Task Name	Target Deliverables
15/11/17	First Meeting	Preliminary Design Document
17/11/17	Record first bags	Pictures and Raw bag data
22/11/17	Exchange of ideas	Basic concept
29/11/17	Knowing Interfaces and State of the ArtFine concept	

Date	Task Name	Target Deliverables
06/12/17		First implementation
...	Testing	Optimized Code
...	Documentation	Duckuments
21/12/17		End of Project

## 2) Data collection

---

Images of duckies on the road.

Video of a duckiebot in duckietown with recordings of the different stages.

To log:

- Distance to middle
- Theta
- Images
- Velocity

## 3) Data annotation

---

Label obstacles

*Relevant Duckietown resources to investigate:*

Image processing

feature extraction

MIT2016 object detection

Lane detection

Anti instagram

*Other relevant resources to investigate:*

OpenCV (filtering, color and edge detection)

## 4) Risk analysis

---

Interfaces (control approach of trajectory)

Computation power

## 5) Risk analysis

---

Risk	Likelihood (1-10)	Impact	Actions required
Non robust State Estimation	tbd	very high	Communication/Collaboration with controlling subteam
Failure of following our desired control commands	tbd	very high	Communication/Collaboration with controlling subteam

Risk	Likelihood (1-10)	Impact	Actions required
Lack in computation power	5	high	early testing of whole system on duckiebot
Failure in duckie detection	4	extremely high	thorough testing on bags
Erroneously detecting the middle lane as duckie	7	middle	more sophisticated detection algorithm

## UNIT L-14

### The Saviors: intermediate report

#### 14.1. Part 1: System interfaces

##### 1) Logical architecture

---

##### Description of the desired functionality:

Detect duckies and cones during lane following. When the ducky/cone is not in the middle of the lane we try to avoid them, if avoidance is not possible, we simply stop. Our obstacle avoidance capability is limited due to the fact that the controllers cannot make our Duckiebot cross the lane. In a first step, if we are in an intersection, our node will stop performing its tasks. However, if we progress really fast we will also have a look at the case of obstacles in intersections.

##### What will happen when we click “start”?

If you click start our nodes are launched and we will start to detect duckies.

*In general our obstacle\_detection\_node is always active. The “flow” is as follows:*

1. Our “obstacle\_detection”-node is activated and runs continuously. This node creates an instance of the class “detector” at the very beginning.
2. We have an incoming filtered image stream from the “anti\_instagram”-node to our node with the frequency, the “anti\_instagram”-node is publishing. Our “obstacle\_detection”-node regularly (i.e. with at least 2 Hz) calls the class functions of the “detector” instance.
3. For each frame the detector was called, the class function decides whether obstacles (duckies or traffic cones) are within the range of vision or not. Afterwards the “obstacle\_detection”-node publishes an array containing all coordinates of the obstacles which is empty if there are none. The published topic is called “obstacle\_coordinates”.
4. The “obstacle\_avoidance”-node takes this array as input and analyses it regarding which actions should be performed. The following scenarios are possible:

- 4a) The array is empty, therefore no action is performed and no flag will be pub-

lished.

**4b)** At least one obstacle (e.g. a Duckie) is within the range of  $\frac{1}{4}$  to  $\frac{3}{4}$  of the lane (i.e. in the middle of our lane) such that it cannot be avoided without going to the opposite lane or driving outside of the street. In this case the “obstacle\_avoidance”-node will set the “emergency\_brake\_flag” to true by publishing the topic “emergency\_brake\_flag”. This will indicate to “The Controllers” to immediately stop the Duckiebot.

**4c)** If an obstacle is detected and it lies within the left or right quarter of our lane it can be avoided. In this case we will set the “obstacle\_flag” to true by publishing this topic and also provide the controllers an input d (again by publishing a topic) to drive on the right side or the left side of the lane respectively.

5. If we passed the obstacle or if the obstacle disappeared we reset the “emergency\_brake\_flag” or the “obstacle\_flag” and the controllers can take over again.

6. Additionally we are listening to the “lane\_following\_flag” which indicates, if true, that the lane following is active. This tells us to perform our tasks. Otherwise our module should be inactive as our commands wouldn’t have any effects anyway. In this way we can save computing resources and avoid misunderstandings.

7. Additionally we implemented an additional file “obstacle\_detection\_visualizer” which can be used for visualization of the detected obstacles. It is thought to run on an laptop (which is connected to the same rosmaster) because it is only used for the evaluation of the quality of the detection as well as the 2D-3D projection. In principle it can run on the raspberry pi as well. It subscribes to “obstacle\_coordinates” and visualizes the obstacles in the 2D image by encircling the obstacles with a green rectangle (and publishing this modified image) as well as plotting them in the 3D coordinate frame (e.g. visualizable in RVIZ) as marker which shows the position as well as the size of the detected obstacles. With this additional software the optimization and the debugging are simplified significantly.

#### Expected target values for the following quantities:

- Detection Distance: 30-40cm
- Certainty of Detections: 90 percent
- Max. Amount of Detections in one frame: 3
- Max. Ratio of False Positives: 20 percent
- Obstacles avoided (without crash) to Obstacle detected ratio: 80 percent
- Object detection frame rate: at least 2 Hz

#### Assumptions on other modules:

- Control reaches desired d at most 10cm after request. Our request is “continuous”.
- Steady state control error smaller than 2cm
- d (perpendicular position to lane direction) Position estimate accuracy smaller than 2cm
- when setting the emergency\_flag, we stop within the next 5 cm

## 2) Software architecture

List of nodes which are to be developed:

- obstacle\_detection
- obstacle\_detection\_visualizer
- obstacle\_avoidance

Published and subscribed topics for each node, including an estimate of the introduced latency for the topics being published and an assumption on the latency for all subscribed topics:

### *obstacle\_detection*

*published topics:*

- obstacle\_coordinates (max 0.5s)

*subscribed topics:*

- cameraImage (no latency)

### *obstacle\_detection\_visualizer*

*published topics:*

- obst\_detect/image/compressed (max 0.5s)
- obst\_detect/visual/visualize\_obstacle (max 0.5s)

*subscribed topics:*

- obstacle\_coordinates (max 0.5s)
- cameraImage (no latency)

### *obstacle\_avoidance*

*published topics:*

- desired d (computing time)
- obstacle\_avoidance\_active\_flag (max. 0.3s)
- obstacle\_emergency\_stop\_flag (max. 0.3s)

*subscribed topics:*

- obstacle\_coordinates (max 0.5s)
- state\_estimation (current d, velocity, probably theta) (0.2s?)

## 14.2. Part 2: Demo and evaluation plan

### 1) Demo plan

---

How do you envision the demo?

Duckiebot driving around Duckietown with duckies in the street (on straights, (in curves), not in intersections)

- First the standard code will be running, which will result in Duckiebots crashing

into duckies on the street.

- Afterwards we will launch the Software from last year which will clearly show some strong losses in the quality of the obstacle detection.
- With our code running, the Duckiebot will avoid duckies by driving around them or stopping. If stopped we will remove the duckie which will make the bot drive again.

### What hardware components do you need?

- Enough duckies and traffic cones for being able to perform the evaluation. Traffic cones are already ordered and will be shipped on Monday 11th of December at the latest.
- The minimum city size to properly perform the demo is in our opinion an equivalent size as the one in the ML building.

## 2) Plan for formal performance evaluation

---

How do you envision the performance evaluation? Is it experiments? Log analysis?

### 1. Performance evaluation of the Saviors only:

*This first evaluation will be a general evaluation based on logs. We basically want to check if obstacles are detected correctly, and if the avoidance reacted as expected to prevent crashes.*

#### 1a) Detailed Evaluation Parameters:

- compare the labelled data (= groundtruth) to the results when putting the same image into our pipeline
- checking the actual frequency of our node
- test duckie recognition at different orientations and evaluate the maximum angle possible in which we still detect duckies
- evaluate the precision when having duckies only on our lane and on both, our lane and the opposite lane!!!
- evaluate the number of false positives in the 3 different situations: on straights, in “normal” curves, in the S-curve in duckietown
- measure the accuracy in centimeters of the real position of a duckie and the position we estimate!!!

### 2. Performance evaluation for the Saviors / Controllers / State estimation:

*In a first step we will evaluate our trajectory generation and control command by our obstacle\_avoidance node in designed situations. This means that we assume having a certain state, position of an obstacle and then we verify that we plan a feasible path around the obstacle without crossing the lane or to make the decision to stop*

#### 2a) Detailed evaluation Parameters:

- percentage of correct decisions

*The second step will be with the real state estimation, lane following and obstacle detection. This evaluation will be based on experiments with our duckiebot driving around town with obstacles in the lane.*

**2b) Detailed evaluation Parameters:**

1. percentage of correct decisions
2. percentage of correct executions when having made a correct decision, meaning really driving around the duckie without hitting it or sending the stop command early enough to not crash into duckies

**2c) the 2 measures above will be evaluated in different situations:**

1. only place duckies on straights
2. only place duckies in turns
3. only place duckies in the S-Turns
4. no limit on placing of the duckies

**2d) All of the above 4 situation will be also evaluated with:**

- Variation 1: only duckies in our lane
- Variation 2: also duckies on the opposite lane possible

In our case, only the performance evaluation on bags can be designed for minimal human intervention whereas all the other test have to be done in presence of a human who is able to stop the system when a potential crash occurs.

### **14.3. Part 3: Data collection, annotation, and analysis**

**1) Collection**

---

**How much data do you need?**

We already recorded 13 rosbags and are testing and improving the current obstacle detection. For the first step we think that this should work out.

**How are the logs to be taken? (Manually, autonomously, etc.)**

We took them manually but tried to “simulate” some non optimal controller behaviour, also because the autonomous mode did not work. For the future we might consider taking autonomous logs but only if we don’t crash into obstacles

**Do you need extra help in collecting the data from the other teams?**

No

**2) Annotation**

---

**Do you need to annotate the data?**

Yes!

At this point, you should have tried using [thehive.ai](https://thehive.ai) to do it. Did you? Are you sure they can do the annotations that you want?

Yes we already tried it and it worked quite well. We use this platform to label duckies

for us so that it will be possible to measure the performance of our duckie detection automatically on logs. Otherwise a human would need to label all of the pictures which would take a lot of time.

### 3) Analysis

---

**Do you need to write some software to analyze the annotations?**

We plan to write some software which applies our algorithm on every image of the log and which will mark the duckies using a box around them. Afterwards it will use the annotated data to read out where they have drawn the box and calculate the distance as well as visualize these results and differences. We then try to calculate the percentage of reasonable results and the percentage of outliers.

**Are you planning for it?**

Yes.

## UNIT L-15

# The Saviors: Final Report

**TODO:** JT: fix math formatting, video aspect ratio, standardize appearance, distribute “preliminaries” contributions in here appropriately throughout the book

This is the final report of the fall 2017 Saviors group from ETH Zurich, namely Fabio Meier (fmeier@ethz.ch), Julian Nubert (nubertj@ethz.ch), Fabrice Oehler (foehler@ethz.ch) and Niklas Funk (nfunk@ethz.ch). We enjoyed contributing to this great project and in case there are any open questions left after having read this report, do not hesitate to contact us.

## 15.1. Structure

[The Final Result](#)

[Mission and Scope](#)

[Definition of the Problem](#)

[Contribution / Added Functionality](#)

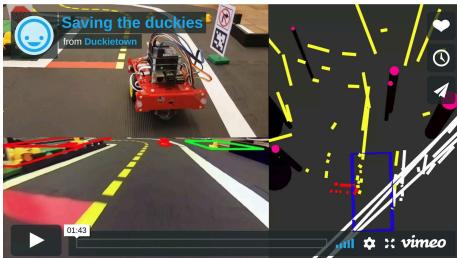
[Formal Performance Evaluation / Results](#)

[Future Avenues of Development](#)

[Theory Chapter](#)

## 15.2. The Final Result

The Saviors Teaser:



**Note:** See the [operation manual \(master\)](#) to reproduce these results.

The code description can be found here in the [Readme](#).

## 15.3. Mission and Scope

**“URBEM ANATUM TUTIOS FACIENDA (EST) - MAKE DUCKIETOWN A SAFER PLACE”**

The goal of Duckietown is to provide a relatively simple platform to explore, tackle and solve many problems linked to autonomous driving. “Duckietown” is simple in the basics, but an infinitely expandable environment. From controlling single driving Duckiebots until complete fleet management, every scenario is possible and can be put into practice. Due to the previous classes and also the great work of many volunteers, many software packages were already developed and provided a solid basis. But something was still missing.

### 1) Motivation

So far, none of the mentioned modules was capable of reliably detecting obstacles and reacting to them in real time. We were not the only ones who saw a problem in the situation at that time: *“Ensuring safety is a paramount concern for the citizens of Duckietown. The city has therefore commissioned the implementation of protocols for guaranteed obstacle detection and avoidance.”* [34]. Therefore the foundation of our complete module lies in the disposal of this shortcoming. Finding a good solution for this safety related and very important topic helped us to stay motivated every day we were trying to improve our solution.

The goal of our module is to detect obstacles and react accordingly. Due to the limited amount of time, we focused the scope of our module to two points:

1. In terms of detection, on the one hand we focused to reliably detect yellow duckies and therefore to saving the little duckies that want to cross the road. On the other hand we had to detect orange cones to not crash into any construction site in Duckietown.
2. In terms of reacting to the detected obstacles we were mainly restricted by the

constraint given by the controllers of our Duckiebots, who do not allow us to cross the middle of the road. This eliminated the need of also having to implement a Duckiebot detection algorithm. So we focused on writing software which tries to avoid obstacles within our own lane if it is possible (e.g. for avoiding cones on the side of the lane) and to stop otherwise.

Besides aforementioned restrictions and simplifications we faced the general problem of detecting obstacles given images from a monocular RGB camera mounted at the front of our Duckiebot and reacting to them properly without crashing or erroneously stopping the Duckiebot. Both processes above have to be implemented and have to run on the Raspberry Pis in real time. Due to the strong hardware limitations, we decided to not use any learning algorithms for the obstacle detection part. As it later transpired, a working “hard coded” software needs thorough analysis and understanding of the given problem. However, in the future, considering additional hardware like e.g. [Tung, “Google offers Raspberry Pi owners this new AI vision kit” \(2017\)](#), this decision might have to be reevaluated.

In practice a well working obstacle detection is one of the most important parts of an autonomous system to improve the reliability of the outcome even in unexpected situations. Therefore the relevance of an obstacle detection in a framework like “Duckietown” is very important. Especially because the aim of “Duckietown” is to simulate the real world as realistic as possible and also in other topics such as fleet planning, a system with obstacle detection behaves completely different than a system without.

## 2) Existing Solution

---



There was a previous implementation from the MIT classes in 2016. Of course we had a look into the old software and found out that one step of them was quite similar to ours: They based their obstacle detection on the colors of the obstacles. Therefore they also did their processing in the HSV color space as we did. Further information on why filtering colors in the HSV space is advantageous can be found in the [Theory Chapter](#).

Nevertheless, we implemented our solution from scratch and didn't base ours on any further concepts found in their software. That is why you won't find any further similarities between the two implementations. The reasons for implementing our own code from scratch can be found in the next section [Opportunity](#). In short, last year's solution considered the image given the original camera's perspective and tried to classify the objects based on their contour. We are using a very different approach concerning those two crucial parts as you can see in the [Contribution](#) section.

## 3) Opportunity

---



From the beginning it was quite clear that the old software was not working reliable enough. The information we have been given was that it was far off detecting all obstacles and that there were quite a few false positives: It detected yellow line segments in the middle of the road as obstacles (color and size are quite similar to the ones of typical duckies) which led to a stopping of the car. Furthermore, ex-

tracting the contour of every potential obstacle is highly computationally expensive. As mentioned, we had a look into the software and tried to understand it as well as possible but because it was not documented at all we couldn't go much into detail. On top of that, from the very beginning we had a completely different idea of how we wanted to tackle these challenges.

We also tried to start their software but we couldn't make it run after a significant amount of time. The readme file didn't contain any information and the rest of the software was not documented as well. This also reinforced us in our decision to write our own implementation from scratch.

#### 4) Preliminaries

Since our task was to reliably detect obstacles using a monocular camera only, we mainly dealt with processing the camera image, extracting the needed information, visualizing the results and to act accordingly in the real world.

For understanding our approach we tried to explain and summarize the needed concepts in the theory chapter (see section [Theory Chapter](#)). There you will find all the references to the relevant sources.

#### 15.4. Definition of the Problem

In this chapter we try to explain our problem in a more scientific way and want to show all needed steps to fulfill the superordinate functionality of "avoiding obstacles".

The only input is a RGB colored image, taken by a monocular camera (only one camera). The input image could look as [Figure 15.1](#).



Figure 15.1. Sample image including some obstacles

With this information given, we want to find out whether an obstacle is in our way or not. If so, we want to either stop or adapt the trajectory to pass without crashing into the obstacle. This information is then forwarded as an output to the controllers who will process our commands and try to act accordingly.

Therefore one of the first very important decisions was to separate the *detection* and *reaction* parts of our **saviors pipeline**. This decision allowed us to divide our work efficiently, to start right away and is supposed to ensure a wide range of flex-

ibility in the future by making it possible to easily replace, optimize or work on one of the parts separately (either the obstacle avoidance strategies or obstacle detection algorithms). Of course it also includes having to define a clear, reasonable interface in between the two modules, which will later be explained in detail.

You can have a look in our [Preliminary Design Document](#) and [Intermediate Report](#) to see how we defined the following topics in the beginning: The problem statement, our final objective, the underlying assumptions we lean on and the performance measurement to quantitatively check the performance of our algorithms. For the most part, it worked out to adhere to this document but for sake of completeness we will shortly repeat them again in the following for each of the two submodules.

## 1) Part 1: Computer Vision - Description

---

In principle we wanted to use the camera image only to *reach the following*:

1. Detect the obstacles in the camera image
2. Visualize them in the camera image for tuning parameters and optimizing the code
3. Give the 3D coordinates of every detected obstacle in the real world
4. Give the size of every detected obstacle in the form of a radius around the 3D coordinate
5. Label each obstacle if it's **inside or outside the lane boundaries** (e.g. for the purpose of not stopping in a curve)
6. Visualize them as markers in the 3D world (rviz)

Since every algorithm has its limitations, we made the following *assumptions*:

- Obstacles are only yellow duckies and orange cones
- Calibrated camera including intrinsics and extrinsics

Those assumptions changed slightly since the *Preliminary Design Document* because we are now also able to detect duckies on the middle line and in intersections.

It was our aim to reach the maximum within these specified limits. Therefore our goal was not only the detection and visualization in general but we also wanted to reach a **maximum in robustness** with respect to changes in:

- Obstacle size
- Obstacle color (within orange, and yellow to detect different traffic cones and duckies)
- Illumination

For evaluating the *performance*, we used the following metrics, evaluated under different light conditions and different velocities (static and in motion):

- Percentage of correctly classified obstacles on our picture datasets
- Percentage of false positives
- Percentage of missed obstacles

An evaluation of our goals and the reached performance can be found in the [Per-](#)

[formance Evaluation](#) section.

Our **approach** is simply based on analysing incoming pictures for obstacles and trying to track them to make the algorithm more robust against outliers. Since we only rely on the monocular camera, we do not have any direct depth information given. In theory, it would be possible to estimate the depth of each pixel through some monocular visual odometry algorithm considering multiple consecutive images. However this would be extremely computationally expensive. The large amount of motion blur in our setup, a missing IMU (for estimating the absolute scale) further argue against such an approach. In our approach we use the extrinsic calibration to estimate the position of the given obstacles. The intuition behind that is that it is possible to assume that all pixels seen from the camera belong to the ground plane (except for obstacles which stand out of it) and that the Duckikebot's relative position to this ground plane stays constant. Therefore you can assign a real world 3D coordinate to every pixel seen with the camera. For more details refer to the [section below](#).

The final output is supposed to look as [Figure 15.2](#).

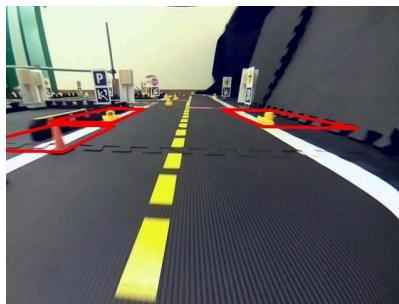


Figure 15.2. Final output image including visualization of detected obstacles

## 2) Part 2: Avoidance in Real World - Description

With the from [Part 1](#) given 3D position, size and the labelling whether the object is inside the lane boundaries or not, we wanted to reach the final objectives:

1. Plan path around obstacle if possible (we have to stay within our lane)
2. If this is not possible, simply stop

The assumptions for correctly reacting to the previously detected obstacles are:

- Heading and position relative to track given
- “The Controllers” are responsible for following our trajectory
- Possibility to influence vehicle speed (slow down, stop)

As we now know, the first assumption is normally not fulfilled. We describe in the [functionality section](#) why this comes out to be a problem.

For measuring the performance we used:

- Avoid/hit ratio
- Also performed during changing light conditions

## 15.5. Contribution / Added Functionality

### 1) Software Architecture

In general we have four interfaces which had to be created throughout the implementation of our software:

1. At first, we need to receive an incoming picture which we want to analyse. As our chosen approach includes filtering for specific colors, we are obviously dependent on the lighting conditions. In a first stage of our project, we nevertheless simply subscribed to the raw camera image because of the considerable expense of integrating the *Anti Instagram Color Transformation* and since the *Anti Instagram* team also first had to further develop their algorithms. During our tests we quickly recognized that our color filtering based approach would always have some troubles if we don't compensate for the lighting change. Therefore, in the second part of the project we closely collaborated with the *Anti Instagram team* and are now subscribing to a color corrected image provided by them. Currently, to keep computational power on our Raspberry Pi low, the corrected image is published at 4Hz only and the color transformation needs at most 0.2 seconds.

2. The second part of our System Integration is the internal interface between the object detection and avoidance part. The interface is defined as a *PoseArray* which has the same timestamp as the picture from which the obstacles have been extracted. This Array, as the name already describes, is made up of single poses. The meaning of those are the following:

The position *x* and *y* describe the real world position of the obstacle which is in our case the center front coordinate of the obstacle. Since we assume planarity, the *z coordinate* of the position is not needed. That is why we are using this *z coordinate* to describe the radius of the obstacle.

Furthermore a negative *z coordinate* shows that there is a white line in between us and the obstacle which indicates that it is not dangerous to us since we assume to always having to stay in the lane boundaries. Therefore this information allows us to not stop if there is an obstacle behind a turn.

As for the scope of our project, the orientation of the obstacles is not really important, we use the *remaining four elements* of the Pose Message to pass the pixel coordinates of the bounding box of the obstacle seen in the bird view. This is not needed for our "Reaction" module but allows us to implement an efficient way of visualisation which will be later described in detail. Furthermore, we expect our obstacle detection module to add an additional delay of about max. 0.3s.

3. The third part is the interface between our obstacle avoidance node and the *Controllers*. The obstacle avoidance node generates an obstacle avoidance *pose array* and obstacle avoidance *active flag*.

The obstacle avoidance pose array is the main interface between the Saviors and the group doing lane control. We use the pose array to transmit *d\_ref*(target distance to middle of the lane) and *v\_ref*(target robot speed). The *d\_ref* is our main control output which enables us to position the robot inside the lane and there-

fore to avoid objects which are placed close the laneline on the track. Furthermore `v_ref` is used to stop the robot when there is an unavoidable object by setting the target speed to zero.

The `flag` is used to communicate to the lane control nodes when the obstacle avoidance is activated which then triggers `d_ref` and `v_ref` tracking.

4. The fourth part is an optional interface between the Duckiebot and the user's personal Laptop. Especially for the needs of debugging and inferring what is going on, we decided to implement a visualisation node which can visualize on the one hand the input image including bounding boxes around all the objects which were classified as obstacles and furthermore this node can output the obstacles as markers which can be displayed in rviz.

In the following ([Figure 15.3](#)) you find a graph which summarises our software packages and gives a brief overview.

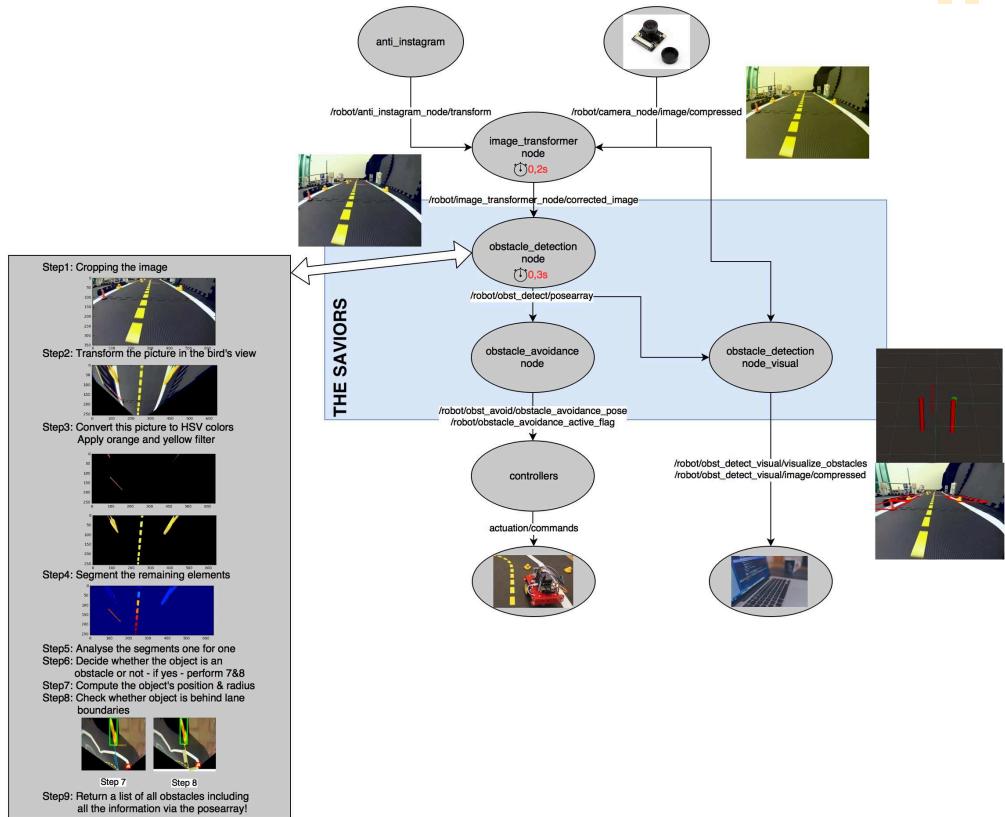


Figure 15.3. Module overview 'The Saviors'

## 2) Part 1: Computer Vision - Functionality



Let's again have a look on the usual incoming camera picture in [Figure 15.1](#).

In the very beginning of the project, like the previous implementation in 2016, we tried to do the detection in the normal camera image but we tried to optimize for more efficient and general obstacle descriptors. Due to the specifications of a normal camera, lines which are parallel in the real world are in general not parallel any longer and so the size and shape of the obstacles are disturbed (elements of the same size appear also larger in the front than in the back). This made it very difficult to reliably differentiate between yellow ducks and line segments. We tried several different approaches to overcome this problem, namely:

- Patch matching of duckies viewed from different directions
- Patch matching with some kind of an ellipse (because line segments are supposed to be square)
- Measuring the maximal diameter
- Comparing the height and the width of the objects
- Taking the pixel volume of the duckies

Unfortunately none of the described approaches provided a sufficient performance. Also a combination of them didn't make the desired impact. All metrics which are somehow associated with the size of the object just won't work because duckies further away from the duckiebot are simply a lot smaller than the one very close to the Duckiebot. All metrics associated with the "squareness" of the lines were strongly disturbed by the occurring motion blur. This makes finding a general criterion very difficult and made us think about changing the approach entirely.

Therefore we developed and came up with the following new approach!

#### *Theoretical Description:*

In our setup, through the extrinsic camera calibration, we are given a mapping from each pixel in the camera frame to a corresponding real world coordinate. It is important to mention that this transformation assumes all seen pixels in the camera frame to lie in one plane which is in our case the ground plane/street. Our approach exactly exploits this fact by transforming the given camera image into a new, bird's view perspective which basically shows one and the same scene from above. Therefore the information provided by the extrinsic calibration is essential for our algorithm to work properly. In [Figure 15.4](#) you can see the newly warped image seen from the *bird's view perspective*. This is one of the most important steps in our algorithm.

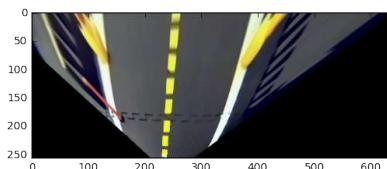


Figure 15.4. Image now seen from the bird's view perspective

This approach has already been shown by Prof. Davide Scaramuzza (UZH) and some other papers and is referred as **Inverse Perspective Mapping Algorithm**. (see: [\[35\]](#),[\[36\]](#),[\[37\]](#))

What stands out, is that the lines which are parallel in the real world are also parallel in this view. Generally in this “bird’s” view, all objects which really belong to the ground plane are represented by their real shape (e.g. the line segments are exact rectangles) while all the objects which are not on the ground plane (namely our obstacles) are heavily disturbed in this top view. This top view is roughly keeping the size of the elements on the ground whereas the obstacles are displayed a lot larger.

*The theory behind the calculations and why the objects are so heavily distorted can be found in the [Theory Chapter](#).*

Either way, we take advantage of this property. Given this bird’s view perspective, we still have to extract the obstacles from it. To achieve this extraction, we first filter out everything except for orange and yellow elements, since we assumed that we only want to detect yellow duckies and orange cones. To simplify this step significantly, we transform the obtained color corrected images (provided by the Anti Instagram module) to the **HSV color space**. We use this HSV color space and not the RGB space because it is much easier to account for slightly different illuminations - which of course still exist since the performance of the color correction is logically not perfect - in the HSV room compared to RGB. For the theory behind the HSV space, please refer to our appropriate [Theory Chapter](#).

After this first color filtering process, there are only objects remaining which have approximately the colors of the expected obstacles. For the purpose of filtering out the real obstacles from the bunch of all the remaining objects which passed the color filter, we decided to do the following: We segment the image of the remaining objects, i.e. all connected pixels in the filtered image are getting the same label such that you can later analyse the objects one by one. Each number then represents an obstacle. For the process of segmentation, we used the following algorithm. (see [\[38\]](#))

Given the isolated objects, the task remains to finally decide which objects are considered obstacles and which not. In a first stage, there is a filter criterion based on a rotation invariant feature, namely the two eigenvalues of the `inertia_tensor` of the segmented region when rotating around its center of mass. (see [\[39\]](#))

In a second stage, we apply a tracking algorithm to reject the remaining outliers and decrease the likelihood for misclassifications. The tracker especially aims for objects which passed the first stage’s criterion by a small margin.

For further informations and details about how we perform the needed operations, please refer to the next chapter.

The final output of the detection module is the one we showed in [Figure 15.2](#).

#### *Actual Implementation:*

Now we want to go more into detail how we implemented the described steps.

In the beginning we again start from the picture you can see in [Figure 15.1](#). In our case this is now the corrected image coming out form the `image_transformer_node` and was implemented by the `anti instagram` group. We then perform

the following steps:

1. In a first step we crop this picture to make our algorithm a little bit more efficient and due to our limited velocities, it makes no sense to detect obstacles which are not needed to be taken into consideration by our obstacle avoidance module. However, we do not simply crop the picture by a fixed amount of pixels, but we use the extrinsic calibration to neglect all the pixels which are farther away than a user defined threshold, which is at the moment at 1.7 meters. So the amount of pixels which are neglected are different for every Duckiebot and depend on the extrinsic calibration. The resulting image can be seen in [Figure 15.5](#). The calculations to find out where You have to cut the image are quite simple (note that it still bargains for homogeneous coordinates):

```
$$ p_{camera} = H^{-1}P_{world} $$
```

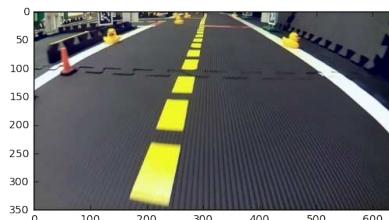


Figure 15.5. Cropped image

2. Directly detecting the obstacles from this cropped input image failed for us due to the reasons described [above](#). That is why the second step is to perform the transformation to the bird's view perspective. For transforming the image, we first use the corners of the cropped image and transform it to the real world. Then we scale the real world coordinates to pixel coordinates, so that it will have a width of 640 pixels afterwards. For warping all of the remaining pixels with low artifacts we use the function `cv2.getPerspectiveTransform()`. The obtained image can be seen in [Figure 15.4](#).

3. Then we transform the given RGB picture into the HSV colorspace and apply the yellow and orange filter. While a HSV image is hardly readable for humans, it is way better to filter for specific colors. The obtained pictures can be seen in [Figure 15.6](#) and [Figure 15.7](#). The color filter operation is performed by the cv2 function `cv2.inRange(im_test, self.lower_yellow, self.upper_yellow)` where lower\_yellow and upper\_yellow are the thresholds for yellow in the HSV color space.

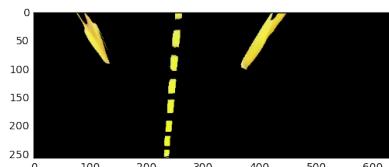


Figure 15.6. Yellow filtered image

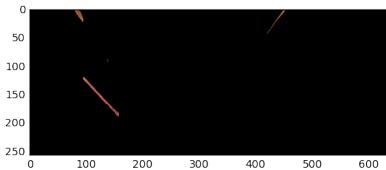


Figure 15.7. Orange filtered image

4. Now there is the task of segmenting/isolating the objects which remained after the color filtering process. At the beginning of the project we therefore implemented our own segmentation algorithm which was however more inefficient and led to an overall computational load of 200% CPU usage and a maximum frequency of our whole module of about 0.5 Hz only. By using the scikit-image module which provides a very efficient [label function](#), the computational efficiency could be shrunk considerably to about 70% CPU usage and allows the whole module to run at up to 3 Hz. It is important to remember that in our implementation the segmentation process is the one which consumes the most power. The output after the segmentation is the one in [Figure 15.8](#), where the different colors represent the different segmented objects.

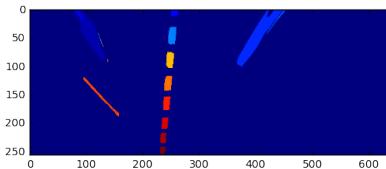


Figure 15.8. Segmented image

5. After the segmentation, we analyse each of the objects separately. At first there is a general filter which ensures that we are neglecting all the objects which contain less than a user influenced threshold of pixels. Since as mentioned above, the homographies of all the users are different, the exact amount of pixels, an object is required to have, is again scaled by the individual homography. This is followed by a more in detail analysis which is color dependent. On the one hand there is the challenge to detect the orange cones reliably. Speaking about cones, the only other object that might be erroneously detected as orange are the stop lines. Of course, in general the goal should be to very reliably detect orange but as the light is about to change during the drive, we prepared to also detect the stop lines and being able to cope with them when they are erroneously detected. The other general challenge was that all objects that we have to detect can appear in all different orientations. Simply inferring the height and width of the segmented box, as we did it in the beginning, is obviously not a very good measure (e.g. in [Figure 15.9](#) in the lower left the segmented box is square while the cone itself is not quadratic at all).

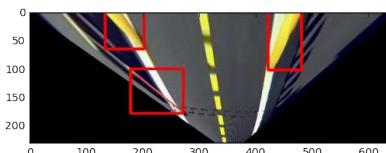


Figure 15.9. Bird's view with displayed obstacle boxes

That is why it is best to use a rotation invariant feature to classify the segmented object. In our final implementation we came up with using the two eigenvalues of the inertia tensor, which are obviously rotation invariant (when being ordered by their size). Being more specific about the detection of cones, when extracting the cone from [Figure 15.8](#) it is looking like in [Figure 15.10](#), while an erroneous detection of a stop line is looking like in [Figure 15.11](#).

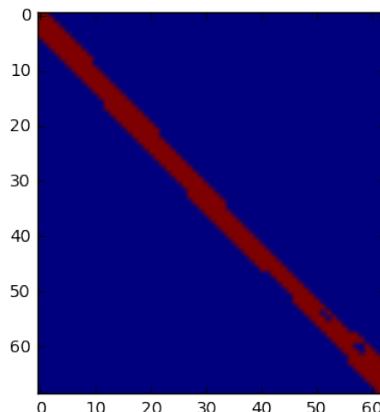


Figure 15.10. Segmented cone

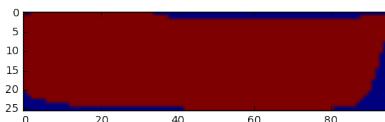


Figure 15.11. Segmented stop line

Our filter criteria is now the ratio between the eigenvalues of the inertia tensor. This ratio is always by a factor of about 100 greater in case the object is a cone, compared to when we erroneously segment a red stop line. This criteria is very stable that is why there is no additional filtering needed to detect the cones.

If the segmented object is yellowish, things get a little more tricky as there are always many yellow objects in the picture, namely the middle lines. Line elements can be again observed under every possible orientation. Therefore the eigenvalues of the inertia tensor, which are as mentioned above rotation invariant, are again the way to go. In [Figure 15.12](#) you can see a segmented line element and in [Figure 15.13](#) again a segmented duckie.

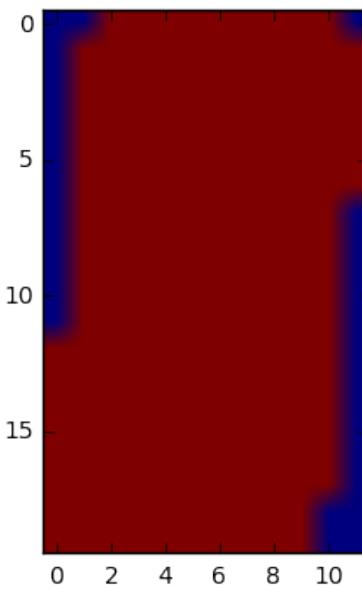


Figure 15.12. Segmented middle line

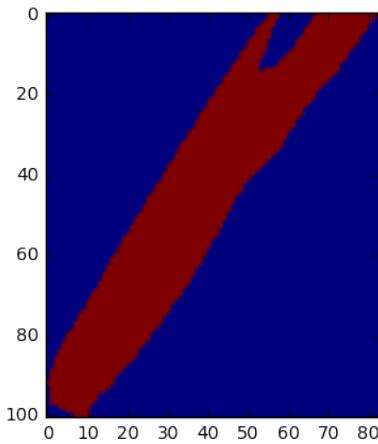


Figure 15.13. Segmented duckie

As the labelled axis already reveal, they are of a different scale, but as we also got very small duckies, we had to choose a very small threshold. To detect the yellow duckies, the initial condition is that the first eigenvalue has to be greater than 20. This criteria alone however includes to sometimes erroneously detecting the lines as obstacles, that is why we implemented an additional tracking algorithm which works as follows: If an object's first eigenvalue is greater than 100 pixels and it is detected twice - meaning in two consecutive images there is a object detected at roughly the same place - it is labelled as an obstacle. However, if an object is smaller or changed the size by more than 50% in the consecutive frames, then a more restrictive criteria is enforced. This more restrictive criterion states that we must have tracked this object for at least for 3 consecutive frames before being labelled as an obstacle. This criteria is working pretty well and a more thorough evaluation

will be provided in the next [section](#). In general those criteria help that the obstacles can be detected in any orientation. The only danger to the yellow detecting algorithm is motion blur, namely when the single lines are not separated but connected together by “blur”.

6. After analysing each of the potential obstacle objects, we decide whether it is an obstacle or not. If so, we continue to steps 7. and 8..

7. Afterwards, we calculate the position and radius of all of the obstacles. After segmenting the object we calculate the 4 corners (which are connected in [Figure 15.14](#) to form the green rectangle). We defined the obstacle’s position as the mid-point of the lower line (this point surely lies on the ground plane). For the radius, we use the distance in the real world between this point and the lower right corner. This turned out to be a good approximation of the radius. For an illustration you can have a look at [Figure 15.14](#).

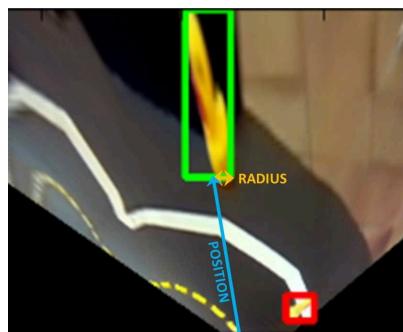


Figure 15.14. Position and radius of the obstacle

8. Towards the end of the project we came up with one additional last step based on the idea that only obstacles inside the white lane boundaries are of interest to us. That is why for each obstacle, we look whether there is something white in between us and the obstacle. In [Figure 15.15](#) you can see an example situation where the obstacle inside the lane is marked as dangerous (red) while the other one is marked as not of interest to us since it is outside the lane boundary (green). In [Figure 15.16](#) you see the search lines (yellow) along which we search for white elements.

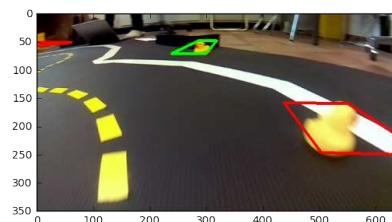


Figure 15.15. Classification if objects are dangerous or not

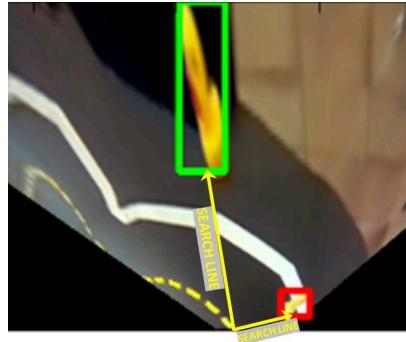


Figure 15.16. Search lines to infer if something white is in between

9. As the last step of the detection pipeline we return a list of all obstacles including all the information via the *Posearray*.

### 3) Part 2: Avoidance in Real World - Functionality

The Avoidance deals with drawing the right conclusions from the received data and forwarding it.

#### *Theoretical Description:*

With the separation of the detection, an important part of the avoidance node is the interaction with the other work packages. We determined the need of getting information about the remaining Duckietown besides the detected obstacles. The obstacles need to be in relation to the track, in order to assess whether we have to stop, can drive around obstacles or if it is even already out of track. Due to other teams already working on the orientation within Duckietown, we deemed it best to not implement any further detections (lines, intersections etc.) in our visual perception pipeline. This saves similar algorithms being run twice on the processor. We decided to acquire the values of our current pose relative to the side lane, which is determined by the *devel-linedetection* group.

The idea was to make the system highly flexible. The option to adapt to following situations was deemed desirable:

- Multiple obstacles. Different path planning in case of a possible avoidance might be required.
- Adapted behavior if the robot is at intersections.
- Collision avoidance dependent on the fleet status within the Duckietown. Meaning if a Duckiebot drives alone in a town it should have the option to avoid a collision by driving onto the opposite lane.

Obstacles sideways of the robot were expected to appear as the Duckietowns tend to be flooded by duckies. Those detections on the side as well as far away false positive detections should not make the robot stop. To prevent that, we intended on implementing a parametrized bounding box ahead of the robot. Only obstacles within that box would be considered. Depending on the certainty of the detections as well as the yaw-velocities the parametrization would be tuned.

The interface getting our computed desired values to impact the actual Duckiebot is handled by *devel-controllers*. We agreed on the usage of their custom message format, in which we send desired values for the lateral lane position and the longitudinal velocity. Our intention was to account for the delay of the physical system in the avoider node. Thus our planned trajectory will reach the offset earlier than the ideal-case trajectory would have to.

Due to above mentioned interfaces and multiple levels of goals we were aiming for an architecture which allows **gradual commissioning**. The intent was to be able to go from basic to more advanced for us as well as for groups in upcoming years. Those should be able to extend our framework and not have to rebuild it.

The logic shown in [Figure 15.17](#) displays one of the first stages in the commissioning. Key is the reaction to the number of detected obstacles. Later stages will not trigger an emergency stop in case of multiple obstacle detections within the bounding box.

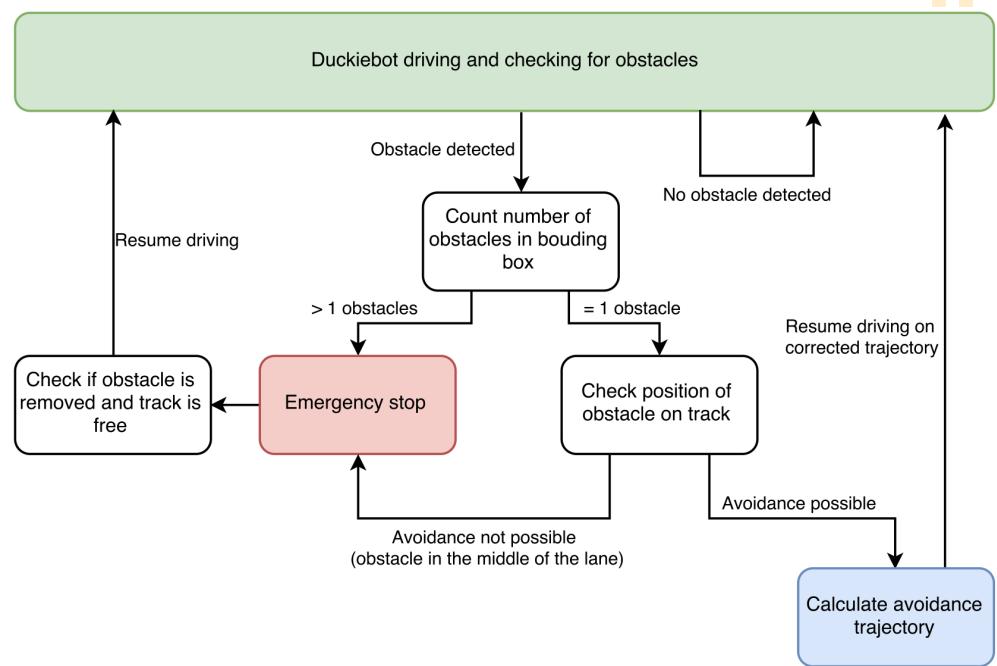


Figure 15.17. Logic of one of the First Stages in Commissioning

Our biggest concern were the added inaccuracies until the planning of the trajectory. Those include:

- Inaccuracy of the currently determined pose
- Inaccuracy of the obstacle detection
- Inaccuracy of the effectively driven path aka. controller performance

To us the determination of the pose was expected to be the most critical. Our preliminary results of the obstacle detection seemed reasonably accurate. The controller could be tweaked that the robot would rather drive out of the track than into the obstacle. Though an inaccurate estimation of the pose would just widen

the duckie artificially.

*Devel-controllers* did not plan on being able to intentionally leave the lane. Meaning the space left to avoid an obstacle on the side of the lane is tight making above uncertainties more severe.

We evaluated the option to keep track of our position inside the map. Given a decent accuracy of said position we'd be able to create a map of the detected obstacles. Afterwards - especially given multiple detections (also outside of the bounding box) - we could achieve a further estimation of our pose relative to the obstacles. This essentially would mean creating a *SLAM-algorithm* with obstacles as landmarks. We declared as out of scope given the size of our team as well as the computational constraints. The goal was to make use of a stable, continuous detection and in each frame react on it.

#### ***Actual Implementation:***



#### **Interfaces**

One important part of the Software is the handling of the interfaces, mainly to *devel\_controllers*. For further informations on this you can refer to the [Software Architecture Chapter](#).

#### **Reaction**

The obstacle avoidance part of the problem is handled by an additional node, called the *obstacle\_avoidance\_node*. The node uses two main inputs which are the obstacle pose and the lane pose. The obstacle pose is an input coming from the obstacle detection node, which contains an array of all the obstacles currently detected. Each array element consists of an x and y coordinate of an obstacle in the robot frame (with the camera as origin) and the radius of the detected object. By setting the radius to a negative value, the detection node indicates that this obstacle is outside the lane and should not be considered for avoidance. The lane pose is coming from the line detection node and contains among other unused channels the current estimated distance to the middle of the lane ( $d$ ) as well as the current heading of the robot  $\theta$ . [Figure 15.18](#) introduces the orientations and definitions of the different inputs which are processed in the obstacle avoidance node.

