The Autolab

KNOWLEDGE AND ACTIVITY GRAPH

Requires: A fully operational Duckietown.

Results: A working Autolab.

Contents

Warning: Autolabs are still experimental. We are working on preparing a set of instructions for you but it is still all in its beta version.

The Big Picture

The goal of an Autolab is to create a human free, automated environment for Duckiebots. This requires a bit more than just a standard Duckietown (unknown ref opmanual_duckietown/duckietowns)

warning next (1 of 9) index

warning

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

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

Location not known more precisely.

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

to work.

An Autolab can serve different purposes. It is used for research on the Duckietown platform, and is also being used to organize the embodied challenges of AIDO.

The aim of this book is to present:

- a set of instructions (labeled BUILDING) on how to build an Autolab to current specifications
- a set of instructions (labeled DEMO) on how to run the Autolab functions or demos
- a set of guides (labeled SOFTWARE) to the different pieces of software that are used and are under development

1) Sections

In each of the following sections you will find the BUILDING, the DEMO and the SOFTWARE pages.

You can find the up-to-date status of each page of the book at the top-right of the page, that will change as we write them down. Please only refer to those marked as either BETA or READY.

- Part A - Autolab definition : The precise definition of an Autolab
- Part B - Autolab minimum requirements : The set of instructions to setup the minimal working environment
- Part C - Auto charging : The set of instructions to build and operate an autocharging station
- Part E - Localization System : The set of instruction to build and operate the localization system
- Part G - Autolab operation manual : The set of instructions to operate the whole Autolab

> Note: Autolabs are currently under development. The naming is not fixed yet, and so the terms “Robotarium” and “Autolab” may have been used interchangeably.

PART A
Autolab definition

excerpt:

KNOWLEDGE AND ACTIVITY GRAPH

Requires: A Duckietown up to specifications from the Duckietown book (unknown ref opmanual_duckietown/duckietowns)

previous **warning** next (2 of 9) index

warning

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

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

Location not known more precisely.

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

.

Results: Knowledge of the fundamental structures that make an Autolab.

Next: Building the autocharging area.

An Autolab is a Duckietown with a set of additional structures:

- A map
- A fleet of Autobots*
- A maintenance area
- A localization system
- An Autolab operation server

*The Duckiebots that are used inside an Autolab are called Autobots and rely on additional specifications.

2) The Map

For an Autolab to work, a precise map is needed. This part is handled in Unit B-4 - Autolab map

3) The fleet of Autobots

An Autolab is nothing without its fleet of Autobots. The Autobots are Duckiebots improved with different parts, mainly for autocharging and localization. They also follow a strict procedure of calibration, and we keep a log of all events that happen throughout the life of an Autobot. This is the role of the fleet roster.

4) The maintenance area

The maintenance area is an area that is designed to include the activities of an Autolab

which are not related to the self-driving goals of Duckietown.

Right now, it is only comprised of the auto-charging area, but could be extended to include, say:

- A parking lot
- An auto-calibration arena

It is accessible via a single entry point (see Figure 0.1).

This area is critical to the full automation of the Autolab, because it will allow for the Autobots to go around indefinitely and recharge themselves, without any human intervention.

5) The localization system

The localization system is comprised of a set of watchtowers distributed in the city and a server to process the data and extract poses of all Autobots. The stream of images from watchtowers also finds use in many other projects than localization system.

6) The Autolab operation server

The Autolab operation server, described in the autolab operation manual and currently under development, is and will be the human interface to control the high level functions of the Autolab, to monitor its status and activity, and to launch experiments.

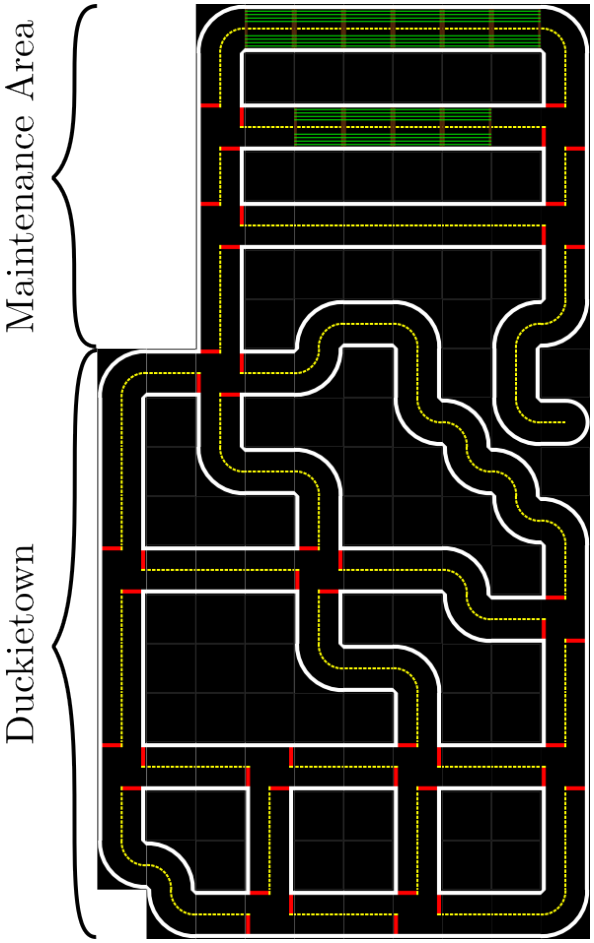


Figure 0.1. Sample map for an Autolab

Keywords: robotics, Autolab, auto-charging, charging, maintenance

PART B
Autolab minimum requirements

KNOWLEDGE AND ACTIVITY GRAPH

Requires: A Duckietown up to specifications from the duckietown book (unknown ref opmanual_duckietown/duckietowns)

previous **warning** next (3 of 9) index

warning

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

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

Location not known more precisely.

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

.

Results: Your software environment is setup.

Next: Learn about Autobot specifications.

Contents

Prerequisite recommendation

If you are building an Autolab, you are a committed developer of Duckietown. We strongly recommend using Ubuntu 18.04. To set up the computer, follow the instructions here.

We suggest that your workflow for software should follow the standard one, using `dt-env-developer` . This is a meta repository that aggregates, with the myrepos tool, the different repositories of code that developers use daily. To set it up, follow the README instructions. The rest of this book will assume that you set it up this way, but you could also clone each individual repository on its own and use it as usual.

UNIT B-1

Autobot specifications and assembly

KNOWLEDGE AND ACTIVITY GRAPH

Requires: a Duckiebot in DB18 configuration

Requires: additional hardware depending on configuration

Results: a Autobot ready to be detected by the localization system

Next: if you have an autocharging area, add the parts for autocharging

1.1. Flashing the autobots

For a functional autolab, we strongly recommend using hostnames of the type :

- autobotXX (with XX starting at 01)

The autobot part can be changed, but the numbers at the end make the whole autolab code easier to use and maintain.

To flash the autobots sd cards, follow the instructions here.

1.2. Adding hardware to the bot

Warning: If you want to use the autocharging feature, you will have to swap the Raspberry Pi to the top part of the Duckiebot. This is due to additional hardware that will be added. So your Duckiebot should look like Figure 1.1. As the batteries do not fit underneath the chassis you will have to get creative about where to place the battery.

Note: If you want to be super safe you might want to add a Pin protector to ensure the pins will not be shorted when the Raspberry Pi is mounted on top of your Duckiebot. You can 3D print the protector with the link provided above or if you do not have a 3D printer available, use one of the many websites that offer 3D printing services such as sculpteo.

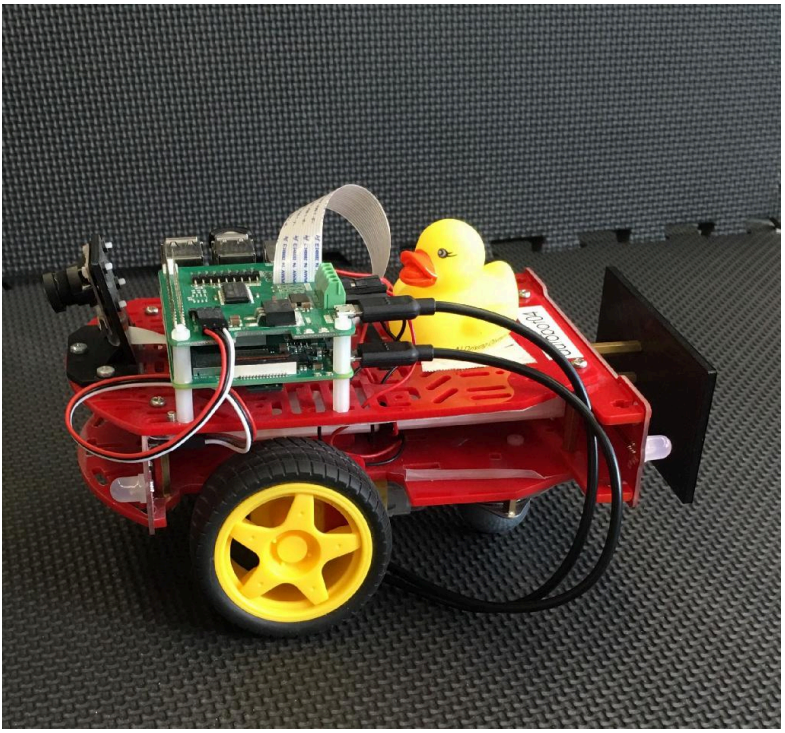


Figure 1.1. Pre-flight setup of the Duckiebot if you want to use autocharging.

1.3. For localization

1) Material needed for the localization standoff

- 4 standoffs M3 x 50mm for example from here
- 8 screws M3 * 8mm
- laser cut top plate
- printed April tags

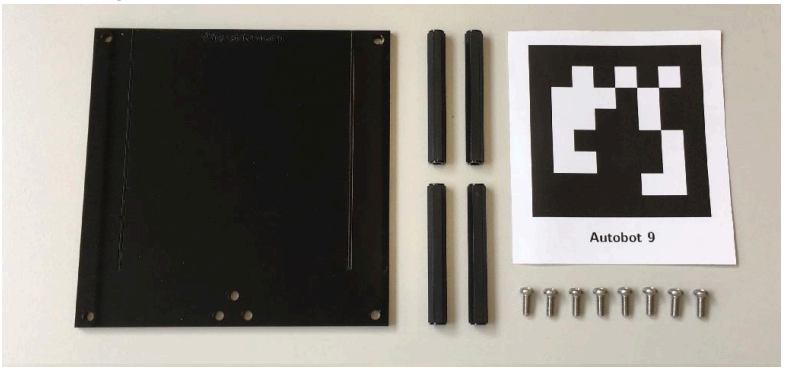


Figure 1.2. Material needed for the localization standoff.

2) Assembly

> **Note:** Do not get confused by having the Raspberry Pi on top in the instructions. If you do not plan to use autocharging, there is no need to have it on top.

First of all, attach the screws and standoffs to the Duckiebot. It is important that you use the same mounting position as we do in Figure 1.4 because the localization system assumes the April tag to sit on a certain spot of the Autobot. If you place it differently, the localization will be imprecise.

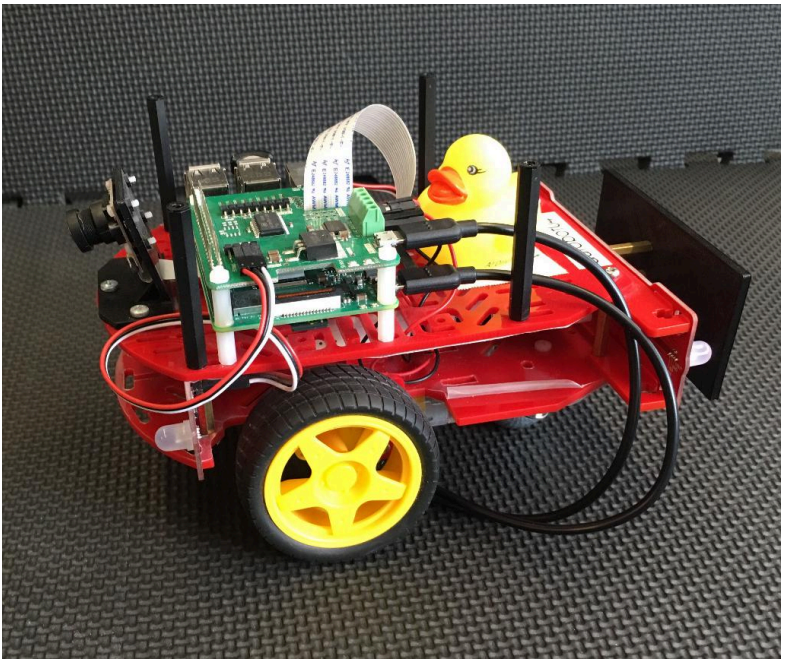


Figure 1.3. Standoffs mounted on the Autobot.

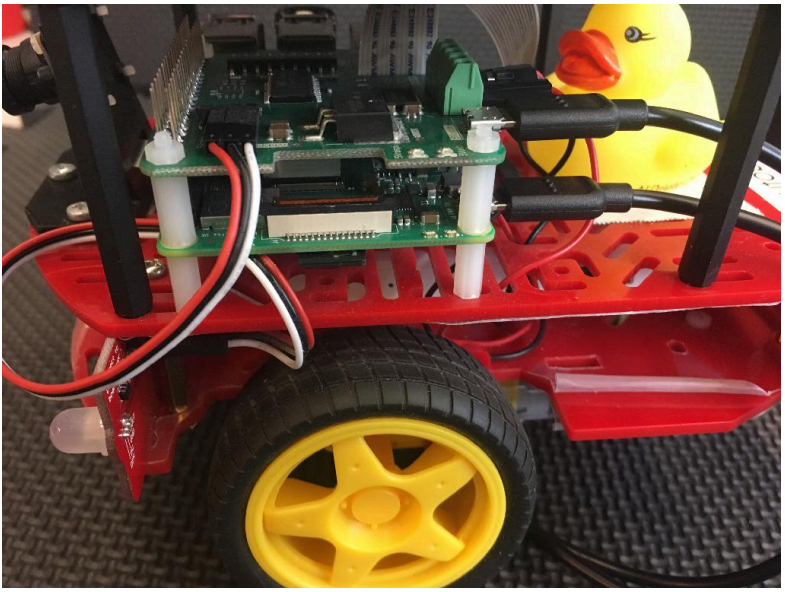


Figure 1.4. Spacers are moved to the very front of tghe mounting holes.

Attach the printed April tag on your top plate. TIP: print the April tags directly onto self-attaching paper so you don't need to add any glue or similar and can easily remove the April tags if needed. Here it is also important that you attach the April tag the exact same way as we do in Figure 1.5 for the same reason as above. The April tag needs to be aligned with the engraving and the front edge and have the same orientation.

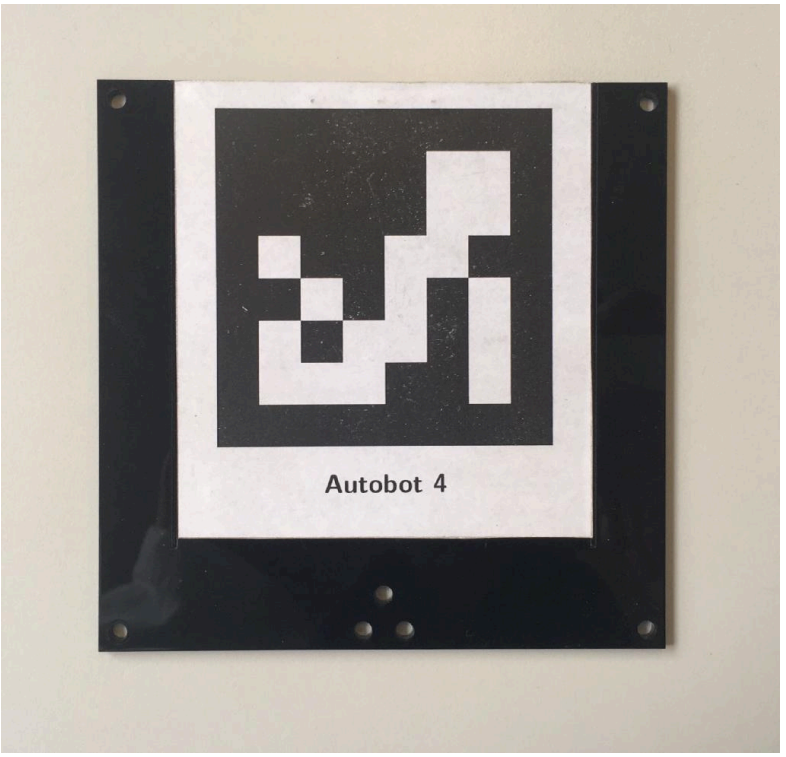


Figure 1.5. April tag mounted on the top plate.

You can now attach the top plate to the Autobot. The result should look like Figure 1.6

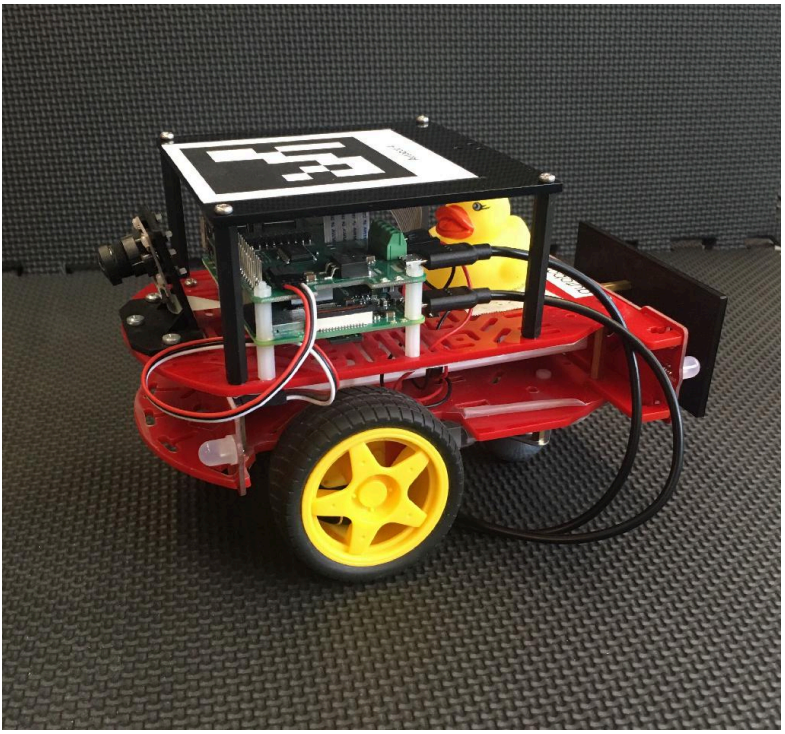


Figure 1.6. Autobot ready to be detected by the localization system.

1.4. For autocharging

In order to let a Duckiebot charge in a charger, additional hardware is needed. This piece is called the current collector.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Material for a single current collector Subsection 1.4.1 - Material needed for a single current collector

Results: A Duckiebot capable of charging Figure 1.7.

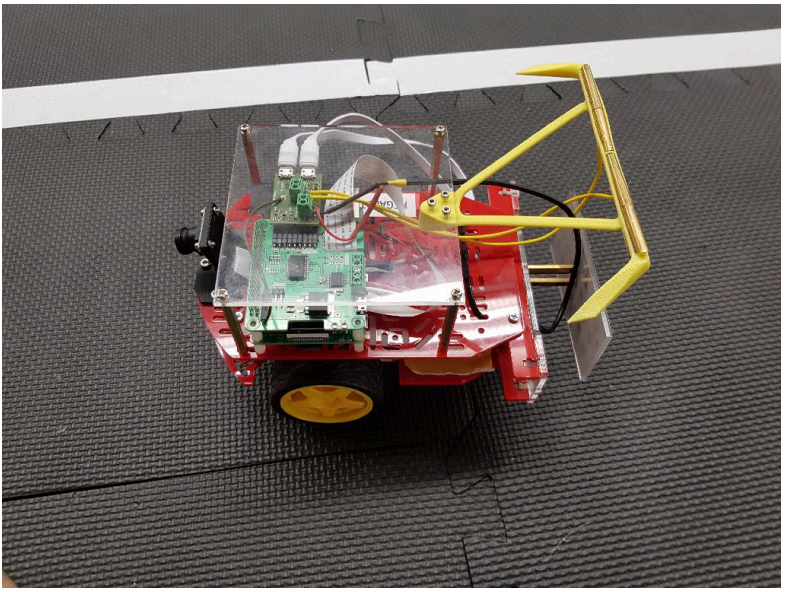


Figure 1.7. A charging capable Duckiebot.