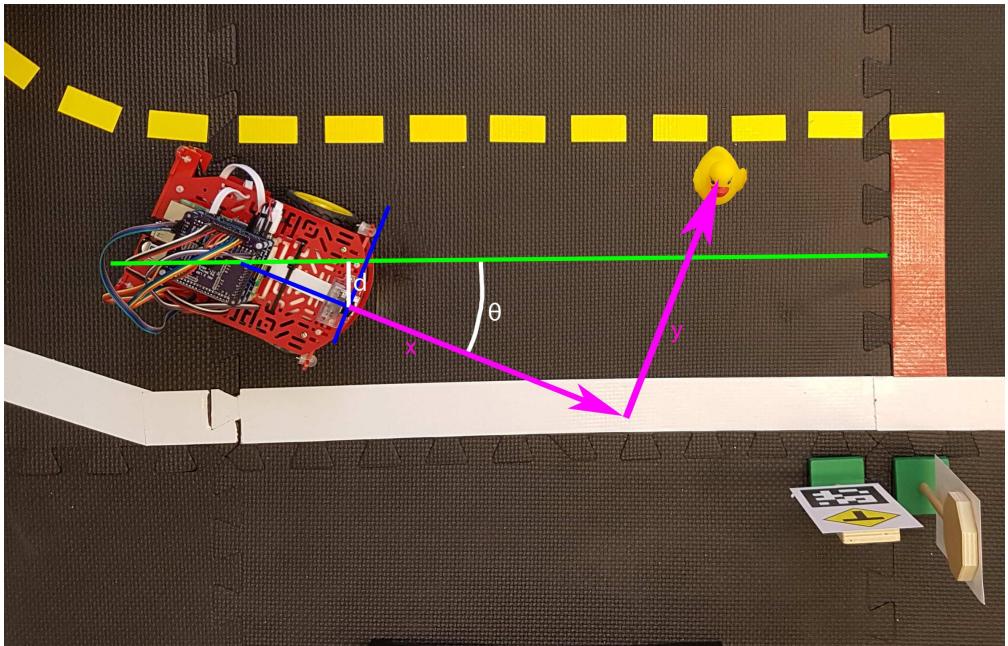


Figure 15.18. Variable Definitions seen from the Top

Using the obstacle pose array we determine how many obstacles need to be considered for avoidance. If the detected obstacle is outside the lane and therefore marked with a negative radius by the obstacle detection node we can ignore it. Furthermore, we use the before mentioned bounding box with tunable size which assures that only objects in a certain range from the robot are considered. As soon as an object within limits is inside of the bounding box, the obstacle\_avoidance\_active flag is set to true and the algorithm already introduced in [Figure 15.17](#) is executed.

#### Case 1: Obstacle Avoidance

If there is only one obstacle in range and inside the bounding box, the obstacle avoidance code in the avoider function is executed. First step of the avoider function is to transform the transmitted obstacle coordinates from the robot frame to a frame which is fixed to the middle of the lane using the estimated measurements of  $\theta$  and  $d$ . Doing this transformation allows us to calculate the distance of the object from the middle line. If the remaining space (in the lane (subtracted by a safety avoidance margin) is large enough for the robot to drive through we proceed with the obstacle avoidance, if not we switch to case 2 and stop the vehicle. Please refer to [Figure 15.19](#)

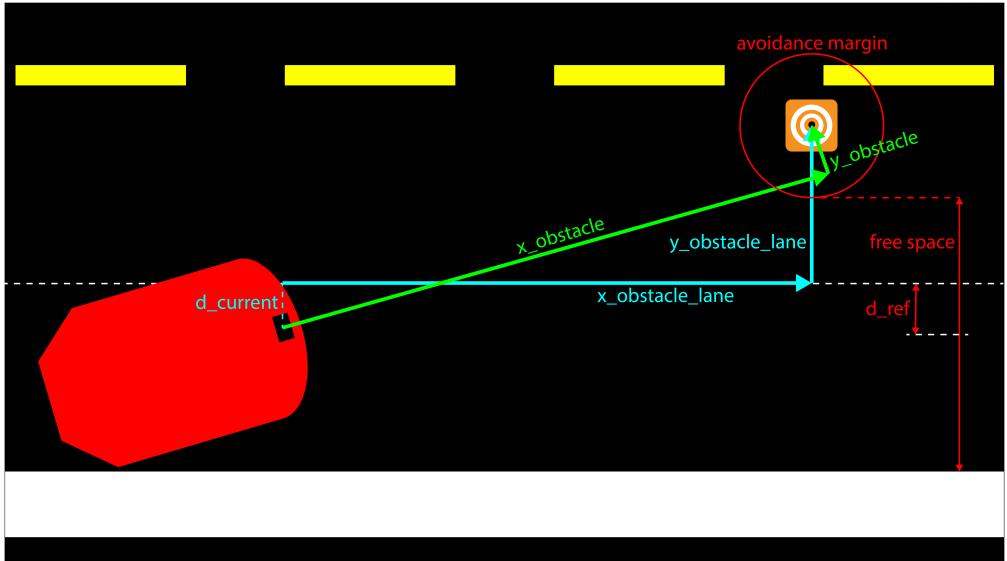


Figure 15.19. Geometry of described Scene

If the transformation shows that an avoidance is possible we calculate the  $d_{ref}$  we need to achieve to avoid the obstacle. This is sent to the lane control node and then processed as new target distance to the middle of the lane. The lane control node uses this target and starts to correct the Duckiebot's position in the lane. With each new obstacle pose being generated this target is adapted so that the Duckiebot eventually reaches target position. The slow transition movement allows us to avoid the obstacle even when it is not visible anymore shortly before the robot is at the same level as the obstacle.

At the current stage, the obstacle avoidance is not working due to very high inaccuracies in the estimation of  $\theta$ . The value shows inaccuracies with an amplitude of  $10^\circ$ , which leads to wrong calculations of the transformation and therefore to misjudgement of the  $d_{ref}$ . The high amplitude of these imprecisions could be transformed to a uncertainty factor of around 3 which means that each object is around 3 times its actual size which means that even a small obstacle on the side of the lane would not allow a safe avoidance to take place. For this stage to work, the estimation of  $\theta$  would need significant improvement.

### Case 2: Emergency Stop

Conditions for triggering an emergency stop:

- More than one obstacle in range
- Avoidance not possible because the obstacle is in the middle of the lane
- Currently every obstacle detection in the bounding box triggers an emergency stop due to the above reasons

If one of the above scenarios occurs, an avoidance is not possible and the robot needs to be stopped. By setting the target speed to zero, the lane controller node stops the Duckiebot. As soon as the situation is resolved by removing the obstacle which triggered the emergency stop, the robot can proceed with the lane follow-

ing.

These tasks are then repeated at the frame rate of the obstacle detection array being sent.

#### 4) Required Infrastructure - Visualizer

---



Especially when dealing with a vision based obstacle detection algorithm it is very hard to infer what is going on. One has to also keep the visual outputs low, to consume as less computing power as possible, especially on the Raspberry Pi. This is why we decided to not implement one single *obstacle detection node*, but effectively two of them, together with some scripts which should help to tune the parameters offline and to infer the number of false positives, etc.. The node which is designed to be run on the Raspberry Pi is our normal `obstacle_detection_node`. This should in general be run such that there is no visual output at all but that simply the `PoseArray` of obstacles is published through this node.

The other node, namely the `obstacle_detection_visual_node` is designed to be run on your own laptop which is basically visualising the information given by the `posearray`. There are two visualisations available. On the one hand there is a marker visualisation in `rviz` which shows the position and size of the obstacles. In here all the dangerous obstacles which must be considered are shown in red, whereas the non critical (which we think that they are outside the lane boundaries) are marked in green. On the other hand there is also a visualisation available which shows the camera image together with bounding boxes around the detected obstacles. Nevertheless, this online visualisation is still dependent on the connectivity and you can only hardly “freeze” single situations where our algorithm failed. That is why we also included some helpful scripts into our package. One script allows to thoroughly input many pictures and outputs them labelled together with the bounding boxes, while another one outputs all the intermediate steps of our filtering process which allows to fastly adapt e.g. the color thresholds which is in our opinion still the major reason for failure. More information on our created scripts can be found in our [Readme on GitHub](#).

#### 5) Recorded Logs

---



For being able to thoroughly evaluate and tune our algorithms, we recorded various bags, which we uploaded to the [Duckietown logs database](#).

### 15.6. Formal Performance Evaluation / Results

---



#### 1) Evaluation of the Interface and Computational Load

---



In general as we are dealing with many color filters a reasonable color corrected image is the key to the good functioning of our whole module, but turned out to be the greatest challenge when it comes down to computational efficiency and performance. As described above we are really dependent on a color corrected image by the *Anti Instagram* module. Throughout the whole project we planned to use their *continuous anti-instagram node* which is supposed to compute a color trans-

formation in fixed intervals of time. However, when it came down we acutally had to change this for the follwing reason: The continuous anti-instagram node, running at an update interval of 10 seconds, consumes a considerable amount of computing power, namely 80%. In addition to that, the image transformer node which is in fact transforming the whole image and currently running at 4 Hz needs another 74% of one kernel. If you now run those two algorithms combined with the lane-following demo which makes the vehicle move and combined with our own code which needs an additional 75% of computing power, our safety critical module could only run at 1.5Hz and resulted in poor behaviour.

Even if you increase the time interval in which the continuous anti-instagram node computes a new transformation there was no real improvement. That is why in our final setup we let the anti-instagram node once compute a reasonable transformation and then keep this one for the entire drive. Through this measure we were able to save the 80% share entirely and this allowed our overall node to be run at about 3 Hz with introducing an additional maximal delay of about 0,3 seconds. Nevertheless we want to point out that all the infrastructure for using the continuous anti instagram node in the future is provided in our package.

To sum up, the interface between our node and the Anti Instagram node was for sure developed very well and the collabroation was very good but when it came to getting the code to work, we had to take one step back to achieve good performance. That is why it might be reasonable to put effort into this interface in the future, to being able to more efficiently transform an entire image and to reduce the computational power consumed by the node which continuously computes a new transformation.

## 2) Evaluation of the Obstacle Detection

---



In general, since our obstacle classification algorithm is based on the rotational invariant feature of the eigenvalues of the inertia tensor it is completely invariant to the current orientation of the duckiebot and its position with respect to the lanes.

To rigorously evaluate our detection algorithm, we started off with evaluating **static scenes**, meaning the Duckiebot is standing still and not moving at all. Our algorithm performed extremely well in those static situations. You can place an arbitrary amount of obstacles, where the orientation of the respective obstacles does not matter at all, in front of the Duckiebot. In those situations and also combining them with changing the relative orientation of the Duckiebot itself, we achieved a false positive percentage of **below 1%** and we labelled all of the obstacles with respect to the lane boundaries correctly. The only static setup which is sometimes problematic is when we place the smallest duckies very close in front of our vehicle (below 4 centimeters), without approaching them. Then we sometimes cannot detect them. However this problem is mostly avoided during the dynamic driving, since we anyways want to stop earlier than 4 centimeters in front of potential obstacles. We are very happy with this static behaviour as in the worst case, if during the dynamic drive something goes wrong, you can still simply stop and rely upon the fact that the static performance is very good before continuing your drive. In [the log chapter](#) it is possible to find the corresponding logs.

This in return also implies that most of the misclassification errors during our **dynamic drive** are due to the effect of motion blur, assuming a stable color transformation provided by the anti instagram module. E.g. in [Figure 15.20](#) two line segments in the background “blurred” together for two consecutive frames resulting in being labelled as an obstacle.

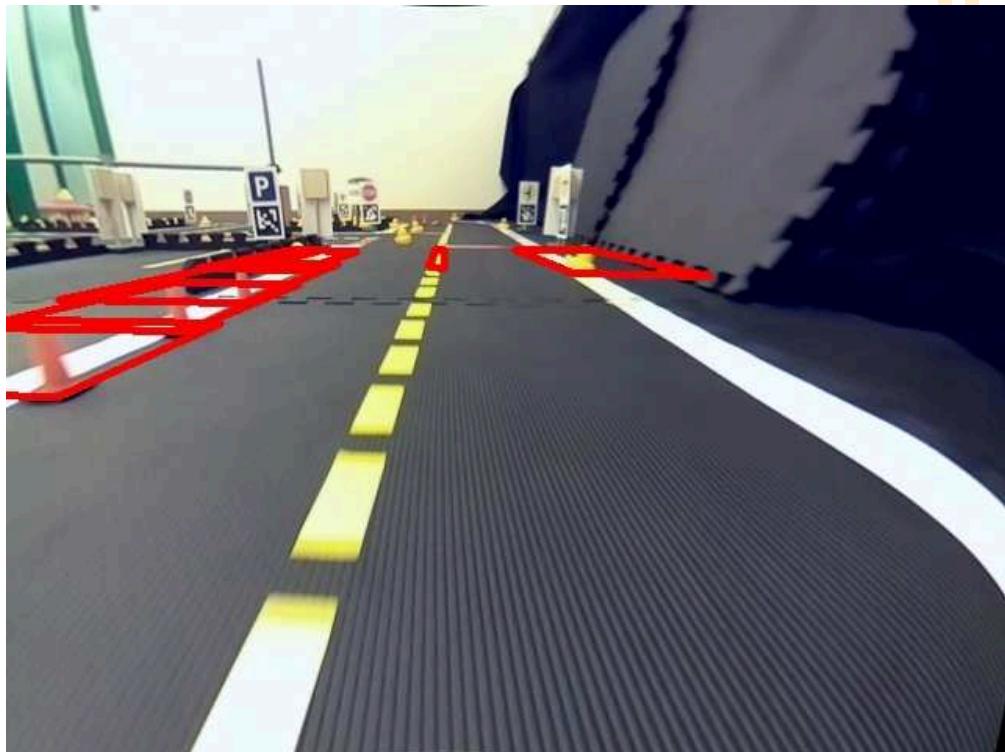


Figure 15.20. Obstacle Detector Error due to motion blur

Speaking more about of numbers, we took 2 duckiebots at a gain of around 0.6 and performed two drives at different days, so also at different lights and the results are the following: Evaluating each picture which will be given to the algorithm, we found out that on average, we detect 97% of all the yellow duckies in each picture. In terms of cones we detect about 96% of all cones in the evaluated frames. We consider these to be very good results as we have a very low rate of false positives (below 3%).

Date	#correctly detected duckies	#correctly detected cones	#missed ducks	#missed cones	#false positive	#false position
19/12/2017	423	192	14	8	9	45
Robot:Arki				3,2%	4%	1,4%
Duration:82s						7,2%
—						—
21/12/2017	387	103	10	5	15	28
Robot:Dori				2,5%	4,4%	3%
Duration:100s						5,7%

When it comes to evaluating the performance of our obstacle classification with

respect to classifying them as dangerous or not dangerous our performance is not as good as the detection itself, but we did also not put the same effort into it. As you can see in the table above, we have an error rate of **above 5%** when it comes to determining whether the obstacle's position is inside or outside the lane boundaries (this is denoted as false position in the table above). We are especially encountering problems when there is direct illumination on the yellow lines which are very reflective and therefore appear whitish. [Figure 15.21](#) shows such a situation where the current implementation of our obstacle classification algorithm fails.

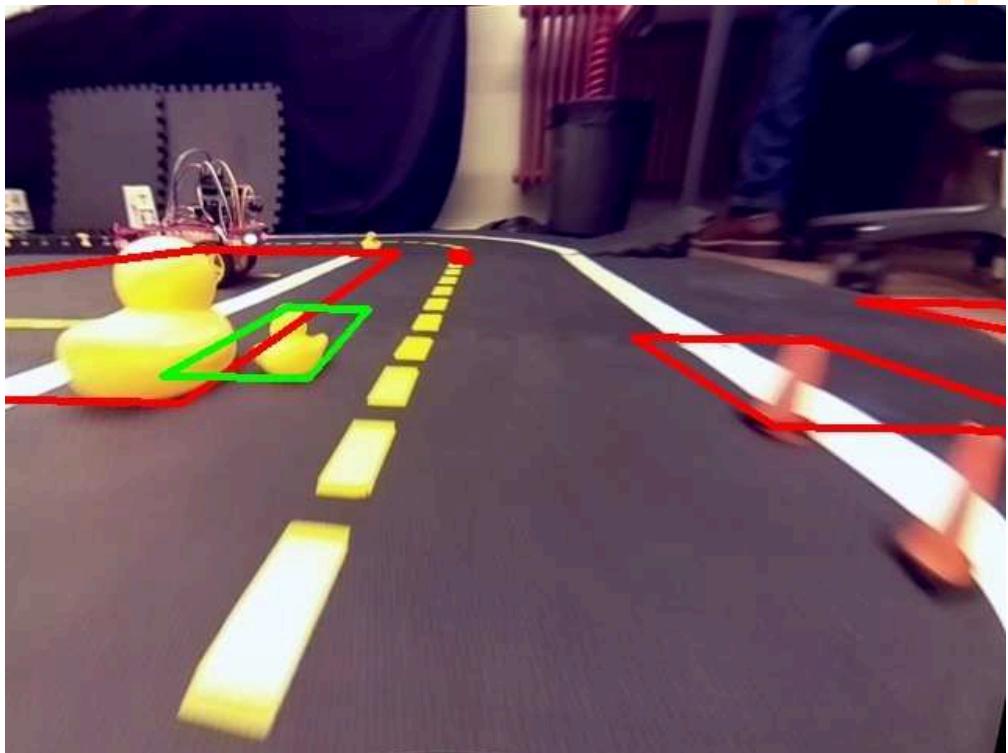


Figure 15.21. Obstacle Detector Classification Error

### 3) Evaluation of the Obstacle Avoidance

Since at the current state we stop for every obstacle which is inside the lane and inside the bounding box, the avoidance process is very stable since it does not have to generate avoidance trajectories. The final performance on the avoidance is mainly relying on the placement of the obstacles:

1. **Obstacle placement on a straight:** If the obstacle is placed on a straight with a sufficient distance from the corner the emergency stop works nearly every time if the obstacle is detected correctly.
2. **Obstacle in a corner:** Due to the currently missing information of the curvature of the current tile the bounding box is always rectangular in front of the robot. This leads to problems if an obstacle is placed in a corner because it might enter the bounding box very late (if at all). Since the detection very close to the robot is not

possible, this can lead to crashes.

**3. Obstacles on intersection:** These were not yet considered in our scope but still work if the detection is correct. It then behaves similar to case 1.

Furthermore there are a few cases which can lead to problems independent of the obstacle placement: **1. Controller oscillations:** If the lane controller sees a lot of lag due to high computing loads or similar its control sometimes start to oscillate. These oscillations lead to a lot of motion blur which can induce problems in the detection and shorten the available reaction time to trigger an emergency stop.

**2. Controller offsets:** The current size of the bounding box assumes that the robot is driving in the middle of the lane. If the robot is driving with an offset to the middle of the lane it can happen that obstacles at the side of the lane aren't detected. This however rarely leads to crashes because then the robot is just avoiding the obstacle instead of stopping for it.

While testing our algorithms we saw successful emergency stops in 10/10 cases for obstacles on a straight and in 3/10 cases for obstacles placed in corners assuming that the controller was acting normally. It is to be noted that the focus was lying on the reliable detections on the straights, which we intended to show on the demo day.

## 15.7. Future Avenues of Development



As already described above in the [eval interface section](#), we think that there is still room for improving the interface between our code and the *Anti Instagram* module in terms of making the *continous anti instagram node* as well as the *image transformer node* more computationally efficient. Another interesting thought which might be taken into consideration concerning this interface is the following: As long as the main part of the anti instagram's color correction is linear (which was in most of our cases sufficient), it might be reasonable to just adapt the filter values than to subscribe to a fully transformed image. This effort could save a whole publisher and subscriber and it is obvious that it is by far more efficient to transform a few filter values once than to transform every pixel of every incoming picture. Towards the end of our project we invested some time in trying to get this approach to work but as time was not enough we could not make it. We especially struggled to transform the orange filter values, while it worked for the yellow ones (BRANCH: devel-saviors-ai-tryout2). We think that if in the future one will stick to the current hardware this might be a very interesting approach, also for other software components such as the lane detection or any other picture related algorithms which are based on the concept of filtering colors.

Another idea of our team would be to **exploit the transformation to the bird's view also for other modules**. We think that this approach might be of interest e.g. for extracting the curvature of the road or performing the lane detection from the rather more undistorted top view.

Another area of improvement would be to further develop our provided scripts to being able to **automatically evaluate** the performance of our entire pipeline. As you

can see in our code description in github there is a complete set of scripts available which makes it easily possible to transform a bag of raw camera images to a set of pictures on which we applied our obstacle detector, including the color correction part of Anti Instagram. The only missing step left is an automatic detection whether the drawn box is correct and in fact around an object which is considered to be an obstacle or not.

Furthermore to achieve more general performance probably even adaptions in the hardware might be considered (see [40]) to tune the obstacle detection algorithm and especially its generality. We think that setting up a **neural network** might make it possible to release the restrictions on the color of the obstacles.

In terms of avoidance there would be **possibilities to handle the high inaccuracies of the pose estimation** by relying on the lane controller to not leave the lane and just use a kind of closed loop control to avoid the obstacle (use the new position of the detected obstacle in each frame to securely avoid the duckie). Applying filters to the signals, especially the heading estimation, could further improve the behaviour. This problem was detected late in the development and could not be tested due to time constraints. Going further, having both the line and obstacle detection in the same algorithm would allow the direct information on how far away obstacles are from the track. We expect that this would increase the accuracy compared to computing each individually and bringing it together.

The infrastructure is in place to include new scenarios like obstacles on intersection or multiple detected obstacles inside the bounding box. If multiple obstacles are in proximity, a more sophisticated trajectory generation could be put in place to avoid these obstacles in a safe and optimal way.

Furthermore the **avoidance in corners** could be easily significantly improved if the line detection would estimate the curvature of the current tile which would enable adaptions to the bounding box on corner tiles. If the pose estimation is significantly improved one could also implement an adaptive bounding box which takes exactly the form of the lane in front of the robot (see [Figure 15.22](#))

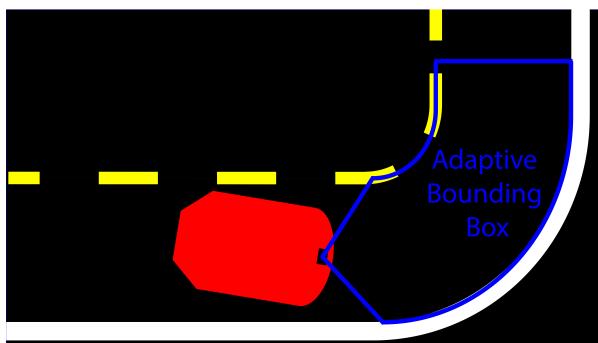


Figure 15.22. Adaptive bounding box

## 15.8. Theory Chapter

## 1) Introduction to Computer Vision

In general a camera is consisting of a converging lens and an image plane (Figure 15.23). In the following theory chapter, we will assume that the picture in the image plane is already undistorted, meaning we preprocessed it and eliminated the lens distortion.

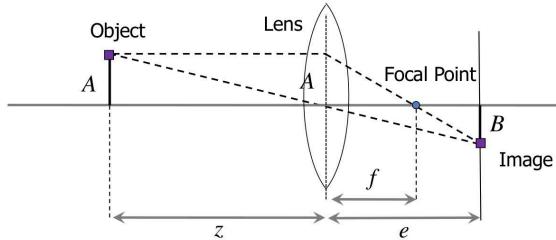


Figure 15.23. Simplified camera model [41]

It is quite easy to infer from Figure 15.23 that for a real world point to be in focus, it has to hold, that both of the “rays” (see Figure 15.23) intersect in one point in the image plane, namely in point B. Mathematically written this means:

$$\text{Equation 1: } \frac{B}{A} = \frac{e}{z} \quad \text{Equation 2: } \frac{B}{A} = \frac{e-f}{f}$$

This last equation  $\text{eqref}\{\text{eq:two}\}$  can be approximated since usually  $z \gg f$  such that we effectively arrive at the pin-hole approximation with:  $e \approx f$  (see Figure 15.24)

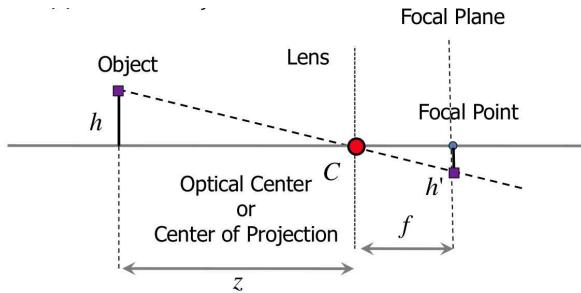


Figure 15.24. Pinhole camera approximation [41]

For the pixel coordinate on the image plane it holds:

$$\frac{h'}{h} = \frac{f}{z} \Rightarrow h' = \frac{f}{z} * h.$$

In a more general case, when you consider a 3 dimensional setup and think of a 2 dimensional image plane you have to add another dimension and it follows that a real world point being at  $\vec{P}_W = \left( \begin{array}{c} X_W \\ Y_W \\ Z_W \end{array} \right)$  will therefore be projected to the pixels in the image plane:

$$x_{\text{pix}} = \alpha * \frac{f}{Z_W} * X_W + x_{\text{offset}} \quad \text{and} \quad y_{\text{pix}} = \beta * \frac{f}{Z_W} * Y_W + y_{\text{offset}}$$

where  $\alpha$  and  $\beta$  are scaling parameters and  $x_{\text{offset}}$  and  $y_{\text{offset}}$  are constants which can be always added. Those equations are usually rewritten in homogeneous coordinates such that we have only linear operations left as:

$$\lambda \left( \begin{array}{c} x_{\text{pix}} \\ y_{\text{pix}} \\ 1 \end{array} \right) = \left( \begin{array}{ccc} \alpha & f & 0 \\ 0 & \beta & f \\ 0 & 0 & 1 \end{array} \right) \left( \begin{array}{c} X_W \\ Y_W \\ Z_W \end{array} \right)$$

$$\begin{aligned} \text{\textbackslash begin\{equation\}} & \text{\textbackslash Leftrightarrow } \lambda * \text{\textbackslash vec\{P\_pix\}} = H * \text{\textbackslash vec\{P\_W\}} \\ & \text{\textbackslash label\{eq:one\}\textbackslash tag\{2\} \textbackslash end\{equation\}} \end{aligned}$$

**Note:** In general this Matrix H is what we get out of the intrinsic calibration procedure and it might happen, that if the World frame and Camera frame are not completely aligned that then the (1,2) and (2,1) entry of the H Matrix are not exactly zero.

This equation [\eqref{eq:one}](#) and especially [Figure 15.24](#) clearly show that since in every situation you only know H as well as  $x_{\text{pix}}$  and  $y_{\text{pix}}$  of the respective objects on the image plane, there is no way to determine the real position of the object, since everything can only be determined up to a scale ( $\lambda$ ). Frankly speaking you only know the direction in which the object has to be but nothing more, which makes it a very difficult task to infer potential obstacles given the picture of a monocular camera only. This scale ambiguity is illustrated in [Figure 15.25](#).

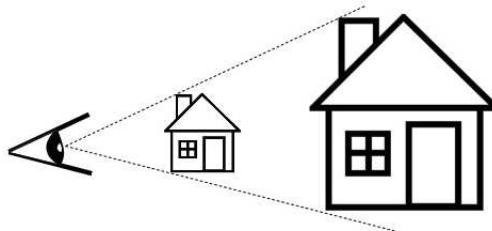


Figure 15.25. Scale ambiguity [\[43\]](#)

To conclude, given a picture from a monocular camera only you have no idea at which position the house really is, so without exploiting any further knowledge it is extremely difficult to reliably detect obstacles which is also the main reason why the old approach did not really work. On top of that come other artifacts such as that the same object will appear larger if it is closer to your camera and vice versa, and lines which are parallel in the real world will in general not be parallel in your camera image.

**Note:** The intuition, why we humans can infer the real scale of objects is that if you add a second camera, know the relative Transformation between the two cameras and see the same object in both images, then you can easily triangulate the full position of the object, since it is at the place where the two “rays” intersect! (see [Figure 15.26](#)).

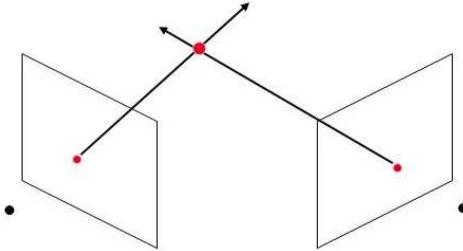


Figure 15.26. Triangulation to obtain absolute scale [44]

## 2) Inverse Perspective Mapping / Bird's View Perspective

The first chapter above introduced the rough theory which is needed for understanding the following parts. The important additional information that we exploited heavily in our approach is that in our special case we know the coordinate  $Z_W$ . The reason therefore lies within the fact that unlike in another more general usecase of a mono camera, we know that our camera will always be at height  $h$  with respect to the street plane and that the angle  $\theta_0$  also always stays constant. ([Figure 15.27](#))

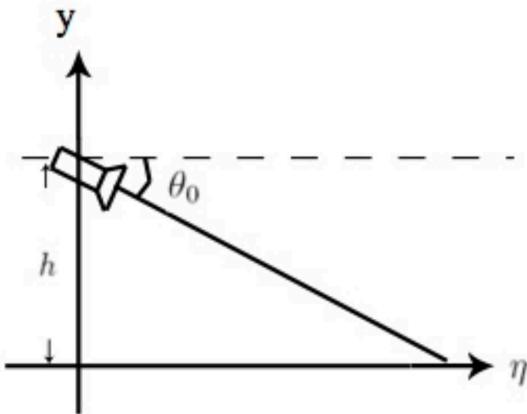


Figure 15.27. Illustration of our fixed camera position [45]

This information is used in the actual extrinsic calibration such that in Duckietown, due to the assumption that everything we see should in general be on the road, we can determine the full real world coordinates of every pixel, since we know the coordinate  $Z_W$  which uniquely defines the absolute scale and can therefore uniquely determine  $\lambda$  and  $H$ ! Intuitively this comes from the fact that we can just intersect the known ray direction (see [Figure 15.24](#)) with the known “ground plane”.

This makes it possible to project every pixel back into the “road plane” by computing for each available pixel:  $\text{vec}\{P_W\} = H^{-1} * \lambda * P_{\text{pix}}$

This “projection back onto the road plane” is called inverse perspective mapping!

If you now visualize this “back” projection, you basically get the bird’s view since you can now map back every pixel in the image plane to a unique place on the road plane.

The only trick of this easy maths is that we exploited the knowledge that everything we see in the image plane is in fact on the road and has one and the same z-coordinate. You can see that the original input image [Figure 15.28](#) is nicely transformed into the view from above where every texture and shape is nicely reconstructed if this assumption is valid [Figure 15.29](#). You can especially see that all the yellow line segments in the middle of the road roughly have the same size in this bird’s view [Figure 15.29](#) which is very different if you compare it to the original image [Figure 15.28](#).



Figure 15.28. Normal incoming image without any obstacle



Figure 15.29. Incoming image without obstacle reconstructed in bird’s view

The crucial part is now what happens in this bird’s view perspective, if the camera sees an object which is not entirely part of the ground plane, but stands out. These are basically obstacles we want to detect. If we still transform the whole image to the bird’s view, these obstacles which stand out of the image plane get heavily disturbed. Lets explain this by having a look at [Figure 15.30](#).

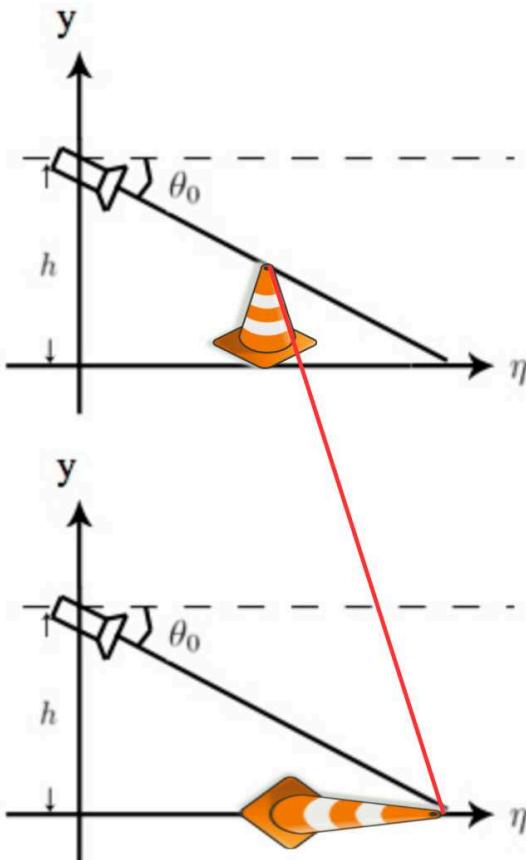


Figure 15.30. Illustration why obstacle standing out of ground plane is heavily disturbed in bird's view, modified: [45]

The upper picture in [Figure 15.30](#) depicts the real world situation, where the cone is standing out of the image plane and therefore the tip is obviously not at the same height as the ground plane. However, as we still have this assumption and as stated above intuitively intersect the ray with the ground plane, the cone gets heavily disturbed and will look like the lower picture in [Figure 15.30](#) after performing the inverse perspective mapping. From this follows that if there are any objects which DO stand out of the image plane then in the inverse perspective you basically see their shape being projected onto the ground plane. This behaviour can be easily exploited since all of these objects are heavily disturbed, drastically increase in size and can therefore be easily separated from the other objects which belong to the ground plane.

Let's have one final look at an example in Duckietown. In [Figure 15.31](#) you see an incoming picture seen from the normal camera perspective, including obstacles. If you now perform the inverse perspective mapping, the picture looks like [Figure 15.32](#) and as you can easily see, all the obstacles, namely the two yellow duckies and the orange cone which stand out of the ground plane are heavily disturbed and therefore it is quite easy to detect them as real obstacles.

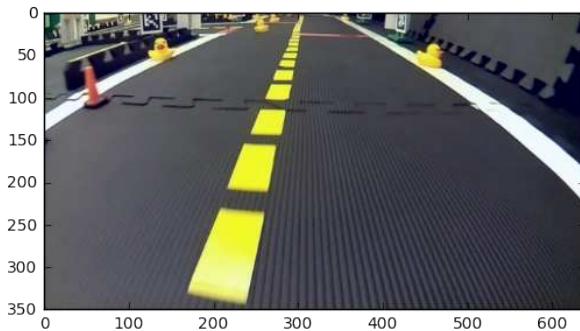


Figure 15.31. Normal situation with obstacles in Duckietown seen from Duckiebot perspective

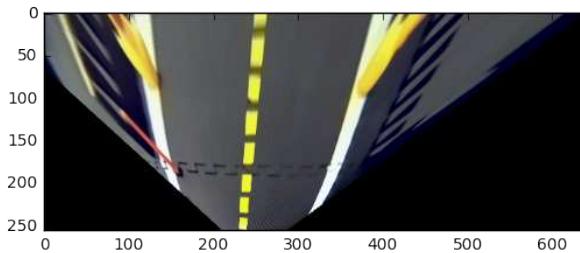


Figure 15.32. Same situation seen from bird's perspective

### 3) HSV Color Space

#### *Introduction and Motivation:*

The “typical” color model is called the RGB color model. It simply uses three numbers for the amount of the colors *red*, *blue* and *green*. It is an additive color system, so we can simply add two colors to produce a third one. Mathematically written it looks as follows and shows the way of how we deal with producing new colors:

$$\$ \$ \left( \begin{array}{c} r_{\text{res}} \\ g_{\text{res}} \\ b_{\text{res}} \end{array} \right) = \left( \begin{array}{c} r_1 \\ g_1 \\ b_1 \end{array} \right) + \left( \begin{array}{c} r_2 \\ g_2 \\ b_2 \end{array} \right) \$ \$$$

If the resulting color is white, the two colors *1* and *2* are called to be complementary (e.g. this is the case for blue and yellow).

This color system is very intuitive and is oriented on how the human vision perceives the different colors.

The *HSV* color space is an alternative representation of the RGB color model. On this occasion *HSV* is an acronym for *Hue*, *Saturation* and *Value*. It is not so easy summable as the RGB model and it is also hardly readable for humans. So the big question is: **Why should we transform our colors to the HSV space? Does it derive a benefit?**

The answer is yes. It is hardly readable for humans but it is way better to filter for specific colors. If we look at the definition openCV gives for the RGB space, the higher complexity for some tasks becomes obvious:

In the RGB color space all “the three channels are effectively correlated by the amount of light hitting the surface”, so the color and light properties are simply not separated. (see: [47])

Expressed in a more simpler way: In the RGB space the colors also influence the brightness and the brightness influences the colors. However, in the HSV space, there is only one channel - the *H* channel - to describe the color. The *S* channel represents the saturation and *V* the intensity. This is the reason why it is super useful for specific color filtering tasks.

The HSV color space is therefore often used by people who try to select specific colors. It corresponds better to how we experience color. As we let the *H* (*Hue*) channel go from 0 to 1, the colors vary from red through yellow, green, cyan, blue, magenta and back to red. So we have red values at 0 as well as at 1. As we vary the *S* (*saturation*) from 0 to 1 the colors simply vary from unsaturated (more grey like) to fully saturated (no white component at all). Increasing the *V* (*value*) the colors just become brighter. This color space is illustrated in [Figure 15.33](#). (see: [48])

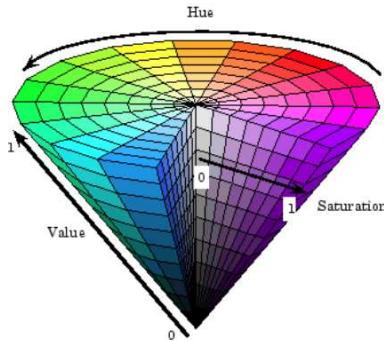


Figure 15.33. Illustration of the HSV Color Space [48]

Most systems use the so called RGB additive primary colors. The resulting mixtures can be very diverse. The variety of colors, called the *gamut*, can therefore be very large. Anyway, the relationship between the constituent amounts of red, green, and blue lights is unintuitive.

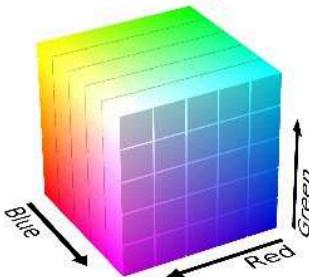
#### Derivation:

The *HSV* model can be derived using geometric strategies. The RGB color space is simply a cube where the addition of the three color components (with a scale from 0 to 1) is displayed. You can see this on the left of [Figure 15.34](#).

## RGB Vs HSV (Color Spaces)

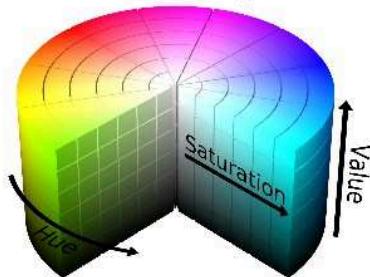
\*Slide produced using PDF-Ray (<http://lab.jorixx.org/>)

### RGB Color Space



- Simplicity in representation and storage
- Some point operations are faster...
- Have lighting and color properties together

### HSV Color Space



- Close to human perception
- Easy interpolation between colors
- Invariant to illumination transforms

Michel Alves -- Five Minute Speech (FMS) - Overview of Activities Developed in Disciplines and Guided Studies - Graduate Program in Systems Engineering and Computing

Figure 15.34. Comparison between the two colors spaces [50]

You can now simply take this cube and tilt it on its corner. We do it this way so that black rests at the origin whereas white is the highest point directly above it along the vertical axis. Afterwards you can just measure the *hue* of the colors by their angle around the vertical axis (red is denoted as  $0^\circ$ ). Going from the middle to the outer parts from 0 (where the grey like parts are) to 1 determines the *saturation*. This is illustrated in [Figure 15.35](#).

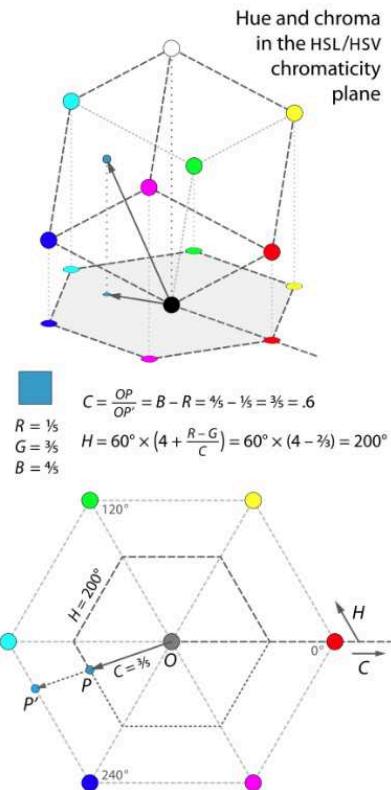


Figure 15.35. 'Cutting the cube' [51]

The definitions of *hue* and *chroma* (proportion of the distance from the origin to the edge of the hexagon) amount to a geometric warping of hexagons into circles (for more informations see: [51]). Each side of the hexagon is mapped linearly onto a  $60^\circ$  arc of the circle. This is visualized in Figure 15.36.

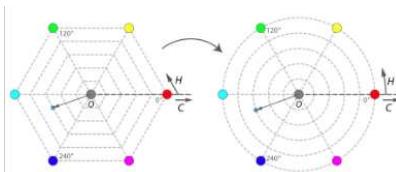


Figure 15.36. Warping hexagons to circles [51]

For the *value* or lightness there are several possibilities to define an appropriate dimension for the color space. The simplest one is just the average of the three components, which is nothing else then the vertical height of a point in our tilted cubic. For this case we have:

$$\$ I = 1/3 * (R + G + B) \$$$

For another definition the *value* is defined as the largest component of a color. This places all three primaries and also all of the “secondary colors” (cyan, magenta, yellow) into a plane with white. This forms a hexagonal pyramid out of the RGB

cube. This is called the HSV “hexcone” model and is the common one. We get:

$$\$ \$ V = \max(R, G, B) \$ \$$$

(see: ([51]))

**In Practice:**

1. Form a hexagon by projecting the RGB unit cube along its principal diagonal onto a plane.

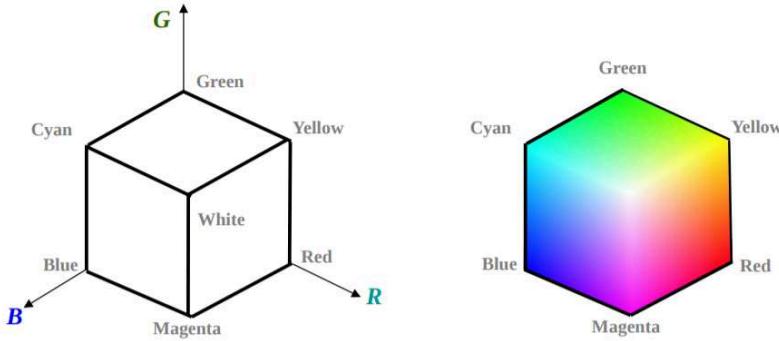


Figure 15.37. First layer of the cube (left) and flat hexagon (right) [55]

2. Repeat projection with smaller RGB cube (subtract 1/255 in length of every cube) to obtain smaller projected hexagon. Like this a *HSV hexcone* is formed by stacking up the 256 hexagons in decreasing order of size.

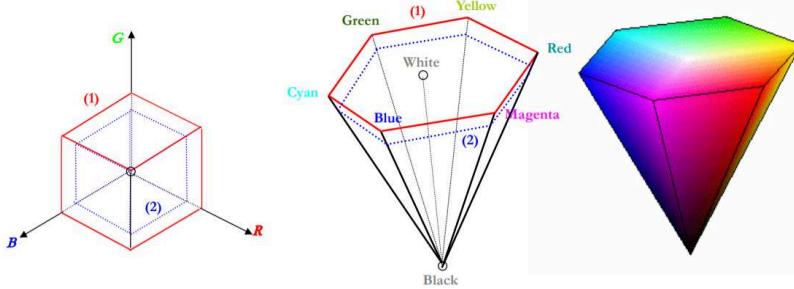


Figure 15.38. Stacking hexagons together [55]

Then the value is again defined as:

$$\$ \$ V = \max(R, G, B) \$ \$$$

3. Smooth edges of hexagon to circles (see previous chapter).

**Application:**

One nice example of the application of the HSV color space can be seen in [Figure 15.39](#).



Figure 15.39. Image on the left is original. Image on the right was simply produced by rotating the H of each color by  $-30^\circ$  while keeping S and V constant [51]

It just shows how simple color manipulation can be performed in a very intuitive way. We can turn many different applications to good account using this approach. As you have seen, color filtering also simply becomes a threshold query.

## UNIT L-16

### Navigators: preliminary report

#### 16.1. Part 1: Mission and Scope

##### 1) Mission Statement

The objective of this project is to implement a method that allows Duckiebots to reliably navigate any kind of intersection they may encounter when driving through Duckietown.

##### 2) Motto

TRANSIBITIS  
(You shall pass)

##### 3) Project Scope

*What is in scope:*

- Navigating three- and four-way intersection of predetermined shape.
- (Absolute or relative) localization within the intersection.
- Computing a path or trajectory that guides the Duckiebots across the intersection to the desired lane.
- Computing control inputs to follow path / track trajectory.
- Limiting travel time across intersection.
- Detecting when the Duckiebot successfully navigated across an intersection and finds itself in a regular lane.

- Proposing hardware modifications to the intersection (e.g. traffic lights, additional markers,...).

*What is out of scope:*

- Understanding that the Duckiebot is at an intersection.
- Deciding where to go at an intersection.
- Coordinating with other Duckiebots at an intersection (e.g. who drives first, ...).
- Object detection and collision avoidance.
- Global localization.

*Stakeholders:*

- Smart Cities
- The Controllers
- Anti-Instagram
- The Identifiers

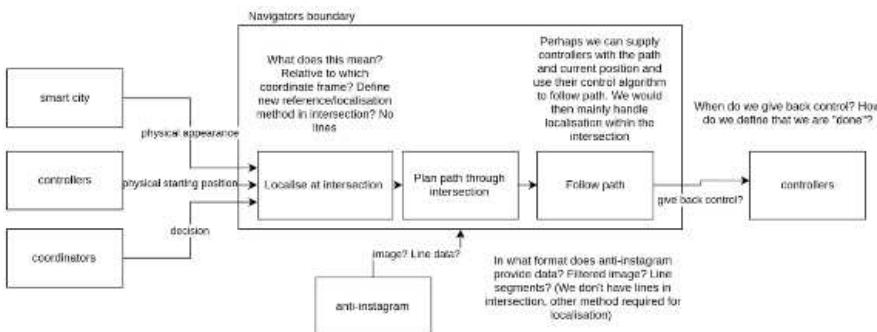


Figure 16.1. Project boundary

## 16.2. Part 2: Definition of the Problem

### 1) Problem Statement

We seek to find a method that allows a Duckiebot to safely navigate an intersection. In particular, we attempt to solve the following problem: Given that the Duckiebot finds itself at rest at an intersection, 1) devise a method such that the Duckiebot can localize itself at the intersection, 2) compute a path or trajectory that guides the Duckiebot to the desired exit while respecting the Duckiebots system dynamics and control input limitations, 3) find control inputs to track the path or trajectory, and lastly 4) detect when the maneuver is finished and the Duckiebot finds itself in a regular lane.

### 2) Assumptions

- Size, shape of intersections is given and fixed.
- Color and size of lane markings are given and fixed.
- The type of intersection (e.g. three-way or four-way intersection) as well as the desired exit (e.g. left turn, right turn or straight) are provided.
- There are fiducial markers (e.g. April tags, stop signs, ...) placed at the intersection at predetermined positions.

- The Duckiebots' are initially at rest and their pose is within a certain range with respect to the intersection (distance to center of road, distance to stop line, orientation within the lane).
- The intersection is free of obstacles.
- Good light conditions (e.g. no illumination problems,...)

### 3) Approach

---

The task of navigating an intersection can be roughly split into two tasks: 1) localization and 2) path/trajectory planning and control. The latter problem is assumed that be relatively easy compared to the first. Paths or trajectories can, for example, be found using simple motion primitives such as splines and it is then straightforward to track these using control techniques such as proportional-derivative-integral controllers. Hence, we only list different approaches to solve the localization problem when navigating intersections in the following.