1) Material needed for a single current collector

- 6 × laboratory plug CAT I Ø4mm
- 2 × 1mm cable, length 30cm
- Autolab add-on board (get in touch on slack to find out how to get them)
- If printer available: **5*g*** Material for the 3D printer (cutest color is yellow)
- April tag plate
- 16 × plastic spacers M2.5 × 12mm **or** 4 × spacers M2.5 × 50mm
- 7 × screw M2.5 × 10mm and nut M2.5
- Open ended USB cable 20cm
- Soldering iron and solder

2) Assembly

Cut / Order the April tag plate:

If **a laser cutter is available**, then laser cut this file with the dimensions **110 × 110*mm*** (you probably need to scale it, depending on your programs units).

If **no laser cutter is available**, then order this file with the dimensions **110 × 110*mm*** from a page, for example https://www.sculpteo.com/en/.

Print / Order the current collector:

If **a 3D printer is available**, then just follow these instructions.

If **no 3D printer is available**, then order the printed part from this site.

Put together three laboratory plugs:

Take three laboratory plugs and put them together as seen in Figure 1.8.

Figure 1.8. Three laboratory plugs put together.

Prepare the current collector soldering:

Be sure to have everything ready in Figure 1.9

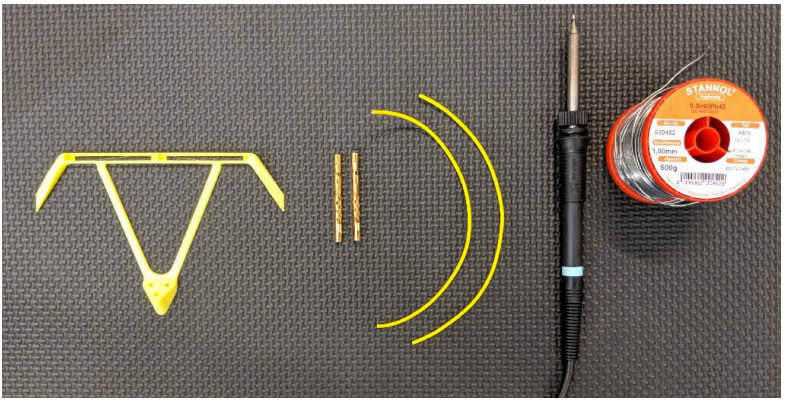


Figure 1.9. Neccessary parts for the current collector soldering

Solder the wires:

Solder the wires to the laboratory plugs as seen in Figure 1.10.

Figure 1.10. A wire soldered to the laboratory plugs.

Put the cables through the 3D printed part as seen in Figure 1.11.

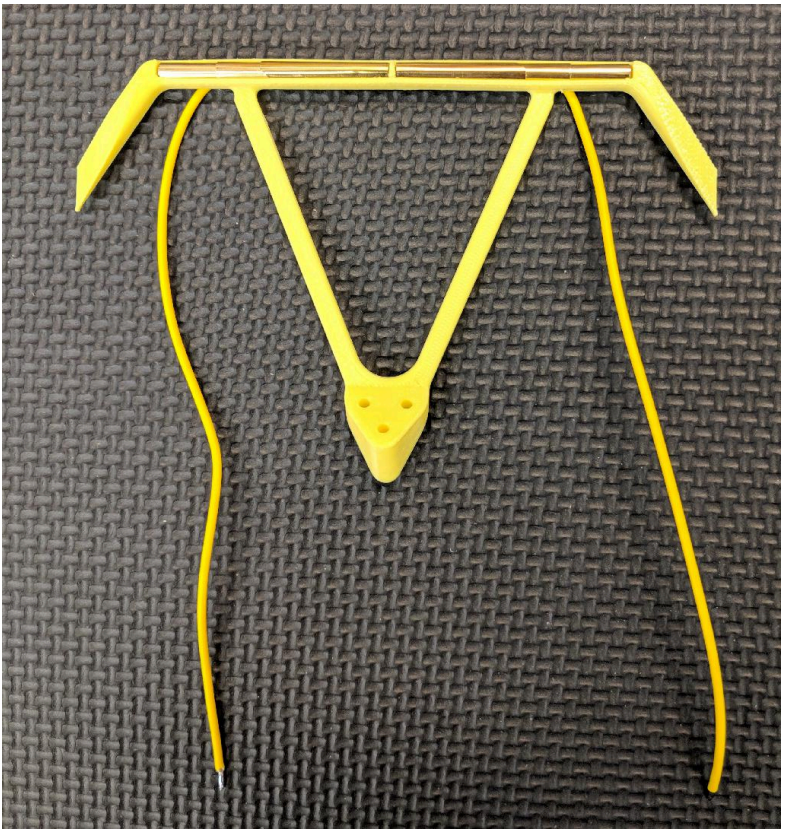


Figure 1.11. A current collector during the soldering process.

Optional: Glue the laboratory plugs:

If for any reason the laboratory plugs do not fit tightly in the 3D printed part, glue them.

Prepare the assembly:

Make the parts ready as seen in Figure 1.12.

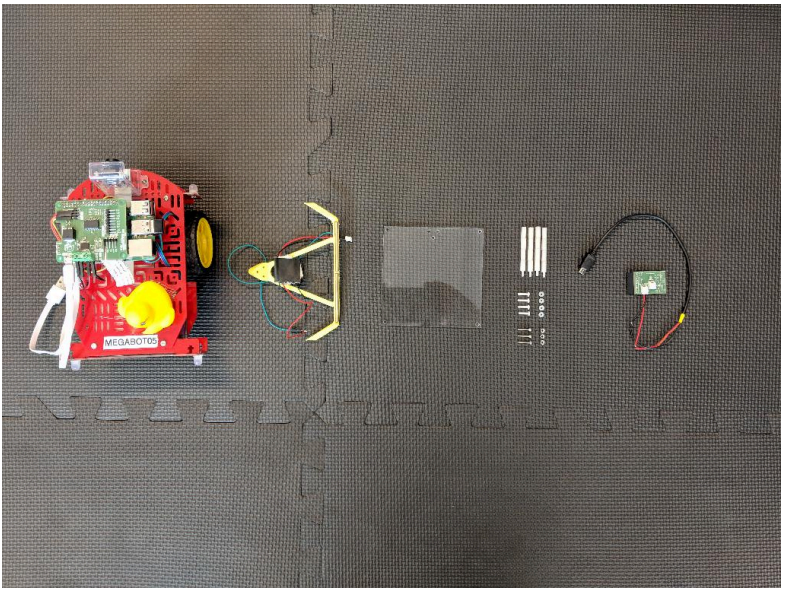


Figure 1.12. Neccessary parts for the assembly.

Assemble the April tag plate:

Assemble the April tag plate by using the acrylic glass, screws and distance keepers as seen in Figure 1.13

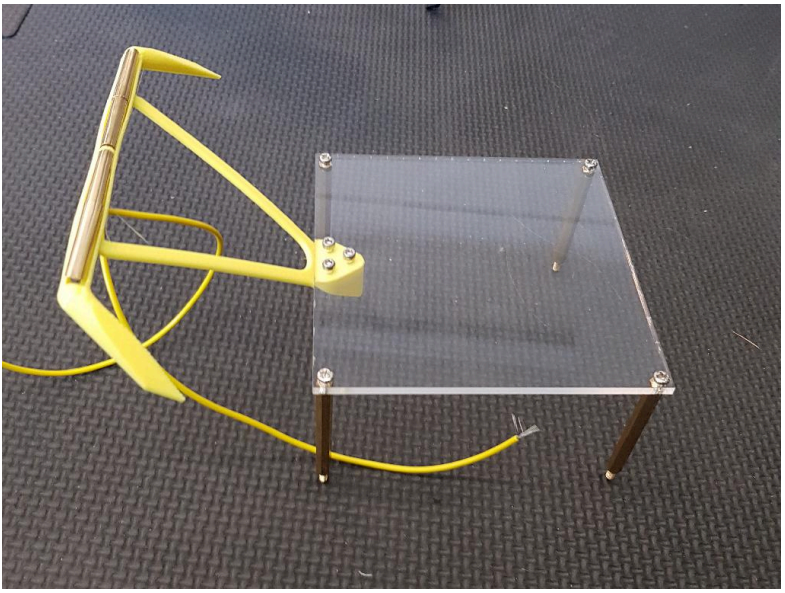


Figure 1.13. The assembled April tag plate

Mount current collector to April tag plate

Mount the current collector by using three screws and nuts as seen in Figure 1.14

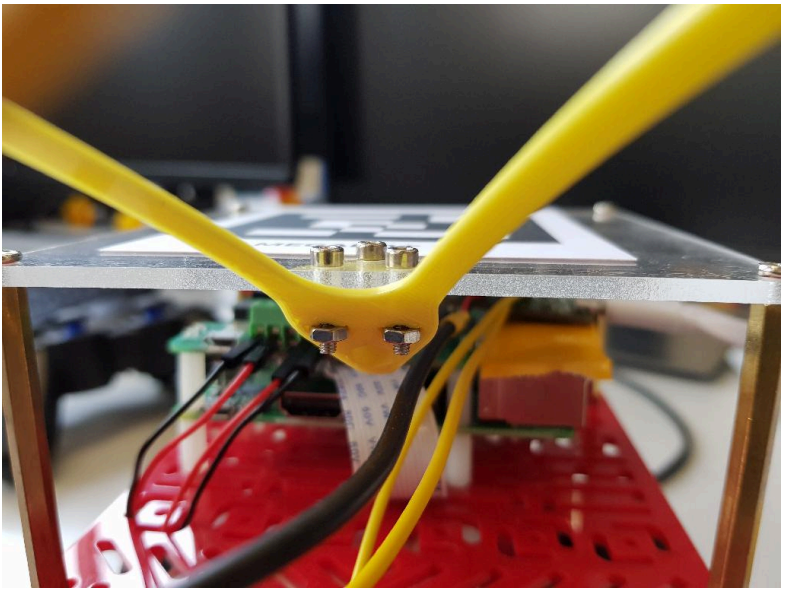


Figure 1.14. The current collector mounted to the April tag plate.

Plug in the Autolab add-on board:

Plug in the Autolab add-on board as seen in Figure 1.15. Also, screw the cables from the current collector as well as the open ends from the open ended USB cable to the add-on board.

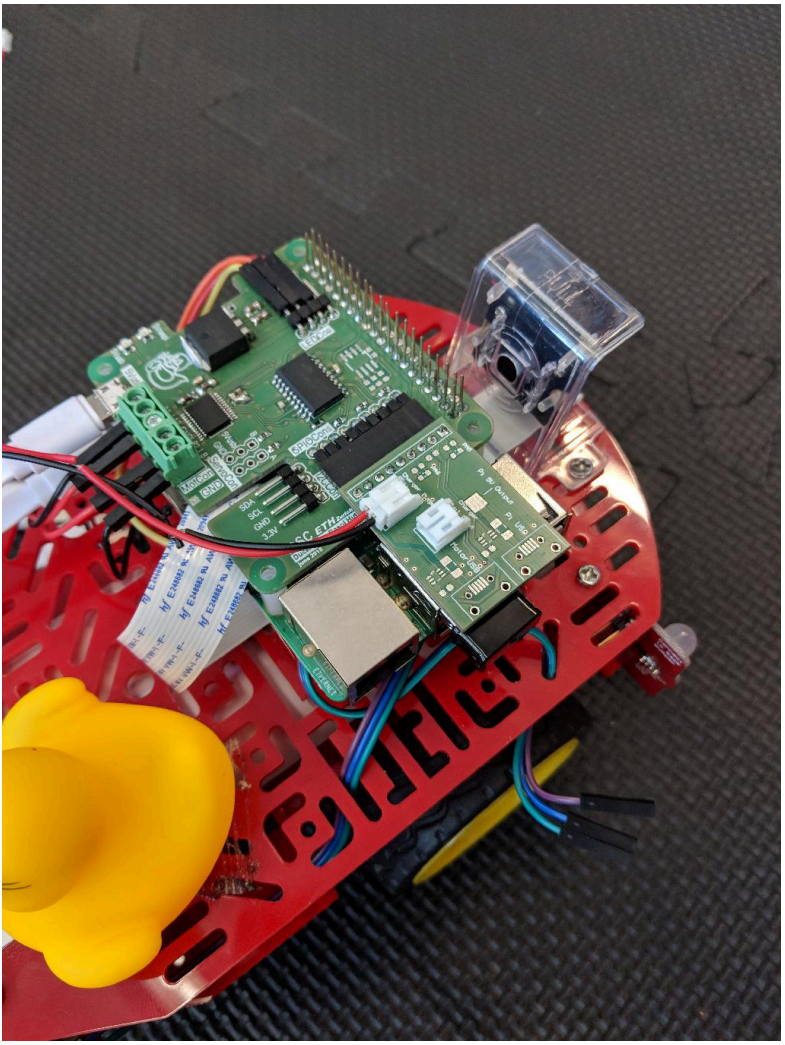


Figure 1.15. The Autolab add-on board.

Mount the structure to the Duckiebot:

Mount the April tag board with the current collector assembled to a Duckiebot as in Figure 1.17 and Figure 1.16. Plug in the USB cable to the battery of the Duckiebot.

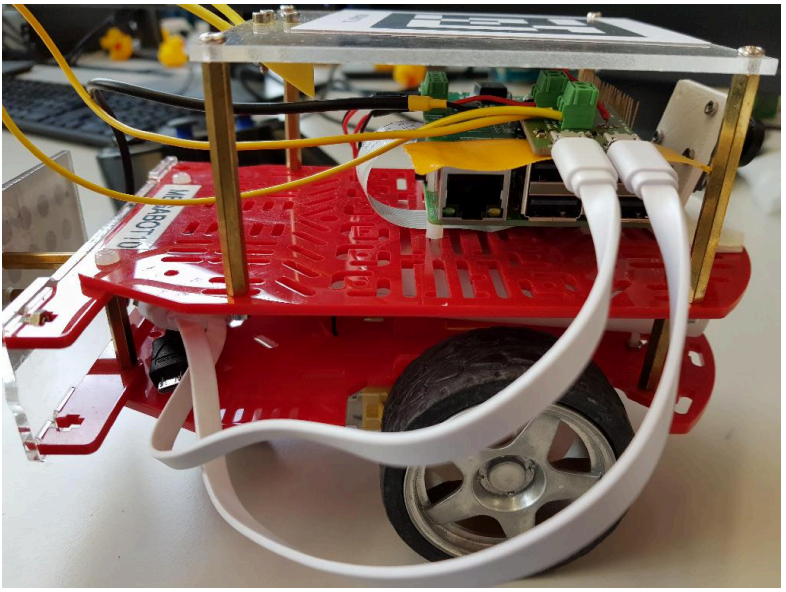


Figure 1.16. The April tag plate mounted to a Duckiebot.

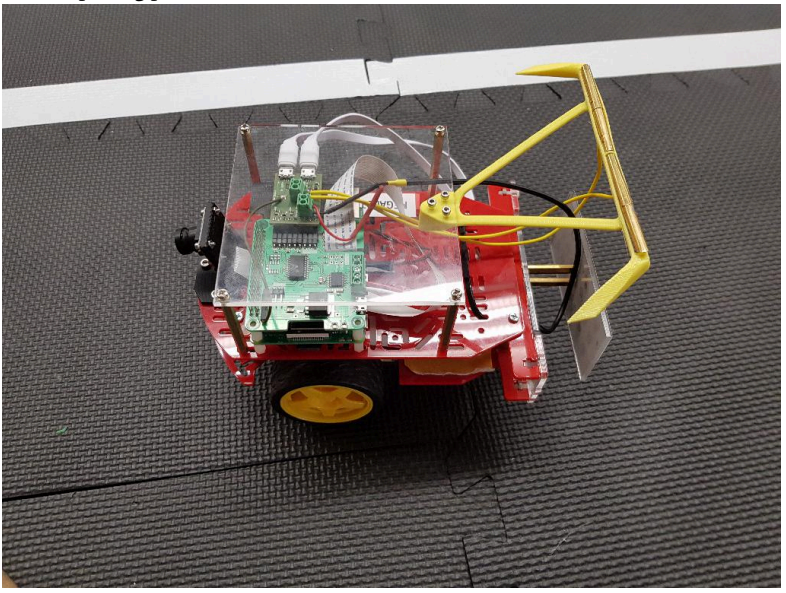


Figure 1.17. The resulting charging-capable Duckiebot.

Test your setup:

Connect the brass pieces to a 5V voltage source and check if the battery signals that it is charging.

UNIT B-2

Autobot hardware compliance checks

We have put together a step-by-step instruction to ensure that every Autobot that enters the Autolab is ready to go. We keep track of them in a GitHub Repository which will be used to keep track of the change history of every Autobot as well as to provide additional information to the user.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: a assembled Autobot

Results: a Autobot which is ready to be used in the Autolab.

2.1. Checklist

1) Step 0

Power on the Duckiebot. Then run the following command on your Computer. It will launch a Docker container on your Duckiebot which creates the suitability test report:

```
$ docker -H HOSTNAME.local run -it --network host -v /data:/data -v
/sys/firmware/devicetree:/devfs:ro duckietown/duck-
iebot_hw_checks:daffy
```

2) Step 1

If you mounted the Raspberry Pi on top, check that a pin protector is mounted on the pins of the hut to prevent pins from shorting out. If no protector is installed yet, then first check that no pins are bent or are making contact and then add the protector. It should slide into place without applying any force. The stl file to 3D print the pin protector can be found in Section 1.2 - Adding hardware to the bot.

3) Step 3

Check if the standoff on the Duckiebot is mounted and an April tag matching the Duckiebot name (hostname) is attached. The April tag needs to be aligned with the front of the top plate and centered. On the top plate you will find engraved lines that show you where to position it. the standoff needs to be mounted in the very front of the bot. A picture can be found in Figure 1.4

4) Step 4

Check if a rubber duckie sits on the bot. If not, put one on it now. As the rubber duckies are rather bad at balancing and lack hands to hold on to the Duckiebot, use double sided tape to fix it to the Duckiebot.

5) Step 5

Make sure the back bumper is properly fastened to the Duckiebot (no wiggling allowed).

6) Step 6

Check that no cables can interfere with the wheels of the Duckiebot. If necessary, use zip-ties to fix everything properly.

7) Step 7

Check that all cables are properly attached. Especially check that the camera cable is plugged in correctly and locked. When the Autobot is on, a small red light appears on the camera. Note that it can sometimes appear but still the camera is badly connected.

8) Step 8

If all above steps are done, press `y` and `ENTER` on your keyboard

9) Step 9

In ETH, we currently use batteries that are no longer produced. To replace them, we manufacture special batteries just for Duckietown which you will have gotten with your Duckiebot. So if you use these that's even better. To be able to write down what battery you use, press `n` and `ENTER`. This will give you the possibility to type in what battery you use.

10) Step 10

Check that a standard actuators (yellow motors) are used to drive the Duckiebot. If that is the case, press `y` and `ENTER` on your keyboard, otherwise press `n` and `ENTER`. Afterwards describe the other motors the Duckiebot is using.

11) Step 11

Fill in your name and press `ENTER`, the container should now close without errors.

12) Step 12

The freshly created suitability test report can be found under `http://HOSTNAME.local:8082/config/YYYY-MM-DD_hardware-compliance.yaml`. Please download it.

13) Step 13

Now upload the file that was created to your `fleet roster`. If you don't know what a `fleet roster` is and how to create one, read this. Once you created your own `fleet roster`, upload your newly created file to it.

if you know how to do upload the files to GitHub, you can skip the next section. If not, we will give you a step by step instruction in the following:

Clone your fleet-roster repository onto your local computer. To do this, open a terminal and type

```
$ git clone git@github.com:YOUR-WORKSPACE/YOUR-FLEET-ROSTER.git
```

create the directory where you want to place the file with the following code. you need to exchange `XX` by the number of the actual autobot and fill in the current date.

```
$ mkdir YOUR-FLEET-ROSTER/autobots/autobotXX/hardware-compliance/
YYYY-MM-DD_hardware-compliance
```

add the new file into the above mentioned folder. The easiest way to do this is just copying the file and placing it in the folder mentioned in step 2 using the file explorer.

now you are set to push things to github. Run the following in the same terminal as above: (again replace XX by the number of the autobot)

```
$ cd YOUR-FLEET-ROSTER
$ git add .
$ git commit -m "hardware compliance test for autobotXX"
$ git push origin aido2
```

14) Step 13

Verify that everythig worked out by checking the yaml file in `YOUR-FLEET-ROSTER/autobots/autobotXX/hardware-compliance/YYYY-MM-DD_hardware-compliance/YYYY-MM-DD_hardware-compliance.yaml`. It should look as follows:

```
verdict: pass
hostname: autobot01
date: 2019-04-01
mac-adress: B8:27:EB:9B:BD:DD
platform: Raspberry Pi 3B+
hat_version: 2018D v1.1
usb-memory: class 10 micro SD
sd-memory: class 10 micro SD
battery: RAVPOWER RP-PB07
actuation: DG01D dual-axis drive gear (48:1)
tester_name: Gyro Gearloose
```

UNIT B-3

Autolab fleet roster

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Knowledge about what a Autolab is Part A - Autolab definition

Results: Knowledge about what a fleet roster is and why it is used

Next: create your own autolab fleet roster and add data t it Unit B-2 - Autobot hardware compliance checks

3.1. Motivation

The goal of an Autolab is to have autonomous operation around the clock. Therefore, it is important that all agents (namely Autobots and Watchtowers) that are in the city are up to specification and in good shape. To keep track of all the changes and checks that a Autobot or Watchtower undergoes during its lifetime, some kind of information storage is needed.