**Open-Loop Maneuver:** Given that the Duckiebot finds itself at an intersection, it simply executes a predetermined trajectory (from the nominal starting position to the desired end position) to cross the intersection. Once the regular lane detection is able to estimate the Duckiebots pose again, the control is handed back to the lane following controller. *Extension:* Model inaccuracies that are likely to affect the performance of the open-loop maneuvers could be compensated for by using iterative learning based on the state estimate of the lane detection method at the end of the open loop maneuver. If the lane detection algorithm finds that the duckiebot is far off of the nominal path, the nominal path could be adjusted iteratively such that systematic errors are suppressed (Note: This could to some extent interfere with the System Identification Project).

**April-Tags:** At each intersection in Duckietown, April-tags tell the Duckiebots at which intersection they are and what kind of intersection they need to cross (e.g. three-way intersection). The April-tags are placed at predetermined position at the intersection and are standardized. Given the (absolute) size of an april tag and the Duckiebot's camera parameters, one can compute the Duckiebot's position relative to the april tag and use this information to track a trajectory (again, relative to an april tag). *Extension:* The environment in Duckietown is very structured and various different objects, e.g. traffic lights or street signs, are placed at predetermined locations. To increase robustness of this method or to handle the case when the April tags are out of sight of the Duckiebots (likely towards the end of crossing an intersection), these objects could also be used for localization.

**Line Localization:** The disadvantage of using fiducial markers for localization is that it is not applicable to the real world where there are no fiducial markers. However, also there also exist in the real world standardized marks, the line marks on the street. One possible approach could thus be the detection of all the line marks visible at intersection (e.g. stop marks of current lane but also of the different exits) and use this for localization. Since the appearance of intersections are standardized in Duckietown, the homography that best explains the detected line marks could be determined in order to localize the Duckiebot.

**Visual Odometry:** The most generic solution would be the use of visual odometry

for the localization of the Duckiebot at the intersection, i.e. matching of distinct features between subsequent camera frames and using these point matches to compute the relative camera pose between the two frames. Visual odometry in general computationally expensive. However, the Duckiebot's dynamic could be used to reduce the computational burden (e.g. 1-point RANSAC). Due to a lack of sensors on the Duckiebot, visual odometry is unable to determine the scale of the scene. To solve this, the information about e.g. the size of April-Tags, line width, etc. can be used.

The above methods can also be combined, e.g. using visual odometry and if available April-tags.

#### 4) Functionality provided

---

- Localization within an intersection.
- Path/trajectory planning to navigate intersection.
- Possibly control strategy to track above.

#### 5) Resources required / dependencies / costs

---

The proposed method(s) can be evaluated using the following measures (in order of decreasing importance):

- Success rate, i.e. 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.
- Accuracy and precision of final state, i.e. how close is the Duckiebot's state relative to the desired final state (e.g. distance relative to the center of the lane, orientation within lane, velocity) and how repeatable is this.
- Duration, i.e. the average time required for the Duckiebot to cross an intersection and if possible an upper limit (worst-case) on the time required.
- Robustness to changes in the size of the intersection (e.g. lane width, size of tiles, ...), i.e. how do the above performance measures change with respect to changes of the size of intersection.
- Path following or trajectory tracking error, using for example the maximum absolute distance error from the desired path.

#### 6) Performance measurement

---

- The success rate can evaluated by simply performing the intersection navigation tasks N times and counting the number of successful trials.
- The accuracy and precision of final state can be evaluated using the already existing lane detection method.
- The average duration can simply be computed from N experiments. A (probabilistic) upper bound to navigating an intersection can be found likewise.
- Either experiments on the real system with slightly different intersections or a sensitivity analysis (analytically or numerically) with respect to parameters defining the size of the intersection can be carried to evaluate the system's robustness.
- An absolute position system (e.g. an overhead motion capture system) can be used to evaluate the Duckiebot's trajectory tracking errors.

## 16.3. Part 3: Preliminary Design

### 1) Modules

---

- Initial position understanding
- Information processing (in which kind of intersection we are, where we want to go, ..)
- Trajectory generation
- Control loop
- Conclusion

### 2) Interfaces

---

The inputs and outputs are also shown on a very high level in the Stakeholders Graph.

### 3) Software modules

---

- Intersection Localization: ROS node
- Path Planner / Trajectory Generator: Python library
- Path Following / Trajectory Tracking Controller: ROS node.

### 4) Infrastructure modules

---

None (assuming that no hardware changes are required).

## 16.4. Part 4: Project Planning

### 1) Data collection

---

Recordings of the camera feed of Duckiebots navigating intersections.

### 2) Data annotation

---

No

*Relevant Duckietown resources to investigate:*

- Current (open-loop) solution
- April tag detector
- Control strategy
- Lane Detector
- Anti-Instagram
- State Machine (switch between lane-following and navigating intersection)
- Visual Odometry
- Duckiebot system dynamics and control input constraints

*Other relevant resources to investigate:*

- April Tags: <https://april.eecs.umich.edu/software/apriltag.html>
- Aruco Markers: <https://sourceforge.net/projects/aruco/files/>
- Visual Odometry: D. Scaramuzza, F. Fraundorfer, “Visual odometry [tutorial]”, IEEE Robotics & Automation Magazine, 2011.

D. Scaramuzza, F. Fraundorfer, R. Siegwart, “Real-time monocular visual odometry for on-road vehicles with 1-point RANSAC”, IEEE International Conference on Robotics and Automation, 2009.

- Localization using Line Detection: J. Barandiaran, D. Borro, “Edge-Based Markerless 3D Tracking of Rigid Objects”, IEEE Conference on Artificial Reality and Telexistence, 2007.

### 3) Risk Analysis

---

- Not enough distinct features for visual odometry.
- Similarly, not enough lane markings.
- Not sufficient computational power on board the Duckiebot.

*Mitigation strategies:*

Mainly the mentioned risks will be related to a closed-loop implementation, which is a basic goal of the project. However they should not affect the development of an improved open-loop solution, ex. using April Tags detection, so this may be a starting point to improve the current state and to start the development of the closed-loop solution.

For the application of a Visual Odometry algorithm, if features in the Duckietown will be not enough, we may plan to add/use additional hardware, which will be possibly developed in accord with the Smart Cities group.

Moreover, the paper “Real-time monocular visual odometry for on-road vehicles with 1-point RANSAC” may be useful for addressing issues about computational constraints thanks to the implementation of the 1-point RANSAC algorithm.

## UNIT L-17

### Navigators: intermediate report

**TODO:** JT: fix intra-duckiebook links

#### 17.1. Part 1: System interfaces

##### 1) Logical architecture

---

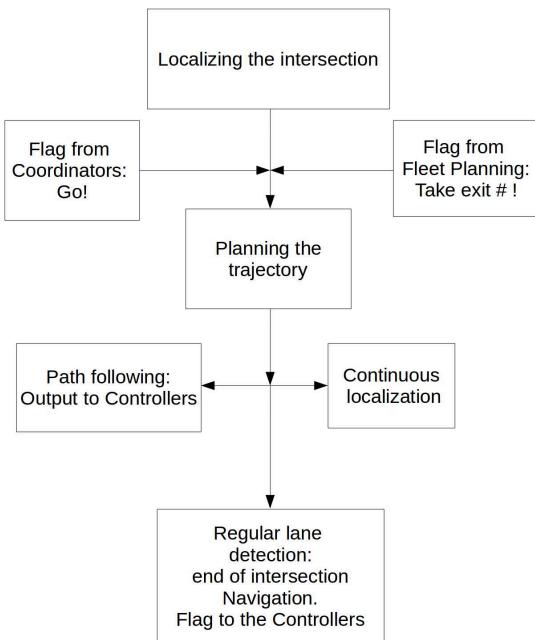


Figure 17.1. Simple step diagram

The intersection navigation is started as soon as the Duckiebot is told that it is in front of an intersection. The following functions are then executed (in chronological order):

- The Duckiebot localizes itself with respect to the intersection.
- The Duckiebot waits until it receives a message on the topic “turn\_type” which exit of the intersection it should take.
- A path is planned that guides the Duckiebot from its current location to the desired intersection exit.
- A path following controller steers the Duckiebot to its final location, while the Duckiebot continuously localizes itself and feeds the estimated pose (i.e. the distance from the desired path and the relative orientation error) to the lane following controller to account for, for example, disturbances or modelling errors.
- The Duckiebot detects when it traversed the intersection, i.e. when it finds itself again in a regular lane, and hands control back to the lane following controller by publishing on the topic “intersection\_done”.

It is assumed that

- the Duckiebot stops between 0.10m and 0.16m in front of the center of the red stop line, i.e.  $d_x \in [0.1m, 0.16m]$ , has an error of no more than 0.03m with respect to the center of its lane, i.e.  $d_y \in [-0.03m, 0.03m]$ , and that the orientation error is smaller than 0.17rad, i.e.  $\theta \in [-0.17rad, 0.17rad]$  (see Fig. 1.2 for details, all values are with respect to the origin of the Duckiebot’s axle-fixed coordinate frame).
- a lane following controller exists that takes as inputs the distance from desired path  $d$  and the orientation error with respect to the path tangent  $\theta$  (see Fig. 1.3 for details).

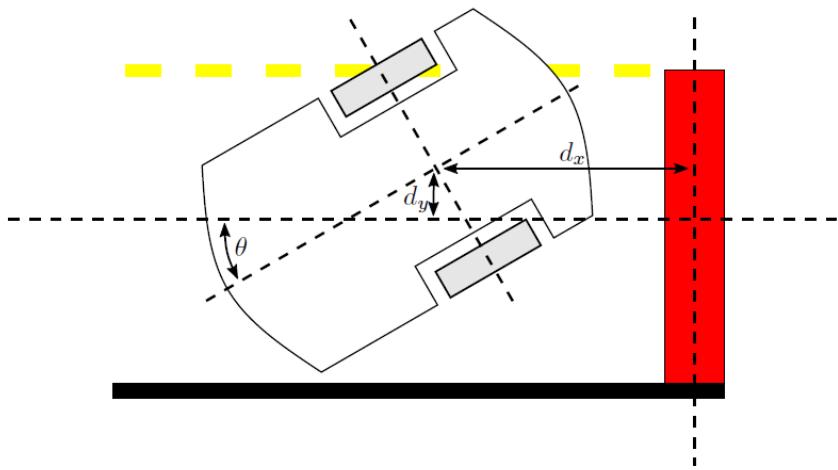


Figure 17.2. Pose of the duckiebot in front of an intersection

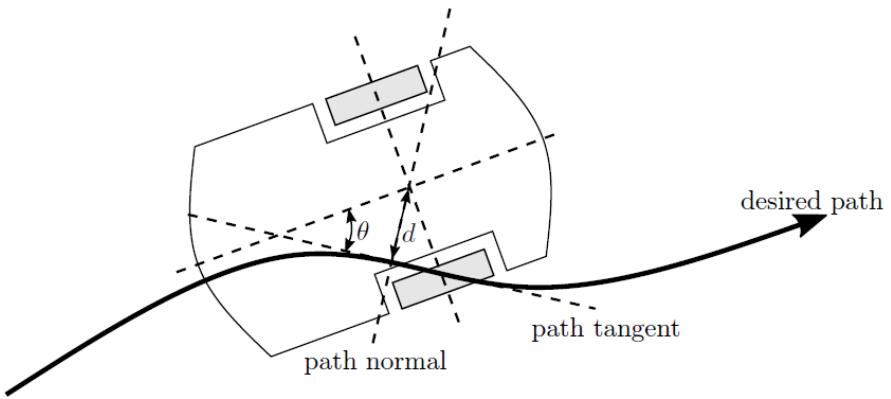


Figure 17.3. Pose of the duckiebot relative to the desired path

## 2) Software architecture

---

We will develop two nodes; “*intersection\_navigation*” and “*intersection\_localization*”. In the following, their functionality and interfaces will be described in more detail.

### “*intersection\_navigation*”-node

The “*intersection\_navigation*”-node is responsible for the high level logic of navigating the Duckiebot across an intersection, planning paths from the Duckiebot’s initial position to the final position, estimating the Duckiebot’s pose and communicating with the lane following controller. It subscribes to the following topics:

- “mode”: Used to detect when Duckiebot is at an intersection or when the intersection control is active, respectively. No assumptions about the latency of this topic will be made. As soon as the mode is switched to “INTERSECTION\_CONTROL”, the “*intersection\_navigation*”-node will take over.
- “turn\_type”: Tells the Duckiebot the type of turn it should take (e.g. left, right, straight, random). No assumption about the latency of this topic will be made. As soon as message is received, the maneuver will be executed.
- “intersection\_pose\_meas\_inertial”: Measured pose of the “*intersection\_localization*”-node with respect to an inertial frame  $\mathcal{I}$  (see Fig. 1.4). This message is used to estimate the pose of the Duckiebot at the intersection, which is then used by the controller to follow the desired pose. It is assumed that this message will have quite some delay (several 10ms), but the delay will be compensated by a state estimator using the timestamp of the message (i.e. camera frame) and using the past commands sent to the vehicle.
- “~image/compressed”: Upon receiving such a message, the Duckiebot’s pose at the time the image was taken will be estimated and sent to the “*intersection\_localization*”-node to initialize the localization problem.
- “~car\_cmd”: The command published by the “*lane\_controller*”-node used to track a desired path. These commands are stored in a queue and will be used to compensate for delays and predict the Duckiebot’s pose. It is assumed this topic has no delay, i.e. that commands are immediately executed.

The “*intersection\_navigation*”-node publishes on the following topics:

- “intersection\_done”: A message on this topic will be broadcasted as soon as the Duckiebot finished traversing the intersection and is used to handback the control.
- “intersection\_pose”: Pose of the Duckiebot with respect to the desired path (see Fig. 3). This topic is basically identical to the “~lane\_pose”-topic from the lane filter and will be used by the “*lane\_controller*”-node in case “mode” is “INTERSECTION\_CONTROL”.

The “*intersection\_navigation*”-node will be estimated to introduce no more than 500ns of delay in regular operation (it only does some logic) and hence an equal delay will be introduced to all the published topics. Initially, after the “*intersection\_localization*”-node is initialized (see below), a path that guides the Duckiebot across the intersection will be planned. This task can be computationally expensive since it needs to be guaranteed that the path is feasible (e.g. not leaving the intersection, curvature constraints, ...) and may take up to 500ms. However, since the Duckiebot is at rest at the intersection, this will not cause any issues.

### **“*intersection\_localization*”-node**

The “*intersection\_localization*”-node is responsible for localizing the Duckiebot at an intersection. For this purpose, it subscribes to the following topics:

- “mode”: This topic is used to detect when the node should start localizing itself at an intersection. No assumption about its latency is made, since it is irrelevant for the node. As soon as the mode is switched to “INTERSECTION\_CONTROL”, the “*intersection\_localization*”-node will start to estimate the Duckiebot’s pose relative to the intersection.
- “~image/compressed”: The compressed camera image is used to localize the Duckiebot within the intersection. No assumption about the latency of this topic is

made. In order to compensate for the expected latency, the timestamp of the camera frame will be also be used to indicate the time for which the pose is estimated.

- “intersection\_pose\_pred\_inertial”: The predicted pose of the Duckiebot at the time when the camera image was taken. This information will be used to initialize the localization problem this node solves. Since the Duckiebot’s pose is predicted for the time the camera image was taken, delays are irrelevant.

The “*intersection\_localization*”-node publishes the following topic:

- “intersection\_pose\_meas\_inertial”: This is the measured pose of the Duckiebot at the intersection with respect to an inertial frame  $\mathcal{I}$  based on the received camera image. The measured pose will be timestamped with the timestamp of the camera image such that the “*intersection\_navigation*”-node can compensate for the latency.

It is estimated that it will take approximately 15ms to estimate the Duckiebot’s pose once the camera image is received, hence about 15ms of delay can be expected on the published topics. However, the delay will be compensated for by the “*intersection\_navigation*”-node.

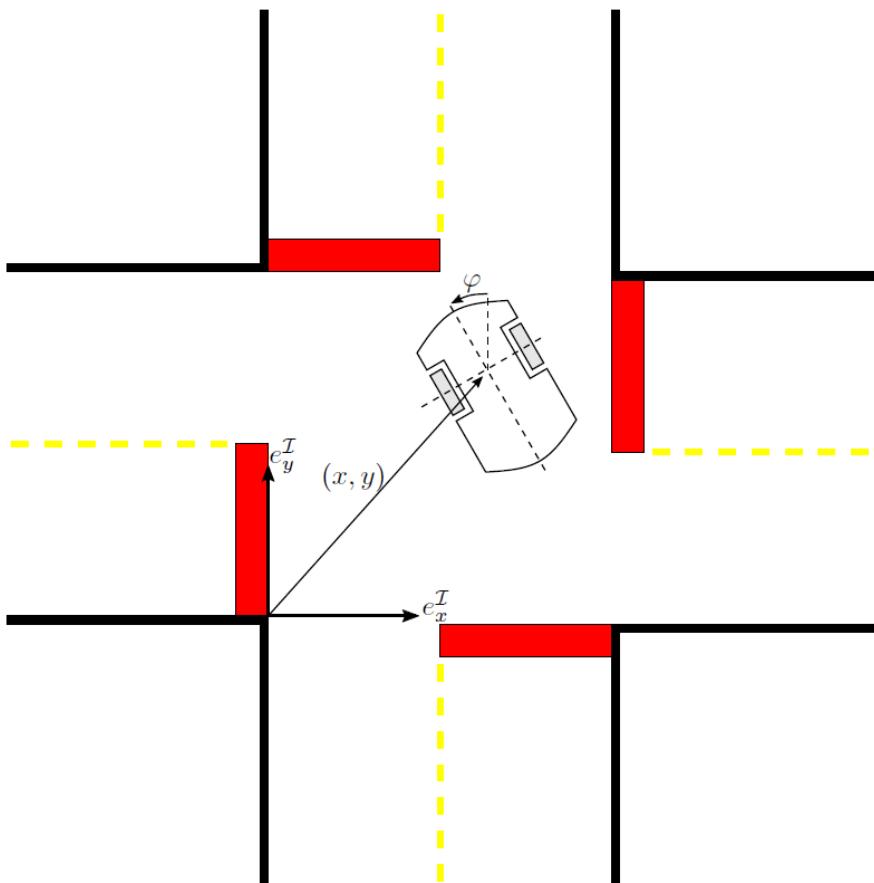


Figure 17.4. Pose of the duckiebot inside a four way intersection

## 17.2. Part 2: Demo and evaluation plan

### 1) Demo plan

---

The proposed closed-loop intersection navigation method will be demonstrated by placing the Duckiebot at an intersection and commanding it via joystick to traverse the intersection to a desired exit (the arrows on left hand side of the joystick will be used to enter the desired exit). The Duckiebot must be placed in a lane in front of the red stop line before executing the demo. The relative position within the lane and its orientation can vary (the initial pose must satisfy the assumption of Part 1). The demo takes as input arguments the desired exit and the navigation method, i.e. the proposed closed-loop navigation or the already existing open-loop navigation. Both arguments are optional.

In addition to the specific intersection navigation demo, the proposed closed-loop intersection navigation will be embedded in the indefinite navigation demo. The indefinite navigation demo can be run with two Duckiebots, one using the proposed closed-loop intersection navigation method and one using the existing open-loop intersection navigation method. In order to be able to distinguish the two Duckiebots, we suggest driving around with a blindfolded duck for the closed-loop method and with a duck that has its eyes wide open for the open-loop method, respectively. However, this demo will require more time to run and differences between the open-loop and closed-loop navigation (e.g. their sensitivity to the initial position at the intersection) will be harder to see.

Both demos can be run in any Duckietown that contains at least one intersection, hence the hardware for a regular Duckietown is required (<http://purl.org/dth/fall2017-map>).

### 2) Plan for formal performance evaluation

---

The performance of the intersection navigation is evaluated experimentally using the following measures (in order of decreasing importance):

- Success rate, i.e. 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 simply performing the intersection navigation task N times and counting the number of successful trials.
- Accuracy and precision of final state, i.e. how close is the Duckiebot's state relative to the desired final state (e.g. distance relative to the center of the lane, orientation within lane, velocity) and how repeatable is this. The accuracy and precision of final state is estimated using the already existing lane detection method, and is measured for different initial conditions.
- Accuracy and precision of estimated pose during traversing the intersection. The estimated pose will be logged and an external motion capture system, which directly outputs the pose of the Duckiebot, will be used as ground-truth.
- Duration, i.e. the average time required for the Duckiebot to cross an intersection and if possible an upper limit (worst-case) on the time required. The average dura-

tion is computed by running a series of N experiments. A (probabilistic) upper bound to navigating an intersection is found likewise.

### 17.3. Part 3: Data collection, annotation, and analysis

#### 1) Collection

---

The use of the provided platform for data collection, annotation and analysis is not needed since we are using logs and recordings of the camera feed of Duckiebots navigating intersections.

#### 2) Annotation

---

No data will be annotated.

#### 3) Analysis

---

No data needs to be analyzed.

## UNIT L-18

### Navigators: final report

TODo: JT: add operation manual, fix bibliographic references, math formatting, various typos

### 18.1. The final result

Video of the final result:



Figure 18.1. The Navigators Demo Video

### 18.2. Mission and Scope

The objective of this project was to implement a method that allows Duckiebots to

reliably navigate any kind of intersection they may encounter when driving through a regular Duckietown.

Motto:

TRANSIBITIS (you shall pass)

What is in scope:

- Navigating three- and four-way intersections of predetermined shape.
- Absolute localization within the intersection.
- Computing a path that guides the Duckiebots across the intersection to the desired exit.
- Tracking the path.
- Detecting when the Duckiebot successfully navigates across the intersection and finds itself in a regular lane.
- Limiting travel time across intersection.

### 1) Motivation

---



We seek to find a method that allows a Duckiebot to safely navigate an intersection. Duckiebots navigate through regular streets in Duckietown using a lane following method. It includes a localization method but also a position control method. Based on this scheme, the need for a controlled crossing of intersection emerged. The main motivation is therefore based on the safety and the reliability of intersection navigation. Ideally, the vehicle should be able to not only go from exit 1 to exit 2, but also recognize where it is relative to those 2 points (localization). Once the Duckiebot receives information about its current position, it will be able to correct any mistake relative to a precalculated path.

### 2) Existing solution

---



The project starts from the current solution, in which the Duckiebot navigates the intersections in open loop. In this solution, after arriving at an intersection, the Duckiebot uses the AprilTags to know the kind of intersection and the feasible exits. Then it randomly chooses one of them and executes standard commands to navigate. This solution did not include localization implying that the current position of the vehicle was not known during the intersection navigation. Since one of the objectives of the project was to use the controller, that the Controllers group designed, the open loop implementation could not be used any further. For practical reasons, we almost started from scratch, and the new solution would have very few common points with the previous implementation.

So our main improvement to the current solution is the use of the camera to introduce vision during navigation. This allow us to introduce feedback into the system with all the benefits that closed loop control has with respect to open loop, and so regulating the control inputs based on the information of the system state.

### 3) Opportunity

---



As mentioned previously, the drawback of the existing solution is the missing infor-

mation about the Duckiebot's position during the navigation. The result is that the system inputs, linear and angular velocities are independent of the system state so that the navigation is not robust and the percentage of failures is high (around 50% for some group members' Duckiebots for short right turns).

Our main contribution is to use the camera as well as a state estimator to estimate the absolute position of the robot with respect to the intersection. Moreover, we compute a path to the desired exit. The planned path is a cubic spline computed with two control points and directions, placed in the initial and final desired intersection positions.

The localization is done by comparing the images from the camera with a template of the intersection. Namely, edges are detected in the current image and some control points are defined so to minimize the distance between their 2D projections in the image frame and the detected edges. In this way, we can estimate the initial position and localize our robot during the navigation.

Reference paper, from which we took inspiration is J. Barandiaran, D.Borro, "Edge-Based Markerless 3D Tracking of Rigid Objects", IEEE Conference on Artificial Reality and Telexistence, 2007.

### 18.3. Definition of the problem

#### 1) Problem Statement

---

We seek to find a method that allows a Duckiebot to safely navigate an intersection. In particular, we attempt to solve the following problem: Given that the Duckiebot finds itself at rest at an intersection, 1) Initial localization of the rest position, 2) compute a path that guides the Duckiebot to the desired exit while respecting the Duckiebots' system dynamics and control input limitations, 3) Continuous localization thanks to a state estimator which integrates wheel commands and pose updates from images, the same method of the initial localization is used frame by frame, 4) track the path providing localization feedback to the lane controller, and lastly 5) detect when the maneuver is finished and the Duckiebot finds itself in a regular lane.

#### 2) Assumptions

---

- Size, shape of intersections are given and fixed.
- Color and size of lane markings are given and fixed.
- The type of intersection (e.g. three-way or four-way intersection) as well as the desired exit (e.g. left turn, right turn or straight) are provided.
- There are fiducial markers, mainly AprilTags, placed at the intersection at predetermined positions.
- The Duckiebots are initially at rest and their pose is within a certain range with respect to the intersection (distance to center of road, distance to stop line, orientation within the lane).
- The intersection is free of obstacles.
- Good light conditions (e.g. no illumination problems, ...)

### 3) Stakeholders

- Smart Cities
- The Controllers
- Fleet planning
- Coordinators

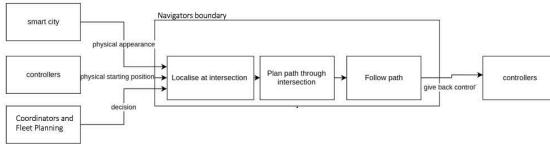


Figure 18.2. Stakeholders Diagramm

### 4) Performance measurement

Success rate, i.e. 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 simply performing the intersection navigation task N times and counting the number of successful trials. Accuracy and precision of final state, i.e. how close is the Duckiebot's state relative to the desired final state and how repeatable is this. The accuracy and precision of the final state is estimated using the existing lane detection method, and is measured for different initial conditions. Duration, i.e. the average time required for the Duckiebot to cross an intersection and an upper limit (worst-case) on the time required. The average duration is computed by running a series of N experiments.

#### 18.4. 4 Contribution / Added functionality

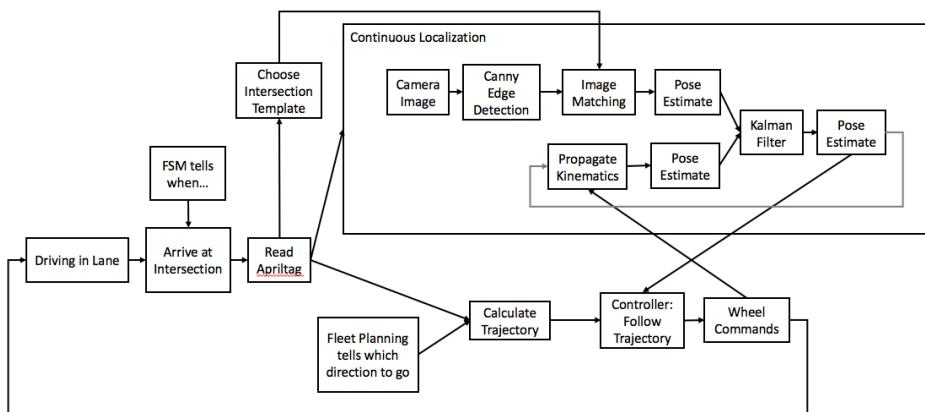


Figure 18.3. Logical architecture diagramm

The intersection navigation is started as soon as the Duckiebot is told that it is in front of an intersection. The following functions are then executed (in chronological order): *The Duckiebot localizes itself with respect to the intersection, given the intersection type*. The Duckiebot waits until it receives a message “turn\_type” indi-

cating which exit of the intersection it should take, and a message “go” indicating that the navigation can start. *A path is planned that guides the Duckiebot from its current location to the desired intersection exit.* The lane following controller, adapted for path tracking, steers the Duckiebot to its final location. During the navigation, the Duckiebot continuously localizes itself and feeds the estimated pose (i.e. the distance from the desired path and the relative orientation error) to the lane following controller to account for disturbances or modelling errors. \* The Duckiebot detects when it traversed the intersection, i.e. when it finds itself again in a regular lane, and hands control back to the lane following controller by publishing on the topic “intersection\_done”.

It is assumed that: *the Duckiebot stops between 0.10m and 0.16m in front of the center of the red stop line, i.e.  $d_x \in [0.1m, 0.16m]$ , has an error of no more than 0.03m with respect to the center of its lane, i.e.  $d_y \in [-0.03m, 0.03m]$ , and that the orientation error is smaller than 0.17rad, i.e.  $|\theta| \in [-0.17rad, 0.17rad]$*  (see Fig. 4 for details, all values are with respect to the origin of the Duckiebot’s axle-fixed coordinate frame). a lane following controller exists that takes as inputs the distance from desired path  $d$  and the orientation error with respect to the path tangent  $\theta$  (see Fig. 5 for details). This is done by the new lane following controller. However, we needed to slightly modify the controller to account for thresholds wheels’ speed.

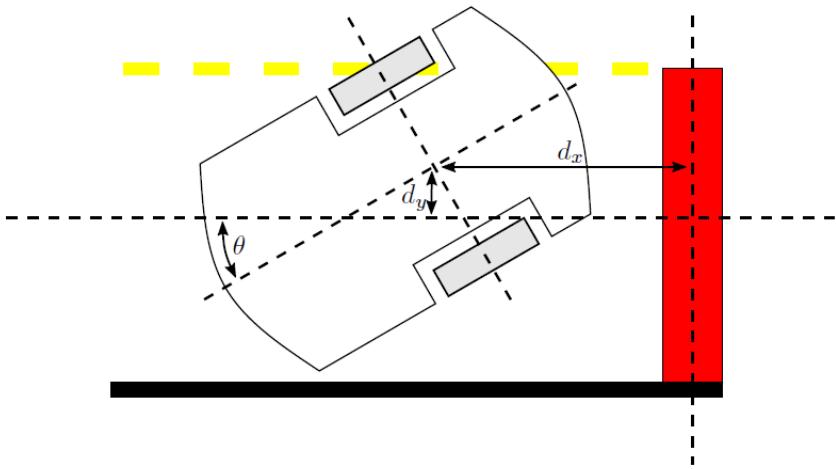


Figure 18.4. Duckiebot's position relative to the red line.

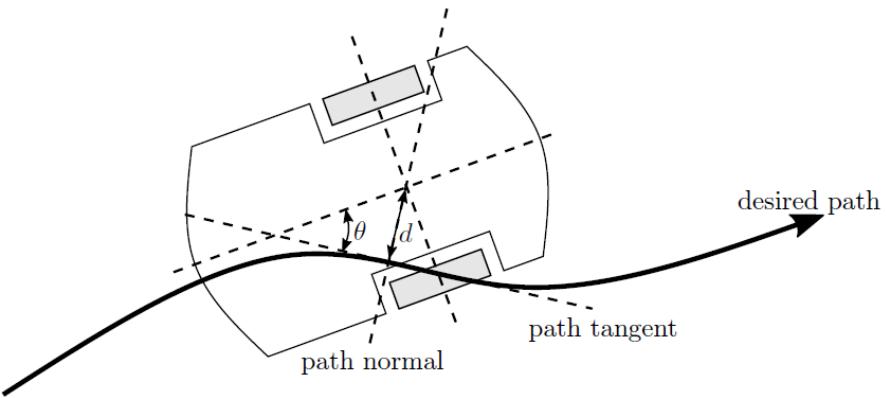


Figure 18.5. Duckiebot's pose relative to the desired path.

## 1) Software architecture

---

Two nodes were developed: “*intersection\_navigation*” and “*intersection\_localization*”. In the following, their functionality and interfaces will be described in detail.

### “*intersection\_navigation*”-node

The “*intersection\_navigation*”-node is responsible for the high level logic of navigating the Duckiebot across an intersection, planning paths from the Duckiebot’s initial position to the final position, estimating the Duckiebot’s pose and communicating with the lane following controller. It subscribes to the following topics:

- “~fsm”: Used to detect when Duckiebot is at an intersection or when the intersection control is active, respectively. As soon as the mode is switched to “INTERSECTION\_COORDINATION”, the “*intersection\_navigation*”-node will take over.
- “~turn\_type”: Tells the Duckiebot the type of turn it should take (e.g. left, right, straight, random).
- “~pose\_in”: Measured pose of the “*intersection\_localization*”-node with respect to an inertial frame  $\mathcal{I}$  (see Fig. 5). This message is used to estimate the pose of the Duckiebot at the intersection, which is then used by the controller to follow the desired pose. This message will have quite some delay (several 10ms), but the delay will be compensated by a state estimator using the timestamp of the message (i.e. camera frame) and using the past commands sent to the vehicle.
- “~image/compressed”: Upon receiving such a message, the Duckiebot’s pose at the time the image was taken will be estimated and sent to the “*intersection\_localization*”-node to initialize the localization problem.
- “~cmds”: The command published by the “*forward\_kinematics\_node*”, linear and angular velocities. These commands are stored in a queue and will be used to compensate for delays and to predict the Duckiebot’s pose.
- “~in\_lane”: The command published by the lane filter. It is true when the robot finds itself in lane.

The “*intersection\_navigation*”-node publishes on the following topics:

- “`~intersection_done`”: A message on this topic will be broadcasted as soon as the Duckiebot finished traversing the intersection and is used to handback the control.
- “`~pose_img_out`”: Estimated pose of the Duckiebot with respect to an inertial frame  $\mathcal{I}$  at the time when the camera image is taken. This topic is subscribed by the “`intersection_localization`”-node in order to initialize the localization problem.
- “`~intersection_navigation_pose`”: Pose of the Duckiebot with respect to the desired path (see Fig. 5). This topic is basically identical to the “`~lane_pose`”-topic from the lane filter and will be used by the “`lane_controller`”-node in case “`fsm`” is “`INTERSECTION_CONTROL`”.

### `“intersection_localizer”-node`

The “`intersection_localization`”-node is responsible for localizing the Duckiebot at an intersection. For this purpose, it subscribes to the following topics:

- “`~pos_img_in`”: The predicted pose of the Duckiebot with respect to an inertial frame at the time when the camera image was taken as well as the raw image from the camera. This information will be used to initialize the localization problem this node solves. Since the Duckiebot’s pose is predicted for the time the camera image was taken, delays are irrelevant.

The “`intersection_localizer`”-node publishes the following topic:

- “`~pose_out`”: This is the measured pose of the Duckiebot at the intersection with respect to an inertial frame  $\mathcal{I}$  based on the received camera image. The measured pose will be timestamped with the timestamp of the camera image such that the “`intersection_navigation`”-node can compensate for the latency.
- “`~localizer_debug_out`”: This topic is used in the visualizer node. The visualizer node can be launched on the laptop, it allows to visualize the current frames and the estimated position.

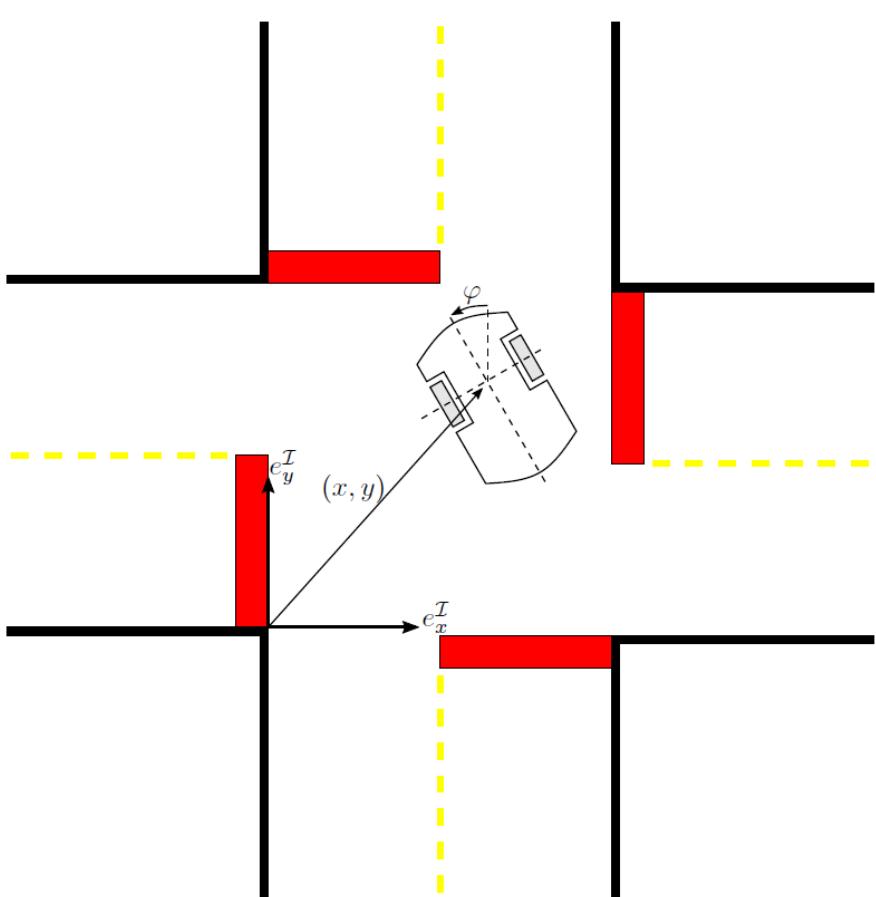


Figure 18.6. Pose of the duckiebot with respect to the Inertial Frame.

## 2) Algorithms

There are two main algorithms in our implementation about localization and path planning respectively

### *Localization algorithm::*

The algorithm is composed by the following steps:

- Process raw image: The image from the camera is processed. The processing is composed by the rectification, conversion to gray scale and edges detector by the Canny edges algorithm.
  - Compute pose: The current pose is estimated. The algorithm starts from a range of poses centered in the previous pose, for the initial localization we use information from the nearest April tag detected at intersection. We use a range of +- 2.5 cm and +- 5 deg around the pose to make the following optimization more robust with respect to not accurate enough camera calibration. Next step is to compute control points from the template model along with their 2D projections onto the image plane. In order to do it, we defined different templates which contain edges of the intersections. The next step is a least squares optimization, defined as (Formula from

the paper cited in [Subsection 18.2.3 - Opportunity](#)):

$$\begin{aligned} \text{\textbackslash begin\{equation\}} & W = \min \sum_i (A_{iW} l_i)^2 \text{\textbackslash end\{equation\}} \end{aligned}$$

Where A is the nx3 matrix which contains the n control points, W is the motion vector defined as  $W = [w_z \ t_x \ t_y]^T$  where we can obtain the Rotation matrix from the vector w applying the Rodriguez's formula. And  $l_i$  are the displacements between the projections of the control points and the edges detected in the image.

Then we obtain the new pose from W, which tells us the relative position and orientation between two consecutive poses, and the previous pose.

*Path planning algorithm::*

The path planned to traverse an intersection is a polynomial of order three. The polynomial coefficients are chosen such that the path starts at the Duckiebot's current pose (i.e. position and orientation) and ends at a desired pose. However, this only defines the coefficients partially. In particular, the orientation of the Duckiebot only determines the direction of the velocity at the initial and final position, but not its magnitude. The magnitude of the initial and final velocity are thus optimized to minimize the curvature of the path. During the optimization, it is also verified that path does not contain any loops and that it satisfies the Duckiebot's maximum curvature, i.e. only feasible paths are planned.

## 18.5. Formal performance evaluation / Results

### 1) Performance evaluation

All the experiments are taken in a duckietown with appearance in accord to [appearance specifications \(master\)](#). It is a Duckietown with 3 and 4-ways intersections and intersection April Tags well visible. We consider an experiment is valid, when the Duckiebot correctly stops at the red line. The term correctly refers to the thresholds defined in section 8 Logical Architecture. We let the Duckiebot navigate Duckietown for 2 runs of 30 minutes randomly choosing the exit to take.

- Success rate: Our implementation has success rate of 80% for the upper left turns and for the straight exit, whereas a lower rate of 65 % for the short right turn. The main failures are: right wheel touches the track boundary white line for the upper left turn, left wheel touches the middle dashed line for the straight exit and the right turn. Mainly the lower success rate of the right turn is due to the fact that in such short maneuver the feedback controller cannot compensate, factors as wheels slippage and inaccurate kinematic calibration. However, our implementation improves the current solution, in which the Duckiebots hits a lane marking 50% of the times and often fails in navigating the short right turn.
- Accuracy and precision: We define a final state as accurate when the Duckiebot finds itself in lane after the intersection navigation is done. Our results show that 95% of the times that the navigation is concluded the robot detects itself in lane and successfully switches to the lane following control. To note that, if the path is concluded but the robot does not find itself in lane, it slows down and goes straight for 2 seconds. If it finds itself in lane during this time, we hand back the control over to

the lane following controller, otherwise the robot stops.

- Duration of the intersection: The time is computed from when the Duckiebot arrives at the red line, the fsm mode is at “intersection\_coordination”, and the intersection navigation is done, publishing of the topic “intersection\_done”. The average time is 17 seconds and the upper limit (worst-case) is 21 seconds.

Moreover, we estimate the accuracy and precision of the estimated pose during traversing the intersection with a visualizer node, which can be run on the laptop. The visualizer node outputs the images from the camera and the edges projections from the estimated current pose.

The time between when the Duckiebot stops at the red line and when it is ready to start the navigation as well as the pose estimation accuracy are the biggest challenges, where mainly our solution may be improved.

In the next section, we give some insights about a possible ways of improvement.

## 18.6. Future avenues of development

The main improvements can be done about the accuracy of the localization, which will also have a positive impact on the computation time.

Specifically, our localization algorithm is very sensible to the camera calibration. Since the calibration matrices are used in the 2D projection of the control points in the image frame, with not adequately good calibration, the optimization problem will minimize quantities that are affected by offsets. In order to compensate for it we optimize, as described in section 8 Algorithms, over a range of initial conditions, but this increases the computation time.

A solution could be to improve the camera calibration procedure and introducing metrics to evaluate its performance. It would allow to decrease the range of initial positions used in the least squares optimization and so to have benefits on both localization accuracy and computation time.

## UNIT L-19

### Parking: preliminary report

#### 19.1. Part 1: Mission and scope

##### 1) Mission statement

Implement parking feature and design specifications (feature and physical)

##### 2) Project scope

Implement parking feature and design specifications (feature and physical)

*What is in scope:*

- forward parking
- bot localization with april tags
- open spot localization
- path generation (coming in and out of parking space)
- how to drive backwards
- parking lot full signal
- physical parking lot design specification

*What is out of scope:*

- Develop new algorithms to filter new lane line colors
- Fleet level coordination
- Handle multiple parking events at once

*Stakeholders:*

- Single SLAM or distributed-est
- Controls
- Smart City
- Anti Instagram

## 19.2. Part 2: Definition of the problem

### 1) Problem statement

---

We need to park N Duckiebots in a designated area in which they are able enter and exit in an efficient manner.

### 2) Assumptions

---

- Four tile structure with defined inlet, outlet and color scheme
- Known design specification of parking lot
- Assume when leaving parking space, path is free of other Duckiebots
- When entering lot and searching for parking space, parking lot is in static state

### 3) Approach

---

- Enter parking lot in designated inlet lane
- Localize based on april tags within field of view with known locations
- Control with feedback along predetermined path
- Detect parking space status (full/free) of each parking space in sequential manner
- Locate a free parking space
- Paths generated for maneuvering into parking space
- High fidelity control into parking space
- Signal generated to signify parking space is full
- When we want to leave a space, generate path out of parking space
- High fidelity control out of parking space (with caster wheel dynamics taken into account for feedback control)
- Control with feedback along predetermined path
- Exit parking lot in designated outlet lane

### 4) Functionality provided

---

aims to address: “how is the functionality of this feature measured”

- Probability of a successful parking maneuver per parking maneuver attempt
- Number of Duckiebots within the parking lot boundary per hour

#### 5) Resources required / dependencies / costs

---

- Size of parking space
- Resources required to develop Duckiebot trajectory
- Number of april tags and infrastructure to support april tags

#### 6) Performance measurement

---

- Starting at parking lot entrance, measure the number of parking maneuvers completed within boundaries of designated parking spot (over N attempts)
- Starting in designated parking space, measure the number of Duckiebots able to arrive at the exit of the parking lot (over N attempts)
- Average time (for N vehicles) to enter and exit parking lot

### 19.3. Part 3: Preliminary design

#### 1) Modules

---

##### *Perception:*

- Lane filtering
- April tag detection
- “Fleet communication”: detecting

##### *Localization and parking map generation:*

- Ego localization
- Localization other Duckiebots
- Parking map design

##### *Planning:*

- Parking space allocation
- Path planning

##### *Control:*

- “Fleet communication”: publishing

#### 2) Interfaces

---

##### *Perception:*

##### **Lane filtering**

- Used for pose estimation at entrance and exit of parking lot and maybe at parking space
- Input: camera image
- Output: location lanes

##### **April tags detection and triangulation**

- Use for pose estimation while driving on the parking lot when no lanes can be identified, every parking space has its own april tag, relative Duckiebot-tag pose is extracted using computer vision
- Input: camera image
- Output: location of april tag, relative position Duckiebot-tag

### **“Fleet communication”: detecting**

- Blinking LEDs are used for communication: while parking signal who is driving (one at the time), while parked signal which parking lot is taken (Duckiebot on parking space 2, blink led in parking space 2 specific frequency)
- Input: camera image
- Output: other occupied signals (other means blinking signals is not from own Duckiebot)

### *Localization and parking map generation:*

#### **Localization other Duckiebots**

- Determines the pose of other Duckiebots using specific blinking LEDs
- Input: other occupied signal(s)
- Output: pose other Duckiebot(s)

#### **Parking map design**

- Static map (physical known map with defined parking spaces and areas to move to them, without Duckiebots) is merged with pose of other Duckiebots to generate a {occupied, free} map of the parking lot

- Input: static map (has to be defined offline), pose other Duckiebots
- Output: parking map

#### **Ego localization**

- State estimation of position (x, y) and heading (theta) of own Duckiebot using lanes (at entrance/exit of parking lot) and april tags
- Input: parking map, relative position Duckiebot-tag(s), localization april tag(s), location lane(s)
- Output: pose Duckiebot

### *Path planning:*

#### **Parking space allocation**

- Allocates a parking space to the Duckiebot given the parking map and the authority to move, executed once per Duckiebot
- Input: parking map, pose Duckiebot
- Output: pose parking space (x, y, theta)

#### **Path planning**

- The actual path planning module
- Input: pose parking space, pose Duckiebot, parking map
- Output: reference path or reference trajectory

### *Control:*

- High fidelity control algorithm to drive Duckiebot on reference trajectory/path to allocated parking space, flag when parked
- Input: reference path, pose Duckiebot, (maybe parking map → constrained control)
- Output: motor voltage, parking status = {going to parking space, parked, want to leave, exiting parking space}

### *“Fleet communication” publishing:*

- Flag status using specific frequency on LEDs (or color for human eye)
- Input: parking status
- Output: own\_occupied\_signal

## **3) Preliminary plan of deliverables**

- Need: infrastructure, localization algorithm using april tags (maybe fusion with lane detection), high fidelity control algorithm, map generation algorithm, localiza-

tion (ego and other Duckiebots), parking space allocation, path planning algorithm

- Exists: Lane detection, color filters, lane control, LED communication, april tag detection, control algorithm (maybe has to be improved),

#### 4) Specifications

---

Yes, we need to add parking lot specifications.

#### 5) Software modules

---

A collection of ROS nodes.

#### 6) Infrastructure modules

---

Yes, we will include infrastructure modules to specify parking lot specifications.

## 19.4. Part 4: Project planning

#### 1) Data collection

---

- April tag localization data
- April tag distance data (detection in a range of ~10 cm until ~1 m away from sign)
- Duckiebot to Duckiebot communication using flashing LEDs

#### 2) Data annotation

---

No

*Relevant Duckietown resources to investigate:*

- April tag detection and localization (what is done already?)
- Control algorithm with good enough precision
- Transforming pose to configuration space
- Path planning algorithm (RRT\*)
- Driving backwards (together with the control guys) while updating the pose of the Duckiebot

*Other relevant resources to investigate:*

- Transforming pose to configuration space
- Path planning algorithm (RRT\*)

#### 3) Risk analysis

---

- Localization fails while driving backwards
- Traffic jam at entrance of parking lot
- Fleet communication fails: incoming Duckiebot does not see currently parking Duckiebot, two Duckiebots leave at the same time
- Detection of april tags and extracting pose of the robot
- Map generation is wrong if Duckiebot is not parked to specification
- Control: level of precision adequate for parking
- Exit parking maneuver conflicts: who can drive first (Duckiebot which is exiting does probably not see anything)

# UNIT L-20

## Parking: intermediate report

### 20.1. Part 1: System interfaces

#### 1) Logical architecture

---

Description of the functionality. What happens when we click start?