3.2. Implementation

To solve the above challenge, a so called `fleet roster` was created. It contains all the history of compliance and suitability tests that were done for agents in the Autolab. Every time a hardware check or calibration is done, this is noted in the fleet roster. So make sure that whenever you change anything related to your agents in your Autolab to put it here. To make sure we can read out this data automatically and provide it to the operator of the Autolab, the fleet roster needs to be well structured. When you create your own fleet roster for your Autolab, follow the structure below:

```
my-autolab-fleet-roster
  * autobots
    * autobotXX
      * camera-verification
        * YYYY-MM-DD-HH-MM-SS_camera_verification
          * calibrations
            * camera_extrinsic
              * autobotXX.yaml
            * camera_intrinsic
              * autobotXX.yaml
            * kinematics
              * autobotXX.yaml
          * (camera_test_report.yaml)
          * (config.yaml)
      * hardware-compliance
        * YYYY-MM-DD_hardware_compliance.yaml
      * (system-identification)
        * (YYYY-MM-DD-HH-MM-SS_system-identification)
          *...
  * watchtowers
    * watchtowerXX
      * intrinsic-calibration
        * watchtowerXX.yaml
```

Note: `The camera_intrinsic.yaml`, `camera_extrinsic.yaml` and `kinematics.yaml` are created when you calibrate your Autobot using the calibration procedures described in Unit C-12 - Calibration - Camera and Unit C-13 - Calibration - Wheels.The `YYYY-MM-DD_hardware_compliance.yaml` file is created using one of our helper scripts. The procedure to follow can be found here.

3.3. Creating your own fleet-roaster

You can create your own `fleet roster` repository on GitHub using the structure shown above. If you know how to create a new repository you can proceed right away. If not, read the instructions here to find out how you can create a new repository on GitHub.

Make the name of your `fleet roster` explicit, like for instance `YOURINSITUTION_fleet_roster`, and remember this name as it will be asked later by the Autolab control interface.

UNIT B-4

Autolab map

KNOWLEDGE AND ACTIVITY GRAPH

Requires: The city layout is fixed and ready.

Results: The city layout is transcribed in duckietown-world, to be used for visualization tools, for localization and much more.

Next: todo

In the following chapters of the book, there are many uses to a software defined map of your Autolab. It is for instance required for a working localization pipeline.

Warning: this part is in chapter B for structural design of the documentation. However, in Part C - Auto charging you will probably modify the layout of your city. We strongly recommend to do it first then come back here once the map is fixed (to avoid having to do it again).

4.1. Setting up duckietown-world

1) Creating a virtual environment for python3

We recommend that you create a virtual environment for duckietown-world.

Here is one possible method, but you can use whichever you prefer.

Note: duckietown-world currently needs python3.7 or higher.

First, install venv:

```
$ sudo apt install -y python3-venv
```

Then, cd into your `duckietown-world` repository (it should be in the `src` folder of `dt-env-developer`), and create the venv.

```
$ cd ~/dt-env-developer/src/duckietown-world
$ python3.7 -m venv duckietown-world-venv
$ source duckietown-world-venv/bin/activate
```

Warning: if you create the venv as suggested inside duckietown-world, be very careful not to add it to your git commits.

To verify the version of Python running inside the virtual environment run:

```
$ python --version
```

This should be 3.7.x or higher.

Note: to get out of the virtual environment, just run `deactivate`

2) Pip setup of duckietown-world

Now, you can setup duckietown-world. Inside of the virtual environment (you should see "(duckietown-worl-venv)" in front of your prompt line), please run:

```
$ python3 -m pip install --upgrade pip
$ python3 -m pip install -r requirements.txt
$ python3 -m pip install jupyter
$ python3 setup.py develop --no-deps
```

Note: in case the last command returns the error 'Permission denied: 'src/duckietown_world_daffy.egg-info/PKG-INFO ', then go to the folder `src/duckietown_world_daffy.egg-info` and change permission for the folder itself as well as all the files within the folder using `sudo chmod 777`. After doing this, run again:

```
$ python3 setup.py develop --no-deps
```

4.2. Using duckietown-world to create your map

The best way to create a map is to do it interactively with the notebook.

Still in duckietown-world with your virtual environment, run

```
$ jupyter notebook
```

Warning: Please make sure it opens with some other browser than Firefox, which has a known bug with visuals of maps. Chrome works fine.

Now, navigate to notebooks and open `30-DuckietownWorld-maps`. You can run each step in order to get familiar with the code functions.

Once you get to the `Creating new maps` section, you can happily start creating your map. In the section below it is all the tile names and representation to help you.

You can add your map line by line, until satisfied with the result. Once this is done:

- Adjust the tile size parameter. The size is the interior to exterior size (on most tiles it is 0.585 meters).
- Create, in `duckietown-world/src/duckietown_world/data/gd1/maps`, a new map with your name (eg `autolab-ETHZ-01.yaml`).
- Copy the working map description that you just created in there.
- Check that you haves the tiles and the tile-size.

Now, in the root of duckietown-world, run

```
$ dt-world-draw-maps NEW_MAP_NAME
```

This should generate a new folder name after your map in `out-draw_maps`. You can now move this folder to the `visualization/maps/` folder.

4.3. Verification

If you now go back to the notebook and run the line that lists all maps, you should find yours in the list.

4.4. Committing

Now that the map is ready, you can commit :

- the map in `duckietown-world/src/duckietown_world/data/gd1/maps`
- the generated folder is `visualization/maps/`

Note: You may note have the rights to push to the duckietown repository. Instead, create a fork of the repository and remember the user account you use, as it will be useful later on. Also remember the name you gave to the map.

Warning: again, please do not commit any virtual environment.

Part C
Auto charging

0.5. Purpose

Duckiebots have a limited battery capacity. Therefore to create a fully autonomous Duckietown, a charging station is a crucial part. This chapter describes how to setup and operate a charging area. It is highly recommended to have a 3D-printer available in order to tune the flexibility of some parts - if this is not the case, URLs for ordering the parts are provided.

Below you may see a video of the expected result:

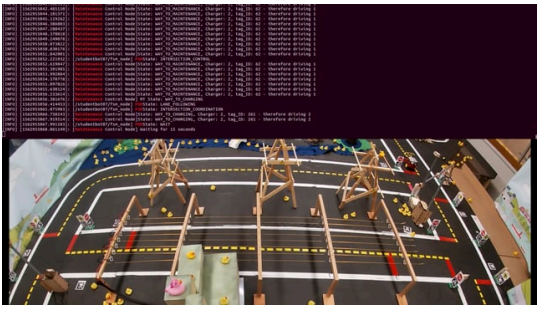


Figure 0.1. Expected results.

Please keep in mind that this solution is still not perfect - in Unit C-13 - Future projects, you find the issues which should be solved next.

Keywords: robotics, megacity, Autolab, auto-charging, charging, maintenance

UNIT C-1

Definitions

1.1. Maintenance area

The actual Duckietown city and the charging and calibration area are strictly separated by a single road. Every tile which purpose is the servicing of Duckiebots is part of the maintenance area. This includes every intersection and road which guides the Duckiebot from the maintenance entrance to a charging module and back.

1.2. Charging area

A charging area is the combination of multiple charging modules (Section 1.8 - Charging module) and intersections, straights and curves which connect the charging modules to the maintenance entrance. Figure 1.5 shows a charging area.

Inside a charging module, Duckiebots drive in a charger (Section 1.9 - Charger) and charge their battery by making contact between the attached current collector (Section 1.3 - Current collector) and the above mounted charging rails (Section 1.4 - Charging rail).

1.3. Current collector

The current collector Figure 1.11 is a flexible shaft mounted on top of a Duckiebot. Its purpose is to make contact with the charging rails.

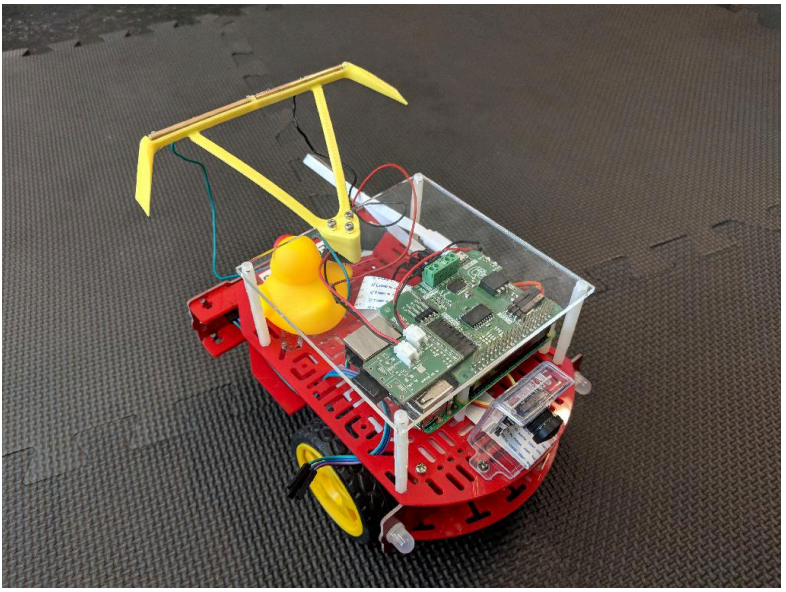


Figure 1.1. A Duckiebot with a current collector.

1.4. Charging rail

A charging rail is a brass tube mounted over a Duckietown with the help of high voltage poles (Section 1.5 - High voltage pole).

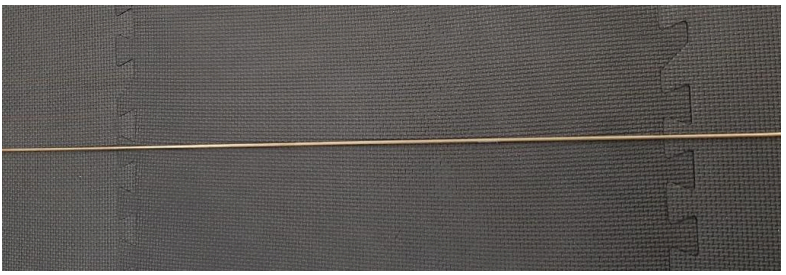


Figure 1.2. A brasstube.

1.5. High voltage pole

The purpose of the high voltage pole is to mount the charging rails over the street.

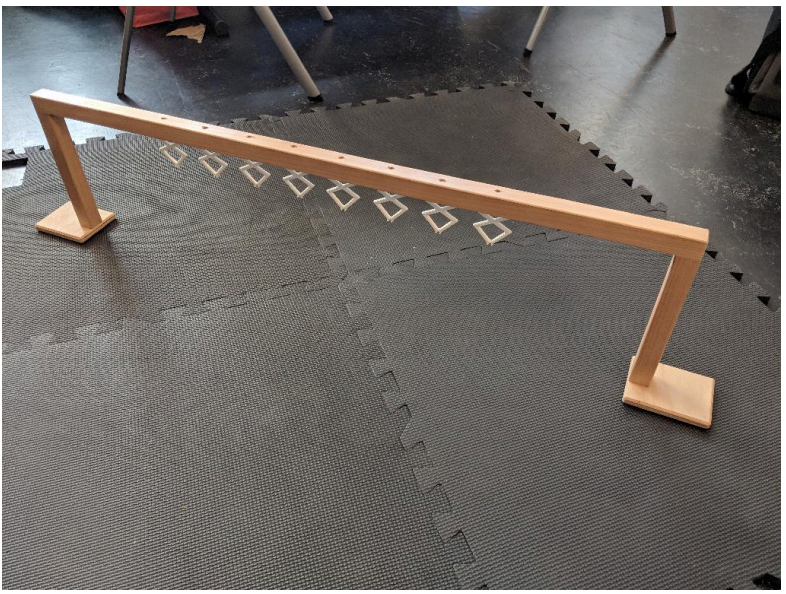


Figure 1.3. A high voltage pole.

1.6. Insulator

The insulator connects the charging rails to a overhanging structure.

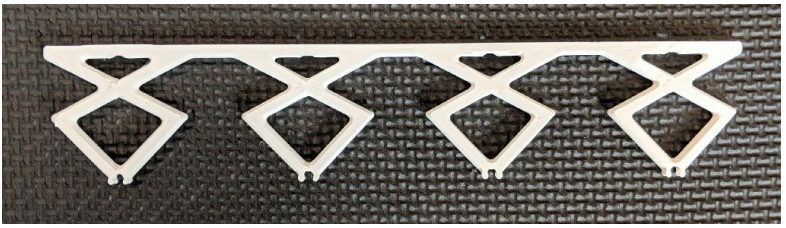


Figure 1.4. An insulator.

1.7. Charging rail tiles

Charging rail tiles are tiles with charging rails. Duckiebots can charge in both directions. Each charging rail tile has 8 brass rails. See Figure 1.5.

1.8. Charging module

A charging module describes the combination of a charging rail and all connected straights and curves up to the next intersection. See Figure 1.5.

1.9. Charger

A charger describes one single lane of a charging module. See Figure 1.5.

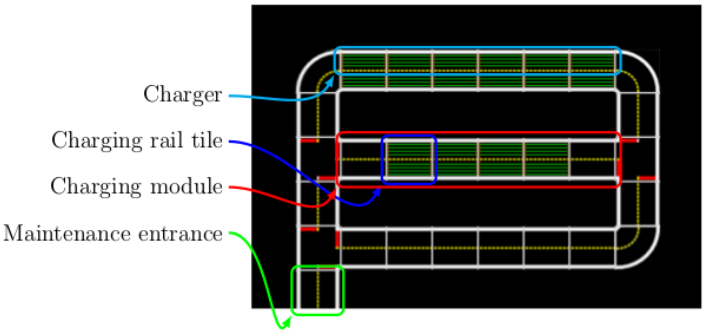


Figure 1.5. conventions

1.10. Maintenance Intersection

The maintenance intersection is a 3-way intersection. A direction on the maintenance intersection leads either to a charging intersection (in case of module 2) or to a subset of chargers (in case of Module 1). See Figure 1.6

1.11. Charger Intersection

A charger intersection is a 3-way intersection where the charger entrances and exits of seperate chargers meet. See Figure 1.6

1.12. Charging Manager

A charging manager is basically a Watchtower with a traffic light. Its task is to tell Duckiebots to which charger they should drive in. The charging manager must be allocated on the maintenance intersection. See Figure 1.6

1.13. Doorkeeper

A doorkeeper is a Watchtower that detects which charger a Duckiebot entered or exited. It must be allocated on the charging intersection. See Figure 1.6

Below you can find the definitions of maintenance intersection and charger intersections. You can also see also an example of the placement of charging manager and doorkeepers. See Figure 1.6

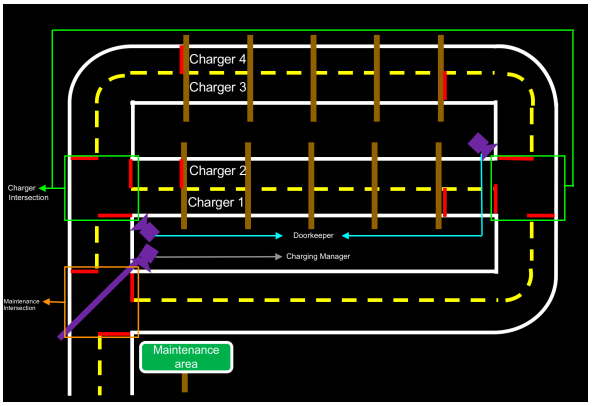
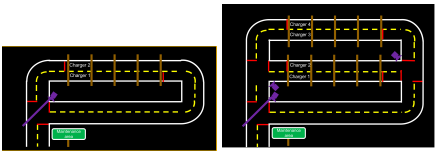


Figure 1.6. Definitions of Intersections and Placement of Charging Manager and Doorkeepers for Module 2

UNIT C-2

Infrastructure of Maintenance Area

This part contains the steps to plan and build a maintenance area.
Before starting, here is a general schematic view of a maintenance area for module 1 and module 2.



(a) (b) Module 2

Figure 2.1. Module 1

UNIT C-3

Plan your maintenance area

We highly recommend that the maintenance area fulfills the specifications of Duckietown.

Charging modules are scalable structures. Under good light conditions, roughly **3** Duckiebots fit in one tile. To ensure robustness, multiple charging modules should be used - that way, if a Duckiebot gets stuck for any reason, the affected charging module can be closed while all others can still be used.

3.1. Decide how many Duckiebots need to fit

Charging of a Duckiebot takes roughly the same time as discharging it. However, currently there is no way to park Duckiebots. Therefore it is recommended to be able to fit every Duckiebot inside the charging area.

3.2. Calculate the amount of charging rail tiles

$$ChargingTiles = \frac{Duckiebots}{3}$$

3.3. Plan the charging area

In between an intersection and a charging rail tile, there needs to be at least one straight. This restriction comes from the fact that the current collector may not touch a charging rail while traversing through an intersection to avoid additional disturbances.

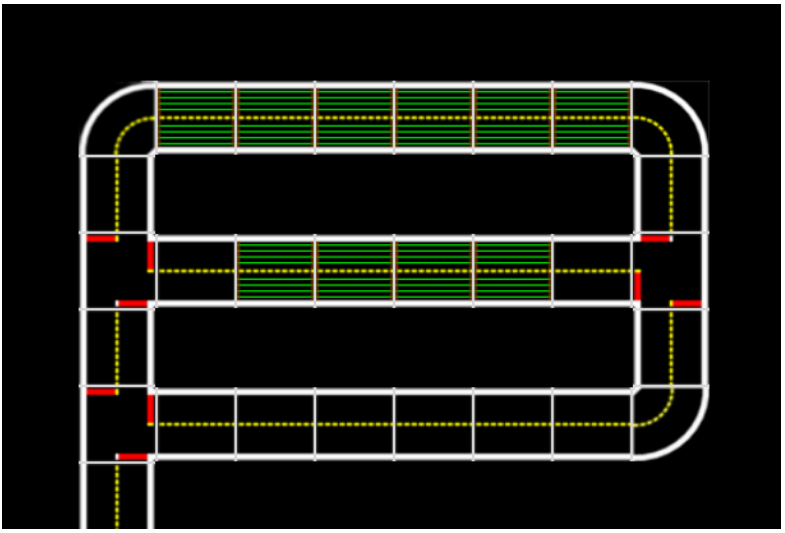


Figure 3.1. An example charging area which fulfills the specifications. This charging area can fit $3 \times 10 = 30$ Duckiebots.

The charging area in Figure 3.1 fulfills the specifications and has four chargers (two per module).

3.4. Calculate the amount of power supplies

The charging current of a Duckiebot is approximately 2Amps at 5V. By using a 5v30A power supply, you are able to charge 15 Duckiebots. However, we recommend to use one power supply per charger. That way, you may use shorter cables and therefore have less voltage drop.

Unit C-4
Charging module

For the construction of a charging module, you have many degrees of freedom. This includes the choice of material of the wooden structure (may be metal as well), the type of connection between the structure elements (screws, glue, nails), the length of the cables and many more.

However, the following requirements need to be satisfied:

- Insulators need to be mounted $210mm$ above tiles (Figure 4.4) for optimal pressure between current collector and charging rails
- Insulators need to be centered with respect to a lane to obtain good contact in center of lane
- There must be at least $150mm$ in between the white lines and the pylons to provide enough space to drive in a lane
- Enough high voltge poles (one per tile) must be used to provide a stable charger

In the following, you find the description on how we did it in Zurich.

Knowledge and activity graph

Requires: Material for one charging module Section 4.1 - Material for one charging module

Requires: Extra tools for assembling Section 4.2 - Extra tools

Results: A charging module for charging Duckiebots Figure 4.1.

Next: Assemble a Autobot that is capable of autocharging Unit B-1 - Autobot specifications and assembly

Figure 4.1. A resulting charging module during operation.

4.1. Material for one charging module

In this list, X will denote the number of charging rail tiles in a charging module. *Reminder*: One charging rail tile can fit **3** Duckiebots (**1.5** per lane).

- $8\times$ brass rod $\text{Ø}\ 4mm$, length $(10cm + X \times 59cm)$
- $(X+1)\times$ wooden structure top piece $(2 \times 2 \times 90cm)$
- $(2 \times (X+1))\times$ wooden structure side piece $(2 \times 2 \times 20cm)$
- $(2 \times (X+1))\times$ wooden structure floor piece $(10 \times 10 \times 1cm)$
- $(4 \times (X+1))\times$ woodscrew for high voltage pole, i.e. screw $\text{Ø}\ 3.2 \times 40mm$
- $(8 \times (X+1))\times$ screw $M3 \times 30mm$ and $M3$ nuts
- $(2 \times (X+1))\times$ insulator - self-print here or order here
- Drill $\text{Ø}\ 4mm$ and $\text{Ø}\ 6mm$
- $2\times$ power supply which enables you to drive 5V and 30Amps
- $\frac{1}{per power supply}\times$ power cable
- $\frac{3}{per power supply}\times$ cable-end-sleeve
- $18\times$ cable shoes M4 $\text{Ø}\ 4 - 6mm^2$
- $4\times$ M4 screw 10mm and M4 Nut
- $\text{Ø}\ 4mm \times 6m$ red cable
- $\text{Ø}\ 4mm \times 6m$ black cable

- 8× laboratory plug CAT I Ø4mm

4.2. Extra tools

In order to put things together you may need the following extra tools:

- crimping tool
- wire stripper
- hot glue gun
- solder iron and solder
- drill
- screw driver

4.3. Building a charging module

1) Assemble the wooden structure

Assemble the wooden structure as in Figure 4.2. This part may differ from our reference part. The important and neccessary specifications are: (i) the structure must be larger than one tile such that a road (with margins on both sides) may fit underneath and (ii) the space between the tile and the bottom part of the crossbar must be exactly 21cm (see Figure 4.4).

Figure 4.2. An assembled wodden structure.

2) Prepare for mounting the insulators

Make sure you have the parts ready seen in Figure 4.3.

Figure 4.3. Parts needed to prepare and assemble a high voltage pole.

3) Drill the holes

Drill 3mm holes such that the insulators will be centered after mounting, seen in Figure 4.4

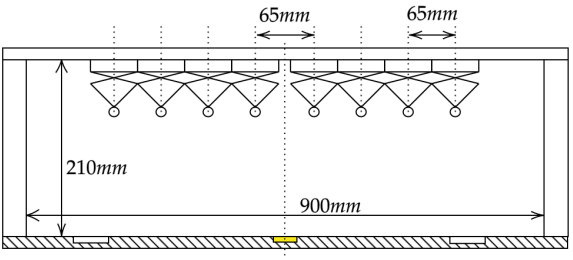


Figure 4.4. 2D sketch of a high voltage pole.

The 6mm holes (depth roughly 5mm) are optional and act as a hideout for the screw heads. The resulting holes should look like Figure 4.5.

Figure 4.5. Drilled holes.

4) Mount the insulator

Mount the insulator 3D prints as seen in Figure 4.6 and Figure 4.7.

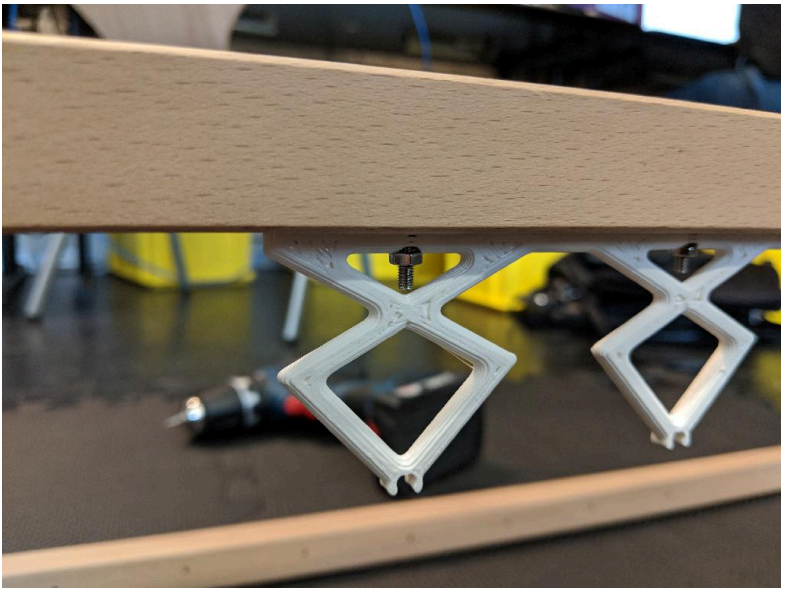


Figure 4.6. The assembled insulator.

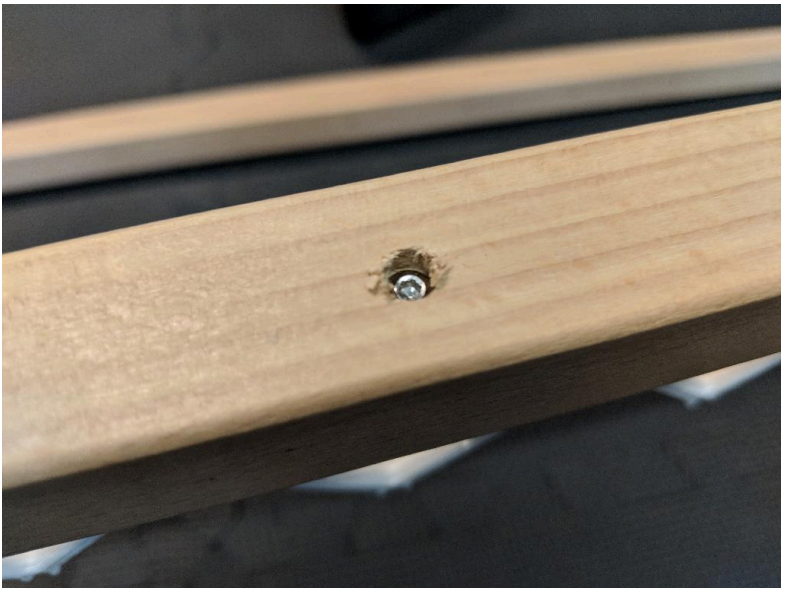


Figure 4.7. How the screw head is hidden.

5) Fix the high voltage poles to tiles

Use double-sided tape to mount the high voltage poles to the tiles (Figure 4.8). Make sure that the high voltage poles are aligned throughout the whole charging module.

Optional: you could also use hot glue instead of double-sided tape.

6) Bend brass rails and mount them

Bend the charging rails 5cm on both sides (in the same direction) to ensure that Duck-

iebots do not get stuck when arriving at the charging rail tiles (Figure 4.8).
Then, clip the brass rails into the insulators.

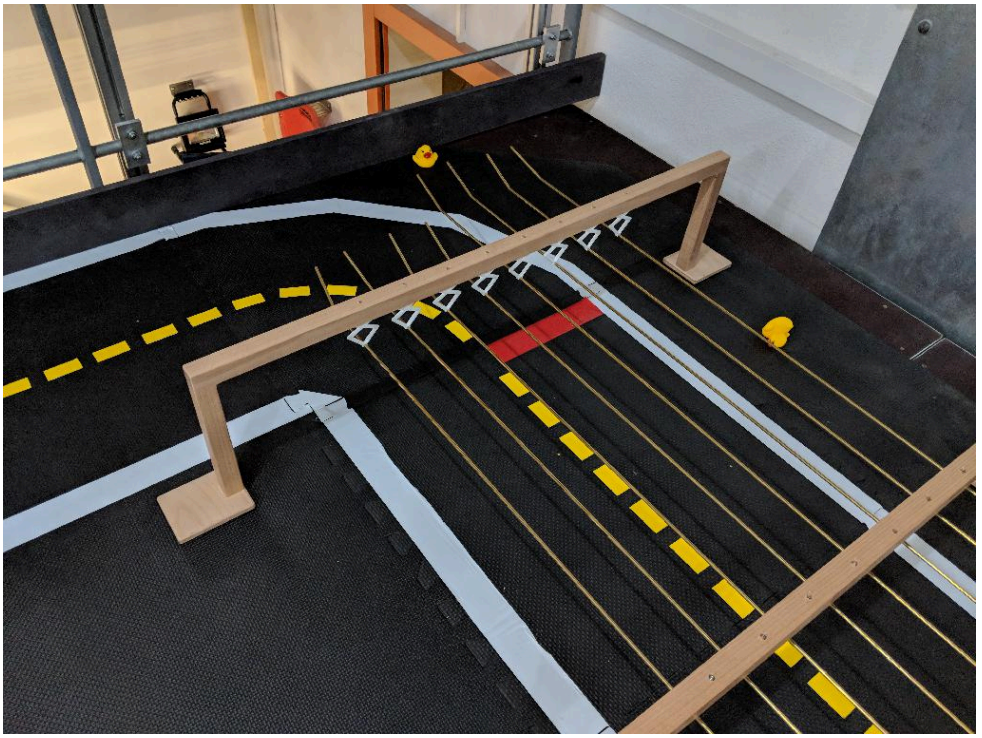


Figure 4.8. Glued high voltage poles with clipped in charging rails.

7) Solder laboratory plugs

Strip the insulation of the four red and four black cables on both sides with a wire stripper off. Then solder four red and four black cables each to a laboratory plug as seen in Figure 4.9. These cables should be approximately 20-25cm long.

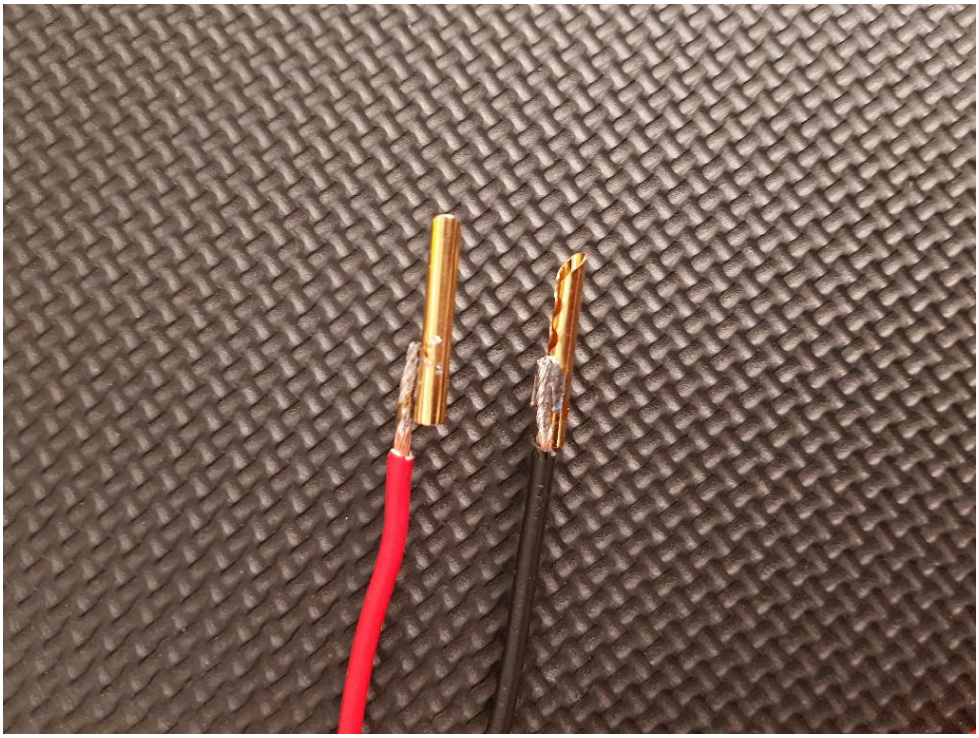


Figure 4.9. Soldered laboratory plugs to the cables.

8) Attach cable shoes

In order to crimp a cable shoe onto a cable, you need the following things which you can see in Figure 4.10.

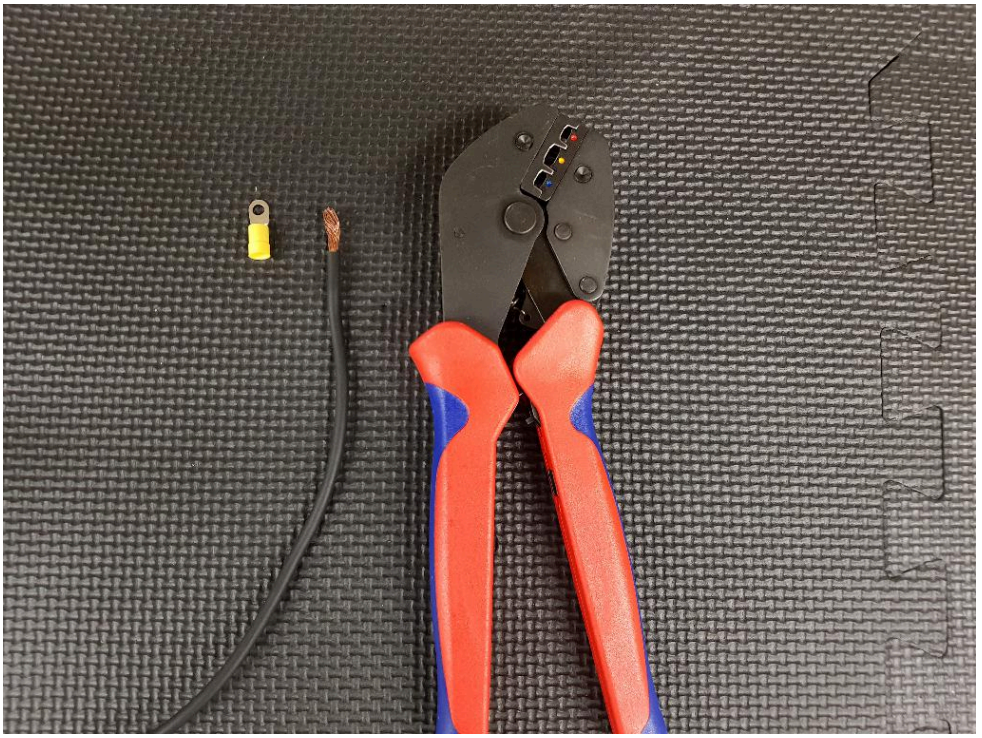


Figure 4.10. Crimping tool with a cable and a cable shoe M4 $\emptyset$ $4 - 6mm^2$.

First put the cable shoe onto the uninsulated cable such that it looks as in Figure 4.11. Then take the crimping tool, put the cable inside the yellow hole and press the cable shoe on it.

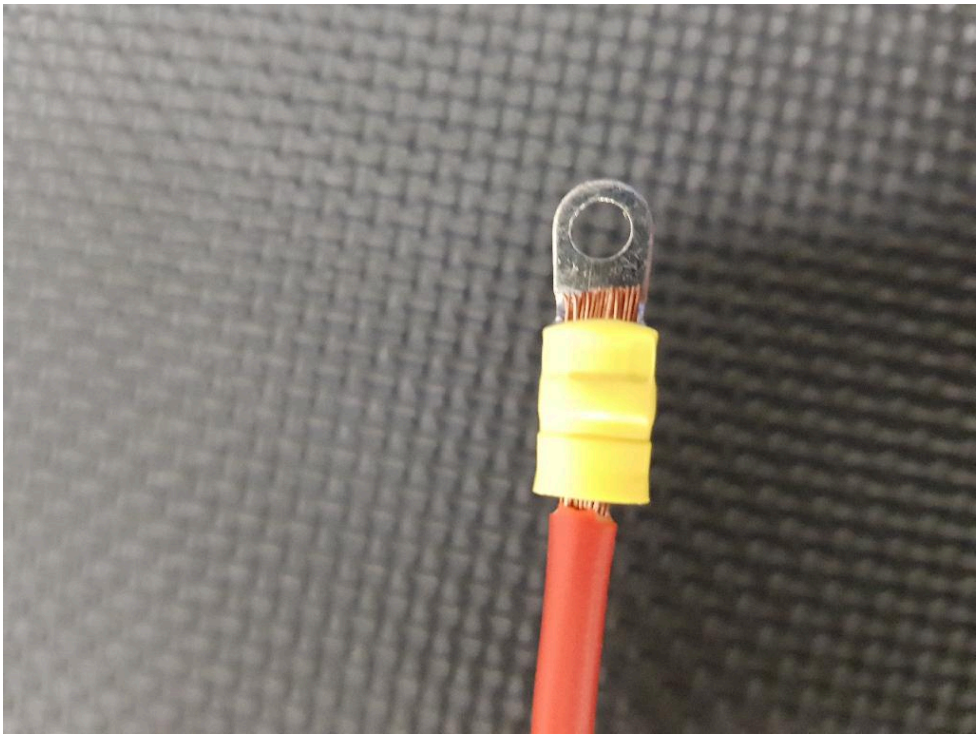


Figure 4.11. Crimped cable with a cable shoe.

9) Connect the cable to the power supply

Mount a cable shoe $\emptyset\,4-6mm^2$ with a crimping tool on the other side of these cables you have soldered. Then connect two red and two black cables respectively together with another cable shoe with a M4 screw and a M4 nut as seen in Figure 4.12. Then connect to the third cable shoe the corresponding red/black cable which will go towards the power supply. This third cable should be long enough to reach the power supply. Also attach to the end of the third cable a cable shoe.

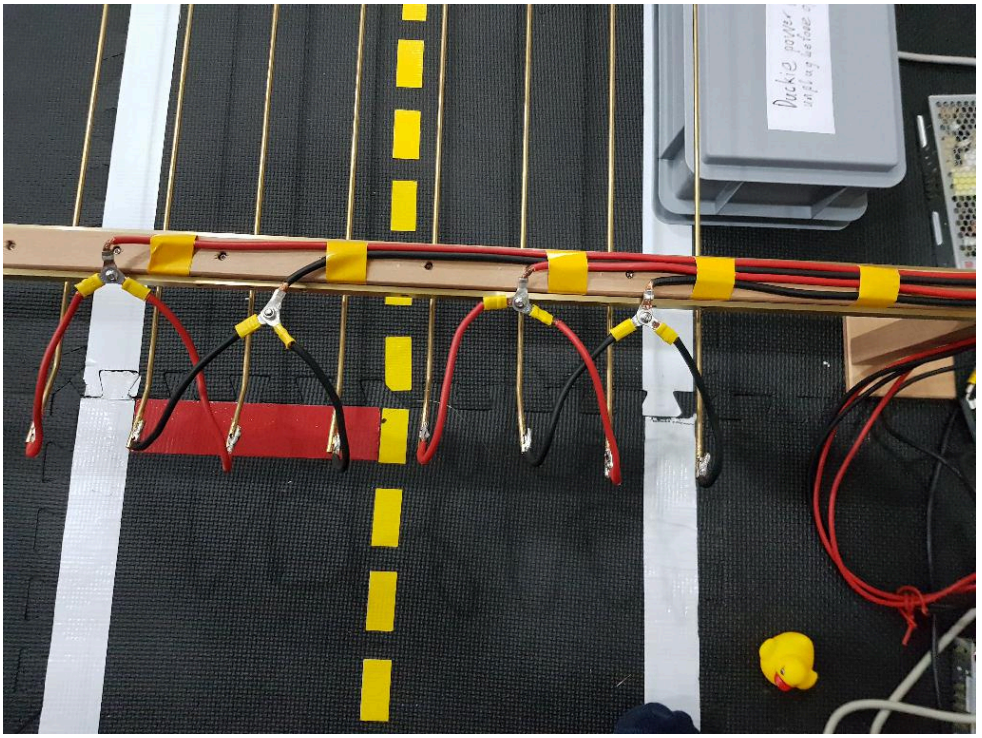


Figure 4.12. Connection of the cables to the rails.

10) Plug the soldered laboratory plug cable to the brass rail

Connect the laboratory plugs to the bended ends of the charging rails as seen in Figure 4.13. The cables of the brass rods must be polarized as seen in Figure 4.14. Make sure that you connect the four rods on the left to one power supply and another four rods on the right to the other power supply.