- As soon as you arrive to the parking lot and see the corresponding entrance april tag, the Duckiebot switches from normal driving mode to parking mode. Parking mode is only allowed in the parking lot. If the bot exits the parking lot, it sees another april tag and it switches from parking mode back to normal driving mode (starting at a four way intersection).
- Inside the parking lot the robot estimates his pose ( $x$ ,  $y$  and theta) using a bunch of april tags. There is one (maybe also two) april tag per parking space and some additional (entrance, exit, etc.) To do so, a new state estimation algorithm has to be implemented using the library ‘AprilTags C++’. It estimates the relative position of the robot with respect to the april tag. The location of the april tag is encoded in the QR code. As soon as you see one (better two) tags, the pose can be calculated. We assume that we always see at least one tag.
- Given a prior information about the parking lot (where are the parking spaces, where can the robot drive etc) and real time vision information the robot chooses a parking space. At first we assume that the parking lot is empty or that other Duckiebots are static (do not move) and this is encoded in the parking map (places where the robot is not allowed to drive).
- We use RRT\* to generate a path given the pose of the robot, the pose of the assigned parking space and the parking map. To do so we use the ‘open motion planing library (OMPL)’.
- We control the robot to the optimal path using a sufficient controller using visual feedback.
- For driving to the exit, we generate a path and control our robot to this path which includes driving backwards to leave the parking space and turn to get to the exit in a forward motion.

Target values:

- accuracy: the error is a combination of localization accuracy and the offset due to the maximum allowable controller error. To park two Duckiebots next to each other within the space boundaries, the path planning accuracy has to be less (or equal) than 5 cm (which is the distance from the robot edge to the parking lane)
- the point of the robot which is the furthest away from the parking mid line should be less than half of the parking space width while the heading of the robot must be less than a constant (20 degrees) relative to the parking space boundary lines.

Assumptions about other modules: - we assume that the robot finds itself at the entrance of the parking lot whenever it wants to get a parking space

- once in the parking lot: parking is decoupled from everything else

## 2) Software architecture

---

`rosnode list:`

- someone
  - publishes: driving\_mode
- /vehicle/parking\_perception\_localization
  - subscribes: driving\_mode, camera\_image,
  - publishes: parking\_mode, space\_status, pose\_duckiebot, ,
- /vehicle/parking\_path\_planning
  - subscribes: parking\_mode, pose\_duckiebot, space\_status
  - publishes: reference\_for\_control, (path)
- /vehicle/parking\_control
- we copy this node from ‘the controllers’
- subscribes: reference\_for\_control
- publishes: motor\_voltage
- /vehicle/parking\_LED
- subscribes: parking\_mode, space\_status
- publishes: -

`rostopic list: - /vehicle/driving_mode - values = {driving, parking} - frequency: ~ 1 Hz`

- /vehicle/parking\_mode
  - values = {parking, staying, leaving, observing}
  - frequency: ~ 1 Hz
- /vehicle/space\_status
  - 1xN array, N = number of parking space
  - values = {taken, free, not\_detectable, my\_parking\_space}
  - frequency: ~ 1 Hz
- /vehicle/pose\_duckiebot
  - x,y,theta
  - frequency: inherit from camera\_image (~30 Hz)
- /vehicle/path
  - x,y,theta array
  - frequency: very low - only updated once (if my\_parking\_space = {1:3}) or twice (for my\_parking\_space = {4:6}, first path is to go to the middle and observe which parking spaces are free, second path is to go to the associated parking space)
  - computation time ~ 10 s
- /vehicle/reference\_for\_control
  - d (orthogonal distance to path), c (curvature), phi (differential heading path and Duckiebot)
  - frequency: first step (path generation) uses a lot of time ~ 10 s, afterwards fast (~ 30 Hz)
- /vehicle/motor\_voltage
  - two values for the two motors
  - frequency: fast (~ 30 Hz)

Introduced latency to other modules:

- we need some additional lines for the april tag detection in order to switch the driving\_mode to parking (this needs a new publisher in this node) -> latency should be negligible
- otherwise we do not introduce delay to other modules since parking is decoupled

## 20.2. Part 2: Demo and evaluation plan

### 1) Demo plan

---

The parking feature including the design specification is new and will be implemented from scratch. Therefore, the desired functionality cannot be compared again a past version. The demo will be split in multiple parts.

In a first demo just a single Duckiebot will enter the parking lot and take the first parking space, knowing beforehand that this space will be free. The Duckiebot is supposed to park within the marked parking space and be able to leave the parking space after a given terminal command.

In a second demo, the Duckiebot will park in space 5 or 6. These are the spaces which are not visible from the beginning. Its need a two stage path panning. 1) Driving from the entrance to the middle of the parking lot and observe if parking spaces 4 to 6 are free. 2) Driving from the middle of the parking lot to space 5 or 6. The demo is completed when the duckie successfully drives to the exit after a given terminal command.

In a later step multiple Duckiebots (one by one) will enter the parking lot. The Duckiebot will first search for an available parking space (out of max. six), undertake a parking maneuver like in the first demo. The Duckiebot will signal the other Duckiebots that the parking space is taken by outputting a sequence with the LEDs at the back. After a given time each robot (one by one) will leave the parking lot again.

Needed hardware components:

- parking lot setup including April tags
- lanes marking the parking spaces
- Duckiebots (obstacles at parking spaces)

### 2) Plan for formal performance evaluation

---

The performance evaluation will be experimental. We will repeat the demo multiple times and measure the probability of a successful parking maneuver.

## 20.3. Part 3: Data collection, annotation, and analysis

### 1) Collection

---

We will need april tag data to understand at what distances and angles we can localize the Duckiebot from.

We will collect data at a distance up to 1.2m at increments of 10cm. We will collect data at an angle up to 40 degrees at increments of 10 degrees. We will do an “angle sweep” at each distance interval

Logs are taken manually

## UNIT L-21

### Parking: final report

**TODO:** JT: fix math and put it in latex environment, link video properly, link operation manual properly, format everything

#### 21.1. Part 1: The final result

Please see a video of the results [here](#)

Note that the video only includes the simulation results and not the Duckiebot parking autonomously as the control feature of our parking pipeline requires further development. Our parking pipeline works well up until feedback control is required. Please see the demo operation manual for further details:

DUCKUMENTS\_ROOT/docs/atoms\_20\_setup\_and\_demo/30\_demos/17\_parking.md

#### 21.2. Part 2: Mission and Scope

- Motivation
  - **Introduction:** A desirable feature of Duckietown is the ability to park Duckiebots in a safe static state.
  - **Relevance:** The parking feature replicates the familiar scenario in real world driving when the driver no longer needs the transportation service offered by the vehicle. Parking allows the vehicle to be stored in a safe static state, without obstructing active Duckietown traffic, until the vehicle is summoned for further transportation services within Duckietown. Additionally, a parking feature allows for a variety of other benefits within Duckietown such as decreased traffic congestion on the roads as well as the potential for the recharging of the Duckiebot batteries while in a parked state.
- Existing Solution
  - The parking feature was implemented from scratch.
- Opportunity
  - There was no previous implementation of parking within Duckietown. In order to approach the problem, we first designed a physical parking lot to park the Duckiebots. A specification for the parking lot was therefore determined. In order to actually park the Duckiebots, we split the problem into three main

pieces of a parking pipeline:

- **Duckiebot localization:** Localization was implemented based on the existing AprilTag C++ library found [here](#).
- **Path planning:** A path was planned using Dubins paths and, during the instance of path obstacles, RRT Star with Dubins paths. A general description of Dubins paths can be found [here](#). An existing library for RRT Star with Dubins paths was implemented with help from the library [here](#).
- **Feedback control to the planned path:** As mentioned in part 1, this is the piece of the pipeline that currently requires development. We found that the localization via AprilTags takes several seconds to compute on the Raspberry Pi. The time lag proved to be insufficient in supplying the lane controller with sufficiently frequent state updates to control to. For further details regarding this issue, please see part 6 of this report (Future avenues of development).

### 21.3. Part 3: Definition of the problem

#### 1) Problem statement

---

We need to park N Duckiebots in a designated area in which they are able enter and exit in an efficient manner.

#### 2) Assumptions

---

- The parking lot is a four tile structure with defined inlet, outlet and colour scheme.
- The specification of the parking lot is known.
- When entering the lot and searching for a parking space, the parking lot is in a static state (there are no actively parking Duckiebots).
- Assume that when leaving the parking space, the parking lot is in a static state.
- The robot is limited in curvature, it exists a minimum curvature radius
- The robot can drive any desired curvature within the minimum curvature radius in forward driving mode
- The only possibility for backwards driving is straight, this a result of the used controller
- The robot must move in a car like behaviour, e.g. no side slip and no turning without forward movement is allowed, this is encoded in the equations of motion:

$$x'(t) = v * \cos(\theta(t))$$

$$y'(t) = v * \sin(\theta(t))$$

$$\theta'(t) = v / r_{turn}(t)$$

This results in a discrete time system (time discretisation  $T_s$ )

$$x[k+1] = x[k] + T_s * v * \cos(\theta[k])$$

$$y[k+1] = y[k] + T_s * v * \sin(\theta[k])$$

$$\theta[k+1] = \theta[k] + T_s * v / rturn[k]$$

### 3) Performance measurement

---

#### *Localization:*

- Localization involves computing a state estimate of the Duckiebot's position (x, y, theta)
- **Quantitative performance metric**
  - accuracy of state estimate in x[mm], y[mm] and theta [degree]

#### *Path Planning:*

- Path planning consists of planning a collision free path from the current state estimate into or out of a parking space given a static map (no actively parking Duckiebots)
- **Quantitative performance metric**
  - Percentage of collision free paths (# of collision free path / # of total paths)[%]

#### *Control:*

- Once a state estimate is computed and a path is planned, the Duckiebot must be controlled to the computed collision free path with a sufficiently high frequency of state updates.
- **Quantitative performance metric**
  - Starting at parking lot entrance, measure the number of parking manoeuvres completed within boundaries of designated parking spot (over N attempts)[%]
  - Starting in designated parking space, measure the number of Duckiebots able to arrive at the exit of the parking lot (over N attempts)[%]
  - Average time (for N vehicles) to enter and exit parking lot[seconds]

## 21.4. Part 4: Contribution / Added functionality

Initially, the theoretical descriptions and implementations of the three main parts of our parking pipeline (localization, path planning and control) are described. As you will notice in the descriptions below, there is no technical description or implementation description for control, as we intended to use the existing lane controller for control. In order to interface with this controller, we developed a state propagation strategy to send high frequency state updates to the lane controller. Therefore, we have included our implementation strategy for control and refer to it as "state propagation".

Following these descriptions, the logical architecture and software architecture of the pipeline as a whole is described.

### 1) Theoretical Descriptions

---

#### *Localization:*

The localization is based on the relative transformation of the Duckiebot to the Apriltags within the parking lot and their known position in the world frame.

A rectified images is needed to detect the AprilTags within the image. The used

wide angle camera on the Duckiebot provides a distorted barrel image. In a barrel distorted image each pixel is position closer to the optical center as it would be in a rectified image. The distortion is non linear and can be modelled by a polynomial function depending on the pixel distance to the optical center:

$$f(r) = 1 + k_1 r + k_2 r^2 + \dots + k_n r^n$$

$$r^2 = (u - u_0)^2 + (v - v_0)^2$$

The intrinsic camera calibration estimates the distortion parameters  $k_1$  to  $k_4$ . The rectified image can be computed by positioning each pixel of the distorted image at its actual position using the estimated parameters and the distortion model.

The rectified image is first converted to a gray scale image and afterwards thresholded to a binary image. Next the AprilTags in the binary image are detected.

The relative position of the camera to the each tag can be calculated, after one or multiple AprilTags are detected. The four pixels corresponding to the corners of each AprilTag in the image, as well as the position of the corners in the body frame of each AprilTag are known. Using this information and the intrinsic camera matrix the relative position of the camera and the AprilTag can be computed by using the PnP algorithm.

Once the relative position of the camera to each AprilTag is computed, the absolute position of the Duckiebot in the world frame can be calculated. First the position of the Duckiebot in the world frame can be calculated for each single AprilTag by combining transformation of the AprilTag in the world frame, the relative transformation of the camera and the AprilTag and the relative transformation of the Duckiebot and the camera. Next a more reliable state estimate can be computed by taking the average all estimated Duckiebot transformations.

### *Path Planning:*

We have to find a path in a predefined parking lot with given objects like other Duckiebots, walls or duckies. The area around those non-driveable objects define the obstacles. To be exact, every object is blown up by the distance of the center point of the robot to the most-distant point. This results in a problem of finding a path from start pose ( $x, y, \theta$ ) to end pose within a map where the information about where the robot is allowed to drive is encoded.

We implemented a two stage algorithm. The first stage is using Dubins curves where the second stage uses rapidly exploring random trees.

#### **Stage 1: Dubins path**

Given the assumption mentioned above (forward driving for a car like robot with given minimum curvature radius) the optimal path in an unlimited and obstacle free space on a Dubins path. This path is a combination of driving on a circle with minimum curvature radius and straight lines. This means that the Dubins path from start pose to end pose is calculated in the first stage. A collision checker is applied to the found path afterwards. If the path is completely within the parking space and does not enter the non-driveable region then we are done and we found the optimal path. If not, we switch to stage 2.

## Stage 2: RRT\*

An alternative way to find a path is the piecewise addition of small path segments with collision check on the fly. A point is randomly sampled within the parking space. The nearest point to the sampled needs to be found. The sampled point is connected to the nearest point using a Dubins curve if and only if the path candidate is collision free. A graph optimization is performed in a local area around the sampled point in the end. This procedure is repeated until a pre-defined number of nodes are sampled. This method is called rapidly exploring random trees with path optimization (RRT\*). This method converges to the optimal path with unlimited number of nodes. In practice, we stop earlier and have an approximation to the optimal path.

### *State Propagation:*

As there is no real theory involved in our strategy to interface successfully with the lane controller, please see the implementation section for an implementation strategy for state propagation.

## 2) Implementation

---

### *Localization:*

We slightly modified the previously implemented localization pipeline.

The localization pipeline takes a rectified image as an input. Therefore we need to undistort the barrel distorted images provided by the wide angle camera in a first step. To do so we use the distortion parameters determined in the intrinsic camera calibration. The previously implemented image rectification node undistorts the image in a way that is usable for the lane following pipeline, but unacceptable for a reliable state estimation. It is necessary for the state estimation based on apriltags to work that the undistorted image corresponds to the intrinsic parameters of the camera.

The previously used pipeline uses the image rectification node from the ROS library that is based on openCV. The node works in the following way:

- The node takes the distorted image (e.g. 480x360 pixels) as an input and undistorts the image using the default “initUndistortRectifyMap” openCV function.

Undistorting a barrel distorted image using all pixel information will result in an image with black areas along the edges as shown in figure 15.3 in the figure [here](#). The default openCV function will cut the biggest rectangular section out of the image that contains information for every pixel within the rectangle (red area) and map the image into a new image with the same size as the original image (i.e. 480x360).

This causes two problems. First of all, the ratio of the cutout section is not the same as the one of the original image. Forcing the selected section back in the original ratio will distort the image by stretching the image more in one direction than in the other causing a rectangle to become oblong.

Secondly, neglecting the distortion the overall scale of the image does not corre-

spond the focal length anymore.

Both problems cause a miss-match between the image and the intrinsic parameters that could easily be compensated by using scaling the focal length and using a different focal length in x and y direction resulting in a new intrinsic matrix. Instead we came up with the following solution:

- We compute the mapping of every distorted pixel coordinate to the undistorted pixel coordinate for a given intrinsic camera matrix and image size using the openCV “initUndistortRectifyMap” function with non-default parameters. We use the intrinsic camera matrix determined in the camera calibration and size of the original image. This way we overcome both problems that we had with the ROS image rectification node, while not changing the intrinsic camera matrix.
- For the apriltag detection we use the AprilTags for ROS library from the Robotics and Intelligent Ground Vehicle Research Laboratory. After an update of openCV 3 the algorithm did not work anymore due to a variable type error that we fixed.
- The apriltag detection node outputs all detected apriltag IDs and the corresponding transformation of the camera to each apriltag. We send these information to the previously implemented apriltag post processing node. The post processing node computes the transformation from the Duckiebot to each apriltag by including the static transformation between the duckiebot and the camera. Afterwards, the localization node computes the pose of the Duckiebot in world frame for each apriltag and averages the transformations to a more reliable estimate. We are able to calculate the pose of the Duckiebot in world frame, because the position of each apriltag in world frame is known to the robot. We extended the node by another custom message that includes the x and y position, as well as the orientation of the Duckiebot, because only this information are important for the path planning and control of the Duckiebot.

### *Path Planning:*

The path planning algorithm is written in Python. The “PythonRobotics” library from “AtsushiSakai” GitHub account is used as a basic for Dubins Paths and RRT\* implementation.

### **Initialization**

The parking lot is parameterized as a rectangle with given length (`lot_width`, `lot_height`). The path must be completely inside this rectangle.

Objects can be defined as rectangles with given properties (`x`, `y`, `dx`, `dy`, `colour`, `driveable`).

Obstacles are computed automatically based on the non-driveable objects. The rectangles are blown up. The non-driveable region is showed in magenta.

The parking spaces are numbered from 1 to 6. The entrance has index 0, the exit index 7. The pose of the parking space or entrance / exit can be computed with the function `pose_from_key(key)`. Input value is an integer with the index of the space. The output is a (`x`, `y`, `theta`) pose tuple.

To start the simulation the python script (parking\_main.py) can be launched with

two arguments. To get a path from the entrance to parking space 2 we type the following command. `./parking_main 0 2`

The path can either be printed on a (interactive) figure and/or it can be saved in the folder images.

### Stage 1: Dubins path

A Dubins path is calculated in stage one. The only argument is the minimum curvature radius. If the path is valid e.g. collision free, it is printed in green, otherwise in magenta.

### Stage 2: RRT\*

Since RRT\* has random character we need to define a stopping criteria. This is done in limiting the number of nodes which is equal in the number of iterations. The design parameter `maxIter` changes this. For the local graph optimization we need to define the area, this can be changed in changing the parameter `radius_graph_refinement` which holds a distance in mm.

### Path variation

The robot can only be controlled on a path when forward driving. When backing up we run the robot in an open loop fashion and only straight backwards driving is allowed. The distance travelled back can be adjusted with the variable `distance_backwards`.

### Generate necessary control output

To provide all necessary control signals, the pose of the robot and the path are combined to define an estimated distance from the path (`d_est`) and a reference distance (`d_ref`). The differential heading between the robot pose and the of the point on the path with lowest distance is calculated (`theta_est`). Furthermore, the reference velocity (`v_ref`) and curvature (`c_ref`) are given to the controller.

This can be tested in the file `project_point_to_path.py` with two input parameters (`start_index` and `end_index`).

### *State Propagation:*

In order to control to the planned path, the lane controller is utilized. We developed a path planning node that “fills in” the state update time lag gaps with a feedforward state update. In addition to the feedforward feature, the algorithm allows the Duckiebot to stop for a set period of time in order to plan a new path. The time for which the Duckiebot is stopped is ensured to be sufficient in order to produce an accurate state estimate. As such, the algorithm behaves as follows:

- 1) Process AprilTags in view and estimate a state while static (Duckiebot velocity is zero)
- 2) Plan a path based on this state
- 3) A “time to plan” threshold has passed
- 4) Use feedforward state estimates to broadcast inputs to the lane controller
- 5) Stop the Duckiebot after a “time to control with feedforward” has been passed
- 6) Return to 1)

In order to perform the above steps, an alternate node, named `devel_path_planning_node.py` was constructed. This node can be found in the `src` folder of the

parking package. As mentioned before, this is an experimental node and needs further development to function with the other parking nodes. The `dev_el_path_planning_node` introduces new functions, namely a `stopping_callback` function and a `get_intermediate_pose` function. Please see below for a description of each.

- The `get_intermediate_pose` function takes the time since a pose was last calculated as an input. The function then uses that time, along with the Duckiebot's velocity to estimate where along the path the Duckiebot is. The estimate is then broadcasted to the controller.
- The `stopping_callback` function allows the Duckiebot to stop for a sufficient amount of time for the camera to estimate a new state via any AprilTags in view and plan a new path. While in the function the Duckiebot velocity is set to zero.

The following logical architecture describes the parking pipeline as whole.

### 3) Logical architecture

---

The logical architecture is a description of the functionality: what happens when we click start?

- As soon as you arrive to the parking lot entrance and see a parking AprilTag, the Duckiebot switches from normal driving mode to parking mode. Parking mode is only allowed in the parking lot. **Note:** the Duckiebot must currently be manually placed at the entrance of the parking lot for the parking feature to engage.
- If the Duckiebot exits the parking lot, it views a “parking exit” AprilTag and it switches from parking mode back to normal driving mode (starting at a four way intersection). **Note:** this feature is not currently implemented in the parking pipeline as it currently stands.
- Once at the parking lot entrance, the Duckiebot estimates her pose (x, y and theta) using as many AprilTags as possible within view. The localization of course requires the camera nodes, AprilTag detector node, Apriltag Postprocessing node and localization node to be launched. There is at least one (potentially more) AprilTags per parking space and possibly some additional tags placed at the entrance and exit. To estimate a pose, the state estimation algorithm has been extended (see localization description above) using the library ‘AprilTags C++’. It estimates the relative position of the robot with respect to the april tag. The location of the april tag is encoded in the QR code. As soon as you see one (better two) tags, the pose can be calculated. We assume that we always see at least one tag.
- Given a prior information about the parking lot (where the parking spaces are located, where the robot can drive, etc) and real time vision information, the robot chooses a parking space. Currently, the parking space is chosen as a “hardcoded” value in the launch file, please see the demo operation manual for more information. We assume that the parking lot is empty or that other Duckiebots are static (do not move) and this is encoded in the parking map (places where the robot is not allowed to drive).
- We use Dubins paths to generate a path given the pose of the robot, the pose of the assigned parking space and the parking map. If there is an obstacle in place, we use RRT star with Dubins paths to generate a path (this feature is coded, but not currently implemented within ROS). The above features are launched in the

path planning node.

- We control the robot to the optimal path using a sufficient controller using visual feedback. The control is performed in the lane controller node.
- For driving to the exit, we generate a path and control our robot to this path which includes driving backwards to leave the parking space and turn to get to the exit in a forward motion. **Note:** this feature is currently not implemented within ROS.

Target values:

- accuracy: the error is a combination of localization accuracy and the offset due to the maximum allowable controller error. To park two Duckiebots next to each other within the space boundaries, the path planning accuracy has to be less (or equal) than 5 cm (which is the distance from the robot edge to the parking lane)
- the point of the robot which is the furthest away from the parking mid line should be less than half of the parking space width while the heading of the robot must be less than a constant (20 degrees) relative to the parking space boundary lines.

#### 4) Software architecture

---

`rosnode list` (note that topic names are often remapped in launch files. Please refer to specific launch files for details):

- `image_proc_proportional_node.py`
- subscribes: `/camera_node/image/raw, /camera_node/raw_camera_info`
- publishes: `image_rect`
- `apriltag_detector.cpp`
- subscribes: `image_rect`
- publishes: `tag_detections_image, tag_detections, tag_detections_pose`
- `apriltags_postprocessing_node.py`
- subscribes: `apriltags_in`
- publishes: `apriltags_out, tag_pose, apriltags_parking, apriltags_intersection`
- `localization_node`
- subscribes: `apriltags`
- publishes: `/tf, pose_Duckiebot`
- `path_planning_node`
- subscribes: `pose_Duckiebot`
- publishes: `parking_pose, parking_active`
- `lane_controller_node`
  - **note:** we copy this node from ‘the controllers’
  - subscribes: `parking_pose`
  - publishes: `car_cmd, actuator_limits_received, radius_limit`

`rostopic list :`

- `image_rect`
  - from `sensor_msgs.msg`, type: `Image`
- `tag_detections` - latency: 3 seconds
  - from `Duckietown_msgs.msg`, type: `AprilTagDetectionArray`
  - **note:** this is the topic that is published at a frequency of ~ 1 signal/ 2-3 seconds. As such, this topic is the bottle neck of the algorithm. Please see Part 6 for

- potential remedies for this issue.
- apriltags\_in - remapping of tag\_detections
    - from Duckietown\_msgs.msg, type: AprilTagDetectionArray
  - apriltags\_out - latency: few milliseconds
    - from Duckietown\_msgs.msg, type: AprilTagsWithInfos
  - apriltags - latency: few milliseconds
    - from Duckietown\_msgs.msg, type: AprilTagsWithInfos
  - pose\_Duckiebot - latency: few milliseconds
    - from Duckietown\_msgs.msg, type: Pose2DStamped
  - parking\_pose - latency: few milliseconds
    - from Duckietown\_msgs.msg, type: LanePose

## 21.5. Part 5: Formal performance evaluation / Results

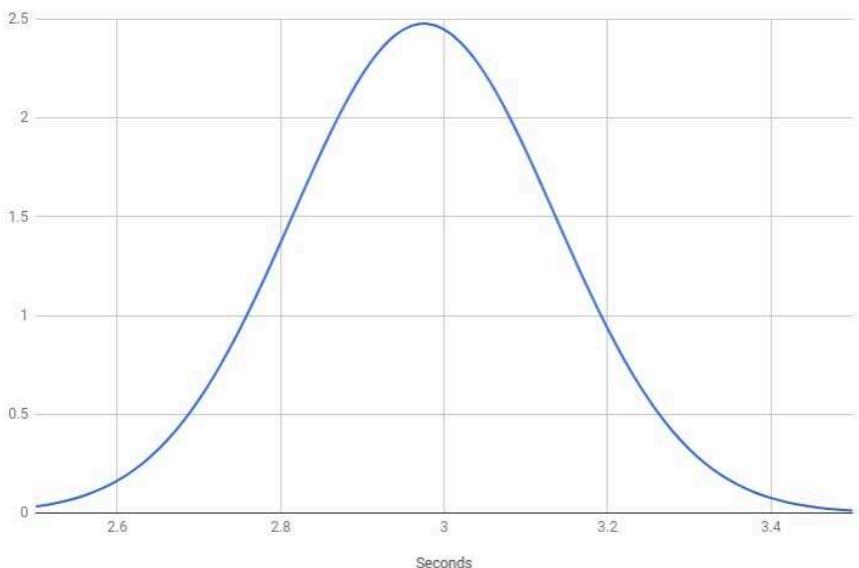
### 1) Localization and State Propagation

---

The localization of the Duckiebot depends on the distance of the camera to the apriltag, as well as the angle between the camera and the AprilTag.

If the camera image plane and the AprilTag are parallel and the Duckiebot is no further than 0.3m away from the AprilTag, the precision of the distance is +/- 0.5mm and the angle +/- 3 degrees. This precision is sufficient to localize the Duckiebot reliably within the parking lot. The localization performs to the metrics presented earlier (even with a single AprilTag in sight of the camera) as long as the AprilTag is facing the Duckiebot with a relative angle less than 45 degrees.

A major problem is that the detection of the AprilTag takes 3 seconds (mean 2.9 sec, std. dev. 0.19 sec, see the figure below).



This is currently the bottleneck of the parking pipeline. To control the duckiebot, a real-time state estimate or a reliable state propagation is needed.

Unfortunately the state estimation is based on the AprilTag detection. Updating the state estimate every 3 seconds is too slow to control the robot. The lower velocity of the Duckiebot is limited to 0.1 m/s due to the Adrafruit motor drivers. This causes the Duckiebot to move 0.3m between control updates. If the robot is supposed to do a small left turn and then go straight, it will continuously drive in a circle as the controller attempts to correct toward an inaccurate state update (again, due to the time lag).

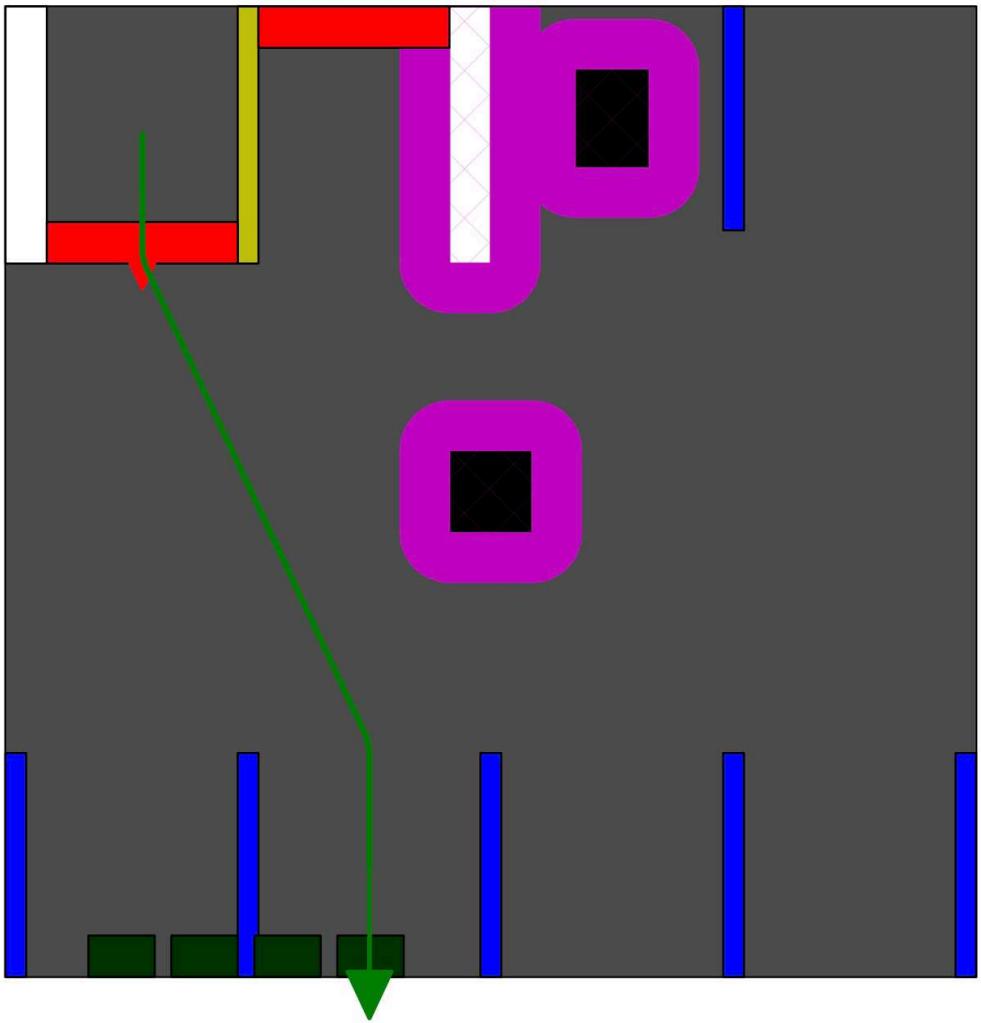
To overcome this problem we started to estimate the state of the Duckiebot using a simple mathematical model by integrating the distance that both wheels have traveled.

This state propagation proved insufficient as the amount each wheel rotates based on the control input is not accurate. This is caused by the slippage of the wheels and a non-linear and inaccurate relationship between the input voltage and the output momentum of the DC motors. This resulted in the robot to driving slower or faster then the commanded velocity as well as to turn on a smaller or bigger turning radius. In order to compensate for this, we introduced calibration factors for the commanded velocity to actual velocity and commanded radius to actual radius. We achieved better results using the state propagation approach, but were still unable to park the Duckiebot in one of the parking spaces.

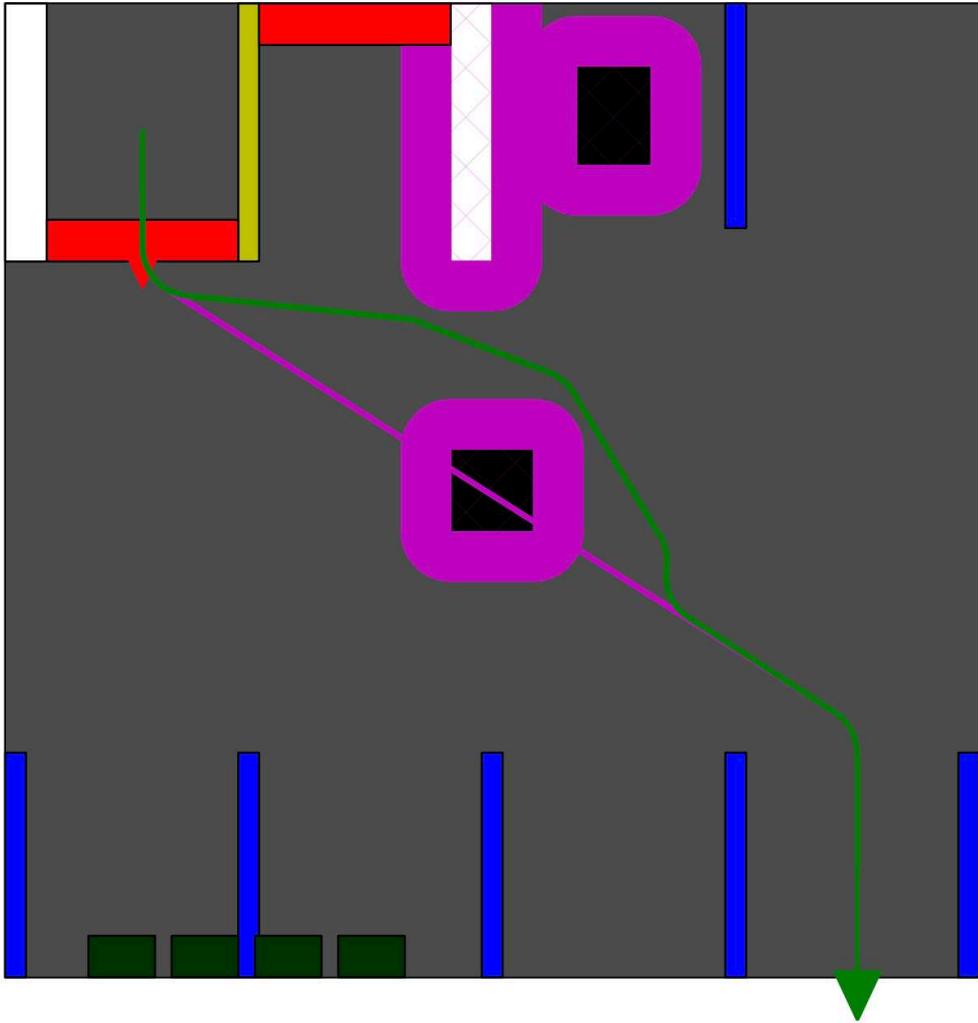
## 2) Path Planning

---

The simulation works smooth and almost always finds a path. We successfully implemented Dubins pathes and RRT\*. The simulation is fast if a solution with dubins path is found. The computation time is under 0.2 seconds. It needs much more time if RRT\* is used, a solution is eventually found after 20 seconds for hundred iterations.



--



## 21.6. Part 6: Future avenues of development

As mentioned before, the main area of work needed to get the parking pipeline working is to successfully implement some sort of control in order to autonomously park a Duckiebot. Currently, there are three main options for doing this, described in the following sections. Each avenue of development may be explored individually or a combination of multiple could prove to be the best way forward.

### 1) Increase the speed of the state estimation

- As seen in the part 5 of this report, there is a considerable time lag in the state estimation via AprilTags. An avenue of investigation should be to look deeper into the `extractTags` method from the AprilTag C++ library. The node where this method is called is found here, on line 67:

### `ag_detector.cpp`

- Any development which can increase the speed of this ‘extractTags’ method, which takes a grayscale image and detects tag number(s) in view, would be very beneficial for an increased state estimate frequency.
- Another avenue of development may be to increase the computing power of the Duckiebot. The parking pipeline was currently run on a Raspberry Pi. A more powerful computer may improve the time lag issue.

### 2) Successful integration of state propagation

- More development could be made on the `devel_path_planning_node` node that propagates the state estimate at a high frequency for use with the lane controller. Please see the “State Propagation” section of part 4 for how this algorithm is intended to work.
- As of the writing of this report, the parking group was unable to successfully integrate the state propagation. More work is needed to allow the algorithm to work as designed.

### 3) Development of a dedicated parking controller

- 1) and 2) above rely on the use of the lane controller while parking. It may be beneficial, however, to develop a dedicated parking controller which can better handle the parking feature pipeline.

## UNIT L-22

# Explicit Coordination: preliminary report

### 22.1. Part 1: Mission and scope

#### 1) Mission statement

---

Coordinate intersection navigation safely through explicit communication.

#### 2) Motto

---

IN HOC SIGNO VINCES

(with this sign, you will win)

#### 3) Project scope

---

Employ LEDs based communication to efficiently manage traffic at intersections. First, LEDs should be used to reliably communicate Duckiebots positions and/or intentions. Then, optimal control theory or game theory might be considered to safely clear the intersection in reasonable time. By safely, we mean that no colli-

sions occur (100% success). This requires robustness in message interpretation.

We aim to solve the problem of clearing the intersection in max. 60 seconds (meaning that even in the case of four Duckiebots participating we can solve the problem below this time). A decentralized solution should be implemented.

*What is in scope:*

The following aims are identified:

- List the messages that need to be exchanged between Duckiebots;
- Encode the messages in LED language;
- Produce the LED signal;
- Detect the LED signal;
- Decode the LED signal;
- Act safely upon the received information.

*What is out of scope:*

The following aims are beyond the scope of the project:

- Changing the communication protocol, i.e., we will stick on using LEDs.

*Stakeholders:*

The following pieces of Duckietown will be interacting with the project:

- Fleet planning;
- Implicit-navigation (determinate where the Duckiebot in the row has to stop);
- Anti-instagram filtering;
- Integration heros;
- Map designers;
- Traffic navigation;
- Controllers.

## 22.2. Part 2: Definition of the problem

### 1) Mission

---

Duckiebots must be able to cross an intersection in the defined reasonable time and without collisions.

### 2) Problem statement

---

The following problems are to be tackled:

- LED-based communication;
- Coordination at intersections.

### 3) Assumptions

---

The following assumptions are made for the LEDs communication:

- The Duckiebot is of type DB17-1.

- One to four Duckiebots are at the intersection with a certain position and orientation with respect to the stop line (projection of Duckiebot on 2D lane):
  - Min. 0 cm behind red line;
  - Max. 6 cm behind red line;
  - Max. +/- 2 cm from center of the line;
  - +/- 10° of rotation.
- Duckiebots are able to see the vehicles in front and on the right with respect to their position.
- LEDs work properly and emit the signals with the right color and frequency, and we can also detect the LEDs as the correct state that they represent, and attribute those LEDs to the correct Duckiebot that is displaying the LEDs.
- The Duckiebots do not move while “waiting” in the intersection, but they can move on the spot to look around to see the left.
- Camera works properly (frequency 30Hz, resolution of 64x48).

The following assumptions are made for the coordination:

- Signals are correctly recognized and associated to the corresponding messages.
- The intersection is among one of the standard intersections of Duckietown. That is, weird intersections will be ignored (at least in a first approach to the problem).
- The intersection type and the presence of a traffic lights are known.
- Intersection navigation is guaranteed to be working safely.
- One Duckiebot navigates the intersection at the time.

#### 4) Approach

---

Firstly, the existing code and see its limits should be tested. Benchmark tests have to be designed in order to carefully measure the performances of the implementation. Then, the literature should check to see what has already been implemented.

The problem can be splitted in LED-based communication and intersection coordination. We further distinguish between intersections with traffic lights and intersections without traffic lights (recall that it is assumed that the Duckiebots know the intersections they entering). In all cases we aim for a decentralized solution.

Possible approaches for LEDs communication:

- Last year’s algorithm;
- Alternative approach 0.

Possible approaches for intersection coordination without traffic lights:

- Last year’s approach;
- Alternative approach 1;
- Alternative approach 2;
- Alternative approach 3.

Possible approach for intersection coordination with traffic lights:

- Detect the signal of the traffic light and behave accordingly.

*Approaches for communication :*

*Last year's approach:* Communication works through LEDs blinking at different frequencies. Colors are just for human understanding and are operational meaningless. See last year's documentation for further details.

*Alternative approach 0:* Communication works through LEDs blinking at different frequencies, colors, and patterns.

#### **Approaches for coordination without traffic lights :**

*Last year's approach:* See last year's documentation.

*Alternative approach 1:*

Assumption: every Duckiebot does not see on the left. Here, Duckiebots do not need to turn in place to see the Duckiebot on the left

There are two messages:

- One signal to say if the Duckiebot can see someone on its right (e.g., a red LED blinking, signal A);
- One signal to say whether the Duckiebot is at intersection or is about to go (e.g., same LED blinking at different frequencies, signal B).

The coordination plan works as follows:

1. If Duckiebot:
  - a. does not see signals from the right and
  - b. it does see signal A from the opposite side of intersection. Then it will enter the intersection. That could happen with 1, 2, 3 Duckiebot at intersection.
2. If 4 Duckiebots are at intersection, that means, every Duckiebot is emitting signals A and B, each Duckiebot turns off with a certain probability (say 25%), so that one is going to see nobody on the right and will navigate the intersection.
3. If one Duckiebot does not see any kind of signal (A or B) it will proceed to navigate the intersection.
4. One case still need to be analyzed, i.e., the one in which we have 2 Duckiebots one opposite to the other. In this case:
  - a. The Duckiebot does not see signals from the right and
  - b. Does not see signal A from the opposite Duckiebot (but it does see signal B).

Then each Duckiebot turns off with some probability (say 50%) so that it will see no more signals and then navigate the intersection, as explained in case 3.

*Alternative approach 2:* There are four messages:

- No color: waiting to enter the red queue;
- Red: waiting to enter the negotiation;
- Yellow: about to enter the intersection but still looking around, can go back to green;
- Green: negotiating.

In the following, let you be a Duckiebot.

1. You currently have no color. You stop at the intersection, turn left 20° in order to see all existing bots. You have no lights on at the moment. Possible scenarios:

- If there are cars with yellow or green colors, you turn red. You will start negotiation after current negotiation ends and Duckiebots involved in the current negotiation pass the intersection.
  - If there is no green or yellow lights, but there is at least one Duckiebot with red color, you wait until you see a green or yellow color. You turn red. You go to 2 a).
  - If no lights are present, you turn yellow and wait for t seconds. If after t seconds there are still no lights on or all lights are red, you enter the intersection.
  - If there is at least one vehicle with yellow lights, you turn green (you need to negotiate with yellow), go to 2 b).
2. 1. You are currently red. Wait until yellows and/or greens are gone (assuming we also know their spots: opposite, left, right), check if there are any other reds: \* If no reds are present, turn yellow and enter the intersection (you can execute immediately because you were red in the beginning of this step so if there are no reds when you checked, Duckiebots should be waiting for you to turn yellow or green so that they can turn red, therefore there cannot be a synchronization problem here). \* If reds, turn green and go to 2 b) (if some reds turned green too quickly so that one red was unable to catch the red color, it still knew the spots of greens/yellow that existed in the beginning of 2 a), so if you see greens in the new spots can, you can still turn green and go to 2b). 2. You are currently green. Wait for random (0.2, 1) seconds: \* If there exists a yellow negotiator wait for t seconds and repeat 2 b). \* Else (if all negotiators show green), turn yellow, wait for t seconds. \* If still all negotiators show green, enter the intersection. \* If there is a yellow negotiator wait for t seconds and repeat 2 b).

(after the above algorithm, we could also implement the below to increase throughout)

Back lights: If you are entering the intersection, turn back lights to green. Else (negotiating, or waiting), turn off back lights.

Duckiebots behind a Duckiebot that stopped: If see a green back light, get ready to run, turn front lights to yellow and immediately enter the intersection after the car in front of you. If no green light on the back of the Duckiebot in front of you, stop at the red line as usual and start executing the intersection algorithm.

Improvements: We could possibly give frequency to back lights to let more than 2 Duckiebots enter the intersection, such that the first Duckiebot that is about to enter the intersection has constant green lights on the back, 2nd one behind it has green back lights with frequency x, and 3rd behind 2nd one knows it is the 3rd since it sees green light with frequency so it also enters the intersection immediately (and has lights off on the back so that 4th has to stop at the red line and execute the default intersection algorithm).

Therefore, this alternative needs 3 different signals, such as green, red, yellow and it will also use the no color case as a 4th signal.

### *Alternative approach 3:*

This approach consists of an exponential backoff model (assuming that two oth-

er visible cars is a low enough number to have it run quickly enough). Here:

- No turning is needed.
- If you see a Duckiebot at the right, give it “right of way”.

The intersection policy works as follows:

START:

- go to CHECKING (blue)

CHECKING:

- if a bot is “out of place” (in the intersection)
  - go to WAIT (red)
  - go to CHECKING
- if no Duckiebots in CHECKING,
  - GO (green)
- if Duckiebot at right or front is CHECKING
  - go to WAIT (red)
  - go to CHECKING

WAIT:

- exponential backoff

## 5) Functionality-resources trade-offs

---

*Functionality provided:*

Metrics for LEDs communication:

- Maximize percentage of success in detecting a LED light or LED blinking in a picture taken from the Duckiebot camera.
- Maximize percentage of failed attempt of communication in a Duckietown intersection.
- Minimize time needed in order to detect signals.

Metrics for coordination:

- Maximize the times the intersection is cleared safely, i.e., without crashing.
- Minimize the time needed to clear the intersection for each Duckiebot and for the fleet.
- Maximize the number of successful intersections cleared safely below a threshold time (which may depend on the intersection itself and on the number of Duckiebot at the intersection).

*Resources required / dependencies / costs:*

The following resources are needed in order to test the behaviour:

- Four Duckiebots are needed to test the coordination.
- A Duckietown intersection.
- Traffic lights.

*Performance measurement:*

Performance measurements for LEDs communication:

- We put one Duckiebot in a lane at the intersection, one Duckiebot on the opposite lane across the intersection, and then one on the right. First the opposite and then the right Duckiebot will emit signals, the observer will receive them. We will see what happens if we let LED-detector-node run. We should be able to detect in which regions are LEDs blinking (See f23-presentation, Minute 11:20 in Google Drive).
- We put four Duckiebots at an intersection (with and without traffic light) and let them communicate. This allows to test whether the communication was successful or failed and the time needed to perform it.

Performance measurements for coordination:

- First, test the algorithm used to clear the intersection with computer simulations to see if there are any theoretical problems with the algorithm itself.
- After having an algorithm that computes a good “plan” to clear the intersection, we run it x times and see how many times the intersection is cleared safely.
- We measure how many seconds it takes to clear the intersection.

### 22.3. Part 3: Preliminary design

*Modules:*

Following subprojects are detected:

1. LED communication
  - a. Duckiebot A emits a signal encoded in LED;
  - b. Duckiebot B detects the signal from Duckiebot A;
  - c. Duckiebot B interpret signal from Duckiebot A.
2. Coordination at intersection
  - a. Each single Duckiebot computes “the plan” to clear the intersection;
  - b. Each Duckiebot leave the intersection according to “the plan”.

*Interfaces:*

1. For the LED communication:
  - a. Led\_emitter: Input for the emission is the state of Duckiebot A (waiting, entering, or navigating the intersection). The output is the corresponding LED signal.
  - b. Led\_detection: Input is the image of the camera. The output is the frequency/color/etc of the detected LED(s).
  - c. Led\_interpreter: Input is the frequency/color/etc of the detected LED. The output is the corresponding message.
2. For the coordination: The inputs are the type of intersection (with or without traffic light and number of streets) as well as the interpretation of the signals emitted by the other uckiebots. The output is decision on when to go, taken accordingly to the coordination policy.

#### 1) Preliminary plan of deliverables

---

*Specifications:*

The Duckietown specification do not need to be revisited.

### *Software modules:*

The software will be organized as follows:

- One ROS node for the emission.
- One ROS node for the detection.
- One ROS node for the interpretation.
- One ROS node for the coordination.

The existing code has already a similar structure, meaning that part of the code might be reused.

### *Infrastructure modules:*

No infrastructure modules are needed.

## 22.4. Part 4: Project planning

### 1) First steps for the next phase

---

We are going to implement and test the alternative coordination algorithms along with the last year's algorithm and check safety and performance metrics.

TABLE 22.1. TIMETABLE

Week	Task	Expected Outcome
12.11.17	Kick-off Hardware up to date Specifications for other groups	Preliminary design
19.11.17	First tests	
20.11.17-	Test existing code Implementing new algorithms	Benchmark results
26.11.17		First implementation
27.11.17-	Implementing new algorithms continues	Performance of the new algorithms-
03.12.17	Benchmark the new algorithms	Deadline for Chicago
		Implementation
04.12.17-	Try more sophisticated implementations such as algorithms that include communication with the bots behind, through back lights	
10.12.17		
11.12.17-	Benchmark new algorithms	Benchmark results
17.12.17		
18.12.17-	TBD	TBD
24.12.17		
25.12.17-	TBD	TBD
31.12.17		
01.01.18-	TBD	End of the project
07.01.18		

### *Data collection:*

- Test what is seen at different initial orientations. This will be employed as specifications for the controller group and for the smart city group.
- Run experiments, take pictures, and screenshots for both LEDs detection and

and intersection coordination.

*Data annotation:*

No data needs to be annotated.

*Relevant Duckietown resources to investigate:*

These are some of the important features used in the past course:

- Produce LED signal (protocol for emitting signal) - led\_emitter.
- Detect LED blinking signal from camera - led\_detection.
- Interpreting signaling data includes - led\_interpreter.
  - a. Determining what kind of signal we have (is it a Duckiebot? A traffic light? Something unwanted?).
  - b. knowing how many other cars at intersection (A, B, C) and if there is a traffic light up (present/ absent).
- Coordination policy and ros node from last year.
- MIT 2016 presentation.

*Other relevant resources to investigate:*

Literature and codes from the class of 2016 at MIT. Additionally, research papers in computer vision (How to detect LEDs? How to use the camera? Etc.) and coordination (how to optimally coordinate a fleet? Etc.) might be of interest.

## 2) Risk analysis

---

The following risks for LEDs communication are identified:

- Detecting colors. A possible solution would be to test colors and frequency to see which one is more robust.
- Limited visibility. Slightly turning in place (and then coming back to the initial position) might alleviate this problem.

The following risks for coordination are identified:

- Synchronization problems when Duckiebots arrive at the intersection at different times and when they detect signals from other Duckiebots while Duckiebots are changing signals.

We can summarize the risks in the following table.

Risk	Potential error	Consequence	Likelihood	Impact	Risk Priority Number (0-100)	Action
Detecting color	Camera does not recognize colors	Signal cannot be encoded in color	7	9	63	Alter interpretation strategy
Visibility	Limited visibility from DB camera	Limited exchange of messages	10	8	80	Strategies that not inform from
Synchronization before messages	Problems when bots arrive at the intersection at different times	Wrong communication	3	10	30	Takes account while much strategy Safety ced nee
Synchronization during messages	Problems when bots detect signals from other bots while bots are changing signals. Problems if bot detects signals while other bots are changing signals	Wrong communication	3	10	30	Safety cedu this

## UNIT L-23

### Explicit Coordination: intermediate report

#### 23.1. Part 1: System interfaces

##### 1) Logical architecture

Our job starts when Duckiebots are stationary at the red-line of the intersection (this is communicated to us via controllers/ parking). By clicking "start" the LED-coordination-node tells the LED-emitter-node to turn the LEDs white for all Duckiebots.

Afterwards, the LED-detector-node checks for each Duckiebot if other LEDs are seen and tells it to the LED-coordination-node. Note that here there is, at least in a first approach to the problem, no turning, i.e., LEDs of Duckiebots on the left are not identified. The LED-coordination-node estimates the coordination move (either “hold on” or “go”) for each Duckiebot. The final output is a signal, named move\_intersection, that will be used by the Navigators to start the procedure to navigate the intersection. Thereafter, we are not going to intervene until the Duckiebot finds itself at another intersection. Should the explicit coordination fail (for instance, because of Duckiebots not equipped with LEDs), the task of coordinating the intersection is given to the implicit coordination.

Our LED-detection, LED-emission and LED-coordination nodes affect only the Duckiebots behavior at intersection. Surely, our LED-signal could be seen from other Duckiebots in Duckietown but, at least for now, no group (except for the fleet planning group, see below) needs LEDs-based communication in other situations. A LED-signal will be used by fleet-planning to indicate the status of each vehicle (free, occupied, waiting, etc.). The Fleet planning will be using one LED for implementing this functionality (back-right one) while the other LEDs remain available for coordination purposes.

The following assumptions are made about other modules:

1. When the Duckiebot is made to stop at the red line by the Controllers at an intersection a flag “at\_intersection” will be set and that is when the coordination will start. Most likely this flag will be sent out by the Parking group after it has been verified that after the intersection there is no parking.
2. Controllers guarantee that the Duckiebots will stop at the red line within the agreed tolerances (i.e., Min. 10cm behind center of red line; Max. 16cm behind center of red line; +/- 10° of rotation ; +/- 5 cm offset from center of the driving lane.).
3. Fleet planning and neural-SLAM are the ones responsible to give information about where the Duckiebots should go at the intersection (information that will not be used in any case for determining how the intersection will be cleared).
4. Navigators will take over once the Duckiebot has received the order that it can proceed to navigate the intersection (a signal “go”), moment from which our team, explicit-coordination, will no longer intervene.
5. If the Fleet planning and neural-SLAM decision is not available, the Navigators are responsible to generate a random choice for the direction that each Duckiebot will have to follow in the intersection navigation, once again, the direction that the Duckiebot will take is not of interest for the coordination part that is performed regardless of this information.
6. Explicit coordination and implicit coordination will never run at the same time on a Duckiebot.

## 2) Software architecture

---

Nodes:

1. LED\_coordination:
  - Input: From Parking group “you are at an intersection” (additionally there is a parameter that indicates whether intersections are cleared with explicit or implicit coordination)

- Output: Duckiebot move (“go”/ “not go”)
  - Subscribed topic:
    - flag\_at\_intersection from Parking group, bool message: true/ false
  - Published topic: move\_intersection
    - string message: go/ no\_go
2. LED\_emitter:
- Input: Communication is needed
  - Output: LED turn on or stay off
  - Subscribed topic:
    - LED\_switch from LED-coordination, string message: on/ off
  - Published topics: None
3. 1. LED\_detection: Depending on the algorithm implemented:
- \* Input: camera\_image (possibly after anti-instagram) and message indicating whether detection is needed
  - \* Output: LED detected/ LED not detected
  - \* Subscribed topic: \* LED\_to\_detect from LED\_coordination, string message: yes/ no
  - \* camera\_image from anti-instagram, CompressedImage
  - \* Published topic: \* string message: LED\_detected/ no\_LED\_detected
2. LED\_detection: second option:
- \* Input: camera\_image (possibly after anti-instagram)
  - \* Output: LED detected/LED not detected with position and/or color and/or frequency
  - \* Subscribed topic:
    - \* LED\_to\_detect from LED\_coordination, string message: yes/ no
    - \* camera\_image from anti-instagram, CompressedImage
  - \* Published topic:
    - \* string message: LED\_detected/ no\_LED\_detected with position and/or color and/or frequency

A diagram of our nodes is shown below.

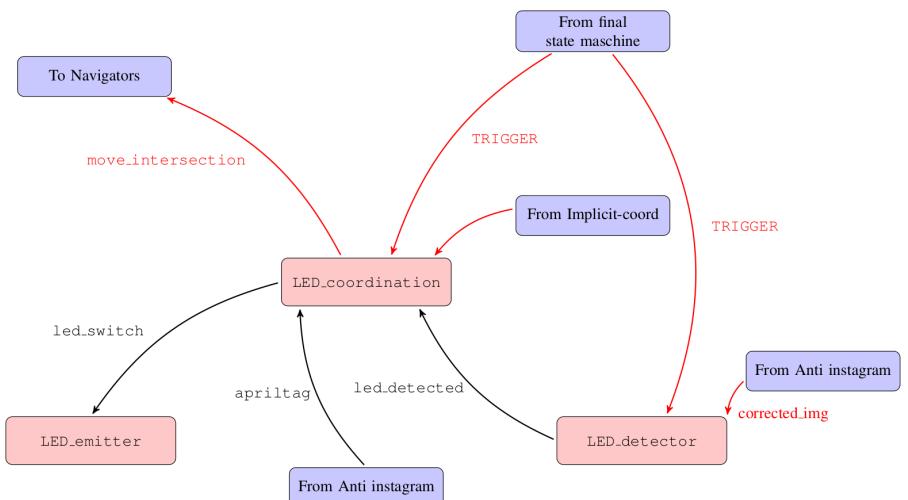


Figure 23.1. Nodes

We subscribe to the following topics:

- Corrected image with maximum assumed latency 1s;
- Flag at intersection with maximum assumed latency 1s.

The following topics are published:

- Flag go/no\_go with maximum latency 60s (this is the time needed to make sure that the intersection can be navigated safely).

## 23.2. Part 2: Demo and evaluation plan

### 1) Demo plan

---

Our demo will be conceptually similar to the MIT2016 “openhouse-dp5”, available from last year [Coordination \(master\)](#). The Duckiebots that are navigating in Duckietown, will stop at the red line and LED-communication and coordination will be performed leading to the eventual clearing of the intersection.

From testing last year’s code we realized that the coordination does not seem to work with the mentioned demo. Duckiebots stop at the red line but they do not communicate so that they never leave the intersection or decide to go independently of the presence and decision of the other Duckiebots. Although we investigated the problem by looking at separate nodes, no solution has been found yet.

We aim to have a working demo that will show an effective clearing of an intersection with a variable number of Duckiebots (1 to 4) regardless of the type of intersection (3-way or 4-way, with or without traffic lights). The intersection should be cleared in a reasonable amount of time (less than 1 min) and be robust to different initial conditions (within the specified tolerances on the pose of the robots). The set-up will be easy and quick: with a small Duckietown as shown in the figure below, up to four Duckiebots will be put on the road and the demo will be started from a laptop with no further interventions required.

The required hardware will therefore be: A four way intersection tile (see image below, center), four three-way intersections tiles, twelve tiles with straight lines, four tiles with curved lines and four empty tiles. In total, twentyeight tiles, red, yellow and white tape as indicated in the figure below. Apriltags and all other required signals at the intersection will also be needed (standard type of Duckietown intersection) as well as a traffic light to illustrate the behaviour in a traffic light type of intersection.

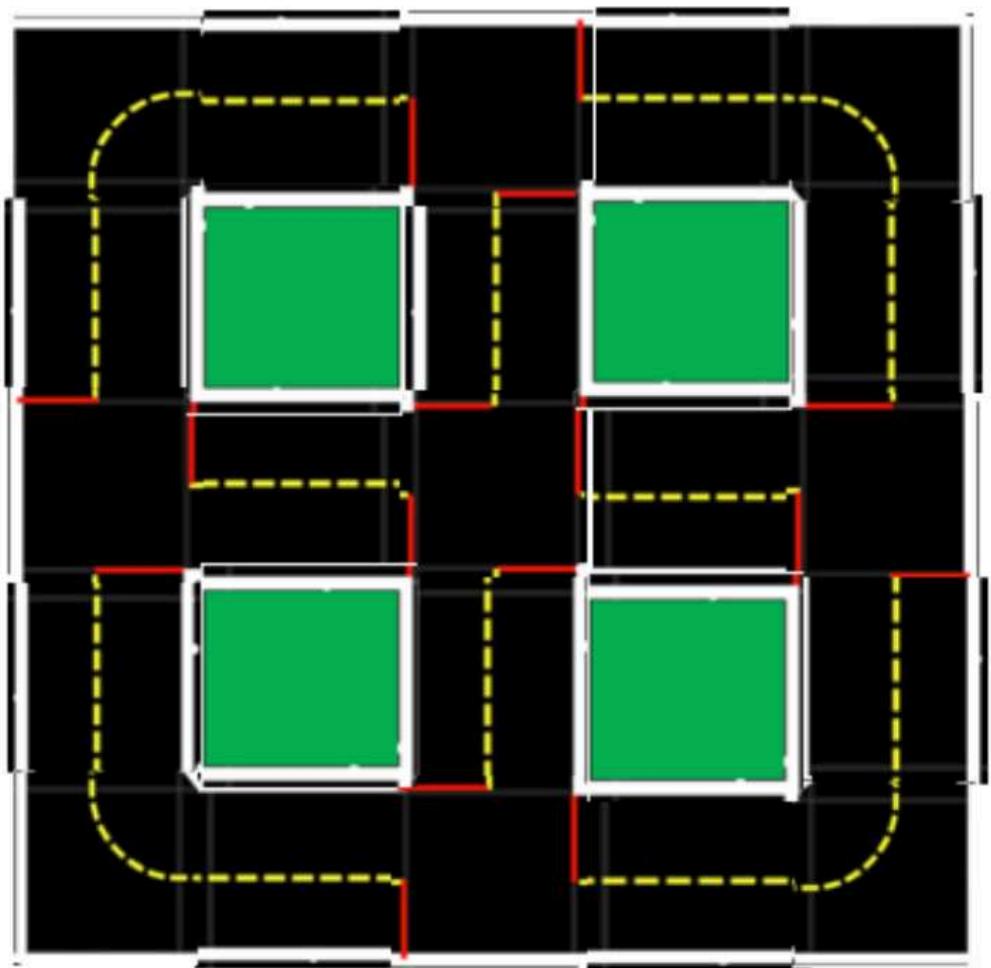


Figure 23.2. Duckietown

## 2) Plan for formal performance evaluation

---

Performance will be evaluated with 3 tests:

TABLE 23.1. PERFORMANCE EVALUATION

What is evaluated	How	Required	Collected quantities	Performance measure(s)
Coordination implementation performance	Run the demo	One to four Duckiebots at an intersection, every case has to be analyzed (three or four way intersection, with or without traffic light)	Time required to clear the intersection and binary variable that tells whether the intersection was effectively cleared without problems (collisions, lack of decision making, wrong detection of signals,etc.) or not.	Mean clearing time or success rate - Both for each case separately and for all combined.
LED emitter and LED detector	Run the specific nodes	Two to three Duckiebots in an intersection configuration (all relative positions have to be analyzed)	Number of unsuccessful LED emissions (the output has not the desired colour and/or frequency) and number of unsuccessful LED detections: the output of the LED detector does not contain all the signals it should have detected or it contains more than the ones effectively present (false positives)	Success rate - For both LED emission and detection.
LED detector	Run the specific node	Two to three Duckiebots in an intersection configuration (all relative positions have to be analyzed)	Time required to perform the LED detection	Mean detection time

### **23.3. Part 3: Data collection, annotation, and analysis**

#### **1) Collection**

---

No data is needed to develop the algorithm. Data might be needed to test the implementation of the detection.

#### **2) Annotation**

---

No data need to be annotated.

#### **3) Analysis**

---

As no data annotation is needed, no software will be developed.

## **UNIT L-24**

### **Explicit coordination: final report [draft]**

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## **UNIT L-25**

### **Implicit Coordination: preliminary report**

### **25.1. Part 1: Mission and scope**

#### **1) Mission statement**

---

Formation keeping and collision avoidance using implicit communication

#### **2) Motto**

---

ALIIS VIVERE  
(Live for others)

#### **3) Project scope**

---

*What is in scope:*

- Relative pose estimation
- Formation keeping: formalize controller that keeps constant distance from vehicle ahead
  - “Follow the leader demo”
  - Avoiding traffic waves
- Intersection coordination
  - **Detection:** position (bounding box) + pose estimation per bot in FoV.
  - **Tracking:** trajectory (sequence of pose) estimation per bot tracked, parameterized in time.
  - **Prediction:** predict bots behavior through intersection by integrating tracking information and rules of the road (no explicit communication allowed).
  - Implement traffic rules
  - Intention of the other car has to be predicted
  - Priority traffic rule: First come first serve. Crossings possible without explicit communication
  - Designing Fiducial markers (April tags)
- (Designing a special T intersection with only one stop)

*What is out of scope:*

- Adding hardware
- Intersection navigation
- Communication with LEDs / explicit communication

*Stakeholders:*

- System Architect
- Software Architect
- Knowledge Tzarina
- Anti-Instagram
- Controllers
- Navigators
- Explicit Coordinators

## 25.2. Part 2: Definition of the problem

### 1) Problem statement

---

**Mission:** Formation keeping and intersection navigation with Duckiebots just using implicit communication.

**Problem statement:** Detect other Duckiebots, follow the leader in a secure distance through Duckietown and coordinate intersections using implicit communication.

### 2) Assumptions

---

- Duckiebots do not use explicit communication, e.g. LEDs, WLAN ...
- Duckiebots have different appearance (different configuration, LEDs on/off, ...)
- All Duckiebots are autonomous, not remote controlled
- All Duckiebots use the same formation and implicit control algorithm

### 3) Approach

---

Formation:

- Vehicle detection
  - April tags
    - Analyze already existing code for vehicle detection ([Software](#), [Duckumentation](#))
  - CNN-based
    - Image annotation
- Tracking
  - model based
  - learning based
- Prediction
  - model based
  - learning based
- Distance Control

Implicit Coordination

- Data Collection
  - Annotation tool: if only detecting bots, any custom manual approach will work, if not, then we will use thehive.ai API.
  - Types of intersections: 4-way, 3-way, 3-way with 1 stop sign.
  - No pedestrians.
  - Motions: moving vs. static traffic, moving vs. static self.
  - Including “look-around” behavior.
  - Illumination variances: add a predominant light to Duckietown so we can have strong shadows and different lighting conditions.
  - Appearance variances: different bots configurations (e.g: with/without the shell, a duck, etc.)
  - Frames will be extracted from videos at a 3 fps framerate.
- Detection -Instance-level 2D/3D Bounding box + pose estimation.
  - Explore OpenCV built-in detection capabilities.
  - Explore supervised learning methods for detection, including existing models (e.g: YOLO2, SDD, etc).
- Tracking
  - Estimate prior trajectory of each bot based on the instance level detection generated.
  - Explore OpenCV tracking algorithms ([Doc](#))
  - Explore MOT algorithms based on Deep Learning ([Paper](#))
- Prediction
  - 3D bounding box + orientation + velocity + Position prediction (1, 5, 20) + estimating a policy (“behavior”) applying rules of the road.
  - APPROACH: TBD.

### 4) Functionality-resources trade-offs

---

*Functionality provided:*

Formation keeping and intersection coordination

- Detection of other Duckiebots within field-of-view
- Tracking
- Prediction

*Resources required / dependencies / costs:*

- Multiple duckiebots
- Duckietown
- April tags

*Performance measurement:*

Robustness of Formation keeping and Vehicle Detection

- Number of duckiebots in the “Follow the leader” demo
- Duration of the demo (time)
- Transient error of distance and perpendicular offset

Implicit coordination

- **Detection:** bounding-box (intersection over union, precision + recall), uncertainty in pose estimation
- Throughput rate of duckiebots through intersection
- Waiting time at intersection Number of successful crossings at intersection before a collision happens
- Robustness to duckiebot pose, appearance, scale, orientation ambiguity and LED-lightning

### 25.3. Part 3: Preliminary design

#### 1) Modules

---

- Detection of Duckiebot
- Estimation of relative pose
- Controller for following the leader
- Intersection behavior

#### 2) Interfaces

---

Detection of Duckiebot

- Input: April tag
- Output: Position in formation and duckiebot parameters

Estimation of relative pose

- Input: Camera image
- Output: Estimate of Relative pose between bots

Controller

- Input: Estimated pose
- Output: Command to motors

Intersection behavior

- Input: State at stopping line
- Output: Preferences at intersection

### 3) Preliminary plan of deliverables

---

*Specifications:*

(Special T-intersection with only one stop in Duckietown)

*Software modules:*

- Detection / Estimation node
- Tracking node
- Controller node
- Implicit coordination behavior node

*Infrastructure modules:*

None

## 25.4. Part 4: Project planning

TABLE 25.1. SCHEDULE

Date	Task Name	Target Deliverables
17/ 11/ 2017	Kick-Off	Preliminary Design Document
24/ 11/ 2017	Review state of the art, theoretical formalisation	List of what has to be implemented and how and what can be reused from last year. Bench- marking current state
1/ 12/ 2017	Implementation	-
8/ 12/ 2017	Implementation and Testing	Code
15/ 12/ 2017	Evaluation, Opti- mization	Performance measurement
22/ 12/ 2017	Buffer	-
29/ 12/ 2017	Duckumentation	Duckuments
05/ 01/ 2018	End of project	-
	End of project	

## 1) Data collection

---

Annotated images

## 2) Data annotation

---

Duckiebots in bounded boxes under different conditions (Illumination, LEDs on/off, Duckiebot configurations, ...)

*Relevant Duckietown resources to investigate:*

Existing intersection safety rules ([Duckumentation](#))

Existing vehicle detection algorithm ([Code](#), [Duckumentation](#))

*Other relevant resources to investigate:*

Robotics Handbook: Ideas for controller for ‘follow-the-leader’ or driving in traffic on pg. 808

## 3) Risk analysis

---

Likelihood from 1-10, where 10 is very likely and 1 very unlikely.

Impact from 1-10, where 10 is a huge negative impact and 1 not so bad.

TABLE 25.2. RISKS

Event	Likelihood	Impact	Risk response Strategy (avoid, transfer, mitigate, acceptance)	Actions required
False Negative in Vehicle detection can lead to crash	4	9	mitigate	The effect of FPs are less bad than those of FNs. Punish FNs more in an eventual cost function.
False Positive in Vehicle detection makes bot stand still.	4	2	accept	-
Inaccurate estimation of distance between bots	3	7	mitigate	Early testing, big enough safety margin, improve controller
Collision of 2 duckiebots in an intersection	3+4	9	mitigate	Combination of Event 1 and 3

## UNIT L-26

### Implicit Coordination: intermediate report

#### 26.1. Part 1: System interfaces

##### 1) Logical architecture

The first part of the project is to detect and track Duckiebots in general. In addition the motion of the Duckiebot should be predicted in a short time horizon. With this detection it should be able to get information of how many Duckiebots are at an intersection and if the way is free to cross the intersection. We will provide a protocol to trigger the driving through intersections.

The detection is also used to implement a demo in which a number of Duckiebots will follow a leader-Duckiebot. The leader will be controlled with forward navigation in Duckietown. The architecture is divided into the following logical submodules:

## Detection

- Object bounding boxes and class outputted in a topic (in image space - per image no tracking) (2pts inside the image 640x480 + class)
- on your laptop you should see in rqt\_image\_view an image with bounding boxes around robots
- For Follow-the-leader:
  - Easy: with fiducial marker detection
  - Hard: With detection see (i) and (ii)

## Tracking

- Output of the 2D trajectory (x,y) (stretch - pose) of each of the detected objects
- visualization of the trajectories in RVIZ

## Prediction

- Output of a 2D class of trajectories with probabilities for future motion of objects
- In RVIZ a class of future object trajectories weighted by something

## Intersection Coordination

- An agent (Duckiebot) infers when it is safe to go without explicit communication. The agent follows a protocol known to each agent on the road.
- Duckietown Intersection Coordination Protocol (DICP):
  - Inspired by CSMA/CD algorithm
  - Constraint: only one Duckiebot at a time crossing the intersection

## Follow-controller

- Controller gives wheel velocity commands to follow a detected/tracked Duckiebot to keep a constant distance behind the Duckiebot in front.
- For centering the Duckiebot behind the one in front of him we will use the lane controller from the “Controllers”.
- Velocity control falls into the scope of our tasks.

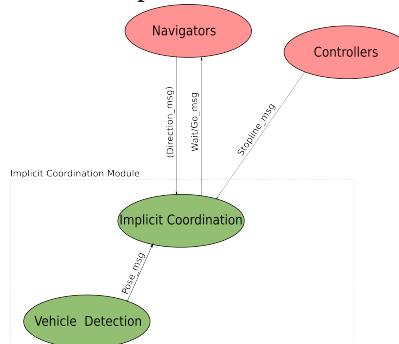


Figure 26.1. The Coordination Module

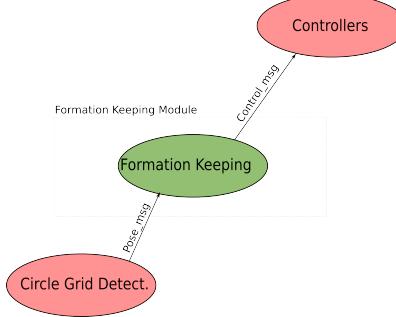


Figure 26.2. The Formation Keeping Module

Easy: Own robot is not moving

Hard : Own robot is moving

### Assumptions about other modules

#### Intersection Coordination:

- Duckiebot knows when it is at an intersection, coordination\_node receives msg when at stopline. Given by Controllers.
- Duckiebot knows how to navigate through an intersection, only needs a wait/go msg from our side (interface with navigators) Formation Keeping:
- We can use the lane controller from “Controllers” to center a Duckiebot behind the Duckiebot in front. We can use the obstacle avoidance for the leader Duckiebot to avoid duckies. Given by Savior.
- The Duckiebot will exit the Formation Keeping mode as soon as the Duckiebot in the front leaves the lane (overtakes or avoids an obstacle) and at intersections.

## 2) Software architecture

---

The software architecture is divided into the following software nodes:

#### Detection:

- One node - input camera image, output detections define a msg type Tracking:
- One node - input detections, output trajectories - define message type
  - For Hard need the odometry of robot

Prediction : later...

#### Intersection Coordination:

- One node - input msg from tracking node, msg stop-line filter - output boolean msg go/wait
  - Input: flag\_at\_stop\_line -> True if Duckiebot is at intersection
  - Input: Tracking\_msg -> Are other Duckiebots at intersection or will arrive at intersection in the prediction horizon
  - Output: flag\_wait\_go -> True if Duckiebot can navigate through intersection

#### Controller (Follow-the-leader/Formation Keeping):

- Easy:
  - One node - input pose of Duckiebot - output car control msg
    - Input: geometry\_msgs/PoseStamped -> Estimated pose from Duckiebot in front to follower
    - Output: Msg for lane controller from “Controllers -> Msg for generating motor commands
    - flag\_follow set true
- Hard:
  - One node - input pose of Duckiebot - output car control msg
    - Input: Tracking/detection msg
    - Output: Msg for lane controller from “Controllers -> Msg for generating motor commands

### Latency introduced by the subscribed modules

Will be added

### Latency introduced by our modules

- We need to do some back of the envelope math for the functionality: turn, look, detect, turn back, go.
- Latency for Follow-the-leader controller: max 15ms
- Latency for boolean msg at intersection: to be benchmarked

## 26.2. Part 2: Demo and evaluation plan

### 1) Demo plan

---

Dream scenario intersection coordination: Duckiebots loop through the map as indicated in [Figure 26.3](#). At the intersections the Duckiebots coordinate implicitly. Demo can run indefinitely without collision. Tunable: density of the traffic (fleet size) and the speed.

Dream scenario follow-the-leader: Leader drives with indefinite navigation through Duckietown and other Duckiebots follow him. The Duckiebots are able to keep up with the leader.

### Required Hardware Components

- Circle grid on Duckiebots
- The tiles to build the map in [Figure 26.3](#)

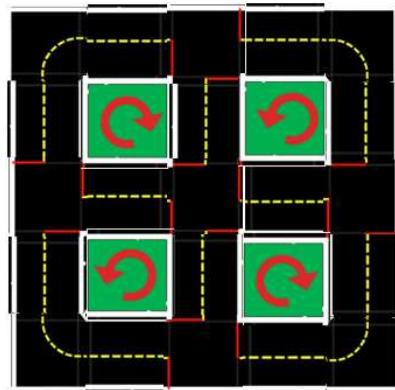


Figure 26.3. The Demo Map

## 2) Plan for formal performance evaluation

TABLE 26.1. PLAN FOR PERFORMANCE EVALUATION

What is evaluated	How	Required	Collected quantities	Performance measure(s)
Coordination implementation performance	Run the demo for intersections	Up to four Duckiebots at an intersection, different configurations have to be analyzed (three or four way intersection)	Time required to clear the intersection	Mean clearing time for each configuration separately and for all combined.
Vehicle detector	Run the detection node	Two to four Duckiebots at an intersection	Number of TP, FP, FN	-Precision -Recall
Follow the leader	Run the demo for follow the leader	More than 4 Duckiebots Duckietown (no special requirements)	Distance between Duckiebots	-Number of Duckiebots in formation -Mean error for desired distance

## 26.3. Part 3: Data collection, annotation, and analysis

### 1) Collection

8000 frames are logged. Sent to the hive.

We don't need extra help in collecting the data from the other teams.

## Method for taking logs

Two robots are taking logs and there are 6-7 other robots around town. Different sizes colors. Shells/no shells.

### 2) Analysis

#### UNIT L-27

### Implicit Coordination: final report [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

#### UNIT L-28

### Single SLAM Project

#### 28.1. Part 1: Mission and scope

##### 1) Mission statement

Enable a bot to map duckietown and know its position in the town

##### 2) Project scope

*What is in scope:*

- Map visualization
- SLAM super-class
  - EKF SLAM
  - Rao-Blackwellized

*What is out of scope:*

- SLAM with lane segments
- April tag detector
- Controlling the bot / finding path to do SLAM with

*Stakeholders:*

- April Tags
- Control

#### 28.2. Part 2: Definition of the problem

##### 1) Problem statement

Every Duckiebot is different in configuration.

Mission = we need to make control robust to different configuration

Problem statement = we need to identify kinematic model to make control robust enough

## 2) Assumptions

---

- Robot will move only in horizontal plane
- No lateral slipping of robot
- The body fixed longitudinal velocity and angular velocity will be provided as well as the timestamp of each measurement
- April tags don't move
- Gaussian errors for EKF slam (relaxed for Rao-Blackwellized)

## 3) Functionality provided

---

- A ROS node that subscribes to image feed and controls
- Publishes estimate of robot pose and uncertainty relative to starting point
- Publishes estimates of april tag poses and associated uncertainties relative to starting point

## 4) Resources required / dependencies / costs

---

- Requires camera calibration
- Process model
- April tags
- A town

## 5) Performance measurement

---

- We will qualitatively evaluate the generated map (Visualization) against town
- For finer tuning we may consider pairwise distance between april tags and compare our estimate to the actual town

## 28.3. Part 3: Preliminary design

### 1) Modules and interfaces

---

- State data-structure holding poses of robot/features and associated uncertainties
- `Visualize :: State -> Image of map`
- `ProcessImage :: Image -> Poses (in the map frame)`
  - Get relative feature pose given robot pose
  - Add new feature to state
- `Predict :: velocity -> new robot pose`
- `Update :: list of poses, old distribution -> new distribution`

### 2) Subclasses and specific methods

---

- Kalman Filter
  - 3 sigma circles around feature/robot mean
- Particle Filter

- Distribution of robot/features heatmap

### 3) Specifications

---

Duckiebots with different hardware configurations for testing

### 4) Software modules

---

- Parameter estimation:
  - runs calibration protocol
- Velocity translation: (Node)
  - get velocity as input and translate it to voltage as output

### 5) Infrastructure modules

---

None

## 28.4. Part 4: Project planning

What data do you need to collect?

### 1) Data annotation

---

Performances of the current implementation

*Relevant Duckietown resources to investigate:*

- Current State Estimation
- Calibration files

*Other relevant resources to investigate:*

[Probabilistic Robotics](#) Chapter 3, 10

### 2) Risk analysis

---

What could go wrong?

- We may not complete the project in the allotted time
- Uncertainty in map may be so high that it is useless

Mitigation strategy:

- Focus on EKF SLAM early

**UNIT L-29**

## Fleet Planning: Preliminary Report

## 29.1. Part 1: Mission and scope

## 1) Mission statement

---

“Create the mobility-on-demand service for Duckietown.”

## 2) Motto

---

VICTORIA CONCORDIA CRESCIT

(Victory through harmony)

## 3) Project scope

---

*What is in scope:*

- Send mobility commands to each duckiebot
- Receive each duckiebot's location
- Calculate a set of mobility commands for all duckiebots
- Implement status LED patterns (i.e. like a taxi)
- (commands to arrange duckiebots in a pre-specified pattern)
- Implement customer request's for pickup and destination at a desired location

*What is out of scope:*

- Hardware modifications
- Fleet wireless communication
- Improvements to existing graphical representation of map and locations
- Fleet localization improvements (e.g. accuracy..)

*Stakeholders:*

- Fleet-comms: for querying their API (contact: )
- Devel-coordination and multi-slam:
- The Architects (smart city): accurate map of city, sufficiently big map to accommodate ~25 duckiebots at once

## 29.2. Part 2: Definition of the problem

### 1) Problem statement

---

We need to combine parts of many different stakeholders to achieve planning for a fleet of duckiebots.

### 2) Assumptions

---

- Sufficiently large duckietown to accommodate all duckiebots
- Collision avoidance and navigation works well to allow duckiebots to reach target destination
- Duckiebots can never park (i.e. stop still anywhere, unless waiting for other duckiebot at intersections etc.).

### 3) Approach

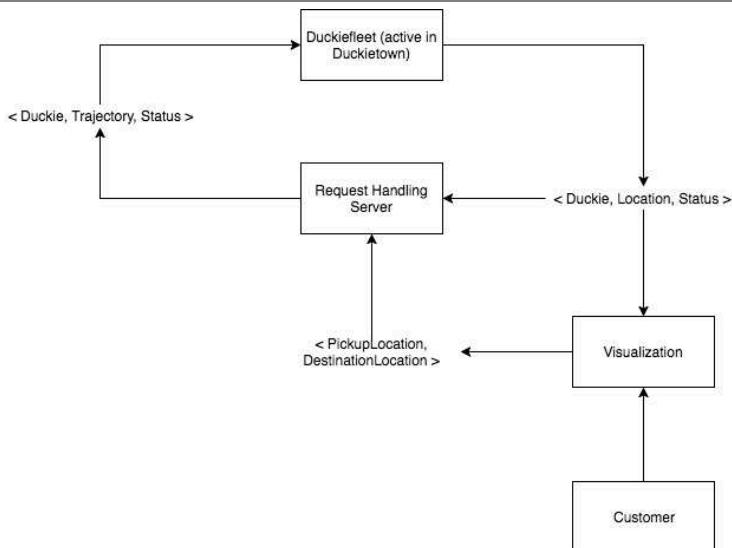


Figure 29.1. Fleet-level planning diagram

Necessary steps:

- [See Part 4: Project planning](#)

### 4) Functionality-resources trade-offs

Functionality includes:

- Visualization of N duckiebots
- Pick up and drop-off on request
- Functional standby distribution of duckiebots waiting for a pickup/ drop-off request
- Ability to arrange duckiebots in formations related to christmas videos
- Taxi status lamps

Metrics:

- Minimize time for each service request to be completed

### 5) Functionality provided

- Average handling time for each request
- Get the location for all duckiebots via the fleet-comms team and agree on the interface
- Get topological map of duckietown for planning

### 6) Resources required / dependencies / costs

- Calculate time taken to complete request
- Number of requests served per time

## 7) Performance measurement

---

- Calculate time taken to complete request
- Number of requests served per unit of time

## 29.3. Part 3: Preliminary design

### 1) Modules

---

See [Figure 29.1](#).

### 2) Interfaces

---

Duckiefleet - request handling server:

- List of duckiebots and corresponding locations and statuses - will be sorted with the fleet-wireless-communications team, see Resources required / dependencies / costs

Customer - request handling server:

- Pickup location and desired target location via clicking on map

Request handling server - Duckiefleet:

- List of target locations for each duckiebot such that request is completed
- Each duckiebot displays its status via its LEDs

### 3) Specifications

---

No revision of existing duckietown specification necessary.

### 4) Changes to existing Software

---

Revisit visualization of Duckiebots on map and adapt it for visualization of N Duckiebots

### 5) Software modules

---

- ROS node for the request handling server
- Offers ROS service for customer requests
- (platinum) map on which customer can click to issue request, potentially as a separate ROS node

### 6) Infrastructure modules

---

None

## 29.4. Part 4: Project planning

TABLE 29.1. FLEET-LEVEL PLANNING PROJECT PLAN

Week of	Task	Deliverable
13.11.2017	Project kick-off and planning	Preliminary Design Document
20.11.2017	Look at state of current infrastructure	Running visualization of 1 duckiebot on map as currently implemented
27.11.2017	Visualization of n duckiebots	
04.12.2017	Mission planner, implement m-stochastic queue median policy (or similar, tbd with Claudio)	
11.12.2017	...continued	Run test cases (e.g. send n reference locations to n duckiebots)
18.12.2017	...continued	Run test cases (e.g. send n reference locations to n duckiebots)
25.12.2017	Implement customer request handling	Run test cases to establish reliable customer request handling routine
01.01.2018	Physical visualization of status, ETH formation	Verify that it works

### 1) Data collection

None

### 2) Data annotation

None

*Relevant Duckietown resources to investigate:*

According to meeting notes:

- Click and send for a single duckiebot is (probably) possible -> find corresponding code
- Graphical representation is running -> find corresponding code
- Read current documentation on tile-level localization
- We want to be able to send a go-to-position to a Duckiebot.
- Already implemented. Video: <https://www.dropbox.com/s/93pbcktmwln4fgo/dp6b-draft.mov?dl=0>
- DP6 : link to the report
- Navigation to a point already implemented -> look at the code
- Visualization of 1 Duckiebot on a 2D map
- Look at code from Claudio re m-stochastic queue median policy + [paper](#)

*Other relevant resources to investigate:*

Papers:

“Fundamental performance limits and efficient policies for Transportation-On-Demand systems“ by Marco Pavone, Kyle Treleaven and Emilio Frazzoli [link](#)

### 3) Risk analysis

---

- Dependency on the Fleet-communications project. Closely work together with that team to get notified early about any upcoming problems that could delay the delivery of the needed parts for this project.
- See Part 4: Project planning

## UNIT L-30

# Fleet Planning: Intermediate Report

**TODO:** JT: math formatting

This document describes system that the Fleet-planning team is planning to implement. Part 1 describes the interfaces to other parts of the system. I.e. How it communicates with all of them. In Part 2 a plan for the demo and evaluation is presented. And finally part 3 focuses on data collection, annotation and analysis. As for this project not much annotated data is used, part 3 is rather short.

## 30.1. Part 1: System Interfaces

### 1) Logical Architecture

---

The fleet planning system shall provide a graphical interface for visualization of n Duckiebots in a Duckietown. Duckiebots shall provide their location regularly to a central “planning node” (not running on a Duckiebot). Furthermore, an interface should exist to generate “taxi” commands (i.e. pickup guest at tile k and bring him to tile m). For such a request the system shall react with a command sent to one of the duckies to pick that customer up and transport him.

See Figure 1 for an overview of the logical structure of the fleet planning system.

In detail the complete process consists of the following steps:

1. Enter desired pickup and drop off location in GUI.
2. Planning node (see Figure 1) finds a Duckiebot that is available for a pickup. The matching is based on the availability of Duckiebots and their current locations. The planning node knows the location of each Duckiebot because they broadcast their position on a regular interval.
3. The location of the pickup is sent to the selected Duckiebot and that Duckiebot changes its fleet planning status to picking customer up as soon as it receives the message.
4. Based on the Duckiebot’s location and the received target location, the Duckiebot calculates locally a shortest path. This shortest path is nothing else than a list of directions for all the intersection that will be crossed on the path.

5. When the Duckiebot arrives at an intersection (realized by listening to flag), it publishes the instruction for this intersection.
6. Once the Duckiebot arrives at the location where it picks up the customer, the planning node sends the target location to the Duckiebot. Then steps 3, 4, and 5 are repeated until the Duckiebot arrives at the dropoff location.
7. The fleet planning state of the Duckiebot is set back to rebalancing.

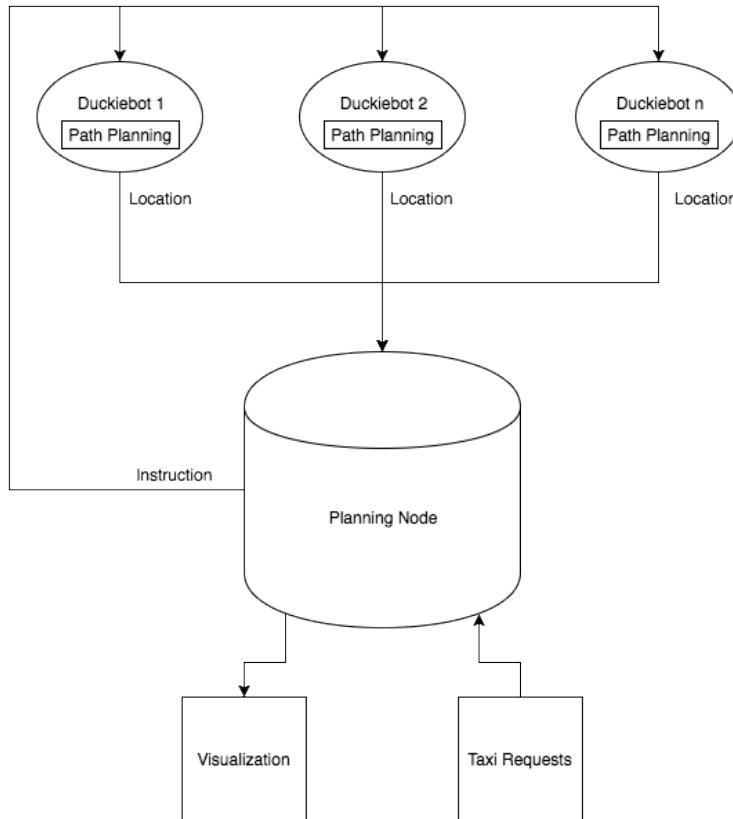


Figure 30.1. Fleet-level planning Logical System Architecture

*Assumptions:*

The communication between Duckiebots and the central planning node relies on the communication team of the distributed estimation project. To exchange messages on a fleet level we need this system to work reliably (i.e. no message loss) and with as little latency as possible (i.e. as little delay as possible between sending and receiving a message). We assume that the distributed estimation team can provide such a system within a reasonable timeframe. In part 2 of this document the interface to the communication layer is described. Based on this interface we can mock the communication and work out the fleet level planning part without a already working communication.

We further assume that the system has a map of Duckietown available. This map can be created by hand or with the system implemented by the distributed estimation team.

As described by the preliminary design document, the localization of the Duckiebot within Duckietown is out of scope. We utilize the existing apriltag based localization from last years Duckietown course. First experiments are very promising and give good results. So we assume that the Duckiebot is able to localize himself within Duckietown if it is given a map of Duckietown. Furthermore, the localization can be enhanced by using the current speed information from the controllers. With that we can get an estimated position between intersections.

General assumptions that collision avoidance, line detection etc. function flawlessly are also made. Furthermore we ignore parking spots and parked Duckiebots and possible parking actions as they are not represented in our map. Lastly we assume that the clocks of the Duckiebots are reasonably in sync.

## 2) Software Architecture

---

New Nodes:

### Fleet planning node [central Laptop]

This node knows the position and status of each Duckiebot in the network. It does the actual planning for the fleet. This consists of matching incoming transportation requests with available Duckiebots in such a way that the overall fleet is used in an optimal way.

Subscribed Topics:

- “Location”: To get the location messages from every Duckiebot.
- “Transportation Requests”: Every transportation request posted on this topic should be handled by the fleet planning node.

Published Topics:

- “Transportation status”: A topic that will get messages whenever something is updated within the fleet planning node. Example messages “”Robot” Picked up customer x at y”, “”Robot” Received transportation order from k to m”. This topic will introduce neglectable latency. As soon as the information is acquired from the sources it will post the message to this topic.
- “Target Location”: here messages are published that contain a robot name and its target location. Based on this the robot will calculate its path to the target location locally. The latency between getting a transportation request and sending out a target location to one of the robots can not be determined offline as it is dependent on the current state of the system. If there is a Duckiebot immediately available, there is no delay. However, it might be that all Duckiebots are busy and therefore no Duckiebot can be assigned to that target location at the moment.
- “Fleet planning active”: Boolean flag indicating that the fleet planning node is active and wants to actively provide instructions at intersections.

### Visualization Node [central Laptop]

This node is responsible of visualizing n Duckiebots on a map.

Subscribed Topics:

- “Location”: The stream of locations that comes in from all the Duckiebots.
- “Target Location”: Combining this knowledge with the messages from the “Location” topic, the calculated path of each Duckiebot can be visualized. The user can decide if the paths shall be shown. (The shortest path is also calculated locally on the Duckiebot, in order to reduce communication overhead).

Published Topics:

- “Visualization”: The rendered visualization as an image

### Taxi command execution node [local]

This node will run on the Duckiebot and listen to commands from the central fleet planning node. Whenever it receives a command it starts the appropriate actions.

A Duckiebot can be in one of three fleet-planning states: - Cruising/Rebalancing - Picking up Customer - Transporting Customer

A Duckiebot receives target locations from the central fleet planning node. It then calculates the shortest route to this location. For this the existing A\*-path planning node is used. Given the Duckiebot’s current location and the target location, the path planning node can calculate instructions for how to get there. These instructions are then passed (on a per intersection basis) onto lower level navigation nodes (i.e. handled by navigator team).

Furthermore this node also handles the back right LED which we are allowed to indicate the taxi status of the Duckiebot. Its status is communicated by the central fleet planning node. Additionally, when a customer is picked up a pattern is played on all the LEDs with very low intensity.

Subscribed Topics:

- “Location”: The location of the Duckiebot on the map
- “Stopline” and “April tag”: Whenever we are at a stop line with the according april tag we know that we are in front of an intersection. As a reaction to being at a stop line this node will publish the instruction that tells the Duckiebot what to do at this intersection. This instruction is based on the path it had calculated to reach its target location.
- “Taxi status”: indicates if the taxi is driving to a customer, is carrying a customer or is in idle (cruising/rebalancing) mode.

Published Topics:

- “Crossing\_instructions”: Whenever the Duckiebot is at an intersection the node will publish on this topic what direction should be taken. As the path the Duckiebot takes was calculated previously there is no latency introduced. I.e. as soon as the Duckiebot gets the message that it stopped at an intersection it will publish the message with the instruction for this intersection to this topic. The message consists of a single integer. To have backwards compatibility with the current system this is one of the following values:
  - 0: left turn
  - 1: straight
  - 2: right turn

- 3: random

#### *Modified Nodes:*

Localization is based on last year's "localization" package. For this purpose the map data was updated to match this year's Duckietown.

The fleet planning package is also based on last year's "navigation" package. It provides software to handle the path planning and a GUI that allows to select start and target nodes and displays the calculated path for a single Duckiebot. Multiple Duckiebot handling does not exist. The Duckiebot is then made to follow these commands. By now, we were not able to reproduce this feature in a stable manner. Also, in this package no location information is taken into consideration, the path planning and execution is executed in an open-loop manner. This shall be closed loop this year.

## 30.2. Part 2: Demo and evaluation plan

### 1) Demo plan

---

The demo from last year consisted of a single Duckiebot. It was possible to click on the node in the graph where the Duckiebot currently is and a target now where the Duckiebot should go. A path planning node then calculated a shortest path to the target location and the Duckiebot drives to that location.

For this years demo we envision a system that builds on top of that. A map is presented to the user that contains the current locations of all Duckiebots. The user can generate a transportation request by using a GUI. The system then assigns one of the Duckiebots to that task. That Duckiebots drives to that location, picks the customer up, drives to the target location and drops the customer again. The pickup and dropoff action are visible in the visualization. Further, the pickup and drop off can be visualized using the LEDs by showing a fancy pattern. There is no physical interaction planned. The system will be able to handle multiple of such requests at the same time. Also, the system shall be robust in the face of dying Duckiebots (may they rest in peace), thus a customer shall be assigned to a new Duckiebot if the original one is lost on it's way. A Duckiebot counts as out of service if he does not publish a new location within a certain time window. No customers waiting forever. Unfortunately we cannot guarantee safety for a customer that is on a lost duckie-taxi.

Setting up this demo is as quick as starting all Duckiebots with the correct mode of operation and putting them on the map.