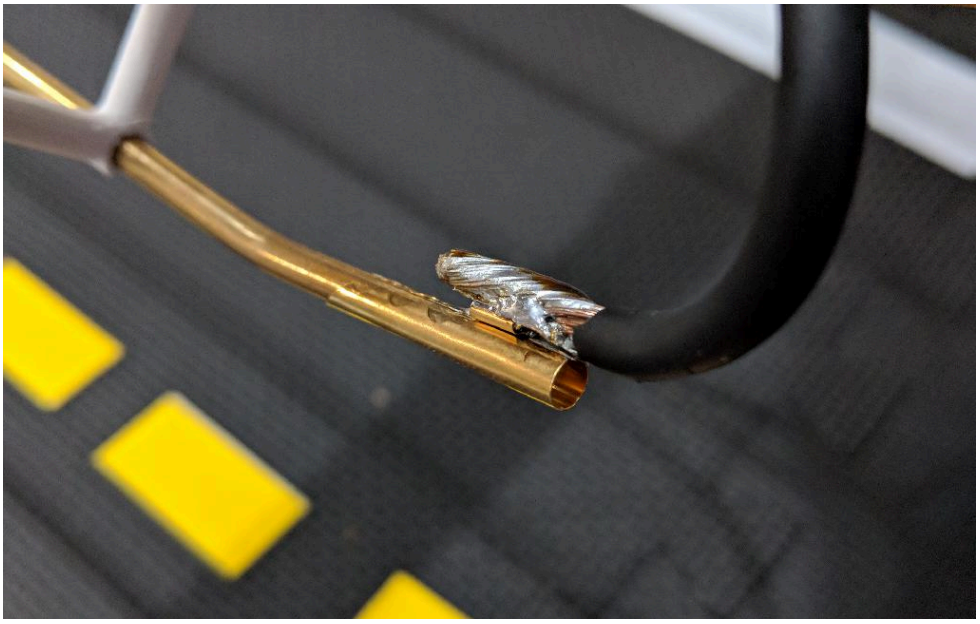


Figure 4.13. Connection between cables of power supply and charging rails.

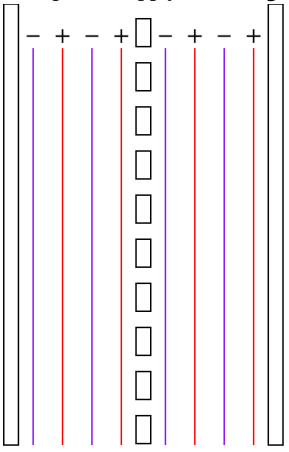


Figure 4.14. The polarities of the brass rods.

11) Prepare the power cable for the power supply

Take the power cable Figure 4.15 and strip the isolation off. Then attach a cable-end-sleeve with the crimping tool as seen in Figure 4.16. Then connect the prepared cable to the power supply exactly as it is shown in Figure 4.17.

Note: **It is important that ground, phase and neutral phase is connected the right way, so the colors need to match.**

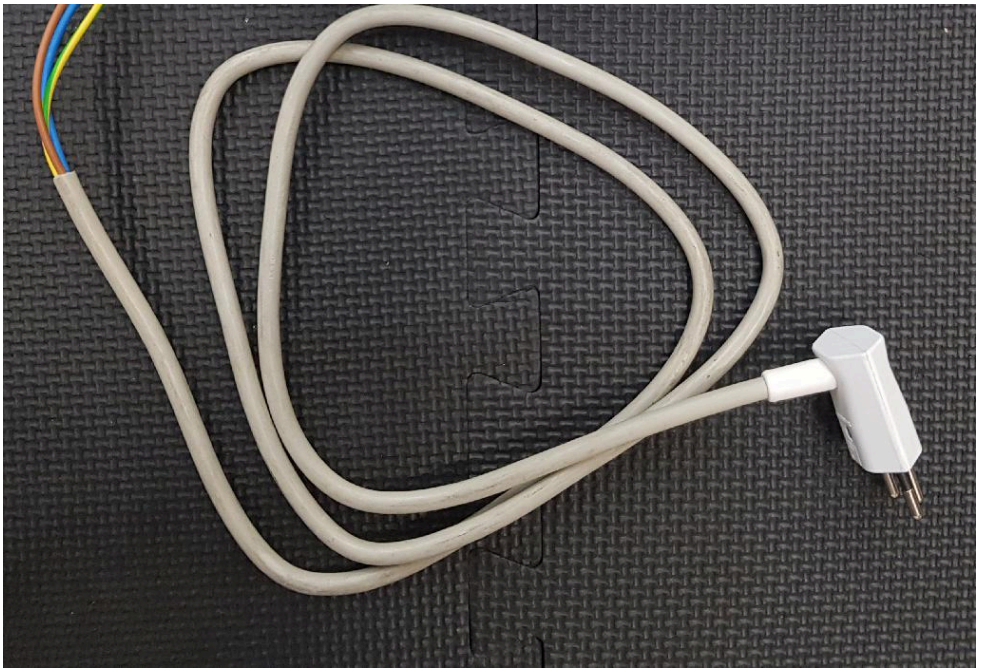


Figure 4.15. A power cable.

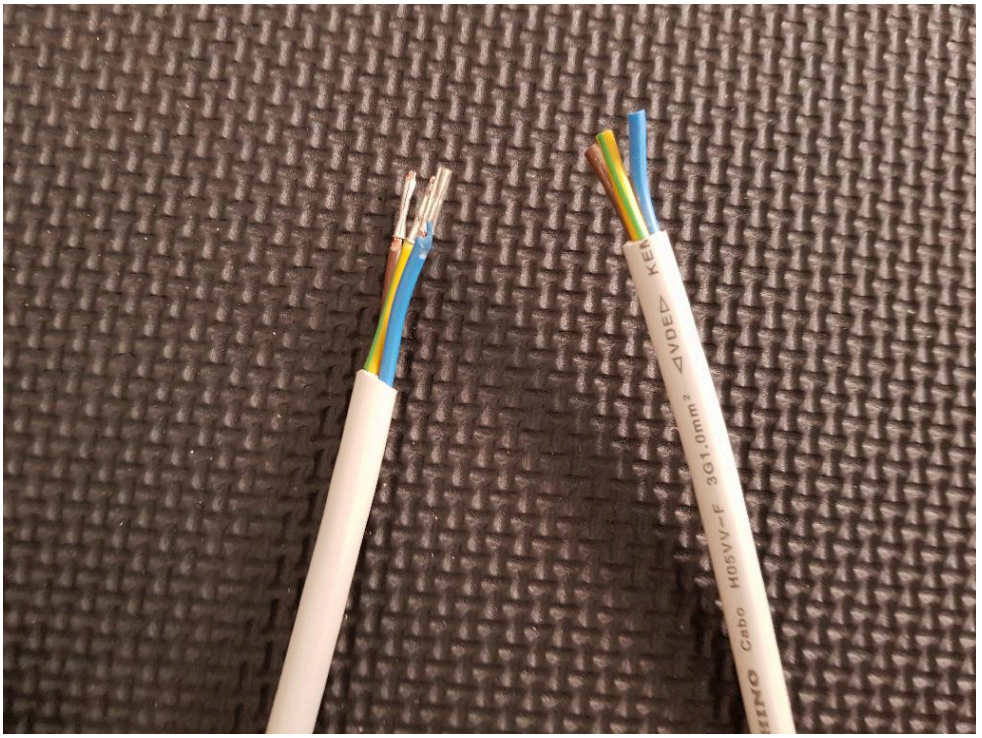


Figure 4.16. One end of the power cable before and after.

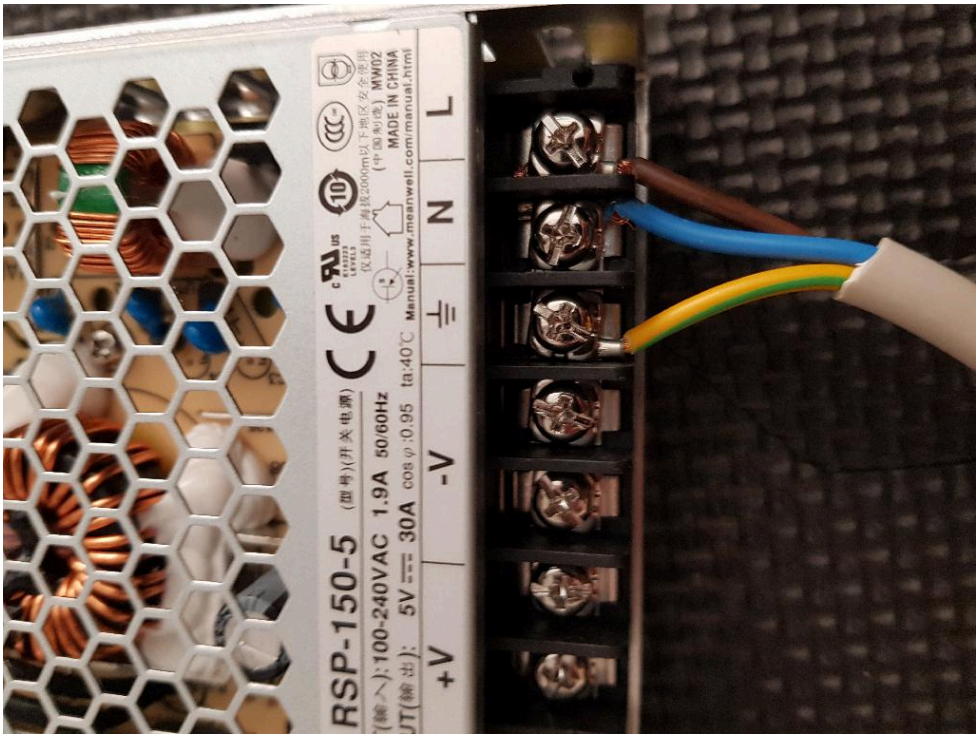


Figure 4.17. Attached power cable to the power supply.

12) Attach to power source

Connect the power supply to the rails. Red cable to V+ and black cable to V- as shown in Figure 4.18. Make also a connection between the two power supplies ground, in order to have a common ground.

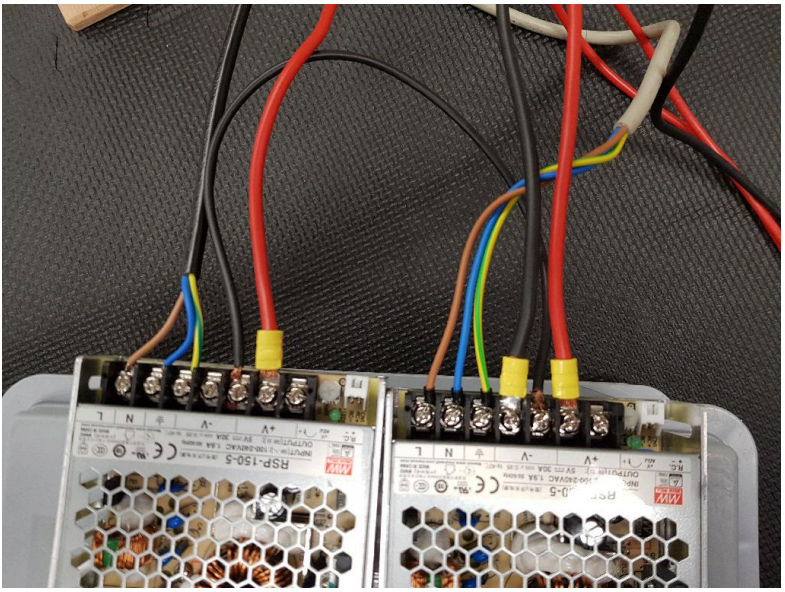


Figure 4.18. The connected power supply.

13) Adjust the output voltage of the power source

Turn on the power supply by plugging in the power cable. There is a voltage regulator - the plastic screw - (see Figure 4.18) next to the V+ connection - there you can adjust the voltage. Take a screw driver and a multimeter and measure the Voltage across V+ and V-. The Voltage should be adjusted to 5.5V.

14) Test your setup

Place an assemblied Duckiebot Figure 1.17 which is capable of charging underneath the charging rails, turn on the power supplies and see if the battery is going to charge.

UNIT C-5

Building a maintenance area

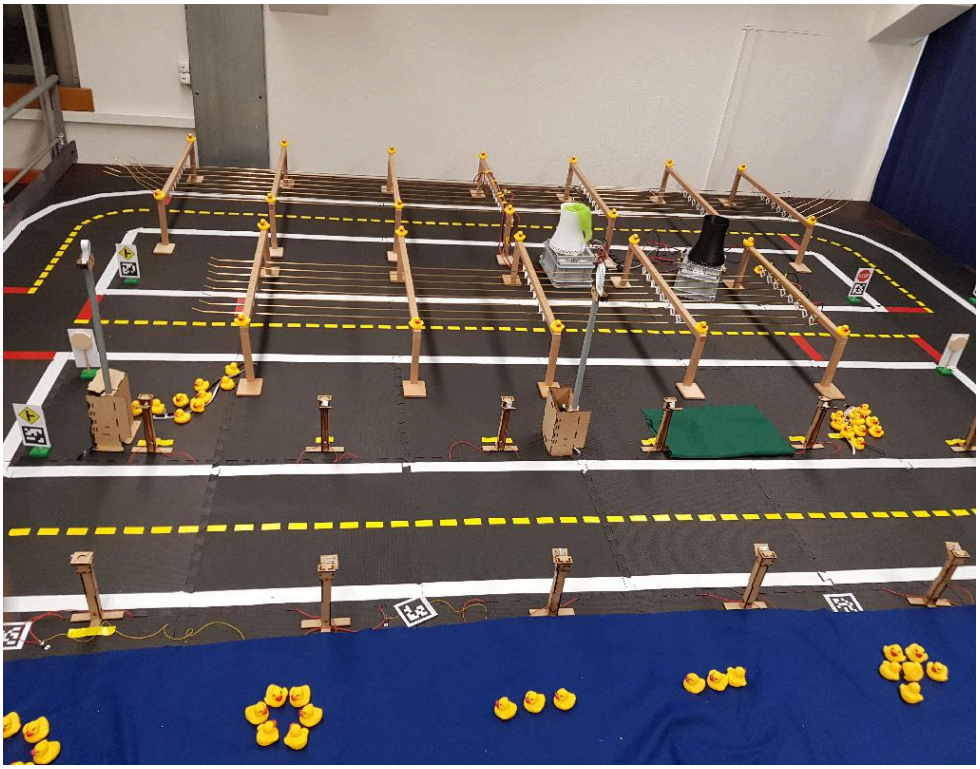


Figure 5.1. A complete maintenance area.

Next, please build the whole maintenance area.

5.1. Put together all tiles

Respect the constraints of Duckietown.

5.2. Add April tags for every intersection

Note down all the **tag IDs** and their **positions** in your maintenance area - you will need them later on to define your map in the software.

5.3. Add a red line and a STOP April tag after each charger

Directly at the beginning of the next tile after a charger, add a red line Figure 5.2. This red line and the stop tag is used to determine whether a Duckiebot is the first in a charger or not.

5.4. Place a traffic light on the maintenance intersection

This traffic light will serve as the charging manager.

5.5. Place watchtowers on charger intersections

Make sure that the intersection is under the scope of the camera

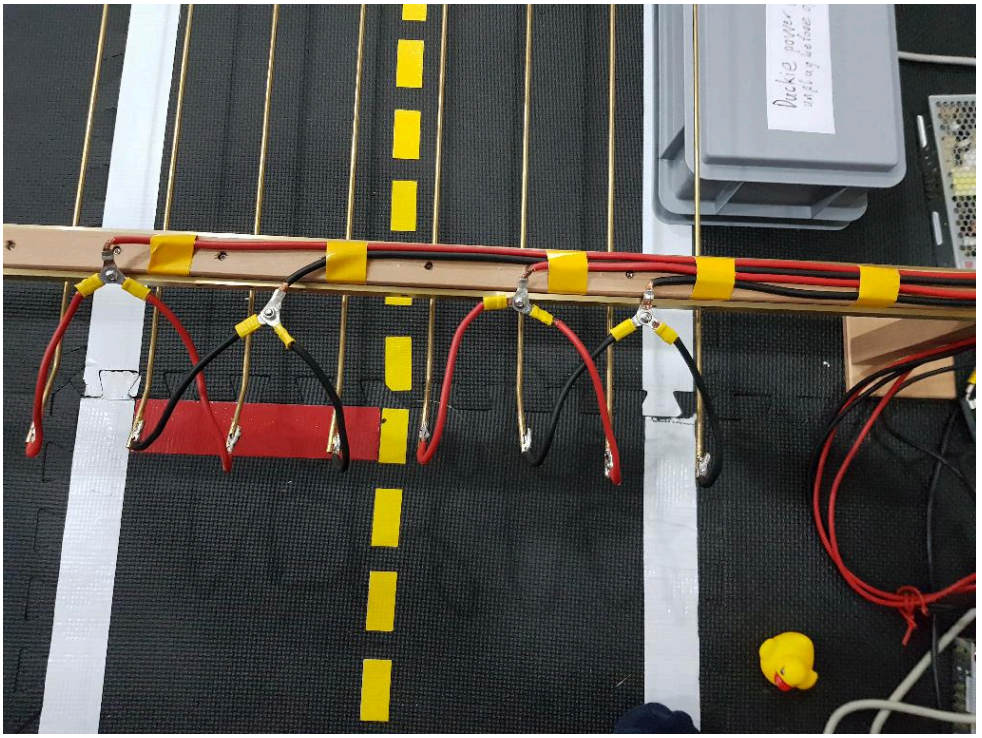


Figure 5.2. The red line after a charger.

UNIT C-6

Setup of Reference Tags

(First, we seperate the apriltags that are only used for charger management with the normal ground apriltags and we call them reference tags.)

This part helps you to place reference tags on the intersections they are needed. They are used for detecting Duckiebots on an intersection. For module 1 the charging manager will be detecting Duckiebots, so you have to place the reference tags on the maintenance intersection following the instructions below.
For module 2 Duckiebots are detected on the charger intersections by the doorkeepers from the apriltag on their apriltag plate, so you must place the reference tags on the charger intersections following the instructions below.

6.1. Steps

1. Take 4 apriltags and place them on the intersection they are needed. Note down the apriltag IDs you are using.

2. Allocate the watchtowers on the edge to the intersection in order to let it see all 3 directions: entrance/exit to the intersection and two directions to two seperate chargers.

3. In Step 2 we will start the CSLAM container.

If you are using module 1, the charging manager will be responsible for apriltag detection. Therefore, you have to run the CSLAM container on charging manager.
If you are using module 2, the doorkeepers will be responsible for detecting apriltags. Hence, CSLAM container must be started on doorkeepers and not on the charging manager.

Use the following command line to run the CSLAM container

0) Clone the github repository for CSLAM:

```
$ git clone https://github.com/duckietown/duckietown-cslam.git
lapotp $ cd duckietown-cslam/scripts
```

1) Open the file watchtowers_setup.sh . In the file set the ROS_MASTER_HOSTNAME and ROS_MASTER_IP to charging manager's hostname and IP (for module 1) or doorkeeper's hostname and IP (for module 2). Change the array to the hostname. An example:

```
ROS_MASTER_HOSTNAME=watchtower22
ROS_MASTER_IP=192.168.1.176

array=(watchtower22)
```

Now change the image tag on the last line to autocharge. The last command in the file

should be like this:

```
docker -H ${array[$index]}.local  run -d \
                                      --name cslam-acquisition \
                                      --restart always \
                                      --network=host \
                                      -e ACQ_DEVICE_MODE=live \
                                      -e ACQ_ROS_MASTER_URI_DEVICE=${ar-
ray[$index]} \
                                      -e ACQ_ROS_MASTER_URI_DE-
VICE_IP=127.0.0.1 \
                                      -e ACQ_SERVER_MODE=live \
                                      -e ACQ_ROS_MASTER_URI_SERV-
ER=${ROS_MASTER_HOSTNAME} \
                                      -e ACQ_ROS_MASTER_URI_SERV-
ER_IP=${ROS_MASTER_IP} \
                                      -e ACQ_DEVICE_NAME=${array[$in-
dex]} \
                                      -e ACQ_BEAUTIFY=1 \
                                      -e ACQ_STATIONARY_ODOMETRY=0 \
                                      -e ACQ_ODOMETRY_UPDATE_RATE=0 \
                                      -e ACQ_POSES_UPDATE_RATE=15 \
                                      -e ACQ_TEST_STREAM=1 \
                                      -e ACQ_TOPIC_VELOCITY_TO_POSE=ve-
locity_to_pose_node/pose \
                                      -e ACQ_TOPIC_RAW=camera_node/im-
age/compressed \
                                      -e ACQ_APRILTAG_QUAD_DECIMATE=2.0
\
                                     duckietown/cslam-acquisition:au-
tocharge || echo "ERROR: Starting cslam-acquisition on ${array[$index]}
failed. Probably this Watchtower wasn't configured properly or we can't
connect via the network."
```

2) Run the script

💻 `$ source watchtowers-setup.sh`

After starting the container, make sure it is running. You have to see logs in CSLAM container as following:

```
...
[INFO/serverSideProcess] Published pose for tag 327 in sequence 10
[INFO/serverSideProcess] Published pose for tag 334 in sequence 10
[INFO/serverSideProcess] Published pose for tag 360 in sequence 10
[INFO/serverSideProcess] Published pose for tag 327 in sequence 11
...
```

3) Now you are going build and run containers which are responsible to interpret the apriltag poses of duckiebots. In order to do that, use the commands:

For module 1:

```
$ git clone https://github.com/alifahriander/charging_manager_mod-
ule1.git
$ cd charging_manager_module1/docker
```

For module 2:

```
$ git clone https://github.com/alifahriander/doorkeeper.git
$ cd doorkeeper/docker
```

In the *docker* directory you find a YAML file. There you can find the parameters relevant for charging manager or doorkeeper. You have to change the following parameters:
direction1
direction2
direction1_tag
direction2_tag
entrance
exit

Now it is explained how these parameters are defined.