### Required Hardware

- Duckietown (At least the same size as ML J44, depending on number of Duckiebots on duty)
- Several Duckiebots
- One laptop to act as the central fleet planning server (provided by fleet-planning group member)

## 2) Formal performance evaluation

---

This project will introduce metrics that will be used to evaluate the performance of the fleet planning, providing a baseline for future groups working on further optimization. The metrics, introduced below, will be applied to a test-setup, which is described in the “proposed performance evaluation” section. The aim of the setup is to test the system’s reliability on the one hand (first scenario) and the capability to handle multiple requests at very high frequency and at the load limit, that is, at full capacity (number of requests at a given time == number of Duckiebots) on the other hand.

**Customer requests fulfilled per minute for given number of Duckiebots (ie. throughput).** This metric measures how many customer request are handled by the server and fulfilled by all Duckiebots combined. This would allow for the evaluation of different path planning algorithms and testing how well the rebalancing works.

**Mean distance of closest Duckiebot to the origin of a request for a set of requests and a given number of Duckiebots.** In the optimal case the Duckiebots will be distributed as homogeneously as possible in the Duckietown, minimizing the expected distance to each customer. This metric will enable the evaluation of how well a given implementation does this.

**[Stretch goal] Lost customers** In certain circumstances Duckiebots will fail (e.g. battery dies, Duckiebots gets stuck, etc.) and if this occurs while a Duckiebot is on the way to pick up a customer the fleet planning system should be able to send another available Duckiebot to fulfill the request instead. For the purpose of evaluation, several Duckiebots can be removed from the Duckietown at random and the system should still be able to fulfill all open requests. (A lost duckiebot can be detected using a timeout on the localization.

*Proposed performance evaluation:*

In the Duckietown in ML-J44, that is a 5x6 Duckietown, 4 Duckiebots will be placed at the intersections in the following locations:

- (0, 3)
- (2, 3)
- (4, 3)
- (2, 5)

See picture below for initial locations.

The Duckiebots should operate at a predefined speed which is consistent across all tests for comparability. There should be two scenarios running 10 minutes each with 10 customer requests per scenario.

Scenario 1: 10 requests evenly spaced out across 10 minutes. Scenario 2: 4 requests within the first minute, then 6 requests at 4,5,6,7,8,9 minutes respectively.

For each scenario, the method is the following:

1. Place Duckiebots in the Duckietown and have them move around in an idle state (i.e moving around randomly).
2. Send a command to move all Duckiebots to predetermined locations to ensure repeatability of the performance evaluation. The locations are
  - o (0,1)
  - o (2,5)
  - o (4,1)
  - o (1,5)
3. Start the evaluation by sending the series of customer requests during a 10 minute interval.

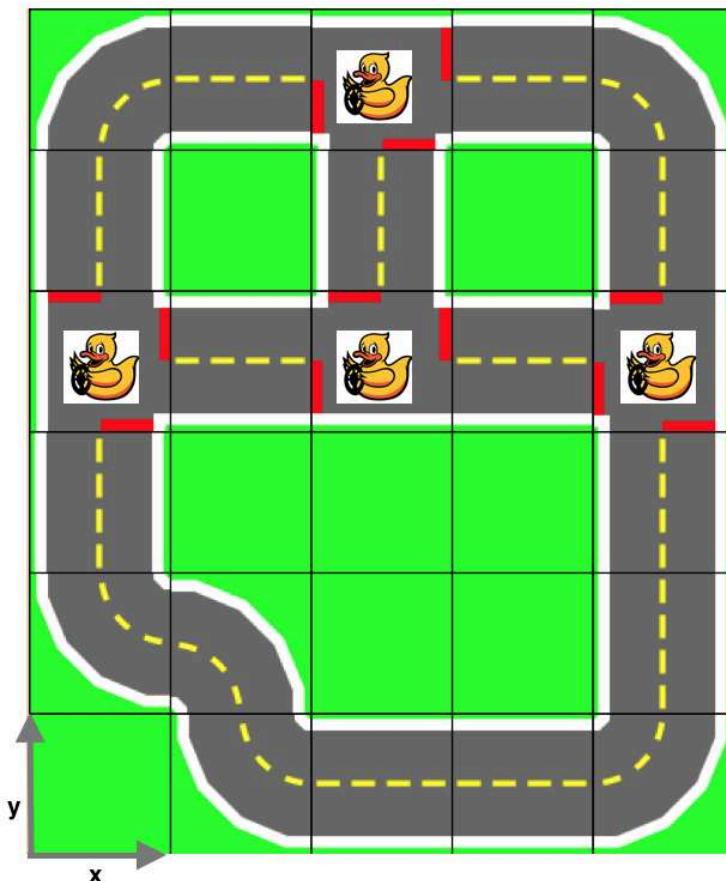


Figure 30.2. Initial position

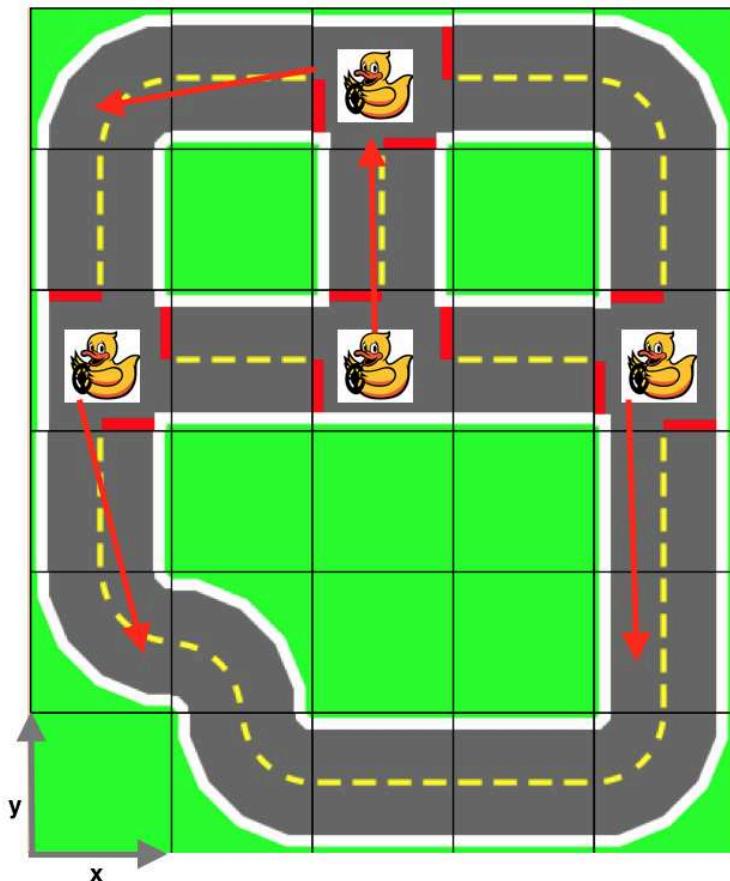


Figure 30.3. Final position

### 30.3. Part 3: Data collection, annotation, and analysis

#### 1) Collection

We need data to do the formal performance evaluation. As the central fleet planning node has all the information about the Duckiebots (i.e. location at every point in time and taxi status) it is enough to log the information flowing through the topics to and from the central fleet planning node.

These logs will be used for the formal performance evaluation as described in Part 2 of this document.

#### 2) Annotation

None needed.

#### 3) Analysis

Analysis is done by hand on the acquired logs. As a stretch goal, a set of functions is made available that automates the process such that future teams working on improving this system can use the same evaluation strategy.

## UNIT L-31

# Fleet Planning: Final Report

**TODO:** JT: fix image sizes

### 31.1. The final result

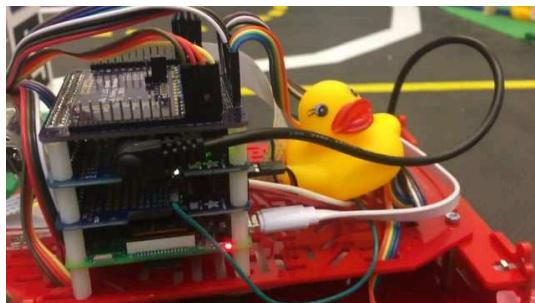


Figure 31.1. The Fleet Planning Demo Video

See the [operation manual](#) to reproduce these results.

### 31.2. Mission and scope

#### 1) Motivation

---

As cities such as Duckietown grow, their inhabitants can no longer walk from each point of the city to the next; especially after a long night out. With the tremendous growth of Duckietown, the need for a mobility-on-demand (MOD) service became unbearable and was addressed in this project. Duckietown being a city in which autonomous Duckiebots roam the streets, it makes sense for them to provide the much-needed MOD service. If Duckiebots share their positions as well as their next destinations, with a central dispatcher node they can work together to deliver an efficient service. The development of a multi-duckiebot MOD service is thus reliant on many other Duckietown components such as localization, mapping, navigation, collision-avoidance and communication.

#### 2) Existing solution

---

The existing codebase was set up to support localization, navigation and visualization for a single Duckiebot in a fixed, hard-coded Duckietown, represented as a tile-map image; see the [specification](#) for the available tile-types.

By combining a set of rotated tiles, a map of any given Duckietown can be formed. In the existing codebase, such a tile-map was supplied as an [image](#) and a CSV file of tile-types and orientations; however neither code for composition of a tile-map nor documentation as to how this was generated in the first place were supplied.

The entire code for localization, navigation and visualization ran on the Duckiebot and was linked to the other modules using a Finite State Machine (FSM).

The codebase can be found [here](#).

#### *Localization:*

Localization was performed by placing AprilTags at each intersection and having the Duckiebot identify each unique AprilTag through image analysis. The AprilTags are defined in the duckybook [signage section \(master\)](#). A Duckiebot could thus compute at which (x,y) coordinate of the map it was and estimate its rotation. The position on the topographic map was not mapped to the corresponding topological graph representation which is required for path planning.

#### *Navigation:*

A graph was computed from the tile-map and given a start and an end node, A\* search was performed on that graph to provide the list of nodes along the shortest path. This list of nodes was then converted into a set of instructions the Duckiebot could follow. These instructions were of the form “straight”, “left”, “left”, ... They were executed in open loop manner, i.e. once the entire path was calculated, the Duckiebot was sent off to execute it without any intermediate checks whether the Duckiebot was doing this correctly. Thus any deviation of the Duckiebot from the path, or noisy FSM-transitions led to failure of the system. The python package graphviz was used to compute the graph and generate a visualization, i.e. an image of the graph.

#### *Visualization/GUI:*

The GUI consisted of a list, from which the user could select the desired start- and destination nodes for one single Duckiebot as well as an overlay of the tile-map and the graph image. The location was indicated by highlighting the corresponding node in a different color, and the path by highlighting the graph edges. No localization was integrated in the demo, and thus the start node had to be set manually and the subsequent execution of the path was completely open loop.

When the user hit the start button, a request was sent to the navigation ROS node, upon which the start and end graph nodes were highlighted and the computed path was indicated on the image.

### 3) Opportunity

---

The existing solution could benefit from scalability, adaptability and localization. It has been set-up, and often hard-coded, to work only with a single Duckiebot in a completely open loop manner. Without human interaction the Duckiebot would not move.

For fleet-level behaviour and our MOD service, the solution needed to be scaled

up to work with n Duckiebots. Also, in order to being able to implement sophisticated fleet planning algorithms, an API handling and tracking the statuses of the Duckiebots and customer requests appeared highly desirable.

Building this infrastructure was a priority to us, since a good software foundation will make future endeavours in the field more productive and will allow to focus on more complex fleet planning concepts. Nevertheless, inspiration for intelligent fleet-level behavior and automatic rebalancing for more efficient request handling were drawn from [1].

Critical components such as the request-handling were desired to move away from running on a single duckiebot. The previous solution also had no stack and could thus handle only one request at a time. A new request simply overwrote its predecessor, which led to incomplete jobs. These were unacceptable limitations to a true MOD service and could be overcome through the implementation of a central node dealing with coordination, called the taxi central node.

In coordination with the fleet-communications team, a method of sharing information and sending commands to multiple Duckiebots was introduced. The GUI was also changed so as not to run on the Duckiebot, but rather directly on the customer's machine to allow multiple customers to utilize the service at the same time.

Furthermore, the existing solution had been set up to work with one specific Duckietown layout only. Since a MOD service should be capable of running in every Duckietown, this issue was also addressed. The tile-map and graph are now automatically generated off of a CSV based description, allowing for easy adaptations to an existing Duckietown and the application of our MOD service in new Duckietowns.

The GUI was also rewritten to scale well with varying Duckietowns and number of Duckiebots and now allows the MOD customer to issue requests more easily. The GUI workflow was also improved with respect to intuitive usability and faster issuing of customer requests. It also has the capability to show all available Duckiebots and their locations. The improved solution is also more flexible with regards to localization. Once this is properly merged with the localization teams' efforts, the resolution of localization will be dramatically improved.

To verify our work, a test environment and virtual Duckiebots were introduced. The previous solution had no means of demonstrating functionality virtually; i.e. without a physical demo.

### 31.3. Definition of the problem

#### 1) Problem Statement

---

Integrate expectations of different stakeholders to achieve planning for a fleet of Duckiebots. This can be broken down into the following tasks:

- Receive location information and status from n Duckiebots

- Visualize n Duckiebots on the current Duckietown map
- Receive and process pick-up and drop-off requests which are issued using a GUI
- Assign customer requests and rebalancing targets to Duckiebots
- Standby distribution (“rebalancing”) of Duckiebots waiting for customer requests
- Send target location (customer requests or rebalancing targets) to n Duckiebots
- Indication of each Duckiebot’s status using their LEDs. A Duckiebot’s status indicates whether it is on its way to a customer, currently transporting a customer or idle.
- Local path planning and execution on Duckiebots

## 2) Assumptions

---

Per the preliminary and intermediate design documents, we assume that the Duckietown is large enough to accommodate all Duckiebots and that collision avoidance, line-detection etc. function reasonably well. We further assume that the system has a CSV map representation of Duckietown conforming to the conventions defined [here](#). This map can be created by hand or with the system implemented by the distributed estimation team. The communication between Duckiebots and the central planning node relies on the communication team of the distributed estimation project. To exchange messages on a fleet level we need this system to work reliably (i.e. without message loss) and with sufficiently small latencies, i.e. less than roughly a second.

## 3) Contracts

---

*Distributed estimation and communication team::*

The fleet communication team provides a means of transporting arbitrary data as a byte array from one Duckiebot to another Duckiebot or to a computer. Communication channels are set up via a configuration file and messages are sent/received by publishing/listening to messages on a ROS topic. This was integral to the functioning of our system, as we required multiple ROS master nodes running on each Duckiebot locally, nonetheless sharing their information with a separate ROS master on a central computer.

*The Architects::*

Initially the idea was for the Architects to design a Duckietown sufficiently large to accommodate a large number of Duckiebots. Closer to the demo day, it emerged that several smaller Duckietowns would be used so this was no longer needed.

## 4) Performance Evaluation and Metrics

---

The performance of our project was primarily evaluated qualitatively, since the focus of the project was on providing a working framework to easily implement further fleet planning strategies. The results are discussed in 3.5.

### 31.4. Contribution / Added functionality

The work on the 2017 fleet planning project was distributed in the following manner: 80% software infrastructure, 10% algorithms and 10% package integration of other teams. The final software can be divided into five parts:

## 1) Path planning and execution

---

Main component can be found [here](#).

This node listens to location updates from the April Tags localization package and target destinations from the taxi central node.

The (x,y) location information is then mapped to the topological graph representation. Using the rotation of the Duckiebot and by finding the red line with the minimum distance to the Duckiebot, the location of the Duckiebot is set to the node that best explains this configuration.

Once the node has received both location and a mission target, it executes A\* path planning and publishes the next intersection instruction, i.e. “left”, “right”, etc., to be received by the intersection navigation package (implemented by the 2016 team). The path is recomputed at each intersection such that deviations from the original plan do not lead to failure of the entire system; this guarantees a certain robustness to errors of other software components. The calculated path and current location is then reported to the central planning node, called taxi central. This node runs locally on each duckiebot

## 2) Fleet planning aka taxi central node

---

Core component can be found [here](#).

Duckiebot and Customer logic is defined [here](#).

- Runs on a central laptop, handles all the fleet planning logic
- Maintains a set of active Duckiebots.
- Receives location updates from each Duckiebot and stores them. If a Duckiebot does not update its location within a certain time window, it is considered dead and removed from the map. Customers onboard are re-assigned to a new Duckiebot, with their pick-up location corresponding to the last known location.
- Receives customer requests from GUI, assigns them to a nearby Duckiebot based on FIFO breadth first search. For details, see below.
- Tracks execution status of customer requests, and thus tracks whether customer start or target location has been reached and stores the timestamps for each state transition and each request. This allows evaluation of fleet planning metrics such as time-to-customer or execution times of the whole request.
- Duckiebots that are not busy are assigned rebalancing goals. These are currently random locations on the map. This appeared to be a reasonable probabilistic approximation to an optimal strategy, assuming a large number of Duckiebots on a uniform map which will then spread rather homogeneously. The software was designed in a way that allows easy extension of the current approach. Also, it is possible to switch between different approaches by setting a single enum variable which can be very useful for testing and comparing rebalancing strategies.

**FIFO breadth first search:** The taxi central stores a list of pending (i.e. not yet assigned) customer requests and idle Duckiebots. Our algorithm then selects the closest Duckiebot to each customer in First-In-First-Out manner:

For each customer in pending\_customer\_requests:

- Find closest Duckiebot via breadth-first search on the map graph
- Assign customer to Duckiebot
- Remove Duckiebot from list of idle Duckiebots

Although this approach is not fully optimal, it is a reasonable approximation for the 2017 Duckietown setup with a low frequency of new customer requests and a small number of operating Duckiebots. Our software is designed in a way that makes it easy to add more sophisticated approaches. This could include strategies that take into account expected distributions of customers and send Duckiebots to anticipated hotspots ahead of time (e.g. dealing with rush hour customer spikes).

### 3) GUI

---

The code can be found [here](#).

Since the existing GUI was running directly on the Duckiebot and was laid out for a single Duckiebot system, it had to be completely rewritten. To design the front end, the QtGui module was utilized; the GUI itself runs as an rqt module.

To keep the GUI scalable and extensible along with the rest of our solution, it is able to run on multiple devices at the same time, as long as each device can communicate with the ROS master that the taxi central node is running on. The GUI communicates with other modules through ROS messages and topic listeners/subscribers and runs largely independently of all other components of the fleet planning module.

The source code is located in this following [folder](#).

In this section, the GUI components and their interactions with the other modules are described. The overall layout follows design principles outlined in Galitz' "The essential guide to user interface design: an introduction to GUI design principles and techniques" [2].

Please note that components (2) through (5) are re-positioned depending on the Duckietown map's size.

[GUI without customer.](#)



Figure 31.2. Map of Duckietown in GUI showing Duckiebot \_Harpy's\_ current location.

[GUI with assigned customer.](#)





Figure 31.3. Map with icons for a customer at node 7. Duckiebot “Harpy”’s target location is also at node 7 to pickup the customer.

[GUI with assigned customer](#).





Figure 31.4. Harpy travelling with the customer to the target location.

*Duckietown Map:*

- A map of the current Duckietown as an image, received from the drawing node
- Displays a selected Duckiebot's location
- Displays the start and target location of the user's last issued request once the start button (4) has been hit
- Displays the calculated route between start and target locations
- The user can intuitively set start and target location by simply clicking on the map
  - The first click sets the start location
  - The following click sets the target location
  - Both can be cleared by clicking the clear button (5)

*Display of start and target location:*

Serves to provide the user with feedback on the state that the GUI is in. Also provides a way to check correctness of the start/target location before issuing a request

*List of active Duckiebots:*

The selected Duckiebot's location is displayed on the map (1) Used for testing and debugging during development List received as ROS message from the taxi central node

*Start button:*

Triggers a customer request to the taxi central node Only triggered if start and target location are set

*Clear button:*

Clears the start and target locations, which are then removed from the map (1) as well as the numerical display (2)

---

#### 4) Map drawing

Code can be found [here](#).

The map drawing node deals with drawing the Duckietown map according to the specifications in the csv file, overlaying the graph on top of the map and drawing the active Duckiebots at the correct locations. As localization only occurs at intersections where Apriltags are located, Duckiebots are only ever drawn at intersections. The Duckiebot is identified by its name, displayed below the Duckiebot icon. When a customer request is assigned to a Duckiebot, a customer icon is drawn at the specified location and an icon is displayed at the target location as well. Once the customer is picked up, his or her icon is drawn alongside the Duckiebot acting as a taxi. See screenshots above for the different states.

---

#### 5) Messaging / Serialization

As described in a previous section, the existing system runs completely on the Duckiebot, including the user GUI. To make the system scalable we needed to have communication between multiple Duckiebots as well as a central planning

node. This change in the architecture requires a communication system for reliable communication. We acquired this functionality by setting up a contract with the fleet communication team. See [here](#).

The outcome was a system which consists of one ROS node on each participant in the network. Via a configuration file you can define which node communicates with which other node. The interface the fleet communication team provides accepts a byte array and transfers this byte array to the endpoint of the communication channel. For a more detailed explanation of how this is transferred we refer readers to their final report

To send data such as target locations and localization results a way to serialize this data to a byte array was needed. A general framework was setup to serialize the basic data types using python's pickle [3] module. Based on this we implemented serializer and deserializer classes specifically for the messages we needed to send over the network.

## 6) Virtual Duckiebot / Simulation

---

Only at the very end of the project all projects we depend on reached a state where we could integrate them all to have a functional system. Therefore we needed a way to test the system, especially the central dispatcher node, without relying on physical Duckiebots. We solved this by implementing a virtual Duckiebot ROS node that acts as if it were a real Duckiebot.

Duckiebots mainly perform two actions, they report their location to the central dispatcher node and they receive commands (customer pickup, target location, ...). The virtual Duckiebot node mimics this behaviour by regularly sending a message with the current position. Additionally it prints all the information it receives and sends to the console for easier debugging. To the central dispatcher node it looks as if it were interacting with a real Duckiebot.

The virtual Duckiebot node can be run in one of two ways:

*Manual mode::*

The virtual duckiebot is started with an initial location. The user uses a ROS service call provided by the virtual Duckiebot to tell it to send a location update message. The user can specify the location it should send. This mode gives power to the user to test specific situations.

*Autonomous mode::*

In this mode, the user only adds a Duckiebot at a desired node. The virtual Duckiebot node then takes care of all the things a real Duckiebot would do: it receives target locations from the taxi central, calculates the path it should take, notifies the taxi central of the calculated path, and then every few seconds it publishes a location update, as if it were really following the calculated path. The user can add as many Duckiebots as desired and can remove them as well. This way, the whole fleet planning software can be tested and simulated from a single laptop, without the need of a Duckietown or a Duckiebot.

### 31.5. Formal performance evaluation / Results

As mentioned previously, the largest portion of the work that needed to be done involved implementing an operational infrastructure that supports actual fleet planning functionality. In summary, this included:

- Generating a Duckietown map from a CSV description of the map Tracking n Duckiebots on the map, and displaying their locations, their targets and their paths
- Creating customer requests through the GUI by clicking on the image of the map directly, thus removing the need to use tedious drop-down menus
- Tracking customer requests and their assignments through completion (useful classes for easy debugging and clean coding)
- Visualization of the taxi status directly on the Duckiebots via LEDs
- A virtual Duckiebot node, that simulates a real Duckiebot in order to test the fleet planning software.

These functionalities can primarily be evaluated in a binary manner, observing whether these components work or not. Furthermore, the overall performance can be analyzed in a qualitative manner. The components described above all work in the set up used for the 2017 ETH demo day. Our fleet planning system is robust to malfunctions in the execution of the planned path (i.e. the system can re-compute paths and is tolerant towards situations where Duckiebots deviate from the optimal trajectory). Furthermore, the loss of a Duckiebot during the transport of a customer is covered by assigning a new, nearby Duckiebot to pick up the customer from where he or she was last seen.

In comparison to the state-of-the-art, where no fleet planning was possible and visualization of only a single Duckiebot on a map was implemented, a large number of new features was added and the infrastructure put in place to develop more advanced re-balancing algorithms.

Our system is relatively high-level in the sense that it requires many other Duckiebot components to work smoothly for successful operation. For example, we need the Duckiebots to reliably find the correct Apriltag at an intersection and the turns at an intersection to work flawlessly. The principal components our project requires are lane following, intersection navigation, collision avoidance, localization and fleet communication. While our system simply ignores malfunctioning Duckiebots and drops them from the roster of active Duckiebots, it is nonetheless important to ensure most Duckiebots work as expected. On a map the size of the 2017 demo day collisions are inevitable if Duckiebots do not stay in their lane and relatively quickly manual intervention using a joystick becomes necessary. Of course, during manual operation of a Duckiebot the fleet planning system cannot meaningfully assign customers to Duckiebots. Thus, if manual control becomes necessary too often, fleet planning cannot do its job.

At the time of the demo day most other projects of this year that we relied on were not yet ready so we resorted to using last year's implementations for some components, notably (open-loop) intersection control, lane following as well as last year's FSM.

What this meant is that very often the system did not function smoothly and manual intervention was often necessary when Duckiebots missed a turn or lane following abruptly stopped working. Coupled with mercurial joysticks this made testing of new features challenging at times. Initially, we had expected to spend far less time on integrating previous features and get them all running in parallel. Instead, we had thought we would need more time advancing the capabilities of the fleet planning package itself. In the end, a basic Duckiebot that could navigate and communicate with reasonable reliability was paramount to doing any development on our package.

Quantitative evaluation of the sort initially planned and described in the PDD did not lead to any sensible results and insights. The metrics suggested were the ‘customer requests fulfilled per minute for given number of Duckiebots’ and the ‘mean distance of closest Duckiebot to the origin of a request for a set of requests and a given number of Duckiebots’. Both require a large map, a large number of Duckiebots and, most importantly, a certain amount of reproducibility in the results. As Duckiebots frequently veered off the side of the road or collided with sign posts, the same experiment could not be repeated in a meaningful way and the proposed metrics did not make a lot of sense. Nonetheless, once this year’s projects have all been merged and a greater overall stability in the system is achieved these metrics would provide a useful method of evaluation.

### 31.6. Future avenues of development

As seen in the previous sections we were able to provide a framework for a fleet level planning system within Duckietown which can serve as a basis for interesting research questions. However, it goes without saying that the current state of the system has some room for improvement. This section lists possible extensions and improvements.

#### 1) Integration with other improvements from 2017

---

As all the teams were working on their projects in the same time frame with unclear finishing dates we made the decision in mid-January to use lane following and intersection maneuvering from the duckietown class of 2016. Therefore the performance of the system is not as good as it could be with the improved implementation of these two components. Integrating these two improved components into the system to replace the old ones would be a low effort, high gain development.

#### 2) Distributed fleet planning

---

The fleet planning system currently requires the taxi central node to run on a computer in the same network as all the Duckiebots. This is a single point of failure. If either communication between the Duckiebots and that computer breaks down or the computer itself fails the whole system fails. This is a very undesirable property. The fleet communication system not only supports communication with a central node but also point to point communication between the Duckiebots. Therefore it is possible to implement the system in a decentralized way. One possible imple-

mentation strategy:

- New customer requests are broadcasted in the network using a flooding algorithm
- Available Duckiebots close to the customer propose that they pick him up
- The duckiebots that broadcasted a proposal reach consensus using an algorithm such as Cheap Paxos [4]
- The decision is again flooded in the network such that every Duckiebot can accordingly update its own knowledge of the system.

Under the assumption of a connected network (i.e. no partitions) such a system is able to achieve the same performance as a system with a centralized node that coordinates all the Duckiebots. However, the implementation of such algorithms is more demanding because of the increase of complexity in the system which makes it harder to debug.

### 3) Switch to Mesh Network Communication

---

The system as is requires a router for passing the messages in the network around. This can be a standard wifi router or a mobile phone used as a hotspot. This is additional hardware that is required to run the system and needs to be setup the right way. I.e. further points of a possible error while running the system. The library from the fleet communication team which we already use for communication also supports a mesh network configuration. In this configuration each Duckiebot uses its wifi stick to create the same Wifi network. The duckiebots can then communicate using this network. Therefore, no additional hardware is needed. The change to enable this kind of network communication is rather small. However, it has not yet been tested. So further development could include the inclusion and testing of the mesh network configuration.

### 4) Parking of unused Duckiebots

---

In a real world scenario of an autonomous taxi system, the demand for vehicles changes over the course of a day, i.e. there are peaks around the morning commute time as well as the evening. One kind of optimality is to only have as many vehicles on the road as needed based on the current demand. Obviously this should be combined with a prediction of the future demand to minimize waiting time for the customer. (The interested reader is referred to [5] and [6] for a more thorough analysis of fleet size and rebalancing strategies).

The current implementation of the system forces all Duckiebots to continuously drive around the city. If they are not fulfilling a customer request they are driving around according to the currently activated rebalancing strategy (random by default). This is not perfectly efficient. Using the outcome from the parking team the two systems can be combined to allow dynamic resizing of the currently active fleet by parking vehicles that are currently not needed.

### 5) Implementation and evaluation of rebalancing strategies

---

Unused vehicles drive to locations according to a rebalancing strategy. The default

rebalancing strategy is “random” and sends the vehicles to random locations. The software architecture allows to easily implement further strategies and use them within the system. As a further development more rebalancing strategies can be implemented and evaluated for their performance in Duckietowns of different sizes. The reader is referred to [6] for a list of possible rebalancing strategies that can be implemented.

## 6) Location estimation and visualization between intersections

---

The current implementation updates the location of the Duckiebots only at intersections. Using wheel encoder information the locations could be estimated in between intersections and thus deliver a more fluid user interface and allow customer pick up in between intersections. A more high powered version might use SLAM to localize Duckiebots at any given point in time, allowing for more fine-grained localization and consequently better fleet planning.

## 7) Online Map Generation

---

The current implementation of the MOD system depends on a map of the Duckietown generated a priori. It can be extended with the SLAM functionality implemented by the team in Montreal. Using the map that’s generated while traversing the map would remove the step of manually creating the map and thus make the system more user friendly.

## 31.7. Conclusion

In summary, the fleet planning project at current allows for the high-level control of a large number of Duckiebots, the visualization of the duckiebots on the map in a GUI, the assignment of customer requests an the execution of taxi services. The system works but relies heavily on smooth functioning of other components and is only as robust as these components are. The Duckiebot classes are extensively documented and designed in a way that allows easy extension with different fleet planning and rebalancing algorithms. This paves the way for future updates, some of which were discussed in the previous section.

## 31.8. References

- [1] M. Pavone, K. Treleaven and E. Frazzoli, “Fundamental performance limits and efficient policies for Transportation-On-Demand systems” 49th IEEE Conference on Decision and Control (CDC), Atlanta, GA, 2010, pp. 5622-5629.
- [2] W. Galitz , “The essential guide to user interface design: an introduction to GUI design principles and techniques” John Wiley and Sons, 2007.
- [3] <https://docs.python.org/2/library/pickle.html>, Accessed: February 2017
- [4] L. Lamport and M. Massa, “Cheap Paxos,” International Conference on Dependable Systems and Networks, 2004, pp. 307-314.

[5] K. Spieser, K.Treleaven, R. Zhang, E. Frazzoli, D. Morton and M. Pavone, “Toward a Systematic Approach to the Design and Evaluation of Automated Mobility-on-Demand Systems A Case Study in Singapore” Chapter in Road Vehicle Automation, Gereon Meyer, Sven Beiker (Editors). Berlin: Springer, 2014, pp.229-245

[6] Pavone, M., S. L. Smith, E. Frazzoli, and D. Rus, “Robotic load balancing for mobility-on-demand systems.” The International Journal of Robotics Research 31, no. 7

## UNIT L-32

### Transferred Lane following

TODO: JT: move to operation manual section of teh book

This is the description of transferred lane following demo.

#### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Wheels calibration completed.[wheel calibration \(master\)](#)

**Requires:** Camera installed in the right position.

**Requires:** Joystick demo has been successfully launched.[Joystick demo](#)

**Requires:** PyTorch installed on duckiebot and laptop. (On duckiebot, you can either build from source([link](#)), or download the [pytorch-0.2 wheel file](#) we built)

#### 32.1. Video of expected results

[link 1 of lane following](#) [link 2 of lane following](#)

#### 32.2. Duckietown setup notes

A duckietown with white and yellow lanes. No obstacles on the lane.

#### 32.3. Duckiebot setup notes

Make sure the camera is heading ahead. Tighten the screws if necessary.

#### 32.4. Pre-flight checklist

Check: turn on joystick.

Check: Enough battery of the duckiebot.

### 32.5. Demo instructions

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

Step 1: On duckiebot, in /DUCKIERTOWN\_ROOT/ directory, run command:

```
duckiebot `roslaunch deep_lane_following lane_following.launch`
```

Wait a while so that everything has been launched. Press R1 to start autonomous lane following. Press L1 to switch to joystick control.

The following is the same as demo-lane-following: Empirically speaking, no duckiebot will successfully run the demo for its first time. Parameter tuning is a must. The only two parameters that you can modify is the gain and trim. The parameter pair which makes your bot go straight will unlikely work for the lane following due to the current controller design. Facts show that a gain ranging from 0.5 to 0.9, as long as paired with a suitable trim, will all work on this demo. Start with your parameter pair obtained from wheel calibration. Increase gain for higher speed. Increase trim to horizontally move the bot to the center of the lane. Decrease will do the inverse.

Step 2: On laptop, make sure ros environment has been activated, run command:

```
laptop `rqt`
```

In rqt, the images can be visualized are /(vehicle\_name)/camera\_node/image/compressed

### 32.6. Train the network from scratch

Step 1: Download the simulator docker from [link](#), and launch it by following the similar instructions in [link](#)

Step 2: Clone [link](#) and run

```
laptop `python collect_data.py`
```

This will generate ~10100 images under `images` folder.

Note: you can skip the first two step If you want to download the images we collected. Here is the [link](#))

Step 3: Clone [link](#), Assume duckietown\_project\_pose and gym-duckietown are under the same folder.

Step 4: Pretrain on simulation images:

```
laptop `cd duckietown_project_pose_estimation; python main.py --augment --use_model2`
```

You can add `--cuda` if you have gpu available.

Step 5: Download real images from [link](#), and decompress under gym-duckietown folder.

Step 6: Finetune on real images, by run: laptop `python finetune.py --use_model2 --epochs 200`

Step 7: Checkout branch cbschaff-devel and copy model. duckiebot `git checkout cb-schaff-devel` duckiebot `catkin_make` duckiebot `roscore` deep\_lane\_following duckiebot `cp model_file include/deep_lane_following/model.pth.tar`

Step 8: Run the ros package.

```
duckiebot `roslaunch deep_lane_following lane_following.launch`
```

## 32.7. Troubleshooting

Contact Chip Schaff or Ruotian Luo(TTIC) via Slack if any trouble occurs.

## UNIT L-33

# Transfer Learning in Robotics

TODO: JT: move file to appropriate location / rename

## KNOWLEDGE AND ACTIVITY GRAPH

**Results:** Understanding transfer learning and the domain randomization technique for transfer learning.

This unit introduces the concept of Transfer Learning and how it can be applied to Robotics.

## 33.1. Transfer Learning Definition

Transfer learning is a subfield of machine learning that focuses on using knowledge gained while solving one problem to solve a related problem.

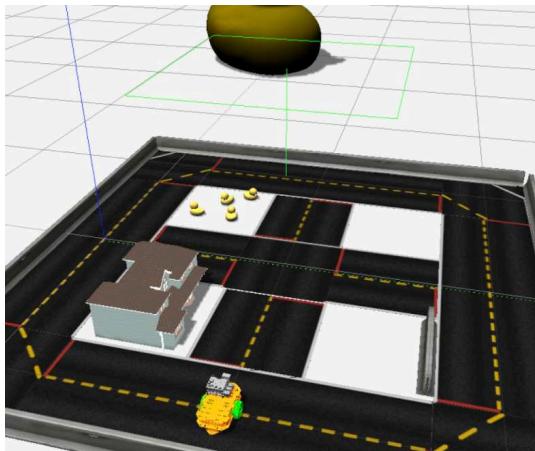
### 33.2. Why is transfer learning important in autonomous driving (or duckietown)

A known problem for autonomous driving (in real world or duckietown) is the lack of data. Today, the most successful methods use a form of machine learning called deep learning. Deep Learning is extremely powerful but is known to require a large amount of data to achieve good performance. However, it is time intensive and expensive to collect labeled data on a real system. Additionally, in reinforcement learning, the agent learns by trial and error, which can lead to large safety concerns for the vehicle and the people around it.

A solution to the data problem is to build a simulator, in which we can safely collect data and train deep learning systems. However there is a discrepancy between simulation and reality because the simulator does not perfectly model the world. So, we need transfer learning techniques to utilize models trained in simulation on real systems.

### 33.3. Transfer Learning in duckietown

In our case, the simulator has clean background and rooftop, but real duckietown has cluttered background. And also, the texture of the road are not exactly the same as duckitown, and there's no illumination changes in the simulator. However the lane width, camera setting are similar. Additionally, the dynamics in the simulation will not directly match the real Duckietown.



(a) Simulator image



(b) Real Image

Figure 33.1. Simulator images and real images

Need to have images with the correct filename in the folder

### 1) Domain randomization

---

Domain randomization is a common technique to enable transfer from simulation to the real world. The idea is to continually randomize the dynamics and look of the simulator. The intuition behind this idea is simple: the real world is going to look and act unlike the simulator, so we should force our policy to be robust to these factors. For example, let's imagine we wanted to train a policy to control the duckiebot directly from images. As long as you can determine the location of the lane lines, the exact look of the environment is unnecessary for the task. However, it is easy for the model trained in simulation to rely on the exact look of the simulation, making it useless in Duckietown. By forcing the policy to be robust to the lighting, coloring, and textures specific to the simulator, it will focus on details such as the position and shape of objects, which are equivalent to that in Duckietown. A policy which only relies on this information will then work when deployed in Duckietown.

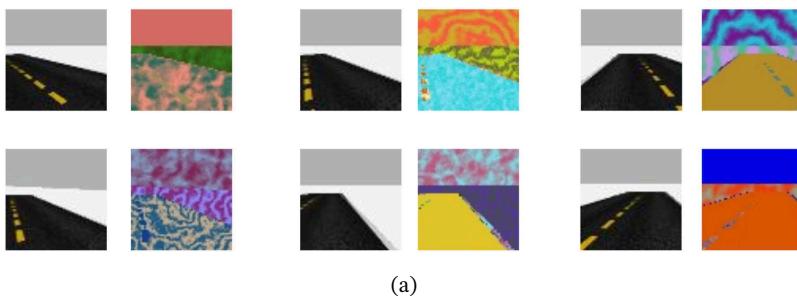


Figure 33.2. Simulators after domain randomization

### 2) Specific task to transfer

---

Here we have a simple example of training neural network for pose estimation using transfer learning. We replace the pose estimation module in current lane following package with the trained network.

The network takes the camera image as input, and outputs pose d and theta. (d and theta are the angle with respect to the front and )

### 3) Training pipelines:

---

Collect images by placing duckiebot in the simulator with different poses (The poses are recorded and later used to train the model). Train the neural network on simulator images. Each image is augmented differently by domain randomization technique during training. We use a CNN which contains 5 convolution layers and 2 fully connected layer. We use mean square error and use Adam to do optimization. Fine-tune on real images. (We collected by driving the bot around the duckiebot, and manually label the d and theta by bare eyes) Deploy on real bots.

### 4) Reference.

---

<https://arxiv.org/pdf/1703.06907.pdf>

## UNIT L-34

# Supervised learning: preliminary report

**TODO:** JT: adapt links to book style

## 34.1. Part 1: Mission and scope

### 1) Mission statement

---

To learn policies which match the results from recorded data from agents in the real world, so that the vast volumes of the data in the real world can be made useful.

### 2) Motto

---

In rete tuo videbimus lumen  
(In your net we see the light)

### 3) Project scope

---

*What is in scope:*

- Verifying whether Deep Learning can be used successfully in duckietown.
- Motivated by the concept of ‘data processing inequality’, using supervised and imitation learning to control the duckiebot end-to-end with data from a recorded policy.
- Using supervised or unsupervised learning to model specific aspects of the autonomous driving task.
- Focusing on indefinite lane navigation by learning based tools.

*What is out of scope:*

- Doing on-policy RL (i.e. running the robot with our learned policy to collect data).
- Collecting our own datasets by running with either our own policy (by hand).

*Stakeholders:*

**All current teams \*** Those who wish to use deep learning in the real world could benefit from our pipeline and experiments in using DL in the duckietown.

**For future teams \*** If a future team does on-policy RL in the duckietown, initializing with our imitation learned policy could be smart.

## 34.2. Part 2: Definition of the problem

### 1) Problem statement

---

We have recorded data of the lane following algorithm running smoothly in the duckietown. Our goal is to learn a policy which performs as well as, or better than the policy which produced the data.

### 2) Assumptions

---

- The policy used to collect the data is reasonably good.
- The training data can be updated when other groups formulate policy which have better performance.
- The errors in imitation learning are sufficiently small to allow a straightforward approach to learn a decent policy.
- Our trained policy can improve the robustness of overall performance.

### 3) Approach

---

- Initial approach is to take image and control parameter data from the lane navigation checkoff/log data from all schools.
- Start with indefinite navigation and lane following data.
- First DL approach is to take the last k-frames (probably could use a smarter selection strategy which picks some older frames) before the control parameters are recorded, and train a neural network to predict the control parameters from the state.
- We will manually downsample the image frames to find the smallest resolution where the lane is clearly visible by inspection.
- We will use an absolute error to predict the control parameters, and measure relative error on held-out data, to figure out if we can learn a network which gen-

eralizes.

- We will train ALI(adversarially learned inference) and VAE(Variational autoencoder) on the images, for the purposes of intellectual curiosity as well as semi-supervised learning (if overfitting is a serious issue).

#### 4) Functionality provided

---

1. Offline metrics
2. Loss for one step ahead prediction on recorded data, with point predictions for the two control parameters.
3. Likelihood under a continuous distribution over the predicted control parameters.
4. Likelihood under a fixed discrete distribution over the predicted control parameters.
5. Online metrics:
6. Actually run the duckiebot in real world duckietown with our learned policy.
7. Visual inspection of trajectories.

#### 5) Resources required / dependencies / costs

---

- Requires neural compute stick on the duckiebot to run. (already got it)



- GPUs for training models (available through MILA and IDSC in ETH).
- Data for training the imitation learning algorithm (ideally use as little as possible).

#### 6) Performance measurement

---

- We can use out-of-sample evaluation for the offline metrics, with care taken so that the train, validation, and test sets cover non-overlapping groups of students.
- Online evaluation will be qualitative, and will be done in the Udemy duckiebot simulator.

etown.

### 34.3. Part 3: Preliminary design

#### 1) Modules

---

- Data collection: a raw collection of the images (state) and control signals.
- Data alignment: Create sets of actions approximately aligned with states (as they're recorded at different frequencies).
- Data Example Construction: Create tuples of (state[t], action[t], state[t-1], state[t-2]).
- Data Train/Valid/Test Split: split the dataset randomly but with different students going into different datasets.
- Model trainer: From collected data trains a model to predict actions from states.
- Model actuator on duckietown: Runs the duckiebot using actions from the trained model.

#### 2) Interfaces

---

- Data collection: Takes student actions and returns a list of ROS bag files saved to some mila filesystem.
- Data alignment: Performed in-memory, takes the ros-bags as inputs and returns a list of aligned (state[t], action[t]) pairs.
- Data Example Construction: Also performed in-memory, and produces (action[t],state[t],state[t-1],state[t-k]) k-tuples.
- Data Train/Valid/Test Split: Uses a fixed random seed to split the examples into different groups, which are then saved to different files on mila filesystem.
- Model trainer: Takes data as input and returns a saved model binary for running on the intel compute stick (todo: figure out just how small binaries need to be).
- Model actuator on duckietown: Takes a trained model as input and is a ROS module which listens for camera data, and sends control signals to the motors.

#### 3) Specifications

---

- No changes to duckietown specification.
- Duckuments for using neural compute stick.

#### 4) Software modules

---

- Python script or small collection of python scripts for data processing.
- Python script using Tensorflow for training model.
- ROS node for running a fixed model on the duckiebot.

#### 5) Infrastructure modules

---

- We don't think any of the modules are infrastructure.

### 34.4. Part 4: Project planning

## 1) Data collection

---

- Collect data generated by other navigation policies.

## 2) Data annotation

---

- Generally speaking, no data annotation required.
- Might need to annotate video to remove crashes, stalls.

*Relevant Duckietown resources to investigate:*

- Taiwan group has done imitation learning with 3-camera setup - we may be able to reuse some of their code or at least learn from their experience.

*Other relevant resources to investigate:*

- See the following papers.

[title](#) [title](#) [title](#)

## 3) Risk analysis

---

- Raw imitation learning may perform badly in practice due to “exposure bias / exploration” issue.
- Simplest solution might be some sort of data augmentation which moves the lane in the training data to create “correction” examples.
- May be smarter ways to improve generalization.
- We may have a hard time figuring out what metrics to trust for offline evaluation of the learned model.
- Workaround might be to report many different metrics and always make sure that the simplest metrics (like per output relative error) are reasonably small.
- Some metrics are hard to interpret: making it difficult to know when to declare success.
- May require some intermediate online evaluation to figure this out.

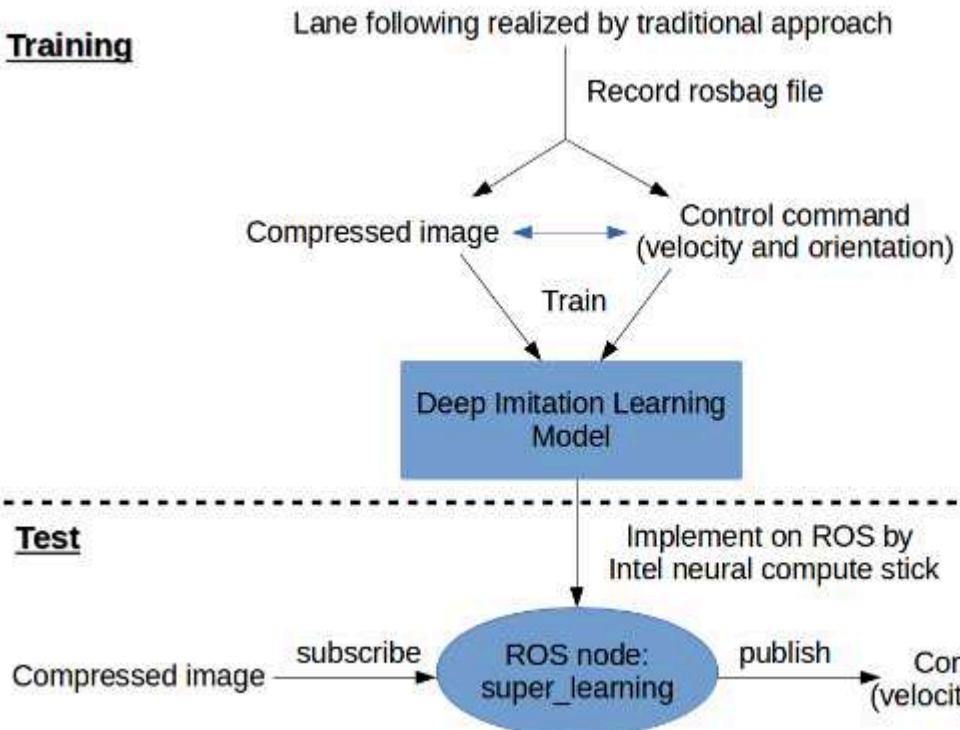
## UNIT L-35

# Supervised learning: intermediate report

**TODO:** JT: fix image sizes, bullet points hierarchy

## 35.1. Part 1: System interfaces

The system architecture is shown below.



## 1) Logical architecture

- Please describe in detail what the desired functionality will be. What will happen when we click “start”?
- The desired functionality is the deep imitation learning network. How the function works is the following: the node of trained deep imitation learning network subscribes to the image published by compressed image node, does computation, then publishes control commands(orientation and velocity) to the inverse kinematic node.
- Please describe for each quantity, what are reasonable target values. (The system architect will verify that these need to be coherent with others.)
- The robot does lane following with a learned end-to-end deep imitation learning system. Look at the activations of the layers and try to understand them. The target can also be extended to Indefinite navigation realized by deep imitation learning if the lane following is fulfilled.
- Please describe any assumption you might have about the other modules, that must be verified for you to provide the functionality above.
- The time latency of other modules are within reasonable range.