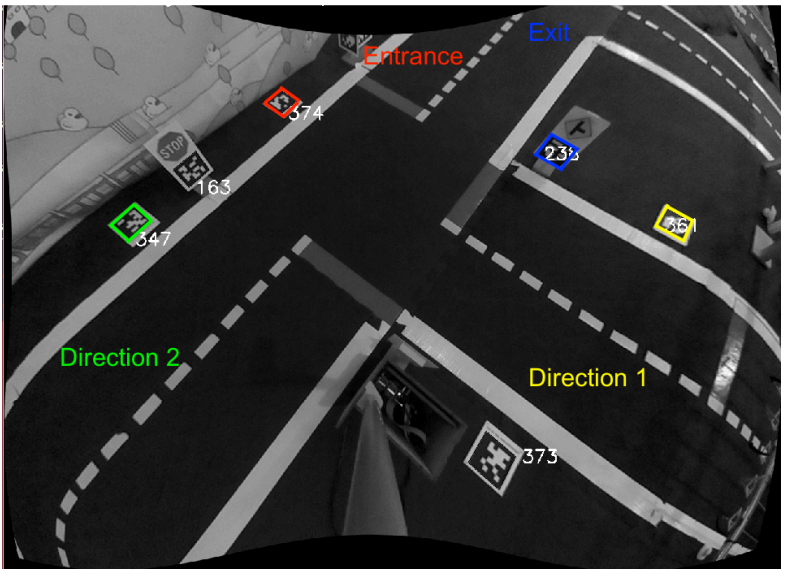


Figure 6.1. View from doorkeeper on charger intersection

In the picture above you can see that the lane on the right side is defined as direction 1 and on the lower left side as direction 2. Define to which charger the direction leads. For example in our case direction 1 leads to charger 2 and direction 2 leads to charger

4. Therefore, according to this picture we have to define direction1 as 2 and direction2 as 4
Then, choose apriltags under the scope of camera in order to assign them as reference tags, for example in this picture the reference tags are selected as following:

```
* entrance tag : 374
* exit tag : 238
* direction1_tag : 361
* direction2_tag : 347
```

If you want to see which apriltags your camera sees, use these command lines:

```
$ dts start_gui_tools HOSTNAME
container $ echo "HOST_IP HOSTNAME" >> /etc/hosts
container $ rqt_image_view
```

Then select the topic /poses_acquisition/test_video/HOSTNAME/compressed
Select the apriltags such that every selected apriltag represents one direction as in the picture above. If it is necessary to choose traffic sign apriltags, you may select them as reference tags.

Type in the parameters to the YAML file.

Now you have to build, push and run the container

```
$ docker build --rm -f "Dockerfile" -t IMAGE_NAME
$ docker push IMAGE_NAME
$ docker -H HOSTNAME.local pull IMAGE_NAME
$ docker -H hostname.local run -it --net host --memory="800m" --memo-
ry-swap="1.8g" --privileged -v /data:/data --name CONTAINER_NAME IM-
AGE_NAME
```

You can check the logs of the container, You have to see the following logs periodically:

```
...
[INFO] [1563805663.275686]: ############################
[INFO] [1563805663.288087]: {'374': {'position': x: 1217.38195801 y:
527.410217285 z: 0}, '238': {'position': x: 252.730285645 y:
602.758789062 z: 0}, '361': {'position': x: 518.172912598 y:
229.96383667 z: 0}, '347': {'position': x: 283.221954346 y:
217.282211304 z: 0}}
...
```

In the logs you see the positions of reference tags are updated periodically. At the beginning all reference tag positions are initialized with 0.0. After you change the above mentioned parameters, you must see that the positions are updated with non-zero values. If that is the case, you accomplished this step.

1. Now you have to place the reference tags such that they refer to a particular direc-

tion. In order to do that, take a duckiebot and place it to the entrance of the intersection.
Now in the logs of charging manager(for module 1)/doorkeeper(for module 2) container you will see that the April tag id is added to a dictionary called MOVING AT(referring the moving apriltags). Its keys refer to the apriltag ids of the duckiebot which arrived to the intersection.

For every apriltag id that is observed on the intersection, there is a dictionary. In it you have some attributes of a duckiebot apriltag:

```
* position :  Position of Duckiebot's apriltag on the image
* first_neighbor : First seen closest reference tag to Duckiebot's april-
tag
* last_neighbor  : Last seen closest reference tag to Duckiebot's april-
tag. This attribute will be updated upon receiving April tag positions
from the acquisition node
* timestamp : The time the information above is saved
```

Have a look on this log. We moved a duckiebot from entrance to direction 2:

```
...
[INFO] [1563805798.282072]: {'400': {'first_neighbor': '374',
'last_neighbor': '347', 'timestamp': 1563805796.091526, 'position': x:
748.847106934
y: 468.132141113
z: 0}})
...
```

There you see that the first_neighbor is the reference tag with ID 374 corresponding the entrance tag. The last_neighbor argument is tag 347, this is the direction2_tag. So one can infer that duckiebot was moved from entrance to direction 2.

After understanding what the logs mean, look at the last_neighbor argument on the logs. If it corresponds to the entrance reference tag and your duckiebot is located near the entrance reference tag, it means, the placement of entrance reference tag works.

If it is not the case, replace the reference tag which is at the moment the closest neighbor apriltag (last_neighbor in terms of logs) further from the intersection entrance along the lane it is located. In this example, you can see that the apriltag 361 is far from the intersection entrance and it is near to the charger exit.Figure 6.1

1. Repeat the previous step for every reference tag

2. Test and verify that reference tags' placement works. In order to do that, follow the instructions below:

   a. Start the *indefinite navigation demo*

laptop $ dts duckiebot demo –demo_name indefinite_navigation –duckiebot_name DUCKIEBOT_NAME –package_name duckietown_demos

   a. Place the duckiebot just before the intersection, the direction it is on does not matter.

b. Start the autonomous mode laptop $ dts duckiebot keyboard_control DUCKIEBOT_NAME and press *A* for switching the Duckiebot to the autonomous mode.

c. Observe the logs of charging manager(for module 1) or doorkeeper (for module 2). You have to see the following: `402 is on WAY 2` This means, Duckiebot with apriltag ID 402 entered the charger 2.

You have to repeat this experiment for other directions and verify that our reference tag placement works.

UNIT C-7

Starting the Containers

7.1. Instructions for CSLAM

1) Clone the github repository for CSLAM:

```
$ git clone https://github.com/duckietown/duckietown-cslam.git
$ cd duckietown-cslam/scripts
```

2) The way how CSLAM is used is the following:
Since the apriltag poses are needed on the device, one has to set the ROS_MASTER to the particular device. Therefore, change lines in the script *watchtowers-setup.sh* as follows:

```
ROS_MASTER_HOSTNAME=watchtower22
ROS_MASTER_IP=192.168.1.176

array=(watchtower22)
```

3) For autocharging a particular version of CSLAM will be run. Therefore, change the image tag for CSLAM container to autocharge. Make sure that the ACQ_DEVICE_MODE is set appropriately. Since we want to get apriltag poses in real time, this parameter has to ve set to live.

```
docker -H ${array[$index]}.local  run -d \
                                      --name cslam-acquisition \
                                      --restart always \
                                      --network=host \
                                      -e ACQ_DEVICE_MODE=live \
                                      -e ACQ_ROS_MASTER_URI_DEVICE=${ar-
ray[$index]} \
                                      -e ACQ_ROS_MASTER_URI_DE-
VICE_IP=127.0.0.1 \
                                      -e ACQ_SERVER_MODE=live \
                                      -e ACQ_ROS_MASTER_URI_SERV-
ER=${ROS_MASTER_HOSTNAME} \
                                      -e ACQ_ROS_MASTER_URI_SERV-
ER_IP=${ROS_MASTER_IP} \
                                      -e ACQ_DEVICE_NAME=${array[$in-
dex]} \
                                      -e ACQ_BEAUTIFY=1 \
                                      -e ACQ_STATIONARY_ODOMETRY=0 \
                                      -e ACQ_ODOMETRY_UPDATE_RATE=0 \
                                      -e ACQ_POSES_UPDATE_RATE=15 \
                                      -e ACQ_TEST_STREAM=1 \
                                      -e ACQ_TOPIC_VELOCITY_TO_POSE=ve-
locity_to_pose_node/pose \
                                      -e ACQ_TOPIC_RAW=camera_node/im-
age/compressed \
                                      -e ACQ_APRILTAG_QUAD_DECIMATE=2.0
\
                                     duckietown/cslam-acquisition:au-
tocharge || echo "ERROR: Starting cslam-acquisition on ${array[$index]}
failed. Probably this Watchtower wasn't configured properly or we can't
connect via the network."
```

4) Run the script

```
$ source watchtowers-setup.sh
```

After starting the container, make sure it is running. You have to see logs in CSLAM container as following:

```
...
[INFO/serverSideProcess] Published pose for tag 327 in sequence 10
[INFO/serverSideProcess] Published pose for tag 334 in sequence 10
[INFO/serverSideProcess] Published pose for tag 360 in sequence 10
[INFO/serverSideProcess] Published pose for tag 327 in sequence 11
...
```

7.2. For Module 1:

1) Start the CSLAM Container

Start the CSLAM container according to the instructions above.

2) Start the Charging Manager Container for Module 1

```
$ docker -H HOSTNAME.local run -it --net host --memory="800m" --
memory-swap="1.8g" --privileged -v /data:/data --name charging_manag-
er CONTAINTER_NAME
```

7.3. For Module 2:

1) Charging Manager

On the charging manager you have to start the following containers:

Start the TCP Server Container:

First clone the github repository for TCP server and then change the YAML file in it. There you have to change the IP address to the IP address of your charging manager:

```
$ git clone https://github.com/alifahriander/tcp_server.git
cd tcp_server/docker
```

Now change the IP address in default.yaml.
Then build, push and run the tcp_server container

```
$ docker build --rm -f "Dockerfile" -t IMAGE_NAME
$ docker push IMAGE_NAME
$ laptop $docker -H HOSTNAME.local pull IMAGE_NAME
$ docker -H hostname.local run -it --net host --memory="800m" --memo-
ry-swap="1.8g" --privileged -v /data:/data --name CONTAINER_NAME IM-
AGE_NAME
```

Start the Charging Manager Container for Module 2:

```
$ docker -H HOSTNAME.local run -it --net host --memory="800m" --
memory-swap="1.8g" --privileged -v /data:/data --name charging_manag-
er  anderalii/charging_manager:module2
```

2) Doorkeeper

Start the CSLAM Container:

Start the CSLAM container according to the instructions above.

Start the TCP Client Container:

First clone the github repository for TCP client and then change the YAML file in it. There you have to change the IP address to the IP address of your charging manager:

```
$ git clone https://github.com/alifahriander/tcp_client.git
cd tcp_server/docker
```

Now change the IP address in default.yaml.
Then build, push and run the tcp_server container

```
$ docker build --rm -f "Dockerfile" -t IMAGE_NAME
$ docker push IMAGE_NAME
$ laptop $docker -H HOSTNAME.local pull IMAGE_NAME
$ docker -H hostname.local run -it --net host --memory="800m" --memo-
ry-swap="1.8g" --privileged -v /data:/data --name CONTAINER_NAME IM-
AGE_NAME
```

Start the Doorkeeper Container:

Use the doorkeeper container you have run for setting up the reference tags

```
$ docker -H HOSTNAME.local run -it --net host --memory="800m" --
memory-swap="1.8g" --privileged -v /data:/data --name doorkeeper
CONTAINTER_NAME
```

UNIT C-8

Software setup on the Duckiebot

8.1. Goal

In order to allow the duckiebot to have the autocharging capability. The duckiebot requires to run a container which does the following tasks.

1) The duckiebot has to be guided from any place on the map to the charging area.
2) At the charging area, it understands the instructions of the autocharging manager, which directs it to the appropriate charging lane.
3) From the autocharging entrance the duckiebot follows the correct path to the charging lane
4) In the charging lane it successfully charges.
5) After charging the duckiebot navigates back to the city.

All these tasks are bundled in the megacity image. However, prior to running it, the appropriate paths of the autocharging area have to be setup.

8.2. Setting up container

Clone the following repository

```
git clone https://github.com/duckietown/rpi-autocharging-demo.git
cd rpi-autocharging-demo.git
```

open with an text editor the default.yaml file. The default.yaml file should have the following outline and structure:

```
# Maintenance gate
maintenance_entrance: {'62': 1, '143': 0}
maintenance_path: {'157': 1, '66': 0,'136': 2, '58': 1, '63': 0, '68':
2,'62': 1, '143': 0}
maintenance_exit: {'135': [1,2]}
maintenance_intersection: {'tag': 261}


# General path back to city
path_to_city: {'238': 1, '144':2, '260': 2, '149': 0, '153': 1, '237':
1, '236': 0, '242': 0, '61': 1}

# Charging station
charging_stations:
  stop_signs: {'tag': [190, 193, 37, 194]}
  station1:
    path_in: {'261': 1, '235': 2, '240': 1}
    path_calib: {'144': 2, '260': 1}
  station2:
    path_in: {'261': 2, '240': 0, '235': 1}
    path_calib: {'236': 0, '153': 0, '243': 2}
  station3:
    path_in: {'261': 1, '235': 1, '240': 1}
    path_calib: {'238': 1, '260': 1}
  station4:
    path_in: {'261': 2, '240': 1, '235': 1}
    path_calib: {'237': 1, '153': 0, '243': 2}
  entrances: {'235': [1,2], '240': [0,1]}
  exits: {'144': [0,2], 236': [0,2], '238': [1,2], '237': [0,1]}

# Calibration station
calibration_station:
  entrances: {'243': 2, '260': 1}
exits: {'242': [0,1]}
```

The parameters in the yaml file correspond to the following map:

picture of map

The parameters in the config file are dictionaries - each key (i.e. ‘150’) stands for an April tag ID and maps to either a single direction (i.e. 1) or to multiple directions, stored in a list (i.e. [0,1,2]). The directions, stored as integers, map as follows:

[0, 1, 2] == [LEFT, STRAIGHT, RIGHT].

8.3. Add your own paths

1) path_in

The “path_in” parameter of a charger should map traffic sign April tag IDs to a single turn type, which in sum guide the Duckiebot to the charger. In Figure 8.1 an example

is given. The path_in of charger 2 would then be

```
path_in: {'261': 2, '240': 0}
```

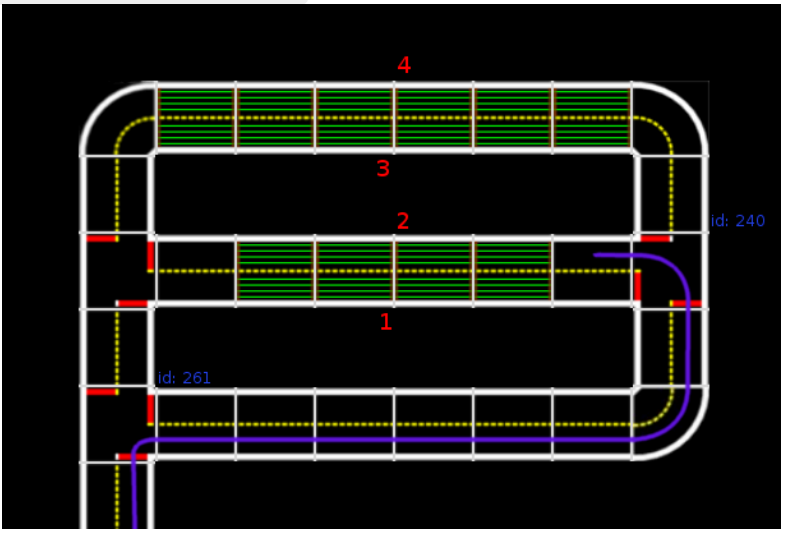


Figure 8.1. An example path from maintenance entrance to charger 2.

2) path_to_city

The dictionary “path_to_city” guides a Duckiebot from every possible leaving position (i.e. charger exit, calibration exit) back to the city. In Figure 8.2, all paths are plotted for an example maintenance area (without a calibration area).

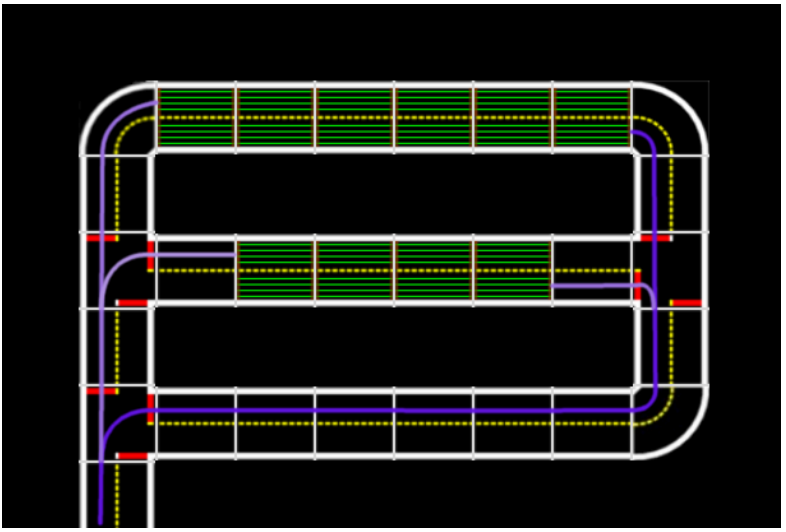


Figure 8.2. All possible exit paths from an example maintenance area (without calibration area).

3) charging_stations: entrances, exits

The dictionary “entrances” and “exits” in the charging_stations parameter contains

every entrance / exit to charging stations. This information is needed in the code to determine when a Duckiebot enters or leaves a charger.

4) maintenance_entrance / maintenance_exit

This dictionaries define which April tag IDs correspond to the entrance / exit of the maintenance area. This information is needed to detect when a Duckiebot enters or leaves the maintenance area.

After setting up all parameters our docker image is ready to be build

8.4. Push to your Docker account

we are building the docker file for your own charging area:

```
$ docker build --rm -f "Dockerfile" -t /[your docker reposito-
ry]/megacity:latest
```

[your docker repository] is the docker hub repository where the image will be saved, from which the image can be pulled to the duckiebot. example for generic autocharging area:

```
$ docker build --rm -f "Dockerfile" -t /duckietown/megacity:latest
```

now we need to push to the remote repository on dockerhub

```
$ docker push [your docker repository]/megacity:latest
```

UNIT C-9

Run the demo

The megacity container has to be built and uploaded to the duckiebot as outlined in here

Step 1 Power on your bot and wait for the `duckiebot-interface` to initialize (the LEDs go off).

Step 2: Launch the demo by running:

if we run the container for the first we need to pull and run the container on the duckiebot

```
$ docker -H hostname.local run -it --net host --privileged -v /data:/data --name megacity duckietown/rpi-duckiebot-base:megacity /bin/bash
```

if the container is already on the duckiebot but has been stopped we can execute

```
$ docker -H hostname.local start megacity
```

Note: Many nodes need to be launched, so it will take quite some time.

Unit C-10
Software architecture

Operating a Duckiebot inside a megacity requires software which can switch from state to state. In this section, we describe how we have fit the maintenance code into the bigger picture.

10.1. Overview

The software responsible for the autocharging capability is running entirely in the following two nodes:

1. Maintenance control node
2. Charging control node

10.2. Maintenance control node

The maintenance control main goal is to transition through the maintenance states that run parallel to the FSM states in order to execute the autocharging procedure. The maintenance states and their transitions are explained in the run_demo part. Each time a FSM state transition occurs the following log info is printed in the megacity container:

```
rospy.loginfo("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")
rospy.loginfo("Maintenance Control Node MT State: " + str(state))
rospy.loginfo("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")
```

If you would like to test a specific maintenance state (i.e. for testing a specific path like path_calib), change the state with

```
rostopic pub -1 "/<robot_name>/maintenance_control_node/set_state"
std_msgs/String "<state_name>"
```

There exist the following maintenance states: [WAY_TO_MAINTENANCE, WAY_TO_CHARGING, CHARGING, WAY_TO_CITY]

10.3. Charging control node

The charging control node is responsible for the duckiebots action when it arrived in the charging area. For example when the duckiebot is under the rails but there is no contact a callback functions triggers a wiggle in order to establish charging contact. Further is checks necessary condition for the duckiebot to leave the charger.

Operating a Duckiebot inside a megacity requires software which can switch from state to state. In this section, we describe how we have fit the maintenance code into the big-

ger picture.