## 2) Software architecture

- Please describe the list of nodes that you are developing or modifying.
- We will develop one node, the trained deep imitation learning model, that maps the compressed images to control commands(orientation and velocity). All other nodes will remain unchanged.

- For each node, list the published and subscribed topics.
- The deep imitation learning node subscribes to /VEHICLE\_NAME/camera\_node/image/compressed.
- The node publishes /VEHICLE\_NAME/car\_cmd\_switch\_node/cmd
- For each subscribed topic, describe the assumption about the latency introduced by the previous modules.
- The latency of image topic can be measured. But during the model training process, we would like to use the map between multiple images and one control command to cover the latency.
- For each published topic, describe the maximum latency that you will introduce.
- The latency that our node introduces will be settled by the computation capability of the Intel Neural Compute Stick and the scale our model. Tests are required before the latency can be finalized.

## 35.2. Part 2: Demo and evaluation plan

### 1) Demo plan

---

The demo is a short activity that is used to show the desired functionality, and in particular the difference between how it worked before (or not worked) and how it works now after you have done your development.

- How do you envision the demo?
- In the final demo, we hope to implement end-to-end deep imitation learning on Duckiebot and make it work for the aim of lane following.
- What hardware components do you need?



- The Intel Movidius Neural Compute Stick.

## 2) Plan for formal performance evaluation

---

- How do you envision the performance evaluation? Is it experiments? Log analysis?
- The performance can be first evaluated manually: 1)Compare the time of lane following by traditional approach and the end-to-end deep imitation learning approach. 2)Observe the deviation of lane following by deep imitation learning approach.
- Other formal evaluation approach will be updated.

## 35.3. Part 3: Data collection, annotation, and analysis

### 1) Collection

---

- How much data do you need?
- A few thousands of image taken by the camera and the corresponding label pairs from images to control commands (orientation, velocity).
- How are the logs to be taken? (Manually, autonomously, etc.)
- The logs will be taken manually. The make log-minimal in branch 'dev-eth-sup-learning' will be enough for the data collection.
- Describe any other special arrangements.
- None.
- Do you need extra help in collecting the data from the other teams?
- None.

### 2) Annotation

---

- Do you need to annotate the data?
- None, The data itself makes enough sense.
- At this point, you should have you tried using [thehive.ai](#) to do it. Did you?
- We have collected the data and extracted the things we need already.

### 3) Analysis

---

- Do you need to write some software to analyze the annotations?
- None. But we did write a python script to write images and data pair from ros bag.
- Are you planning for it?
- None.

## UNIT L-36

## Supervised Learning: final report [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

# UNIT L-37

## PDD Neural Slam

### 37.1. Part 1: Mission and scope

#### 1) Mission statement

---

Build a map from a duckie driving around the road autonomously that will be used for planning

#### 2) Motto

---

À l'idiot sans mémoire tout lui paraît nouveau et miraculeux

#### 3) Project scope

---

##### What is in scope

Train agent to learn to explore an entire map efficiently and keep a representation of the current knowledge of the map.

##### What is out of scope

Need to run live on the duckie. We do not use real images but assume we have a tile detector

##### Stakeholders

Transfer learning team - tbd

### 37.2. Part 2: Definition of the problem

#### 1) Problem statement

---

We want to keep a representation of the current map in memory that could be used for downstream tasks such as planning

#### 2) Assumptions

---

- We work on the tile level, and do not worry about low-level control.
- We assume there are only two types of tiles, road or not road

#### 3) Approach

---

Stage 0: Create an environment where we can simulate a agent

Stage 1: Train an agent using a reinforcement learning method paired with an ex-

ternal memory

Stage 2: Deploy on the real robot and see how the method perform

Stage 3: Use a decoder that can recover the knowledge of the map

#### 4) Functionality provided

---

- A exploration policy
- A map when the exploration is done

#### 5) Resources required / dependencies / costs

---

- GPUs to train our policy
- Chip to run the neural network policy
- A “tile predictor” to infer the tile type from an image
- A simulator to train the agent

#### 6) Performance measurement

---

- Check whether or not the agent actually explores the whole map
- Compare the decoded map with the ground truth map

#### 7) Functionality-resources trade-offs

---

- Robust obstacle detection (many filters,...) vs. computational efficiency
- Maximizing speed (e.g. controllles might want to do that) vs. motion blur

### 37.3. Part 3: Preliminary design

#### 1) Modules

---

- A grid map environment to train the agent.
- A policy network with an external memory
- A decoder to decode the external memory into a map
- A tile detector ??

#### 2) Interfaces

---

##### Simulator

- takes the map size as input and generate a environment with agent output

##### Policy network

- take the current position of the robot w.r.t its original position, reads/write to the memory and give a control action.

##### Decoder

- takes the internal memory as input and give the actual map as output.

#### 3) Specifications

---

No need to revise duckietown specifications

#### 4) Software modules

---

- Pytorch / Tensorflow
- Grid world simulator

### 37.4. Part 4: Project planning

#### 1) Timeline

---

- Build the environment to train the agent
- Implement the agent to be trained with an external memory for the exploration task
- Train the agent
- Check if the map can actually be decoded

#### 2) Data collection

---

- No need for data collection

#### 3) Data annotation

---

- No need data annotation

*Relevant Duckietown resources to investigate:*

Duckietown simulator

*Other relevant resources to investigate:*

- *Neural SLAM*: Jingwei Zhang, Lei Tai, Joschka Boedecker, Wolfram Burgard, Ming Liu
- *Learning to Navigate in Complex Environments*: Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andrew J. Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, Dharshan Kumaran, Raia Hadsell
- *Hindsight experience replay*: Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, Wojciech Zaremba
- *Neural Turing Machines*: Alex Graves, Greg Wayne, Ivo Danihelka Differentiable neural computer Alex Graves et al (DeepMind)

#### 4) Risk analysis

---

- Might have issues to scale
- RL agent might have stability issues paired with an external memory the whole system could be hard to train

UNIT L-38

PDD - Visual Odometry

## 38.1. Part 1: Mission and scope

### 1) Mission statement

---

We will use unsupervised learning to build a depth estimation system for Duckietown. The application could be building a point cloud map for Duckietown for mapping—our overall plan is to have a serviceable deep network that is trained end-to-end with no ground truth data. We will also be using the Movidius chip, hopefully learning to how integrate it well into the current system for future users.

### 2) Motto

---

CARPE DIEM  
(Seize the day)

### 3) Project scope

---

The scope of the project is to use recent work in unsupervised depth estimation as well as training data we gather in Duckietown to create a fully unsupervised depth estimation system.

*What is in scope:*

The deep network and use of Movidius chip.

*What is out of scope:*

Localization. We plan on utilizing the apriltags for localization. Also hardware modification—we plan to do fully monocular depth.

*Stakeholders:*

All of the SLAM teams could (potentially) benefit from our system, though it might be non-trivial to integrate it.

## 38.2. Part 2: Definition of the problem

### 1) Problem statement

---

We need to train a deep neural network to predict depth from monocular video (with no GT depth!)

### 2) Assumptions

---

We assume that it's possible to collect diverse enough training data in Duckietown to train a deep CNN (outside data like KITTI might not transfer to this task).

### 3) Approach

---

We will start with the “Unsupervised Learning of Depth and Ego-Motion from Video” paper and modify the framework for duckietown.

### 4) Functionality-resources trade-offs

---

There's often a tradeoff between size of network and performance—the Movidius chip is more limited than our usual NVidia GPUs.

### 5) Functionality provided

---

We will need to develop some confidence measures for our depth map. Time permitting, we would like to build a point cloud map of Duckietown.

### 6) Resources required / dependencies / costs

---

We will need the Movidius chip for best performance—our metrics are the size of the network (including activations) and making sure that inference is real-time.

### 7) Performance measurement

---

Measuring performance here is tricky since there is no way to obtain ground truth depth data from Duckietown (aside from using a range sensor not available for duckiebots). We will likely develop a surrogate metric based on apriltags.

## **38.3. Part 3: Preliminary design**

### 1) Modules

---

Since it's an end-to-end neural network it's all in one logical module, but we could split up the design of the architecture from the way we use the data (e.g. data augmentation)

### 2) Interfaces

---

Input: RGB Image Output: Depth map (potentially relative depth, discretized)

### 3) Preliminary plan of deliverables

---

The architecture and the data collection and augmentation schemes need to be designed. Tensorflow implementation of architecture is what needs to be implemented. There already exists open source code for unsupervised learning of depth (paper linked below).

### 4) Specifications

---

Do you need to revise the Duckietown specification? N/A

## 5) Software modules

---

The software will be a simple end-to-end CNN going from frames to a depth map (likely a node publishing a depth map)

### 38.4. Part 4: Project planning

Next phase is to start collecting data and experimenting with architectures in Tensorflow.

#### 1) Data collection

---

Video of duckiebot traversing duckietown.

#### 2) Data annotation

---

No data annotation necessary.

*Relevant Duckietown resources to investigate:*

All of the camera geometry and computer vision notes.

*Other relevant resources to investigate:*

1. <https://people.eecs.berkeley.edu/~tinghuiz/projects/SfMLearner/>
2. <https://github.com/tinghuiz/SfMLearner>

#### 3) Risk analysis

---

It's possible that training a deep neural network on only duckietown data will be difficult. We will also consider using the KITTI dataset as additional training data.

## UNIT L-39

# Visual Odometry Project

Here we briefly describe the theory behind the model in the visual odometry project. The discussion begins with a review of epipolar geometry and a description of the depth image-based rendering problem, then moves to the description of the deep learning model used.

### 39.1. Epipolar geometry and DIBR

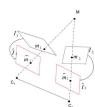


Figure 39.1. Epipolar geometry.

We follow the discussion in Sun et al. (2010). First, consider the stereo setup and recall the relationship between a world point

$$\begin{aligned} m_1 &= \frac{1}{Z} K_1 \cdot R_1 [I] \cdot C_1 M \\ m_2 &= \frac{1}{Z} K_2 \cdot R_2 [I] \cdot C_2 M \end{aligned}$$

If we choose the left camera as the reference, we can set  $R_1=I$ ,  $C_2=0$  in order to get:  $\begin{aligned} m_1 &= \frac{1}{Z} K_1 \cdot [I] \cdot M \\ m_2 &= \frac{1}{Z} K_2 \cdot R_2 [I] \cdot C M \end{aligned}$

and then we can get the relationship with depth  $Z$ :

$$Zm_2 = ZK_2 R_2 K_1^{-1} m_1 + K_2 C$$

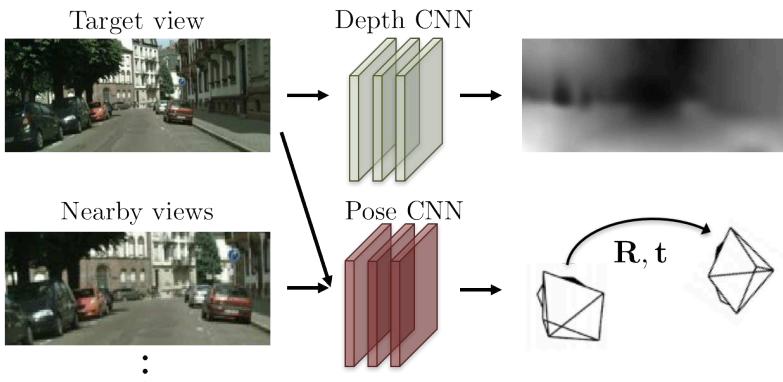
Using this relationship, we can learn a prediction for  $Z$  and use image  $m_1$  to predict image  $m_2$ —we describe the experiments below.

## 39.2. Learning Depth Prediction

We consider an end-to-end CNN-based unsupervised learning system for depth estimation, using the paper Unsupervised Learning of Depth and Ego-Motion from Video by Zhou et al., CVPR 2017. This model was initially trained on the KITTI dataset, and we take the pre-trained weights and evaluate them in Duckietown, showing that the results can be significantly improved by fine-tuning with Duckietown images. Our goal is real-time inference, and at test time we only use the depth prediction network:



(a) Training: unlabeled video clips.



(b) Testing: single-view depth and multi-view pose estimation.

Figure 39.2. The general architecture.

The model consists of two networks—a pose estimation network giving us  $\$R, t\$$  between the source and target frames, and a depth prediction network that gives us  $\$Z\$$  and allows us to warp the source view to the target view using the pose and the RGB values in the source image. Over time, the depth prediction network begins to predict reasonable depth values. The result of fine-tuning on Duckietown data plus KITTI pretraining versus applying trained model on the KITTI dataset directly:

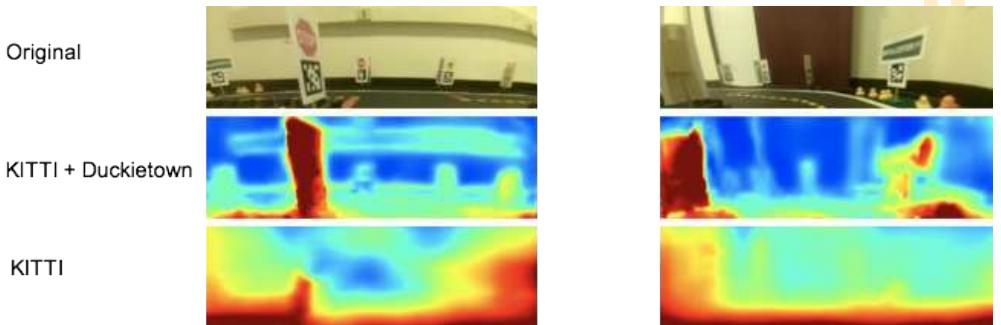


Figure 39.3. Fine tuning on our dataset.

When the bot turns, motion blur heavily affects the model predictions. One way to alleviate this problem would be to preprocess input images. First deblur them and then feed them to the depth prediction network. Having blurred images in the training set would also slightly improve the results:

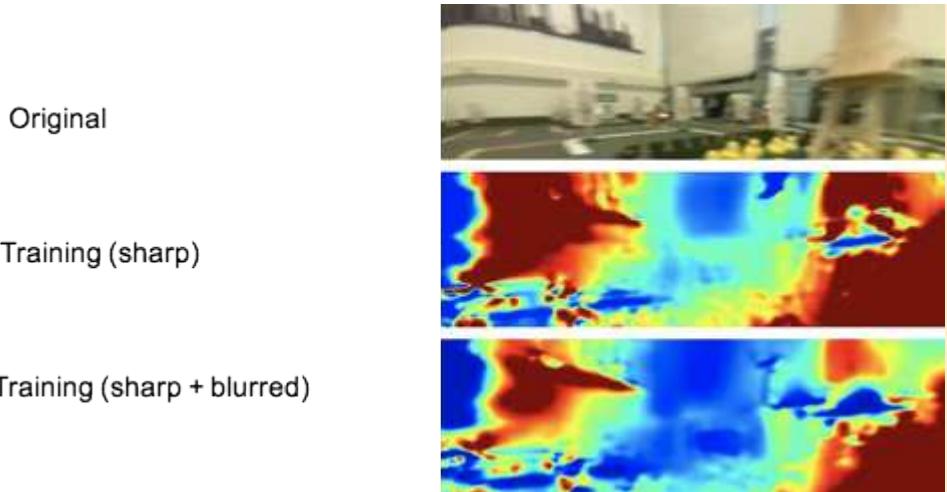


Figure 39.4. Results on motion-blurred images.

### 39.3. True Depth

We benchmark our results against the true distance from the camera we get from April tag detection. In the figure below, we show the predicted depth values versus the estimated depth:

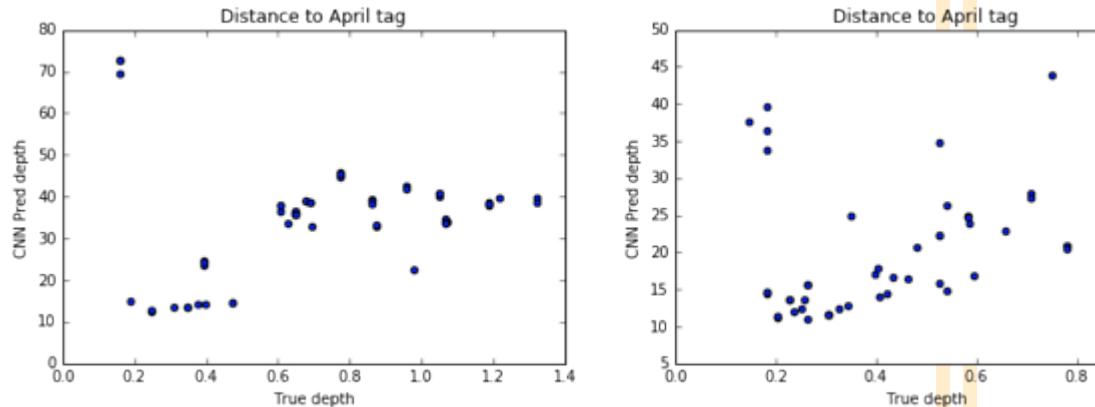


Figure 39.5. Comparison to true depth.

Outliers at low depths are due to lack of texture around April tags. As we can see with proper scaling our estimated depth is well aligned with the estimated depth from April tags—using a sparse set of true depth maps, we can rescale our pixelwise prediction into metric units.

### 39.4. Conclusions

We demonstrated a monocular depth estimation pipeline, trained with no annotated data. The approach gives reasonable depth predictions in Duckietown, but we

found several notable limitations. The results on motion-blurred frames are poor, even after fine-tuning with a small number of blurred images—for good results in fast-moving environments, we would likely need to train with more blur. In addition, our approach does not incorporate any traditional depth post-processing, which should significantly improve results.

Our depth prediction node could be used for a variety of tasks, including point-cloud-based SLAM as well as obstacle detection.

## 39.5. References

[Unsupervised Learning of Depth and Ego-motion from Video](#)

[Depth Image-Based Rendering](#)

**Author:** Igor

**Maintainer:** Igor

**Point of contact:** Igor

UNIT L-40

## Deep Visual Odometry ROS Package

### 40.1. devel-visual-odometry

This package contains a ROS node `dt_visual_odometry` that produces monocular depth estimates on duckiebot. It also contains another node `apriltags_ros_center`, which is slightly modified from `apriltags_ros` to publish pixel locations, in order to benchmark the result on April tags. You need to have Tensorflow installed on your local machine.

To launch the node, clone the repository into `catkin` workspace and download the Tensorflow model checkpoint file from Duckietown Dropbox to `checkpoint_dir`. This Tensorflow model is obtained through taking a pretrained model on KITTI and further train it on Duckietown for 200K iterations with smoothness penalty set as 0.1. Use the command:

```
roslaunch dt_visual_odometry deepvo.launch ckpt_file:=checkpoint_dir robot_name:=robot_name
```

This publishes the depth heatmap the into `robot_name/V0/image/compressed` as well as prints the predicted(from CNN) and actual(generated from April tags node, unit in meters) depth on any detected April tags. The scaling factor is calculated as the average predicted/actual depth for all detected April tags, and published under the topic `robot_name/V0/scale`. To supress April tags detections for a higher refresh rate of depth heatmap, use `apriltags_scaling:=0`.

UNIT L-41

# Anti-Instagram: preliminary design

## 41.1. Part 1: Mission and scope

### 1) Mission statement

---

Make the Duckiebot robust to illumination variation.

Line detection includes different colors of lines, we need to be able to detect these colors accurately.

### 2) Motto

---

SEMPER VIGILANS

(Always vigilant)

### 3) Project scope

---

*What is in scope:*

- Improve software (anti instagram, line detection, ...) such that the line detection is robust to illumination variation
- Sample ground truth pictures
- Influence future changes on Duckietown (e.g. color of parking spots)

*What is out of scope:*

- Geometric interpretation of the line detection (e.g. where is the middle of the road, distance to certain objects, ...)
- Hardware modifications of the Duckiebot
- Hardware modifications of the current Duckietown set up (colors of lanes, stop line, ...)

*Stakeholders:*

System architect	She helps us to interact with other groups. We talk with her if we change our project.
The controllers	A.A. is the interaction person to confirm that we fulfil their requests regarding: frequency, latency, accuracy (resolution), maximum false positives, maximum misclassification error (confusion matrix).
The Parking	We will determine together the best color for the parking lot lines if needed. At the moment the Duckiebot is controlled open loop at intersections. This should be improved.
The Navigators	Probably they need line detection in a certain way. We can help and figure together out what procedure would be the best

## 41.2. Part 2: Definition of the problem

### 1) Problem statement

---

One of the central problems in autonomous mode of the duckiebot is the line detection. Line detection though is very sensitive to illumination variation. This problem has been solved by a color transformation called “Anti-Instagram”. The current illumination correction algorithm, however, is not working well enough. This affects the extraction of the line segments since the extract-line-segments algorithm is very sensitive to illumination changes (e.g. shadow, bright light, specular reflections). There are several reasons why the current implementation fails:

1. Illumination correction is done only once by user input. So we don't do any online/automatic correction. This will obviously fail in an environment where illumination is changing frequently.
2. The algorithm works by detecting different clusters in RGB space for the colors of lines, thus it fails when it's not able to differentiate adequately. (e.g. in certain lighting conditions yellow and white look quite similar, in addition specular reflections distort all of the colors)
3. The color space is fixed to RGB, but it's unclear that this is best.
4. No geometric information is considered in differentiating colors of lines. The color information is completely decoupled from the place it's actually coming from, thus a red Duckiebot may be detected like a stop line, even though their shapes are quite different. (We also don't know whether a pixel is coming from the “street level” or the “sky level”)
5. It is a linear transformation (shift, scale) instead of a possible non-linear transformation, but there is nothing indicating this should be true.
6. Any previous anti-instagram transformation parameters are not taken into account when a new transformation is performed, thus no prior knowledge is leveraged.

### 2) Assumptions

---

1. Lighting:
  - o We assume normal office lighting, including any shadows that may occur be-

cause of occlusions, or spatial variance.

- We don't consider outdoor illumination (e.g. sunlight).
- We assume having illumination (no pitch black scenario).

2. Duckietown Condition:

- We assume the normal colors of lane lines, plus one or two more (for the parking lot).
- We assume no variation in the shapes of the lines besides what is already constructed.

3. Training data:

- We expect to have some pictures in different scenarios with correctly labeled segmentation (lines, street, ...), where a polygon is drawn around each region.
- The pictures will be captured from a Duckiebot camera.

4. We also assume the current method of extracting lane pose from segments is accurate.

### 3) Approach

---

1. Understand current system

- Determine false positives, false negatives, true positives, true negatives of current line detection
- Determine latency
- Compare other color spaces than RGB to see how it affects performance (e.g. HSV or LAB).

2. Use geometric information to better determine the actual existing colors.

- The optimal case would be that the system already knows beforehand which areas it should take into account. The relevant color areas are only the dashed lines area, the continuous line area, the stop line area, the parking line area and the street area. Everything else should not be taken into account since we have no reference color for the other areas.
- It should be possible to define a region in the picture where we can find these specific areas. For example the dashed line starts lower left and goes to direction top middle but it should stop at the “line of horizon” (= middle of the vertical length)
- Distinguish between dashed and non-dashed lane lines to simplify identification of colors.

3. Use time information/parameters from earlier illumination corrections to improve robustness (Online learning).

- In contrast to starting the color analysis from scratch every time, we could consider using the latest transformation parameters as an initial guess. We could consider to update the color analysis every x-th frame during a session, to be more robust to changing light conditions/shadows.

4. Further improvements:

- We are using the color transformation to better estimate the lines. So we know after processing (color transformation, edge detection, ...) where we can find the lines. With that information we could update the color transformation. The color transformation now should take into account only the “important” areas. As a result we should have a more accurate color transformation. We repeat until we converge to a minimum.

### 4) Functionality provided

---

We are assuming to have ground truth pictures. Then it is possible by processing the same picture with our algorithm and compare it to the ground truth to calculate an error.

We are assuming to have ground truth pictures. Then it is possible by processing the same picture with our algorithm and compare it to the ground truth to calculate an error.

We are going to consider true positives, true negatives, false positives and false negatives. This can be done either for only one color/one feature (dashed lines, continuous lines...) or for the whole process at once which means everything should be classified properly. In addition, we would like to measure the accuracy of the lane pose estimation for our algorithm vs. a ground truth, this can be a Euclidean distance.

## 5) Resources required / dependencies / costs

---

Costs:

1. Computational cost
  - If the processing is done online we have to take care that it doesn't take too long.
2. Cost of producing ground truth pictures
  - Will be determined when we have some examples done by hand. (Week of 20th of November)

Resources:

1. Functional Duckiebot
  - Getting sample data for ground truth pictures
  - Try out the algorithm in real conditions

Dependencies:

1. Ground truth images
2. Lane pose estimator

## 6) Performance measurement

---

1. Identify current run times of algorithms implemented at the moment and compare the algorithms intended to implement with the current ones. If the new ones are way more costly than the current ones and the current ones already use the Raspberry Pi to capacity it is probably not a good idea to implement ours. To sum up: Estimate run time of new implementation and compare to old. See whether implementation is feasible.
2. We can compute percentage of success of identification of a line segment, as well as correct color classification.
3. Measure euclidean distance of lane pose estimation using our algorithm and lane pose estimation without to the ground truth.

The performance measurement procedure for the algorithm is described in the section *Functionality provided*.

### 41.3. Part 3: Preliminary design

#### 1) Modules

---

1. Anti-Instagram module: Takes raw picture from camera and estimates a color transformation. The transformation details are returned.
2. Module to classify geometries (e.g., distinguish between dashed lines, continuous lines and stop lines): This could be a standalone routine, which gets called by the Anti-Instagram, we could also combine this with the anti-instagram as described in Approach point 4.
3. Online learning: Takes picture from camera, does Anti-Instagram procedure and transformation. The error (e.g. cluster error of k-means) is estimated and the procedure is repeated until the optimal transformation parameters are found (transformation with the lowest error). The procedure returns the optimal parameters. They are saved and used for the future image processing.

#### 2) Interfaces

---

1. Anti-Instagram: input: Raw camera image. Output: Transformation parameters
2. Geometry Classifier: Input: Raw camera image. Output: List of the different line segments.
3. Online learning module: input: Multiple camera images/continuous stream. Output: Optimal transformation parameters

#### 3) Preliminary plan of deliverables

---

1. Take the current algorithm and find best color space for it, estimate the errors and accuracies discussed previously.
2. Search for other clustering method and optimize current version. (Without considering geometry)
3. Consider geometry (as a first step indicate considerable areas by hand) and see what difference it makes compared to the current optimal implementation (Maybe after 1.) and 2.) are done).
4. Distinguish relevant and non-relevant areas (street surface vs. rest of world).
5. Distinguish dashed and continuous lines.
6. Implement and test a online learning system.

#### 4) Specifications

---

As stated above we are only involved in determining the best fitting color of the (not yet installed) parking lot lines. All the other colors and the environment are assumed to be given.

#### 5) Software modules

---

1. Anti-Instagram Node: Already exists, will most likely be updated or even changed completely according to our approach. The online-learning (if accomplished) will be folded into this node.
2. Geometry Classifier Node: Needs to be written according to the approach.

## 6) Infrastructure modules

---

None.

### 41.4. Part 4: Project planning

Date [MM/ DD/ YYYY]	Task	Target Deliverables
11/20/ 2017	Finish the Preliminary Document, Preliminary Document	
11/27/ 2017	Peer Reading of other team members	
11/27/ 2017	Investigate why current algorithm fails.	Create detailed description when the algorithm fails and when it works. Make it understandable why for everyone
11/27/ 2017	Create data annotation, check how website works	Get 1000 annotated pictures or have a specific date when these images are delivered.
12/1/ 2017	Find out what colorspace is the best for the current algorithm	best color space, performance analysis
12/4/ 2017	Find out what's the best clustering method based on best color space	Best clustering method, best color space, performance analysis is the best color space still the best?
12/4/ 2017	Include geometry in the current color transformation algorithm	Performance analysis
1/8/ 2018	Implement an online system	Performance analysis of supervised system

#### 1) Data collection

---

Around 1000 pictures with the Duckiebot camera from a Duckiebot perspective in Duckietown. The pictures have to be from different environment conditions (illumination, specular light)

#### 2) Data annotation

---

The data collected above has to be annotated. The annotations should state what type and what color it is.

1. Dashed lines: Yellow
2. Continuous lines: White
3. Stopping lines: Red
4. Street: Black
5. (Parking Lot: TBD)

*Relevant Duckietown resources to investigate:*

The whole sum of nodes within the ‘10-lane-control’ folder will be within the

scope of this project. They are:

- anti\_instagram
- ground\_projection
- lane\_control
- lane\_filter
- line\_detector
- complete\_image\_pipeline

*Other relevant resources to investigate:*

1. Color differentiation, [like this](#).
2. Properties of different color spaces
3. OpenCV

### 3) Risk analysis

1. Computationally too expensive algorithms
  - We have to estimate our algorithms carefully and compare them to the existing solutions.
2. No annotated data delivered
  - Build annotated data by ourselves/by hand.
3. Not enough time
  - Create good tasks list to be done. Try to specify time exact for every task.

## UNIT L-42

### Anti-Instagram: intermediate report [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT L-43

### Anti-Instagram: final report

#### 43.1. The final result

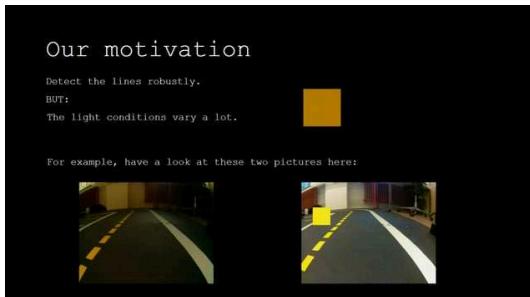


Figure 43.1. Anti instagram video

**TODO:** add link to operation manually, add link to README / code sections (README.txt is in this folder - move to appropriate code section)

## 43.2. Mission and Scope

### 1) Motivation

---

During the process of autonomous driving the Duckiebot has to know where to drive and where not. For example the road marking or obstacles on the road give constraints where the allowable area to drive is.

Let's take the road marking example. For Duckietown it is true that the road is rather black, the stopping lines are red, the side lines are white and the dashed middle lines are yellow.

Knowing about this color information the Duckiebot is theoretically able to know whether it is on the correct lane position or not. By detecting the lines and estimating the position the Duckiebot can know whether it should drive more left or right or stay in the same position.

This is only possible if we are assuming the line detection is done perfectly.

But the line detection and especially the classification is done with the color information.

The problem now is that the color can vary due to illumination variation. So the classification can fail and therefore the performance suffers.

To sum up the lower the variance in the color of the lines the better the classification and the better the autonomous driving performance of the Duckiebot.

### 2) Existing solution

---

Before we started our project a solution was already implemented. The existing approach was estimating a color transformation by using k-Means algorithm with 3 centers.

First "true colors" were defined. These "true colors" should represent the best colors for the lines. So if the road marking lines would be in these predefined red, yellow and white the color classification would work the best.

So this approach took all the pixels of the camera image and estimated three centers with the k-Means algorithm.

Afterwards the three found centers were compared with the previously defined "true colors". The difference of the centers found by k-Means and the "true colors" lead to the color transformation.

Maybe you can better imagine the procedure as follows:

You take all the pixels and their RGB values. Then you plot each pixel in your imaginary R,G,B coordinate system.

The k-Means algorithm now tries to detect clusters in this RGB space and estimates the center of these clusters. The centers of these clusters are now compared to the "true centers" which is the location of the optimal red for example in the RGB space. This leads to a transformation which is applied to every image from the camera of the Duckiebot.

### 3) Opportunity

#### *Advantages of existing solution:*

1. The idea of estimating a color transformation from a captured image based on estimated and true centers is very promising since it really focuses on transforming the colors.

Often other image transformations focus on white balance. But we are concerned the most of the colors. So this clustering approach is a good idea here.

2. k-Means is a fairly simple approach and can be used for unsupervised learning. This is very interesting for a future online implementation.

#### *Disadvantages of existing solution:*

1. The k-Means clustering was initialized only with 3 centers. This is a very rough guess. By analyzing several sample images one sees that there are distinct white, red and yellow clusters. They can indeed be represented by three distinct centers. But the problem is that all the other pixels have to assigned to a cluster as well. This distorts the color transformation.

2. The existing solution was not online. The color transformation had to be estimated explicitly by pressing a button on the joystick. Firstly the system is not fully autonomous anymore since it needs user input (pressing the button). And secondly the user doesn't or cannot really know when the optimal moment is for a color transformation.

We sampled several pictures and tested the old implementation. Following a table with the sample pictures and the outputs.

Example 1:



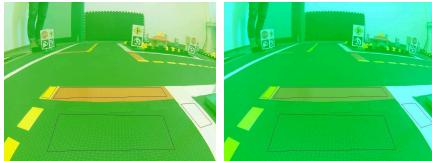
This was a good working example. The euclidean error of “true” centers compared to the centers which are estimated in the picture decreased from 175.541 to 43.724.

Example 2:



This was a medium working example. The euclidean error of “true” centers compared to the centers which are estimated in the picture increased from 115.267 to 56.646

Example 3:



A bad example. The error increased 78.114 to 146.192.

#### 4) Preliminaries

---

Preliminaries to the following topics would be good to have: - [Clustering methods \(master\)](#) - [Convex optimization \(master\)](#) - [Histogram equalization \(master\)](#)

### 43.3. Definition of the problem

The general idea is to find a transformation matrix  $T$  which converts the original image such that we have the best colors for the color classification.

#### 1) Mathematical definition

---

*General color transform:*

A general color transformation is stated as follows:

$$\$ \$ \vec{c}_{\text{transformed}} = T \cdot \vec{c}_{\text{in}} = \begin{pmatrix} k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,1} & k_{3,2} & k_{3,3} \end{pmatrix} \cdot \begin{pmatrix} R_{\text{in}} \\ G_{\text{in}} \\ B_{\text{in}} \end{pmatrix} = \begin{pmatrix} R_{\text{transformed}} \\ G_{\text{transformed}} \\ B_{\text{transformed}} \end{pmatrix} \$ \$$$

Where we have  $\vec{c}_{\text{in}}$  as the input pixel values (a value per channel) and the vector  $\vec{c}_{\text{transformed}}$  is the output of the color transform. But first we used a slightly simpler color transform. Because we wanted to have a color transform which doesn't depend on other channels but only on one. One wants to find for each of the channels (e.g. RGB) a scale factor and a shift value. This leads to the following mathematical formulation:  $\$ \$ \begin{pmatrix} R_{\text{transformed}} \\ G_{\text{transformed}} \\ B_{\text{transformed}} \end{pmatrix} = \begin{pmatrix} k_{\text{red}} & 0 & 0 \\ 0 & k_{\text{green}} & 0 \\ 0 & 0 & k_{\text{blue}} \end{pmatrix} \cdot \begin{pmatrix} R_{\text{detected}} \\ G_{\text{detected}} \\ B_{\text{detected}} \end{pmatrix} + \begin{pmatrix} s_{\text{red}} \\ s_{\text{green}} \\ s_{\text{blue}} \end{pmatrix} \$ \$$

where  $k_i$  stand for the different scale factors for each channel and  $s_i$  are the shift factors for each channel.

RGB color channel is just chosen as an example. The transform is valid for other channels as well. The best channel should be determined later.

Now with the found scales and shifts or with a full matrix as stated first we would have a color transform which can be applied to every image captured by the camera.

This color transform would correct the image such that the colors are best to be detected by the line detector and hence the classification of the lines would work perfectly.

#### 43.4. Contribution / Added functionality

As a first idea we wanted to improve the current implementation. So we further investigated the k-Means approach.

##### 1) k-Means Approach

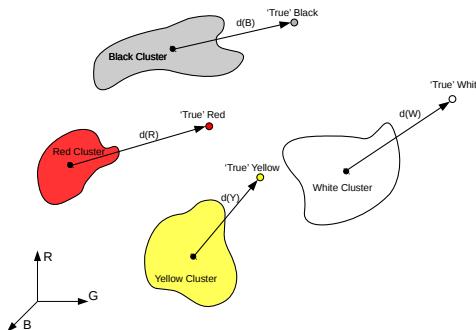
*Implemented in the kMeansClass found in the kmeans\_rebuild.py file. This is set with the parameter lin for the trafomode in the ContAntiInstagramNode.*

Idea:

The core idea of the clustering approach is to define some “true” colors for the colors to be transformed. In the Duckiebot case these are yellow, white, black and red. These “true” colors can be defined differently if wished. It is just important that the definition is coherent with the color classification of the line detection. To determine the color transformation which would ideally map all the red line colors on the “true” red, the yellow line colors on the “true” yellow we have to find out what the error is. This is dependent obviously on the current environment illumination. So depending on the illumination the yellow of the dashed middle line is more or less “wrong” or away compared to the “true” center.

The approach is now to determine the so called real centers of the colors of the lines by using a clustering method. In this case it is k-Means.

We’re going to cluster all the pixels of the input image and try to find the red, yellow, white and black cluster. By calculating the center of each of those clusters we are able to compute the color transform.



Basic idea of the k-Means transform

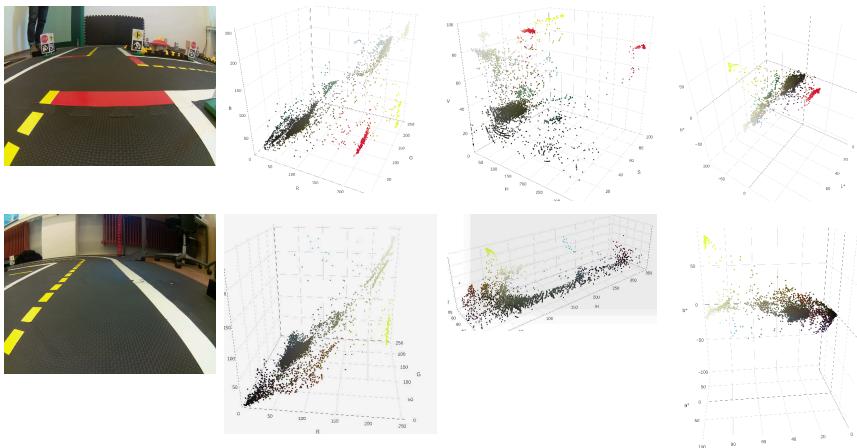
We were convinced that a clustering approach is a good solution because of the following points: 1. The clustering and the consequently determined centers lead to a color transformation focusing only on the problem relevant colors. 2. Clustering can be implemented unsupervised. We were convinced that k-Means is a good solution since it is fairly simple to implement.

We decided to use the RGB space for the clustering approach and therefore as well

as the reference space for the transformation described in [the chapter above](#) + Ref.  
error

I do not know the link that is indicated by the link '#mathematical-general-transform'. [ ]

The following pictures show how sample images look in different color spaces. The far left is the original image where we got the data from. The second from left is the pixel values plotted in RGB space, the third in HSV space and the far right the pixel values in LAB space.



One sees that the color clusters of red, yellow, white and black are nicely distinct in the RGB space. So RGB is a good space to do clustering. As stated in [disadvantages](#) + Ref.  
error

I do not know the link that is indicated by the link '#disadvantages-existing'. [ ] we used more than 3 cluster centers. So we tried it with 10 clusters. But now one has the problem that we don't know anymore which cluster is which. That's why we implemented a function which can determine which color is which.

#### Color determination:

Implemented in the `determineColor` function found in the `kmeans_rebuild.py` file.

If we do k-Means in an unsupervised way we don't know in the end which cluster and therefore which center belongs to which color. We made the assumption that the nearest cluster center is of the same color as the nearest true center.

We thought this assumption is reasonable because looking at some data in the RGB space one sees that if we would want to interchange clusters we need a massive distortion. Such a distortion is in our opinion almost impossible.

So to determine which cluster belongs to which "true" center we do the following steps:

1. The output of the k-Means consists of 10 centers. Or the number of centers you put as an input. Now for each "true" center the error to all found center of k-Means is computed.
2. We iterate through the "true" centers and look which determined center has the lowest value. This center would be assigned to the color of the "true" center. If a "true" center would've been assigned already we would take the determined center with the second lowest error.
3. Now we have a list of centers which we know what colors they are.

Using the least squares approach we can fit with these four centers an optimal color transformation. The very alert fan of Duckietown will already have realized that in this whole color discussion there is one mistake: We don't have always the red stopping line in an image of the camera. There exist sections where you have a simple straight road without any intersections. Therefore you won't find any red color. So we need a solution for that.

That's where color elimination comes into play.

#### *Color elimination:*

*Implemented in the detectOutlier function found in the outlierEstimation file.*

The use of all four colors red, yellow, white and black can lead to wrong results. You just have to think of a situation where you don't find any red stopping line in the picture. That's why we came up with the following "color elimination" procedure: 1. Pick three colors out of the four colors and do a least squares fit to find the color transformation. 2. Estimate the error of the least squares fit. 3. Repeat steps 1 and 2 for all color combinations and keep the three colors which give the lowest error. So this least squares fit would be the one without the outlier. So in the case you don't have any red in the image you would just use white, black and yellow since wrong red cluster would distort the color correction. If there are all the four colors in the picture we observed that most of the time the black cluster is thrown away. This makes quite sense since in the [chapter](#) + Ref. error I do not know the link that is indicated by the link '#k-means-idea'.  above we saw that the black center doesn't really fulfill the assumptions for k-Means since the black cluster is quite elliptical shaped.

Something we observed was that sometimes the procedure gave unreasonably high parameters as an output. So we came up with a convex optimization approach.

#### *Convex optimization:*

*Implemented in the calcBoundedTrafo found in the calcLstsqTransform.py file.*

The idea here is to limit the parameter values of the color transform. We observed that for example a scale factor for a channel of 240 is unreasonably high. So we stated the following:  $\$k_i \leq 3 \quad \$s_i \geq -100$  where  $\$k_i$  and  $\$s_i$  are the scale and the shift of each channel.

We used then convex optimization to limit the scale and shift values.

#### *Online Approach:*

*Implemented in the ContAntiInstagramNode found in the cont\_anti\_instagram\_node.py file.*

*This is set with the parameter ai\_interval in the ContAntiInstagramNode.*

We set the goal for our project to implement an online solution. So we decided to repeat the procedure described before every 30 seconds. This time interval is quite arbitrary and the lower would be the better.

#### *Results:*

All in all the k-Means approach gives good results but it is too slow for an online

approach. If we ran the algorithm on the Duckiebot it consumed almost all the processing power and other computational complex functions were limited.

## 2) Histogram Equalization

*Implemented in the simpleColorBalanceClass found in the simpleColorBalance-Class file. This is trafomode cb for the ContAntiInstagramNode.*

After implementing the k-Means approach we saw that there was need for something else. A very simple white balance would help us a lot to correct the picture to a certain extent and being very fast.

Idea:

Doing research we found a promising approach for a very simple color balance.

Actually the analysis in the [disadvantages chapter](#) + Ref. error I do not know the link that is indicated by the link '#disadvantages-existing'. led us to this idea. Experimenting with gimp we found out that the automatic white balance leads to quite good results:

Example 1:



This was a medium working example. The euclidean error of “true” centers compared to the centers which are estimated in the picture increased from 115.267 to 31.2.

Example 2:



This was a medium working example. The euclidean error of “true” centers compared to the centers which are estimated in the picture increased from 115.267 to 30.35.

This were very promising results comparing to the results of the [disadvantages chapter](#) + Ref. error I do not know the link that is indicated by the link '#disadvantages-existing'.

The idea works as following:

1. Create a sorted list for every channel of the image.
2. Remove a certain predefined percentage of this list from the top and the bottom. The percentage serves as a parameter for the anti instagram node.
3. After removing the highest and the lowest value of this list serve as the thresholds(\$Th\_{low}\$, \$Th\_{high}\$) for the frames afterwards.

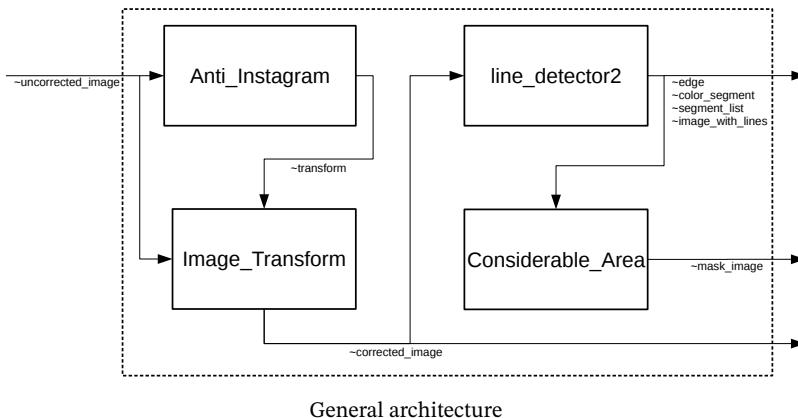
#### 4. Do this for every channel.

So we don't compute any transform anymore. We simply set all the values lower than the low threshold to the lower threshold and the values higher than the high threshold to the high threshold. This transform is applied as follows: 1. Set all values lower than \$Th\_{low}\$ to \$Th\_{low}\$ and higher than the \$Th\_{high}\$ to \$Th\_{high}\$. Do this for every channel. 2. Normalize the image to [0, 255].

*Results:*

This procedure is way faster than k-Means. That's why it is our proposed solution in the end.

### 43.5. General architecture



General architecture

As seen in the picture the most important piece in our code is the `Anti_Instagram` node. It has the ability to get a raw image from the camera and calculate transform parameters out of it. So the core piece is the `ContAntiInstagramNode`.

#### 1) Anti Instagram node

Input parameters for the Anti instagram node: 1. `~ai_interval`: This sets the time in seconds between each computation of the color transformation. Default: 10

1. `~fancyGeom`: States what mask should be used to remove the background. If it is set to `false`, just one third from above is cut away to do the color transformation. If it is set to `true`, the flood fill approach is used. [Attention: Very time consuming and not stable results.] Default: `false`.
2. `~n_centers`: Indicates how many centers for k-Means should be used. Default: 6
3. `~blur`: Indicates what type of blur should be used. Options are '`median`' or '`gaussian`'. Default: '`median`'
4. `~resize`: Defines by what factor the picture should be resized before computing the transformation. Default: 0.2
5. `~blur_kernel`: Defines the kernel size of the blur. Default: 5
6. `~cb_percentage`: Defines the percentage of data points that should be cut away by the histogram equalization. Default: 2

7. *trafo\_mode*: Defines the transformation mode of the color transformation. Options are ‘both’ for histogram equalization first and kMeans after, ‘lin’ for kMeans only and ‘cb’ for histogram equalization only. Default: ‘both’

## 43.6. Formal performance evaluation / Results

In the end we have to admit that k-Means is probably too time consuming for an online approach. Or at least for an online approach with a fixed interval. Following you find computational time on the Duckiebot:

Algorithm:	Running Time [s]
Color Balance	0.0054
Linear 2 iterations, 10 centers	1.9725
Linear 2 iterations, 6 centers	0.6065
Linear 10 iterations, 10 centers	4.1626
Linear 10 iterations, 6 centers	1.3360
Linear 20 iterations, 10 centers	5.1680
Old Anti Instagram	3.1517

## 43.7. Future avenues of development

### 1) Annotated Data

This failed horribly. We didn't manage to get any annotated data since something didn't work with the website. For further improvement of the algorithm and a proper performance evaluation this would be crucial.