10.4. Transition of states

Below you may find a descriptive explanation on the transition of states on Duckiebot for autocharging.

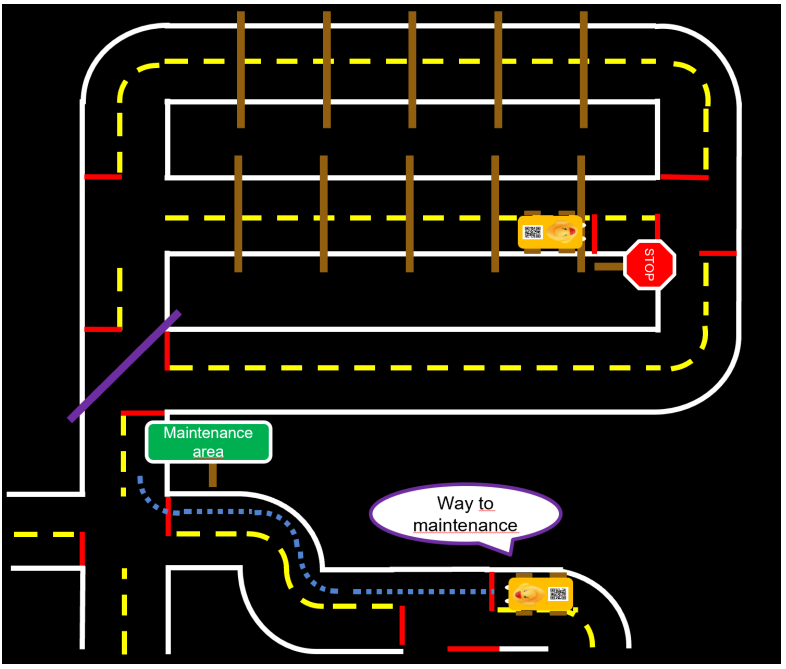


Figure 10.1. WAY_TO_MAINTENANCE

As soon as the megacity container is started, a Duckiebot is placed into the city and is able to drive around. A timer, the charge estimation algorithm or a human being triggers a topic called GO_TO_CHARGING from the maintenance control node in order to recognize the time when we need to put the Duckiebot into the charging area. When this state is switched from NONE to WAY_TO_MAINTENANCE, the Duckiebot is guided towards the entrance of the maintenance area by a fleet management algorithm(The responsible node for this algorithm is the action_dispatcher_node). During its journey it has the priority of way before other Duckiebots. (indicated by purple blinking LEDs).

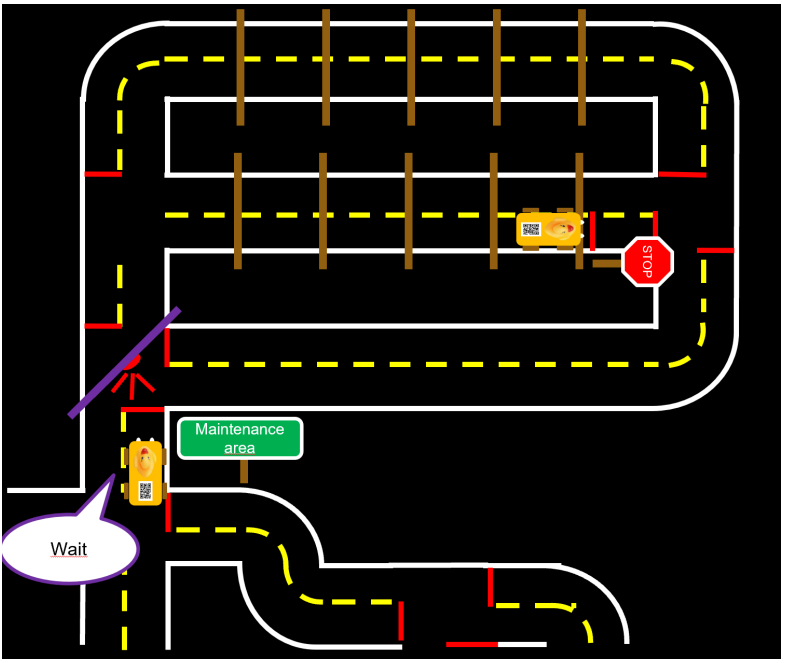


Figure 10.2. WAY_TO_CHARGING LED Detection

Having arrived at the entrance, this state will get switched to WAY_TO_CHARGING. On the maintenance intersection, the Duckiebot will recognize the maintenance apriltag and its FSM State will transit from INTERSECTION_COORDINATION to WAIT. There it will wait for 15 seconds and take multiple measurements for the frequency of charging manager's blinking LED light. The detected frequency will be determined according to the most measured frequency. Since every frequency corresponds to a charger index, the charger which Duckiebot should go will be set. After this, the FSM State will be set to INTERSECTION_COORDINATION and the journey to the charger begins.

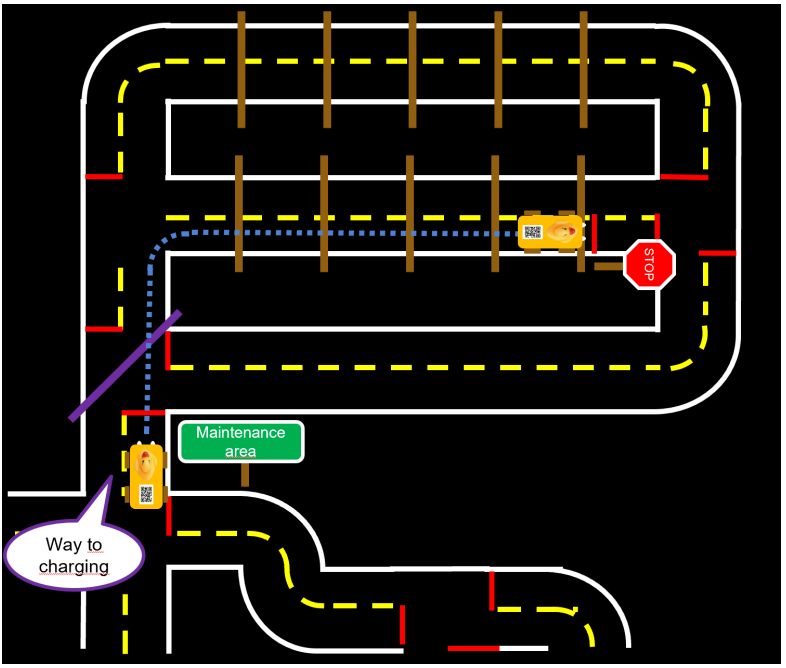


Figure 10.3. WAY_TO_CHARGING Driving to the charger

For every charger there is a predefined route, hence, the Duckiebot will take the turns to reach the charger according to the predefined route.

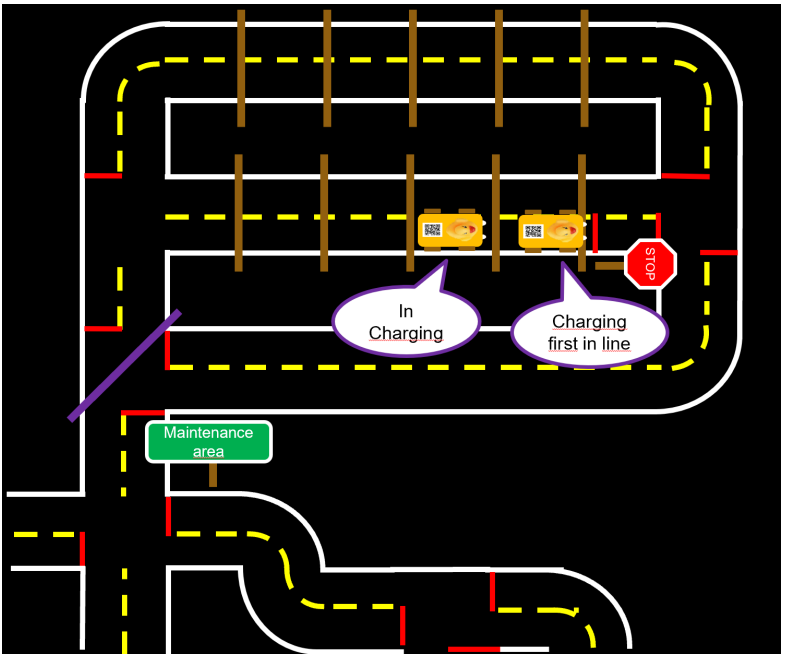


Figure 10.4. CHARGING

After entering the charger, a transition of state from WAY_TO_CHARGING to CHARGING follows and the finite state machine state switches to IN_CHARGING_AREA. These states mean that the Duckiebot is inside the charger and that it will keep distance from other Duckiebots queued up there. As soon as the Duckiebot sees a red line and a stop tag at the end of a charger the finite state is switched to CHARGING_FIRST_IN_LINE.

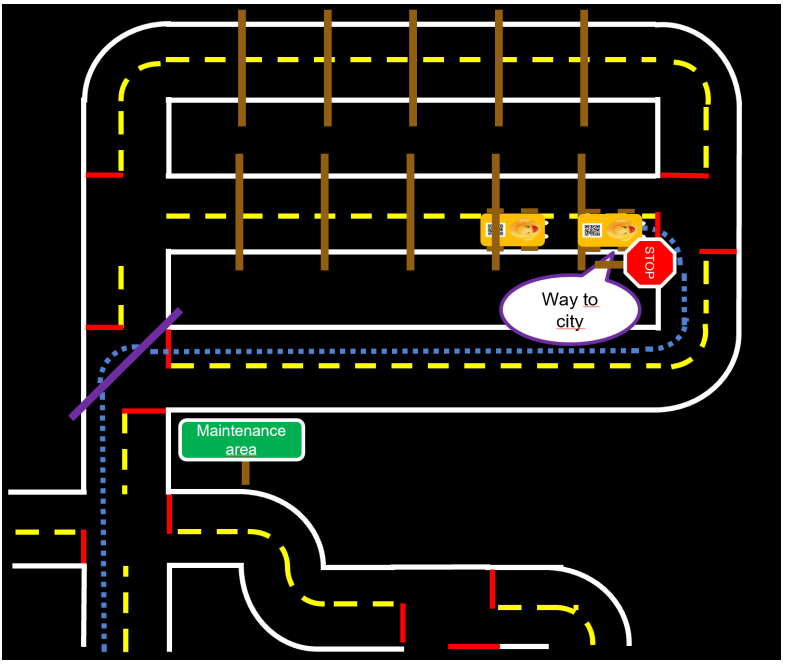


Figure 10.5. WAY_TO_CITY

At this given point of time the Duckiebot is the first one being able to leave the charger and it only needs to wait until it gets fully charged or someone triggers it to exit the charger and free another charging spot. 15 seconds before requesting to leave the charger, Duckiebot's lights on the backbumper will turn blue. In order to leave the charger, the Duckiebot needs to be charged for a certain amount of time and it should not detect another Duckiebot in front of it. While leaving the charger the finite state switches to LANE_FOLLOWING which makes the Duckiebot able to drive and the maintenance control node switches to WAY_TO_CITY. The Duckiebot is now guided out of the maintenance area. If the exit is reached, the maintenance control node is switched to NONE and the Duckiebot is again able to drive around in the city.

UNIT C-11

General Overview

In this part it is explained how the system for module 1 and 2 works.

11.1. Responsibilities of overall managing the chargers

There are 3 main tasks that have to be accomplished :

TASK 1- Evaluating the April tag's position in order to understand where the reference tags are located and in which direction the duckiebot went
TASK 2- Gathering information on charger sizes and finding the next free charger a duckiebot newly arriving the maintenance area should go
TASK 3- Selecting the frequency with which the LED should blink

In Modul 1, all the responsibilities are done by the charging manager. The output of tasks above feeds information to other tasks, for example, in task 1, it will be found out, which duckiebot entered or left the charger and this will be saved in a data structure that can be accessed by the charging manager. In task 2, charging manager will iterate over the data structure in order to discover how many duckiebots occupy the chargers. Through linear search, the most unoccupied charger will be determined and the frequency of LEDs corresponding charger id will be changed accordingly. In task 3 LEDs will blink with the mentioned frequency.

In Modul 2, the tasks are completed in separate devices. Task 1 is accomplished by doorkeepers and the rest will be done by the charging manager. The output of task 1 will be shared with the charging manager through a communication server. For this objective, charging manager serves a python-dictionary of duckiebot Apriltags as keys and charger indices as values (initially all charger indices are set to zero to indicate that the duckiebot does not occupy any charger.). The clients, in this case the doorkeepers, will update this dictionary according to their evaluation of April tag positions, for example, if duckiebot with April tag 400 enters charger 3 then the doorkeeper will change the value of 400 from 0 to 3.

11.2. Workflow in detail

Here you can find step-by-step workflow of module 1 and module 2

1) Preliminary information:

Before starting to read,note that every frequency with which the LED blinks, implies a charger the duckiebot should go. Also, CSLAM Acquisiton node provides many information. Only the data about the position of the center on the image will be used by charging manager(for module 1) or doorkeeper(for module 2).

2) Module 1:

Charging Manager:

1. Charging Manager's LED blinks with the initial frequency
2. **(TASK 1)** CSLAM Acquisition node provides information on the Apriltags in the scope of camera
In the scope of the camera, lies the reference tags and duckiebots' Apriltags if there are any duckiebots arrived to the intersection.
The positions of duckiebots' Apriltags and reference tags will be saved and updated, as the node publishes the topic.
Upon receiving these, the closest neighbor reference tag to duckiebot's apriltag and the timestamp will be computed and saved (Here minimal distance is defined by the minimal euclidean distance of two points on the image.)
(The first computed closest neighbor reference tag will be saved under the variable "first_neighbor" and the other results will be updated and saved under "last_neighbor".)
3. **(TASK 1)** Apriltag Evalutaion:
After not seeing an Apriltag of a duckiebot for more than threshold time, the "first_neighbor" and "last_neighbor" will be evaluated. That means, if the first_neighbor is the entrance tag and the last_neighbor is the direction2_tag, then Duckiebot went in direction2. Since direction2 corresponds to , for example, charger 2, the Duckiebot went in charger 2. According to this outcome the chargers dictionary will be updated and the tag of duckiebot which has entered/exited the charger i and the timestamp of entrance/exit will be saved/deleted under/from charger i's list.
4. **(TASK 2)** Charger sizes will be updated according to step 2
If the most unoccupied charger is changed, it will be noticed and new frequency corresponding the new charger with the most free spaces will be determined.
5. **(TASK 3)** LED blinks with the determined frequency in step 3

These steps will be repeated upon receiving the messages from acquisition node's topic

3) Module 2:

Preliminary information::

In this module a dictionary where keys are the duckiebots Apriltags and values are the charger indices or zero(indicating that the duckiebot is currently not occupying any charger) is served by the charging manager. The clients,namely the doorkeepers, will update this dictionary with charger indices if a Duckiebot enters a charger or with zero if a Duckiebot leaves a charger.

Doorkeeper:

1. **(TASK 1)** CSLAM Acquisition node provides information on the Apriltags in the scope of camera
In the scope of the camera, lies the reference tags and duckiebots' Apriltags if there are any duckiebots arrived to the intersection.
The positions of duckiebots' Apriltags and reference tags will be saved and updated, as the node publishes the topic.
Upon receiving these, the closest neighbor reference tag to duckiebot's apriltag and the timestamp will be computed and saved (Here minimal distance is defined by the minimal euclidean distance of two points on the image.)
(The first computed closest neighbor reference tag will be saved under the variable

“first_neighbor” and the other results will be updated and saved under “last_neighbor”.)

2. **(TASK 1)** Apriltag Evalutaion: After not seeing an Apriltag of a duckiebot for more than threshold time, the “first_neighbor” and “last_neighbor” will be evaluated. That means, if the first_neighbor is the entrance tag and the last_neighbor is the direction2_tag, then Duckiebot went in direction2. Since direction2 corresponds to , for example, charger 2, the Duckiebot went in charger 2. According to this outcome the chargers dictionary will be updated and the tag of duckiebot which has entered/exited the charger i and the timestamp of entrance/exit will be saved/deleted under/from charger i’s list.

3. **(Communication Server)** Sharing the information where duckiebot went After finding out where a duckiebot went, its value in the served dictionary will be either set to the charger index if it entered a charger or to zero, if it left a charger.

Charging Manager:

1. Charging Manager’s LED blinks with the initial frequency

2. **(Communication Server)** Charging Manager checks the dictionary updated by the doorkeepers periodically If the charger index of an Apriltag in the dictionary is changed, then the tag of duckiebot which has entered/exited the charger i and the timestamp of entrance will be saved/deleted under/from charger i’s list.

3. **(TASK 2)** Charger sizes will be updated according to step 2 If the most unoccupied charger is changed, it will be noticed and new frequency corresponding the new charger with the most free spaces will be determined

4. **(TASK 3)** LED blinks with the determined frequency in step 3

These steps will be repeated upon receiving the messages from acquisition node’s topic within doorkeepers

Unit C-12
Testing and debugging

Knowledge and activity graph

Requires: put requirements here

Results: put result here

Next: put next steps here

12.1. Debug the whole charging pipeline

If megacity is launched, a Duckiebot will drive for X minutes and then go to the charging station. After Y minutes, it will leave it again. X and Y are defined in the yaml file for the Charging Control Node. If you would like to test the whole charging procedure, place your Duckiebot on a road which ends in the intersection which is connected to the maintenance area. Then, request the Duckiebot to go charging:

```
rostopic pub -1 "/<robot_name>/maintenance_control_node/go_mt_charging"
std_msgs/Bool true
```

The Charging Control Node will as the TCP server for a free charging spot and if available reserve it.

Now, let your Duckiebot drive autonomously (L1 on your Joystick or 'a' on your virtual joystick).

12.2. Debug a single maintenance state

There exist the following maintenance states: [WAY_TO_MAINTENANCE, WAY_TO_CHARGING, CHARGING, WAY_TO_CALIBRATING, CALIBRATING, WAY_TO_CITY]

If you would like to test a specific maintenance state (i.e. for testing a specific path like path_calib), change the state with

```
rostopic pub -1 "/<robot_name>/maintenance_control_node/set_state"
std_msgs/String "<state_name>"
```

12.3. Troubleshooting

A Duckiebot gets stuck while traversing through a charger

This happens if the friction of the current collector is too high. Try to bend the 3D printed part a little down (multiple times) until the force acting on the charging rails is lower. You could also reprint the current collector with thinner connections (use the Customizer on Thingiverse). Also, ensure that the tiles of the charger are flat.

Duckiebot turns off as soon as rails are touched

Depending on the battery, if no voltage lies across the charging rails, the Raspberry Pi may reboot. This is a known issue and may be solved by adding capacitors to the add-on board - this is in work by Autolab Zurich.

Duckiebot does not charge

There exist multiple reasons for that: did you turn on the power supply? However, most of the times the current collector / the rails are dirty. Clean them with alcohol or sanding paper.

Duckiebot does not stop while waiting in the queue in the charger

Duckiebots are waiting in the queue with the help of vehicle detection and vehicle avoidance control nodes. You can solve this problem by increasing the desired_distance and minimal_distance parameters of vehicle avoidance control node.

Charging Manager was blinking with a frequency for charger 1 but the duckiebot did not drive in charger 1

There can be several reasons for this occasion. Duckiebot could drive to the unintended direction if the intersection control node did not work correctly. You can find this out by looking at the logs of the Duckiebot. You may see logs as following

```
driving to 0
```

This means that the Duckiebot wanted to go left. If the direction the duckiebot wanted to go and the direction it went agree, nothing from the intersection control was wrong. (If it is not the case, you can trim the parameters of unicorn_intersection_node.)

The second reason why this could happen is the LED detection. After arriving to the maintenance intersection, the Duckiebot waits for 15 seconds to detect the LED frequency of the traffic light. The image received by the camera is cut into three pieces. For traffic light detection, the upper part of the image is used. If the traffic light is not in this upper part of the image, the Duckiebot cannot detect the frequency of the LED. You can check this by looking at the led_detection_node/image_detection_TL topic: First start the megacity container. After the container is ready type in the following commands:

```
$ dts start_gui_tools HOSTNAME
```

First publish the topic in order to let the duckiebot drive to the maintenance area
container $ rostopic pub -1 “//maintenance_control_node/go_mt_charging” std_msgs/Bool true
container $ rqt_image_view
Select the topic led_detection_node/image_detection_TL

Now you are ready to debug the led detection process. The duckiebot will come to the maintenance intersection and the led detection node will be switched on. Then you will see a gray image. First check whether you see the LED on the image. If this is not the case, then either the placement of the traffic light was wrong or the Duckiebot arrived to the intersection not correctly. Then, check whether there is a blue circle around the LED.