### 2) Remove unwanted background

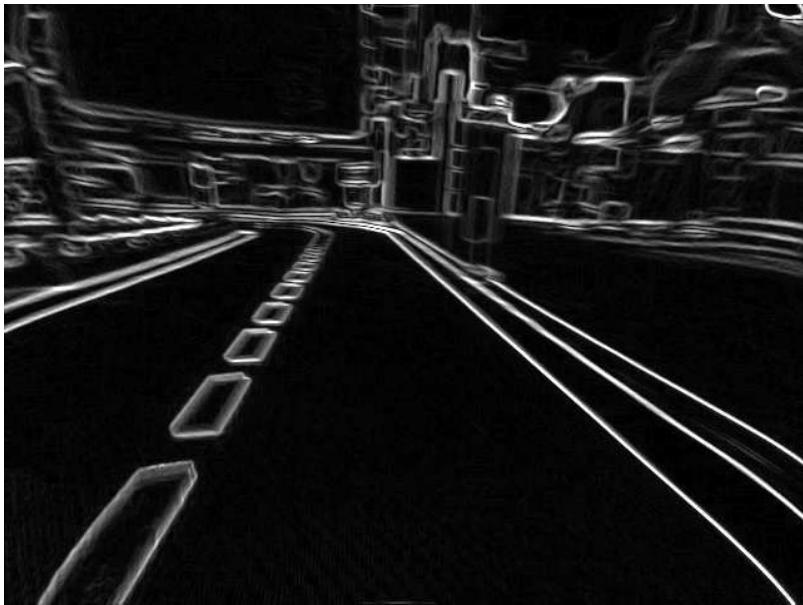
*Implemented in the geom.py file.*

*Idea:*

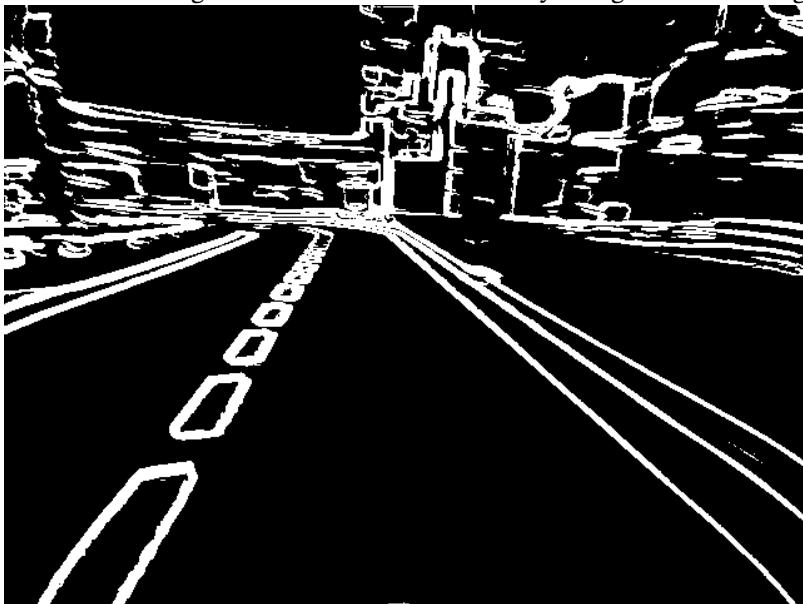
In order to make the k-Means approach faster one could try to remove unwanted background/irrelevant pixels. Since all the information except the road is of no use for the color transformation algorithm it can be removed. Up to now it just consumes more time and makes the algorithm incorrect. So if one could remove the unwanted background we could speed up k-Means because we don't have so many pixels anymore and because probably we don't need this many centers anymore. Since k-Means computational complexity is linear in both parameters this would definitely impact the computational time.

*Lane Surface Identification:*

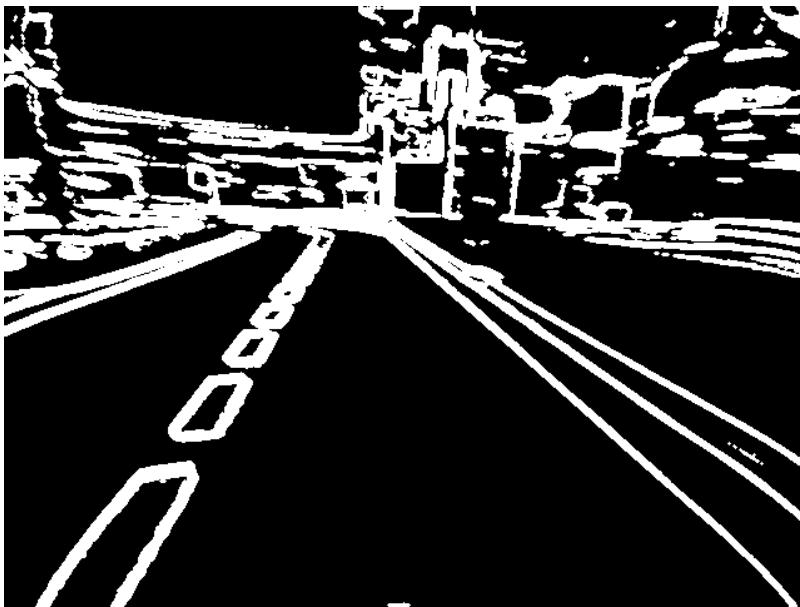
- Prior knowledge:
  - Assumption: Bottom  $\frac{2}{5}$  is lane, top  $\frac{1}{3}$  is not lane.
  - Pixels within a lane element have low gradient.
  - Pixels crossing boundaries, i.e. line edges, have high gradient.
- Detailed steps:
  - a. Compute gradient Compute the image gradient with Sobel operator. The result turns out to be better when done in RGB rather than HSV space.



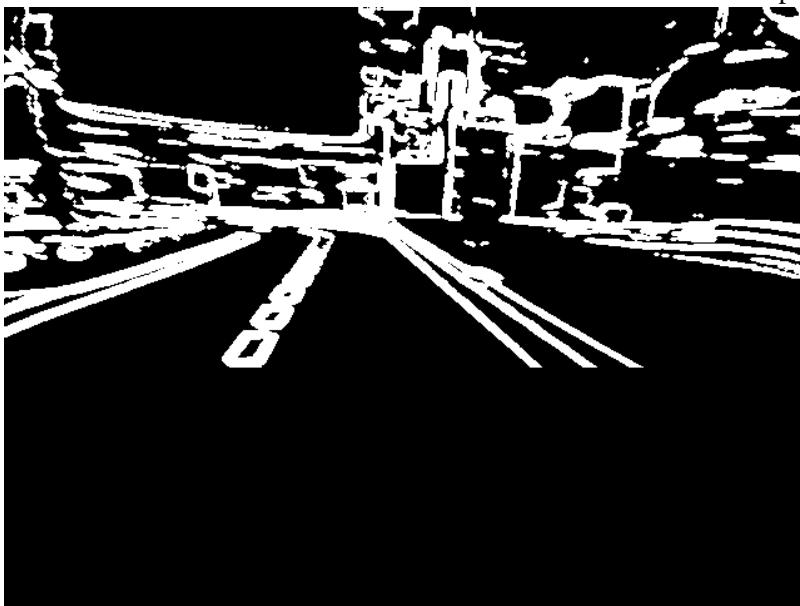
b. Threshold gradient This makes a binary image out of the gradient.



3. Dilate gradient This continues broken lines in the gradient, due to noise (e.g. from motion blur).



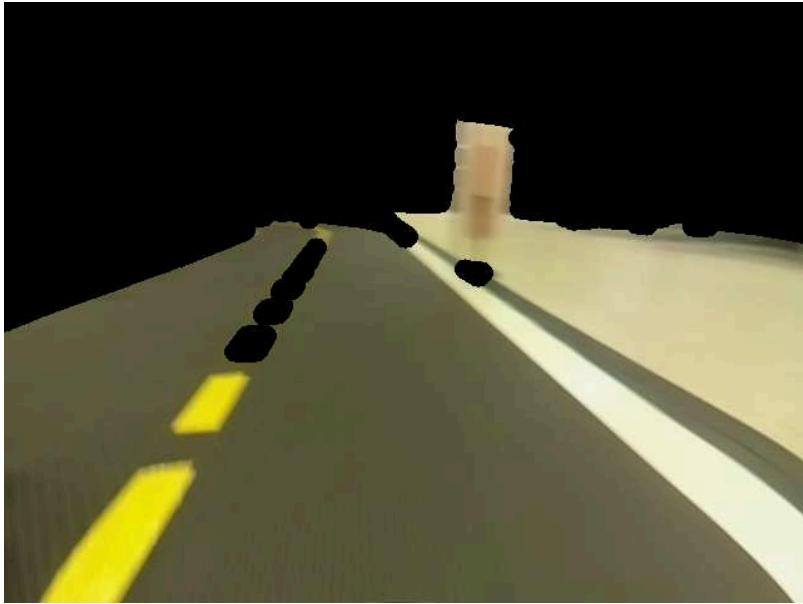
c. Zero fill bottom  $\frac{2}{5}$  This part is assumed to be lane surface. See next step for explanation.



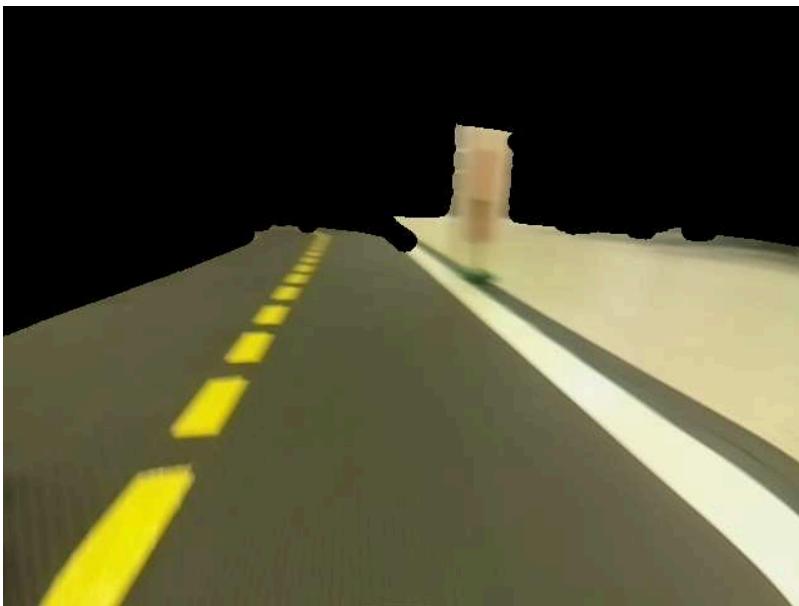
d. Floodfill to get mask This yield the single connected component of low gradient part. The seed is chosen from the bottom  $\frac{2}{5}$ .



e. Close narrow openings Regions inside narrow openings are probably high gradient part within the lane, and thus should be kept. We close it first, which is equivalent to dilation followed by erosion.



f. Fill the holes Then fill the resulting holes by floodfilling from the top, and take the complement.



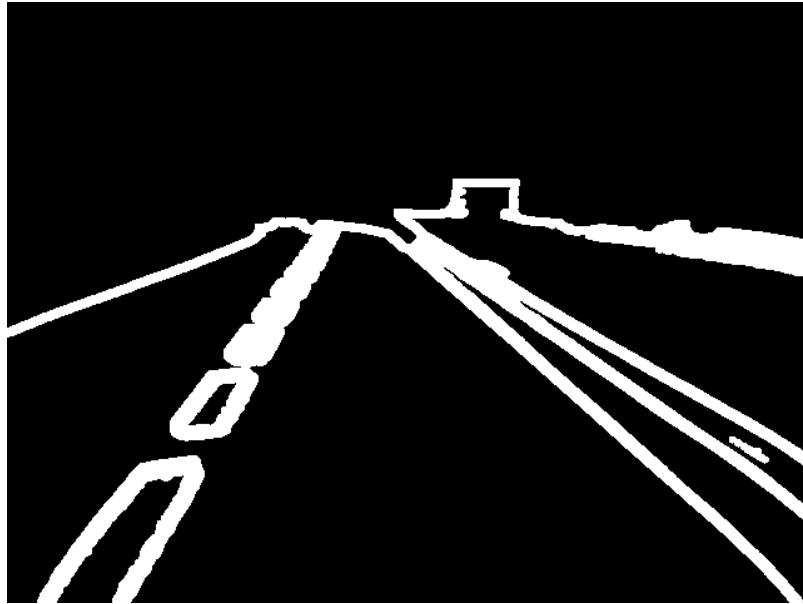
g. Clip off top  $\frac{1}{3}$ . This part are assumed to be not lane surface, so it gets clipped off from the mask.



#### *Boundary Region Detection:*

- Intuitions:
  - We observed that regions of boundaries between lane lines already contain sufficient color information. This is because the line detector only cares about the edges of lane lines, which is the entire reason we are performing anti-insta-gram.
  - In addition, restricting to boundary areas will further remove irrelevant portions of the image while substantially accelerating k-means computation.
- Detailed steps:

a. Compute, threshold and dilate gradient High gradient part corresponds to boundaries. Dilation is meant to cover more of the bordering pixels to get sufficient information.



b. Find contours as masks We want to capture the boundary areas, which can be found as contours in the gradient map above. This is done via the algorithm described in Suzuki, S. and Abe, K., Topological Structural Analysis of Digitized Binary Images by Border Following. CVGIP 30 1, pp 32-46 (1985) (implemented in OpenCV).



c. Remove small contours and fill in Contours that have small size are probably noise. Fill in the holes turn out to provide more information than noise.



The

pixels under this mask then get fed into the k-means algorithm.

#### *Existing code:*

There exists already some code for this idea. The basic concept is to use the flood fill concept. The output would be a mask which can be applied to every picture. But so far the generalization is difficult to make since the road geometry can easily change (e.g. in curves). And secondly this approach is quite time consuming as well.

A simple workaround is the following:

In general one can cut away the upper third of the picture to compute the color transform. One only need the road and its color for this procedure. So everything above the road is uninteresting for computing the color transform. The heuristic approach is cutting away the upper third. This is the default approach!

### **3) Two step k-Means**

---

Do the first transformation with  $n$  iterations and  $k$  centers. Then remember the  $k$  centers. For the next  $2, \dots, z$  images only start from the previous centers from image  $z_{\{i-1\}}$  to compute the next centers from image  $z_{\{i\}}$ .

### **4) Other clustering method**

---

Another clustering method like expectation maximization based on Gaussian mixture models would have more accurate results since some of the color clusters are not really spherical and definitively not of the same size.

### **5) White paper reference**

---

To determine when exactly the moment for the color transformation is a white paper reference would help. So an idea would be to have a small white hardware

piece in camera sight which is always exposed to the environment lighting conditions. There you could see when the best moment for a re-computation of a color transformation would be.

## 6) Polarization filter

---

If you would put a polarization filter in front of the camera we would get rid of the reflections of the tape. This was a big problem we heard of several teams.

## 7) Troubleshooting

---

Contact: [Christoph Zuidema](#)

# UNIT L-44

## Anti Instagram ReadMe

**TODO:** JT: move to appropriate code section.

This is the short description of anti-instagram software.

### KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Camera calibration completed. [Camera calibration](#)

### 44.1. Video of expected results



Figure 44.1. Anti instagram video

### 44.2. Duckietown setup notes

A duckietown with white and red solid lines, yellow dashed lines and black lane surface. No obstacles on the lane.

### 44.3. Duckiebot setup note

Any duckiebot satisfying the basic DB17 configuration will work.

### 44.4. Instructions to use the Anti-Instagram node

#### 1) Description of the Anti Instagram Node

---

Input parameters for the Anti instagram node:

- 1. *~ai\_interval*: This sets the time in seconds between each computation of the color transformation. Default: 10

1. *~fancyGeom*: States what mask should be used to remove the background. If it is set to *false*, just one third from above is cut away to do the color transformation. If it is set to *true*, the flood fill approach is used. [Attention: Very time consuming and not stable results.] Default: *false*.
2. *~n\_centers*: Indicates how many centers for k-Means should be used. Default: 6
3. *~blur*: Indicates what type of blur should be used. Options are '*median*' or '*gaussian*'. Default: '*median*'
4. *~resize*: Defines by what factor the picture should be resized before computing the transformation. Default: 0.2
5. *~blur\_kernel*: Defines the kernel size of the blur. Default: 5
6. *~cb\_percentage*: Defines the percentage of data points that should be cut away by the histogram equalization. Default: 2
7. *trafo\_mode*: Defines the transformation mode of the color transformation. Options are '*both*' for histogram equalization first and kMeans after, '*lin*' for kMeans only and '*cb*' for histogram equalization only. Default: '*both*'

#### 2) How to start the lane following

---

Step 0: On duckiebot, change to /\$DUCKIETOWN\_ROOT/ directory and check-out the lastest version of devel-anti-instagram branch.

Step 1: Run command:



```
$ source environment.sh && source set_ros_master.sh && source set_vehicle_name.sh  
$ roslaunch duckietown_demos lane_following.launch
```

Wait a while so that everything has been launched.

If you accidentally press R1 which starts autonomous lane following, press L1 to switch back to joystick control.

Step 2: On laptop, change to /\$DUCKIETOWN\_ROOT/ directory and perform the following steps:



```
$ source environment.sh  
$ export ROS_MASTER_URI=http://robot_name.local:11311/  
$ rqt_image_view
```

which opens a window to preview image messages. Select the /robot name/camera\_node/image/compressed to view camera image stream. Place the duckiebot somewhere in duckietown.

Step 3:

Run command:

```
rosparam set /robot name/line_detector_node/verbose true
```

so that line\_detector\_node will publish the result.

Step 4:

To view the result, select in rqt\_image\_view the topic /robot name/line\_detector\_node/image\_with\_lines. The anti-instagram node is continually running and will determine a proper transformation.

This can be seen as well with the following topics: *combination of ~corrected\_image and ~uncorrected\_image (shows the raw correction)* ~geomImage (shows the preprocessed image with the mask)

## 44.5. Troubleshooting

Contact czuidema@ethz.ch

UNIT L-45

## PDD - Distributed Estimation

### 45.1. Part 1: Mission and scope

#### 1) Mission statement

---

Enable Duckiebots to communicate with each other wirelessly

#### 2) Motto



UBI UNUM IBI OMNES  
(Where there is one, there is everybody)

### 3) Project scope

*What is in scope:*

**Communication in a centralized network:** *Communication between Duckiebots in one Duckietown Define interfaces for communication Performance testing on the communication system One network for one Duckietown \* TBD: does anything need to be synchronized?*

+ Resource error

I will not embed remote files, such as [https://github.com/duckietown/duckuments/blob/devel-distribution-est-fleet-wireless-communication/docs/atoms\\_85\\_fall2017\\_projects/16\\_distributed\\_est/Duckietown\\_Project\\_Image.png](https://github.com/duckietown/duckuments/blob/devel-distribution-est-fleet-wireless-communication/docs/atoms_85_fall2017_projects/16_distributed_est/Duckietown_Project_Image.png):

Figure 45.1. System Layout

**Option 1:** Communication in a centralized network with a redundant centralized component (multiple routers)

**Option 2:** Communication in a de-centralized network (ad-hoc)

*What is out of scope:*

- Shared network communication between Duckietowns in case of a centralized network
- Communication redundancy with another physical layer
- Data processing → Message data is just de-serialized and provided to other components

*Stakeholders:*

- Distributed Estimation
- Fleet planning
- Integration Heroes

## 45.2. Part 2: Definition of the problem

### 1) Problem statement

**Mission = Enable Duckiebots to communicate wirelessly**

**Problem Statement = Create a communication framework for the Duckiebots**

### 2) Assumptions

- Duckiebots can connect to a wifi network
- Data to be sent is already synchronized ??? TBD

- Duckiebots leave network when they are out of range or switched off
- If a Duckiebot is not connected to the communication network, it is not in Duckietown

### 3) Approach

---

**Step 0:** Define contracts with distributed estimation / fleet planning teams

**Step 1:** Create the communication framework for the duckiebots and test it on a centralized network

- Create wifi communication network (physical hardware e.g. wifi router, configuration, etc.)
- Create a software component that serializes data (create/format datapackets)
- Create a software component to send and receive messages to/from other duckiebots
- Integrate Serialization and Messaging software components into THE communication software component (e.g. into a ROS node)
- Testing:
  - Live visualization of bandwidth (and/or latency, etc. → Network performance measures) of network
  - Live visualization of network topology
  - Are any messages being dropped without arriving at their destination?
  - Etc.

**Step 2 (Optional):** Test the communication framework using a redundant centralized network

- Same as step 1, except for the wifi network → redundant centralized network

**Step 2 (Optional):** Test the communication framework using a de-centralized network

- Same as step 1, except for the wifi network → AD-HOC network

### 4) Functionality provided

---

- Enable duckiebots to join the network.
- Enable duckiebots to pack data and send it to other duckiebots in the same network.
- Enable duckiebots to receive and unpack data such that the data can be used in other software modules.
- Other functionality TBD

### 5) Resources required / dependencies / costs

---

**Bandwidth definition:**

- Number of duckiebots in the network
- Size of the messages
- Sending frequency
- Latency in message transmission
- Extra computation on duckiebots

## 6) Performance measurement

---

- Visualize the network topology → number of duckies
- Visualize messages (wireshark) → message size, latency
- Visualize HW resources → processor, memory, etc.

### 45.3. Part 3: Preliminary design

#### 1) Modules

---

**Libraries:** *Serialization Messaging*

**Integration:** *Libraries integration into component* Communication framework into Duckietown (repository, duckumentation, etc. → contact Integration Heroes) \* Creation of the WiFi network (centralized vs. decentralized)

**Testing:** \* Measuring and visualization of performance metrics

#### 2) Interfaces

---

**Fleet planning** *Send data (SLAM, etc.)* Distribute data through the network

**Distributed Estimation** *Receive data (local maps built from each Duckiebots)* Send data (local maps and/or global map)

**Integration Heroes** *Duckumentation* Contract negotiation \* External libraries (Protocol buffers, ZMQ, etc.)

#### 3) Specifications

---

- WiFi specs

#### 4) Software modules

---

- Library for Serialization
- Inputs:
  - Data to be serialized
  - Data to be de-serialized
  - Type definitions for serialization → Contract with distributed-estimation and fleet-planning teams
- Outputs:
  - Serialized data
  - De-serialized data
- Functionality:
  - Serialize data
  - De-serialize data
- Library for Messaging
- Inputs:
  - Destination
  - Port

- Type of socket
- Serialized data
- Optional: priority
- Outputs:
  - Serialized data
- ROS node for messaging
- Inputs:
  - Messages from ROS topics
  - Messages from other duckiebots
- Outputs:
  - Messages to ROS topics
  - Messages to other duckiebots

## 5) Infrastructure modules

---

- Wifi router for Duckietown
- Network configuration (centralized/decentralized)
- Testing
- Visualize the network topology → number of duckies
- Visualize messages (wireshark) → message size, latency
- Visualize HW resources → processor, memory, etc.
- ...

## 45.4. Part 4: Project planning

### 1) Project plan

---

*Week 9: 13/11/2017:*

- **Tasks:**
  - Project kick-off and planning
- **Deliverables:**
  - Preliminary Design Document

*Week 10: 20/11/2017:*

- **Tasks:**
  - Contract negotiation with the relevant groups
  - Research on testing, redundant centralized and ad-hoc networking
  - Design and implementation of Libraries
  - Design and implementation of ROS node
  - Configuration of centralized network
- **Deliverables:**
  - Contracts

*Week 11: 27/11/2017:*

- **Tasks:**
  - Find suitable tools for testing and test the ROS node (incl. libraries) using these tools
  - Configuration of redundant centralized network
  - Code reviews
  - Duckumentation
- **Deliverables:**

- Network configuration working
- Libraries (tested)
- ROS node (tested)
- First test results

Week 12: 04/12/2017:

- **Tasks:**
  - Ad-hoc networking
  - Duckumentation
- **Deliverables:**
  - Redundant centralized network working

Week 13: 11/12/2017:

- **Tasks:**
  - Ad-hoc networking
  - Duckumentation
- **Deliverables:**
  - Testing “framework” complete

Week 14: 18/12/2017:

- **Tasks:**
  - Ad-hoc networking
  - Duckumentation
  - Solve networking problems
  - Testing
- **Deliverables:**
  - None

Week “15”: 25/12/2017:

- **Tasks:**
  - Ad-hoc networking
  - Duckumentation
  - Solve networking problems
  - Testing
- **Deliverables:**
  - Communicating Duckiebots over ad-hoc network

Week “16”: 01/01/2018:

- **Tasks:**
  - Duckumentation and Buffer
- **Deliverables:**
  - Duckumentation

## 2) Task distribution

---

- Libraries (incl. Testing): Luca and Antoine
- ROS node (incl. Testing): Leonie
- Testing of the framework: Pat and Francesco
- Redundant centralized network: Pat and Francesco
- Ad-hoc networking: Leonie, Luca and Francesco

## 3) Data collection

---

TBD -> other groups

#### 4) Data annotation

---

No data to be annotated.

#### 5) Relevant Duckietown resources to investigate

---

WiFi specs (duckumentation)

#### 6) Other relevant resources to investigate

---

Suggestions on Slack channel: 1. [MAVLink \(born for UAVs also used for other robots\)](#) 2. [ROMANO \(not so good...\)](#) 3. [DDS \(standard for ROS 2.0\)](#)

Computer Networks:

- Introduction to computer networks: <http://intronetworks.cs.luc.edu/>

Serialization and Messaging:

- Protocol Buffers: <https://developers.google.com/protocol-buffers/>
- ZeroMQ: <http://zeromq.org/>

Google python coding style guide:

- <https://google.github.io/styleguide/pyguide.html>

Redundant routers (rollover, cascading):

- <http://www.tomshardware.co.uk/forum/21591-43-design-redundant-wireless-network>
- <https://www.linksys.com/ca/support-article?articleNum=132275> (to be verified)

Static code analysis in python:

- <https://www.pylint.org/>

Pylint configuration:

- <https://stackoverflow.com/questions/29597618/is-there-a-tool-to-lint-python-based-on-the-google-style-guide>

#### 7) Risk analysis

---

Possible Risks? Network problems (*ad-hoc: unstable network, low bandwidth, high latency, ...*) No ad-hoc network solution Sizable amount of redundant data sent over wifi (*chaos*) Synchronization

How to mitigate the risks? Synchronization not part of networking → contract Contracts to prevent redundancy

UNIT L-46

Distributed Estimation: intermediate report

**TODO:** JT: fix formatting, add figures?

## 46.1. Part 1: System interfaces

### 1) Logical architecture

---

Robots can broadcast messages to every other Duckiebot in the network (centralized network for sure, and we'll try to make ad-hoc networking work (as ad-hoc or mesh network)). Robots will post the received message data to the corresponding ROS topic.

Ideally the network should scale from 2 to a town of Duckiebots. The network should also be robust to Duckiebots actively leaving and entering the network.

We will measure the latencies and other performance metrics but do not plan on optimizing the performance in regards of it. This can be improved over the next iteration.

Fleet communication: 1. Assume modules that need to use the communication capabilities e.g. multi-robot SLAM, fleet planning are able to publish and subscribe to ROS topics. 2. Assume modules sends data of serializable type and reasonable size 3. Assume communication is not time critical on sub second scale 4. Assume modules self synchronize (if applicable)

### 2) Software architecture

---

messaging node:

**subscribed topics:** individual outgoing communication (and by outgoing communication, we mean messages we send over wifi) topics published by: - fleet planning (TBD) - Multi-Robot SLAM (TBD).

These teams publish their data to be sent to these topics.

**published topics:** individual incoming communication (and by incoming communication, we mean messages we get over wifi) topics subscribed to by: - fleet planning (flag\_fleet\_planning\_inbox) - distributed estimation (flag\_multi\_slam\_inbox) - maybe other groups, since anyone can subscribe to these topics

Outwards (wifi) communication is realized with protobufs and zmq

Optimizing for latency will be of low priority, since the primary single goal is to pipe through the data. If the data comes in faster than it goes back out from the ROS node, it shall be solved in a next iteration.

We're going to try and make the node configurable such that the code will not needed to be changed in the future (maybe just optimized, since it is quite complex and we do not have much time to implement it). If not, we will hard code the message conversion (ROS message  $\leftrightarrow$  ZMQ message).

How does this work?

Once:

- Team A wants to communicate between bots
- Team A tells us their message structure
- We build serialization
- We define ROS topics: teamAout, teamAin

Perpetually (as long as message structure doesn't change):

- Team A bot A posts message to teamAout
- We automatically serialize message with corresponding serialization
- We send on bot A
- We receive on all other bots
- We deserialize and post to teamAin
- Team A other bots can retrieve message from teamAin

## 46.2. Part 2: Demo and evaluation plan

*Please note that for this part it is necessary for the VPs for Safety to check off before you submit it. Also note that they are busy people, so it's up to you to coordinate to make sure you get this part right and in time.*

### 1) Demo plan

---

Multi-Robot SLAM and fleet planning relies on communication to work, therefore the communication demo is implicit in the SLAM and fleet planning demo. We think a sole communication demo would not be too impressive to the casual demogoer.

At least three Duckiebots (configured for mesh networking i.e. with additional wireless adapters installed, if it works.) Otherwise, a global network (e.g. Duckietown) for the centralized structure.

### 2) Plan for formal performance evaluation

---

There are three main criterias that have to be evaluated: 1. Message transport: 1. Centralized Network: test if a simple message e.g. a string can be sent from one duckiebot to another reliably. 2. Decentralized Network: test message propagation. In a mesh network two Duckiebots (nodes) may not be directly connected, therefore we must test if a message can be propagated through the network to the correct receiver. 3. Check of dropped messages 2. Network traffic: accurately monitor network traffic 3. Network topology: visualize nodes entering and leaving the network reliably

	0	1	2	3	4
Message Transport	Messages cannot be sent	Strings can be serialized	Messages can be re-serialized	Messages can be multicast	Messages can be received

0	1	2	3	4
tcp	on pgm	on pgm	on a mesh	net- work

Network Traf-No trafficCan see trafficAble to iso-Able to identi-Able to visu-fic monitored on the net-late duck-fy specificalize routing work but noiebot traffic packets of specific useful infor-mation ex-tracted packages

Network Network Initial Net-Initial Net-Initial Net-Topology cannot bework can bework can bework can be (centralized established establishe, established, established, established, established, established, established, established) but no newnew nodesnew nodesnew nodes nodes cancan connectcan connect/can connect/ connect to thebut not reli-leave dynami-leave dynami-network, notably, not ro-cally, but notcally, and ro-robust to con-bust to con-robust to con-bust to con-nection loses nection loses nection loses nection loses

### 46.3. Part 3: Data collection, annotation, and analysis

*Please note that for this part it is necessary for the Data Czars to check off before you submit it. Also note that they are busy people, so it's up to you to coordinate to make sure you get this part right and in time.*

#### 1) Collection

---

Initially a set of compiled dummy messages is used to build the fleet-communication. For further implementation and evaluation one ROS-bag of broadcasted messages is needed from the fleet-planning team and from the multi-robot-SLAM product each.

#### 2) Annotation

---

For the fleet-communication no data annotation is needed.

#### 3) Analysis

---

### UNIT L-47

## Distributed Estimation: final report

**TODO:** JT: fix images, formatting

## 47.1. The final result

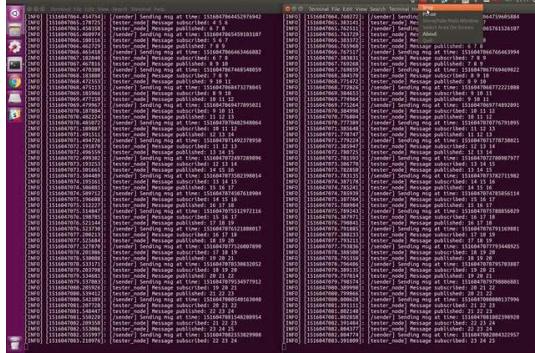


Figure 47.1. The Distributed estimation, a.k.a. Fleet messaging, Demo Video

see the [operation manual](#) to reproduce these results.

## [README](#)

## 47.2. Mission and Scope

With this project we enable Duckiebots to communicate with each other on centralized and decentralized wireless networks.

### 1) Motivation

In the previous state of Duckietown, Duckiebots were individual, autonomous agents, roaming around Duckietown with no way to communicate with each other explicitly (the only method in existence is with the help of LED patterns, resulting in long interpretation times). Since the ultimate goal being an automated taxi system: Duckiebots working together picking up and dropping off customers in the optimal way; the Duckiebots need to be able to communicate with each other efficiently.

One important part of this communication setup is that it can be decentralized, using a mesh network and Duckiebots can join and leave the system without putting the whole network at risk of failing. This also allows for the network to be scaled, given network limitations.

Due to the current state of Duckietown, the communication is needed, but not limited to, fleet planning control to coordinate the fleet in a town with a predefined map.

### 2) Existing solution

There was no prior work to build a communication system upon. Everything was implemented from scratch.

### 3) Opportunity

---

Without any existing work on wireless communication, we came up and built a whole new addition to Duckietown. We implemented a fleet-messaging package that builds an ad-hoc mesh network and lets other teams send messages

### 4) Preliminaries

---

We specifically picked libraries and modules that encapsulates their respective functionalities well. Therefore to fully understand what is going on under the hood, you simply need to read up on the documentation of each package used: - [batman-adv](#) - [zeroMQ](#) - [protobuf](#)

#### 47.3. Definition of the problem

The final goals of the project were to: 1. Create a robust wireless network that can easily be scaled to a larger fleet size and to a bigger Duckietown. 2. Build a communication framework for the Duckiebots that enables the sending and receiving of messages to and from any Duckiebot, which is connected to the above mentioned network. 3. Have a communication framework that is reusable and scalable.

For this we made the following assumptions: 1. Duckiebots can connect to a wifi network. 2. Duckiebots leave network when they are out of range or switched off. 3. If a Duckiebot is not connected to the communication network, it is not in Duckietown. 4. In a first step, the communication network is used by the fleet-planning team.

To evaluate the new framework we: 1. Compared the messages sent and received between two Duckiebots connected over the network and looked for messages dropped. 2. Tested the range of the wifi adapters to see if it is able to cover the size of a demo-sized Duckietown. 3. Test the robustness of the network by taking a Duckiebot out of range of the network and back and restarting the Duckiebot in to see if it would reconnect.

#### 47.4. Contribution / Added functionality

##### 1) Added Functionailities

---

Added Functionalities are as follow: - Duckiebots are now able to communicate with one another directly without a central station to relay the messages. - The network has no centralized point of failure. If a Duckiebot fails (runs out battery, breaks, disconnects abruptly) for any reason, the rest of the Duckiebots remaining in the system are still able to communicate without any noticeable disruption, given they are in range of each other.

Consequently, we believe we have achieved the gold medal outcome for this project: "Ad-hoc networking".

## 2) Package Infrastructure

---

The package infrastructure is as follows: 1. A mesh network that dynamically connects all of the Duckiebots that are currently in Duckietown. 2. A messaging algorithm that allows the sending and receiving of messages over this network. 3. A message encoder that packs the data before it is sent. 4. A message decoder that unpacks the data after it is received. 5. A framework that allows other Duckietown packages to send and receive messages of their defined types.

These individual parts were implemented as described below.

### *Mesh Network:*

Batman-adv is the backbone of the mesh network. In short, it is a specialized linux kernel module that implements a network routing protocol. It emulates a virtual network switch of all nodes participating. Hence, all nodes appears to be linked locally and are unaware of the network's topology and is also unaffected by any network changes. Once setup properly, batman-adv manages the mesh network for us.

### *Messaging Algorithm:*

DuckieMQ is based on zeroMQ, a framework used to send messages over sockets. The serialized messages (protobuf) are broadcasted on a specified port into the network. For this to work, we need to know the name of the network interface, the desired port initialization of a messaging socket, which then can either be used as receiver or sender. Multiple sockets can and usually will run on one bot. Also because we use a multicast protocol (epgm) multiple sender and receiver sockets can run on one port.

Moreover, messaging features of the platform is decoupled from the implementation of the network architecture.

### *Message encoder and decoder:*

In order for the messages to be sent and received, they have to be encoded into ROS [ByteMultiArrays](#). A serialization library was implemented to pack the ROS messages into a byte array such that the data could be sent through a zeroMQ message. After the message is sent, the data is then parsed back into a ROS message and published to the correct inbox\_topic specified by the package that sent the message. We chose to use ByteMultiArray for its flexibility and because it is a std\_msg of ROS. This means that other packages must only publish ByteMultiArray to fleet messaging.

### *Framework:*

For easy use of the messaging algorithm, a ROS package with two ROS nodes was implemented. The two nodes are the receiver\_node and the sender\_node.

The sender\_node subscribes to the outbox\_topic and sends this data to the receiver\_node on all other Duckiebots on the network via the messaging algorithm using zeroMQ. The receiver\_node then publishes the received data to the inbox\_topic.

To use the framework, one simply has to publish to the ROS topic `outbox_topic` (specified by the config file) and subscribe to the `inbox_topic` (also specified by the config file) and listen to the specified message port.

The complete structure of the fleet-messaging package is illustrated below.  
System Infrastructure

+ Resource error  
I will not embed remote files, such as [https://github.com/duckietown/duckuments/blob/devel-distribution-est-fleet-wireless-communication/docs/atoms\\_85\\_fall2017\\_projects/16\\_distributed\\_est/Simple%20Fleet%20Messaging%20Flow%20Diagram.png](https://github.com/duckietown/duckuments/blob/devel-distribution-est-fleet-wireless-communication/docs/atoms_85_fall2017_projects/16_distributed_est/Simple%20Fleet%20Messaging%20Flow%20Diagram.png):

## 47.5. Formal performance evaluation / Results

There are three main criterion that have to be evaluated: 1. Message transport: 1. Centralized Network: test if a simple message e.g. a string can be sent from one duckiebot to another reliably. 2. Decentralized Network: test message propagation. In a mesh network two Duckiebots (nodes) may not be directly connected, therefore we must test if a message can be propagated through the network to the correct receiver. 3. Check of dropped messages. 2. Network traffic: accurately monitor network traffic. 3. Network topology: visualize nodes entering and leaving the network reliably.

To test the first criteria: - Compared the messages sent and received between two Duckiebots connected over the network and looked for messages dropped

To test the second criteria: - Analyse packets sent on wireshark

To test the third criteria: - Tested the range of the wifi adapters to see if it is able to cover the size of a demo-sized Duckietown - Test the robustness of the network by taking a Duckiebot out of range of the network and back and restarting the Duckiebot to see if it would reconnect.

Our conclusions are summarized in the following table that was established for evaluation during the project phase. The performances in bold letters were the states the messaging-package achieved at the end of this project.

	0	1	2	3	4
Message Transport	Messages cannot be sent or re-ceived on tcp	Strings can be sent and re-be received on tcp	Messages can be serialized and sent on and received on multicast	Messages can be serialized and sent on and received on and multicast	Messages can be serialized and sent on and received on a mesh network

Network Traf-No trafficCan see trafficAble to iso-Able to identi-Able to visufic monitored on the net-late duck-fy specificlize routing work but noiebot traffic packets of specific useful infor mation ex tracted packages

	0	1	2	3	4
Network Topology (centralized and decentralized)	Network Initial cannot be established, but no new nodes	Net-Initial bework can be established, but new nodes	Net-Initial bework can be established, can connect to the network, notably, robust to connection loss	Net-Initial bework can be established, can connect to thebut not reli-leave dynami-leave network, notably, not ro-cally, but notcally, and robust to con-bust to con-robust to con-bust to con-bust to connection loses	Net-Initial bework can be established, can connect/can connect/dynam-leave network, notably, not ro-cally, but notcally, and robust to con-bust to con-robust to con-bust to connection loses

## 47.6. Future avenues of development

### 1) One push solution for package setup/installation

*Current Issue:*

Mesh networking can be very finicky because it depends on drivers for the wifi adapters and batman-adv working correctly. From experience, even with the unified Duckiebot hardware this was very much a case by case basis. This made it difficult to develop a one push solution. Nonetheless we implemented one - see operation manual - which seemed to work on most occasion but we still had to do some on the spot debugging.

*Possible Solution:*

Use wireless adapters where the mesh network is currently working (edimax). A refined the bash script already implemented.

### 2) Improve Developmental setup

*Current Issue:*

When developing this package we needed three networks running simultaneously: one connected to the internet (for git purposes), one connected to the local network created by the Duckiebot (for ssh), and lastly the mesh network itself.

*Side note: Technically you can ssh into the Duckiebot through the mesh network, however this becomes problematic if you want to debug the mesh network itself. If your mesh network doesn't work then you can't ssh into your Duckiebot anymore. That is why it is better - at least when developing - to have three different networks running.*

*Possible solution:*

There are already two wifi adapters on the current configuration, and there lies the problem. There are two workarounds in use currently: 1. Use a centralized network created by an access point. This allows you to both ssh and develop your communication platform. However, it is no longer a decentralized mesh network. 2. Use an additional, third (mesh capable) network adapter strictly for mesh networking. There are several drawbacks. Firstly, this means that you need an additional wifi adapter the Duckiebot totalling up to three. Secondly, you also need one

for your laptop to connect to the mesh network if you don't want to use your in-built adapter.

Both workarounds have their drawbacks, so it would be preferable to find a robust solution for laptops connecting to the mesh network.

### 3) Reducing the number of additional hardware

---

#### *Current Issue:*

Following from the last point, adding an additional wifi adapter is costly.

#### *Possible Solution:*

We discovered late into the project that the edimax has mesh capabilities. We tried it and found that it works but never fully tested it to a point that we were confident with its viability.

*Remark: There were some driver problems with some WiFi adapters with respect to mesh network capabilities. It works with the edimax, so it might be of advantage to have two edimax adapters: one for the duckiebot and one for the laptop. With this setup, the edimax adapters can be used to create the mesh network (and the connected laptops would be a part of the network as well). It is also important to know that at this moment, it is not possible to get a connection to the internet through the duckiebot via the mesh network.*

### 4) Improving usability of platform

---

ZeroMQ allows to filter messages according to strings (eg. botname) at the start of messages. Our duckieMQ implementation already supports filtering, however it has not been used by the teams yet as the whole setup gets cumbersome to a certain degree if it has to be implemented on every bot. Also serialized messages need to be manually prepended by the filterstring. It would be useful to extend duckieMQ with the capability to prepend a filterstring to serialized messages automatically.

The configuration files for the different channels are somewhat cumbersome to create for bigger groups of bots. It would be preferable if the bot could create its own config file according to rules laid down in a central config file where all the different groups specify their communication architecture.

E.g fleet level planning want every bot to listen to port 23334, filter it with their name and publish to /taxi/commands and subscribe to /taxi/location and send it on port 23333.

As a final step we could let the config file generator handle ports by itself, so there is no need to keep track of used ports.

### 5) Network Visualization

---

#### *Current Issue:*

With the current implementation there is no way to visualize the topology of the

network. This was very possible but unfortunately we ran out of time.

#### *Possible Solution:*

A very useful function would be to implement a real time visualization of the network. To visualize the network involves installing batadv-vis. Batadv-vis can be used to visualize the batman-adv mesh network. It reads the neighbor information and local client table and distributes this information via alfred - a user-space daemon for distributing arbitrary local information over the mesh/network in a decentralized fashion - in the network. By gathering this local information, any vis node can get the whole picture of the network. But this would have only taken us half the way there as it only gave static snapshots of the network. So to improve on this would be to continuously update/generate the graph so it appears to be live.

## UNIT L-48

### Demo instructions Fleet Communications [draft]

This section has been removed because it is in status 'draft'. [If you are feeling adventurous, you can read it on master.](#)

To disable this behavior, and completely hide the sections, set the parameter show\_removed to false in fall2017.version.yaml.

## UNIT L-49

### Transfer learning: preliminary report

#### 49.1. Part 1: Mission and scope

##### 1) Mission statement

---

The goal is to test transfer learning algorithms trained on simulator and test on the real duckietown.

Solving control problems in reality is hard due to sparse reward signals, expensive data acquisition and the danger of breaking the robot during exploration. It is comparatively easier to train the policy in a simulator as we can “speed up” the reality, and there are no inherent dangers of running arbitrary policies. But policies trained on simulator do not necessarily transfer directly onto the real world, and our goal is try and bridge this gap

##### 2) Motto

---

IPSA SCIENTIA POTESTAS EST,

UT TRANSIRE CALLIDUS EST

(Knowledge is power, transferring that is clever)

### 3) Project scope

Final goal: Implement [DARLA](#) or <https://arxiv.org/pdf/1703.06907.pdf> or <https://arxiv.org/pdf/1710.06537.pdf>

*What is in scope:*

Testing out different RL algorithms, (or unsupervised feature learning algorithms, if any)

Methods for transferring from simulator to duckietown

Baselining (?)

*What is out of scope:*

Hardware modifications

Design modifications that might be needed

Anything not involving the control of the duckiebot

*Stakeholders:*

Simulator (Maxime and Florian)

Supervised Learning (Rithesh and Alex)

Running NN models on the bot (Neural stick ?) who is in charge?

File with all projects [click](#)

## 49.2. Part 2: Definition of the problem

### 1) Problem statement

We will replace the part of the pipeline that uses raw images and give motor controls by using a model trained in a simulator. The model policy will be able to do lane following and navigation. The model can hopefully generalize to imperfect camera calibration, motor calibration and different light conditions.

Mission: Efficient transfer of algos trained on simulators

Problem statement:

- Implement [DARLA](#) or
- [Domain Randomization Paper](#) or
- [Dynamics Randomization Paper](#)
- Apply the algorithms for tasks like, navigation.

## 2) Assumptions

---

We need the simulation to be close enough to real world so as to be transferable.

We need to run inference on the robot at test time.

Maybe allow for fine-tuning in the real world (on a small dataset)

## 3) Approach

---

- Get the simulator up and running
- Start with the most basic lane following environment: A straight lane...
- Train a model to control the duckiebot in the simulator
- Transfer the policy to the real world
- Move onto slightly more complicated scenario ... repeat (curriculum learning setting)

## 4) Functionality provided

---

Lane following or route following (i.e. follow a given route through duckietown and obey traffic laws)

Metrics:

- Quality of navigation (as defined by a reward/loss function)
- Quality of transfer defined by the above reward function, computed automatically or by hand
- Finetuning needed for transfer?

Reward function description (probable):

- Deviation from center (-ve)
- Time taken (-ve)
- Finish position (+ve)
- Collision (-ve)
- Violating traffic rules or conventions (-ve)
- See [Socially Aware Motion planning](#)

## 5) Resources required / dependencies / costs

---

- # of cpu/gpu-hours for training
- Test time computation costs
- Duckiebots/ duckietown

## 6) Performance measurement

---

- In simulation, use access to the true state to compute the reward function
- In duckietown, compute the reward function by hand (or develop a heuristic for computing it)
- Qualitative comparison to current control pipeline
- Baseline wrt other RL algos
- Compare with #devel-super-learning for performance wrt imitation learning policies

### 49.3. Part 3: Preliminary design

#### 1) Modules

---

Module 1: Policy mapping raw images → actions (or Module 1a: policy → disentangled representation → actions)

Module 2: Actions → motor voltages (Joy Mapper node)

Module 3: Training module (Also introduces domain/dynamics randomization in the simulator)

#### 2) Interfaces

---

Module 1: subscribe to raw camera images and publish actions (forward/backward/turn left/turn right = float values)

Module 2: already provided in duckietown stack

Module 3:

- Input: simulator parameters, model
- Output: a trained policy to be used in Module 1

#### 3) Specifications

---

See Modules and Interfaces above.

#### 4) Software modules

---

During training, the modules will be written in Python.

At test time, modules 1,2 will be deployed as a ROS node during test.

#### 5) Infrastructure modules

---

Simulator

### 49.4. Part 4: Project planning

## 1) Data collection

---

- Real world cam pictures corresponding to simulator states (e.g in front of a straight lane/at the beginning of a left/right turn/...) : can be useful if we want to assess the quality of the disentangled representation without the need to run it on the robot.

- Use the simulator to generate training data for RL algorithms

## 2) Data annotation

---

The simulator will annotate data automatically by providing ground truth information about the duckiebot and the environment

+ comment

To be confirmed... We probably need to implement a module that does that.

If needed, then semantically segmented images would be useful

*Relevant Duckietown resources to investigate:*

Simulator

*Other relevant resources to investigate:*

PyTorch rl codebase, OpenAI Gym, OpenAI Baselines

DARLA: <https://arxiv.org/abs/1707.08475>

Transfer by randomization/generalization:

- <https://arxiv.org/abs/1703.06907>
- <https://arxiv.org/abs/1710.06537>

UNREAL: <https://arxiv.org/abs/1611.05397>

Socially Aware Motion planning: <https://arxiv.org/abs/1703.08862>

## 3) Risk analysis

---

Deploying the policy trained in simulator will break the bots in real duckietown.

Risk mitigation:

- Train it properly
- Curriculum learning setting to gradually increase difficulty
- Safety on/off switch
- Cushion the sides of the lane

Page left blank