Duckiebot drove in charger 1 but charging manager thought it went in charger 2

Check whether the reference tags are placed correctly following the instructions for "setup of reference tags".

UNIT C-13

Future projects

There are still multiple tasks which would improve the charging area by a lot. In the following, you find a list of those tasks with a small description, ordered in descending priority.

13.1. Improve the intersection navigation

For traversing through the maintenance area, intersection navigation is a crucial part. With the new Unicorn Intersection Node, the success rate was raised up to over 90%. For a completely autonomous city, this is still way too low.

13.2. Improve the vehicle detection

In order to keep distance to a Duckiebot in front, the vehicle avoidance node detects a circle grid pattern ($\mathbf{7 \times 3}$) and adapts the velocity. However, the detection only runs at $\mathbf{2}\boldsymbol{Hz}$ and fails sometimes.

13.3. Optimize CPU utilization

Currently, CPU utilization is at 100% for all four cores. The most expensive node seems to be the Image Transformer Node (Anti Instagram) which uses alone 30% of the CPU. This slows down the whole megacity.launch and could be cause for multiple bugs.

13.4. Implement the battery capacity estimation

Currently, the time a Duckiebot drives through the city and stays inside a charger are hard coded variables. With a high enough charging time and a low enough driving time, it is ensured that a Duckiebot never reaches a battery level of 0%. The drawback is that the efficiency in terms of the ratio $\frac{\boldsymbol{drivingtime}}{\boldsymbol{chargingtime}}$ is very low. This can be improved by implementing the battery capacity estimation: with the new add-on board, one is able to measure the current going into the battery and therefore integrate it to obtain the battery level.

13.5. Fix the stop line sleep time

In order to avoid stopping at the red line in opposite direction right after an intersection, there is a stop line sleep time implemented. This however does not work robustly - sometimes, that “bad” red line is still detected.

13.6. Consider a cooling system

After operating the Duckiebot for a long time, the Raspberry Pi gets really hot at its interfaces (USB plugs, LAN Port, etc). For a 24/7 operating time that can influence the behavior of the Duckiebot a lot. If one would cool the Duckiebot actively by a little fan, the temperature wouldn't increase that much.

13.7. Adapt autocharging to the newest version of CSLAM

Right now, the charger management runs the version of CSLAM, where apriltag processing is done on raspberry pi. In the newest version of CSLAM, the images of the watchtowers are acquired from the watchtowers and apriltag processing is done on the server computer. This increases the rate of apriltag processing. In order to adapt the usage of CSLAM by charging manager or doorkeeper, one has to let the raspberry pi get the apriltag poses, since they are needed on the device.

13.8. Hardware Redesign

The current collectors are made out of plastic which need to be bend appropriately so they touch the rails well enough to conduct current, but not to hard to not hinder the duckiebot from moving. This solution may be inappropriate in the long term, since the plastic wears out with time. A possible solution could be a hinge with two springs which will hold the current collector in its desired position.

PART D
Watchtowers

This section teaches you how to prepare your Watchtower system. This includes initialization and hardware assembly. The placement of the Watchtowers in the city and the whole localization system is described in the Localization System Section

1) SD card initialization

For Watchtowers and traffic lights we use naming conventions. Have a look at traffic light naming conventions or for information. To flash your SD card with the conventions mentioned in the links before, use these instructions.

2) Sections

(Pointer to beta/draft material that was removed - watchtower-hardware-assembly-WT19-B)

previous **warning** next (4 of 9) index

warning

```
This link points to beta/draft material removed
```

Location not known more precisely.
Created by function `n/a` in module `n/a`.

: This is the currently shipped and used version for Duckietowns and Autolabs around the world.

- Unit D-2 - BUILDING - Watchtower in WT18 configuration: This version is out of production but the section is still left here for reference.

Below you find the pictures of assembled Watchtowers in the above configurations.

(a) WT19-B configuration.

WT18 configuration.

(b) WT18 configuration.

Figure 0.1. Assembled Watchtowers in different configurations.

UNIT D-1

Watchtower Initialization

KNOWLEDGE AND ACTIVITY GRAPH

Requires: An SD card of size at least 32 GB.

Requires: A computer with a **Ubuntu OS** (for flashing the SD card), an internet connection, an SD card reader, and 16 GB of free space.

Requires: Duckietown Shell, Docker, etc, as configured in Unit C-1 - Setup - Laptop.

Requires: Duckietown Token set up as in Unit C-2 - Setup - Account.

Results: A ready watchtower

Next: Place the Watchtower in the city.

1.1. Flashing the SD card

The image setup procedure for Watchtowers is the same as for Duckiebots.

In the Autolab of ETH Zurich, we use the naming convention:

- linux-username: **mom**
- hostname **watchtowerXX** (where XX specify the number of the Watchtower)
- password **MomWatches**.

Note: **Important** : please add `--type watchtower` to the flashing procedure

Note: **Important** : please make sure you have set your dts to version `daffy` using the command `dts set-version --daffy`.

Note: **For RaspberryPi 4**, add `--experimental` to the command

A complete command will look like:

```
$ dts init_sd_card --hostname watchtowerXX --linux-username mom --
linux-password MomWatches --country COUNTRY --type watchtower --
experimental
```

Using the above naming conventions, you can flash your SD cards as is described in Duckiebot Initialization.

1.2. Calibrating the camera

Using the instruction in Unit C-12 - Calibration - Camera, you should do only the intrinsic calibration for the Watchtowers.

Note: Be sure to check the quality of the image. It should be (1296x972) and not (480*640) like on the Autobots. If it is not, this means you didnt flash the Watchtower with the `--type watchtower` argument. To solve this, reflash it.

UNIT D-2

BUILDING - Watchtower in WT18 configuration

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Material: Watchtower components. To obtain them contact info@duckietown.org.

Requires: An initialized SD-card.

Requires: Tools: (strong) wood glue or hot glue gun, tape, double-sided tape.

Results: An Assembled Watchtower system in WT18 configuration.

Next: The next step is to initialize the SD card.

2.1. Before Assembly

1) From Wooden Plates to Chasis Parts (Laser Cut):

The plates should be 4mm thick. To cut a set of traffic light, you need a plates with the size of roughly 347mm x 256mm x 4mm.

2) Tubes Preparation:

We only need one tube here. The spec of the tube is listed directly below.

TABLE 2.1. SYMBOLS TO ESCAPE

Length:	572mm
Diameter:	20mm

We recommend you to buy a tube cutter. It'll save you from lots of troubles. After cutting the tube, you could polish the edges so that it won't hurt you.

2.2. Watchtower Assembly

1) Wooden Box (with Raspberry Pi)

These are all the parts for assembly the wooden chasis.

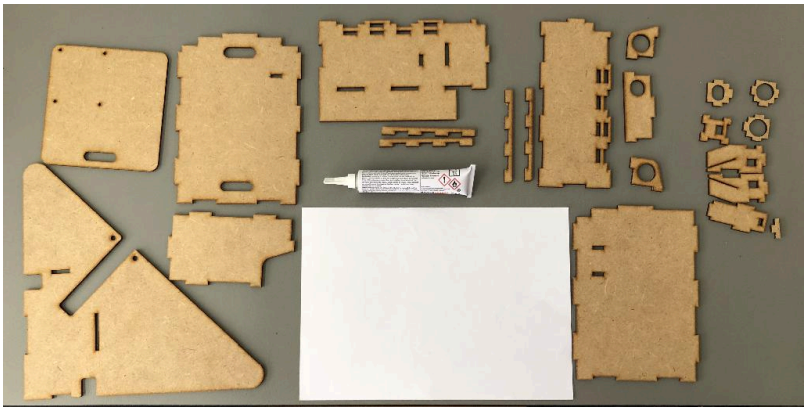


Figure 2.1. The parts for wooden box assemble.

Step 1: Stick the sliders

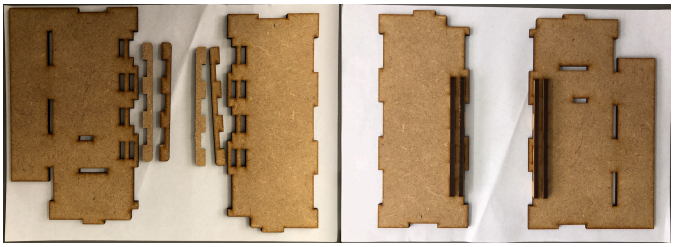


Figure 2.2. Step 1

Step 2: Combine two walls of the box

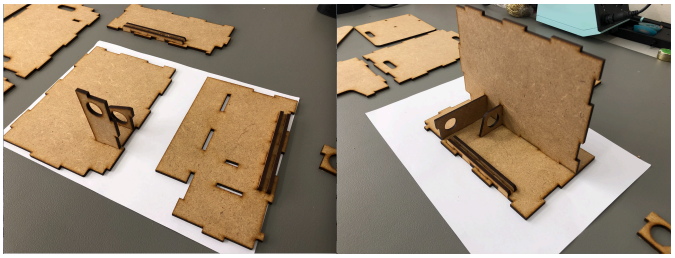


Figure 2.3. Step 2

Step 3: Stick the third wall of the box

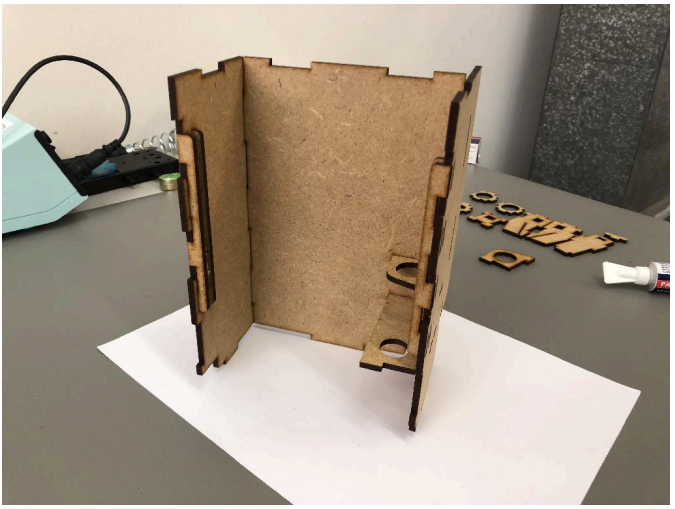


Figure 2.4. Step 3

Step 4: Stick the forth wall of the box

Figure 2.5. Step 4

Step 5: Stick the stabler for the tube

Stick the small part, but DON'T stick the large plate. It should be the lid for the box that can be opened so that we could put Raspberry Pi in.

Figure 2.6. Step 5

Step 6: Get the parts for camera mount

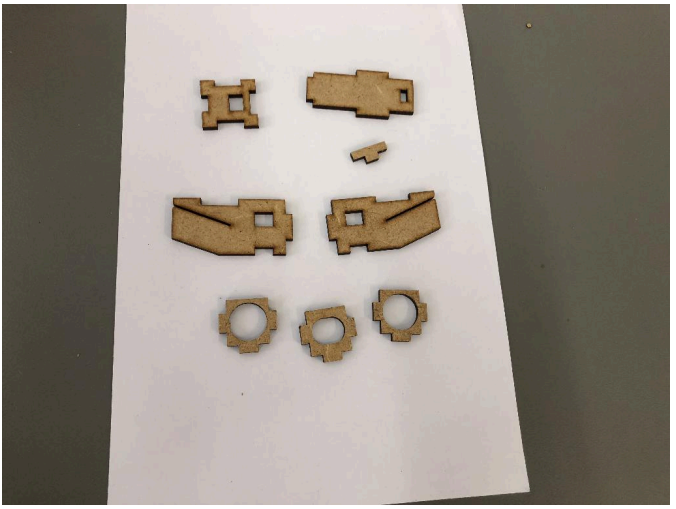


Figure 2.7. Step 6

Step 7: Stick the stablers for tubes on camera mount

Note that there're two parts with larger holes and one part with a smaller hole. Tubes can go through lager hole but not smaller one. Thus, you should place the parts with smaller on the very top.

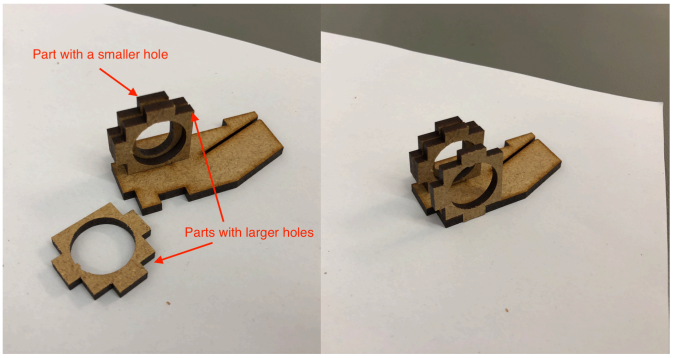


Figure 2.8. Step 7

Step 8: Stick the other side of camera mount

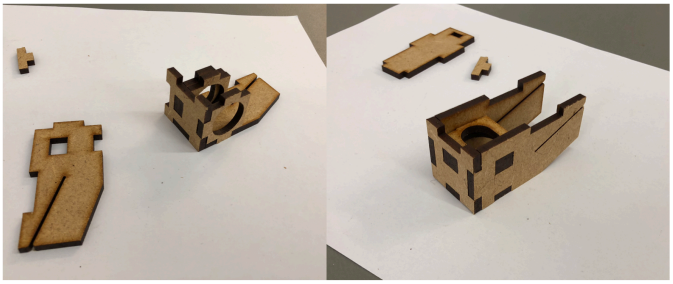


Figure 2.9. Step 8

Step 9: Stick the little T shape part on the lid of camera mount

You don't need to stick the lid on the camera.

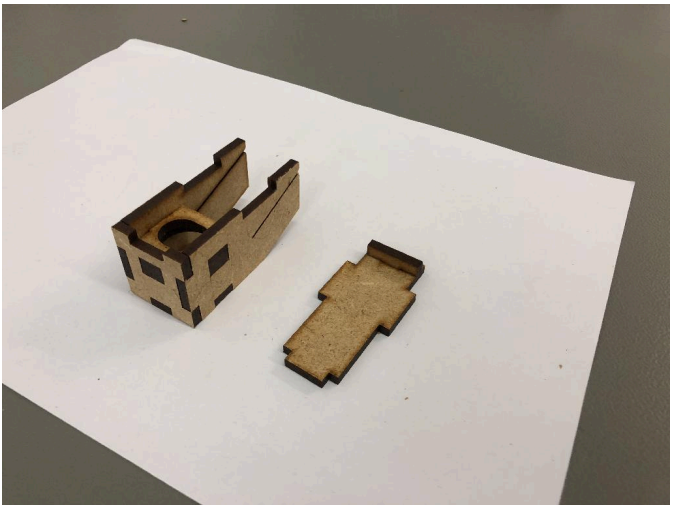


Figure 2.10. Step 9

Step 10: Parts you need for Raspberry PI holder.

Including the plate, two M2.5, 12mm Spacers, two M2.5, 6mm Screws, two M2.5, 4mm Screws and one Raspberry PI 3B+ Model.

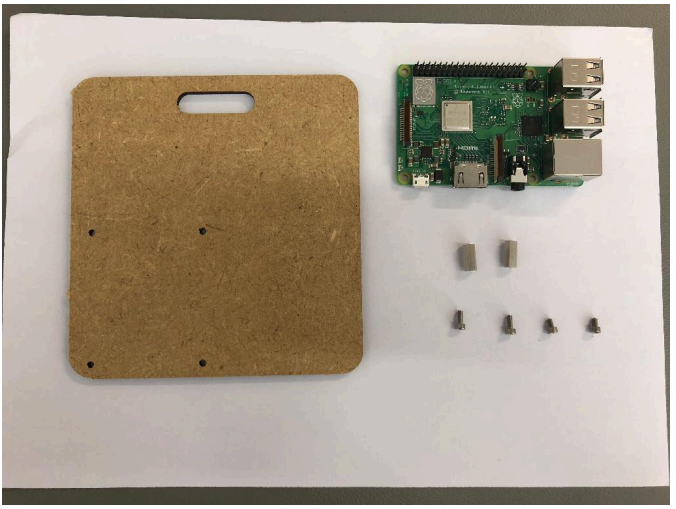


Figure 2.11. Step 10

Step 11: Install the spacers with the 6mm screws.

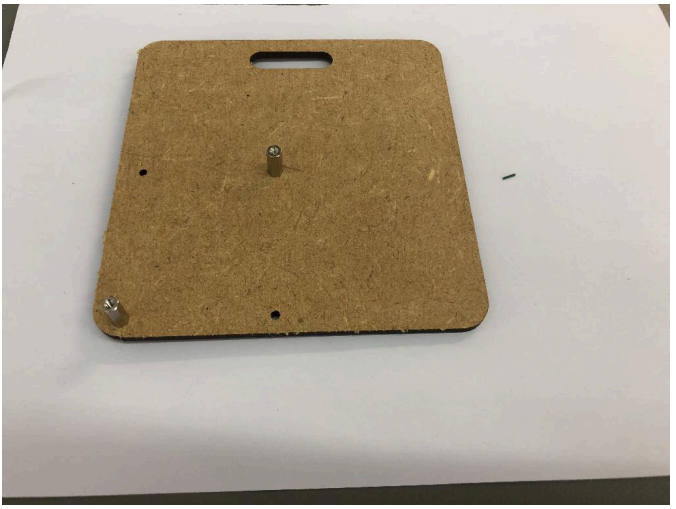


Figure 2.12. Step 11

Step 12: Install the RPI with the 4mm screws.

Figure 2.13. Step 12

After this step, we're finished with the chassis. Now let's assemble everything.

Step 13: Put tubes through the chassis.
Step 14: Install camera mount on the very top of the tube.
Step 15: Install the camera cables on Raspberry PI and let it go through the tube.
Step 16: Install camera on the other side of the cable and put it on the camera mount.
Step 17: Final Results

You finished the assembly! You can now prettify the tube according to your own taste. The final result should look like Figure 2.14.

Figure 2.14. completely assembled Watchtower in WT18 configuration

Part E
Localization System

Knowledge and activity graph

Requires: A fully operational Duckietown (unknown ref opmanual_duckietown/duckietowns)

previous **warning** next (5 of 9) index

warning

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

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

Location not known more precisely.

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

Results: A working localization system to localize all Autobots inside the Autolab

Next: The first step is to build the Watchtowers

The localization system is an important part of the Autolab, as it gives the poses of all Autobots in the city to a server. In the Autolab, the Autobots are Duckiebots that have been improved to Autolab specifications (see Unit B-1 - Autobot specifications and assembly).

Note: Watchtowers are an experimental feature of Duckietown, which are currently only used in Autolabs.

2) Sections

Contents

3) Overview and Concept of the system

The localization system serves multiple purposes:

- Automating the Autolab completely:
- It allows the central system to discover if a Autobot moves out of the track
- It allows more high level fleet control
- It allows the grading of the embodied challenges for AIDO

The localization system mainly relies on AprilTags, which are a conceptually and visually similar to QR codes. Each Autobot is provided with a mounting plate with an AprilTag on top. By tracking these mounted AprilTags, the system evaluates the Autobot poses by solving an optimization problem.

To track the AprilTags, we use Watchtowers. In the spirit of Duckietown, a Watchtower uses most of the same hardware and software as a Duckiebot. A Watchtower is essentially a little tower which is about 60cm in height, and has a pi-camera at the top to look over a part of the city. The Watchtowers are spread all over the city, and by combining the field of view of each tower, it is possible to cover the whole Autolab.

Apart from AprilTags on Autobots, there are other AprilTags that are on the ground, called Ground AprilTags. The precise location for each Ground AprilTag is known in advance. We build a pose graph of all the relative poses bewteen Watchtowers and Autobots, and what they see. By running optimization on the graph, we merge the local influx of data from all agents into a global position graph of all agents, using the Ground AprilTags as global fixed references.

4) BUILDING - Hardware

There are two structural elements required to have a working system:

- The Watchtowers
- The ground AprilTags

The localization system is designed such that Watchtowers don't need to be at a specific height or position as long as they can view "sufficient" area of the Autolab. (TODO: specify what "sufficient" means) However, we still provide the specs of the Watchtowers so that you can produce your own Watchtowers. See hardware part in chapter Part D - Watchtowers.

Moreover, the ground AprilTags need to follow conventions specified in the chapter Unit E-2 - BUILDING - Apriltags specifications.

5) DEMOS - Running Localization

Localization can be run either online or offline.

Running localization *online* means that the data is processed *during* the experiment and the results can be vizualised with some delay. The demo is found at Unit E-3 - DEMO - Localization

Running localization *offline* means that the data recorded but processed only *after* the experiment. The demo is found at Unit E-3 - DEMO - Localization

While the long term objective is to only do online localization, the offline localization has proved very useful for AIDO, because it requires less computing power and does not have the network bandwidth as a bottleneck.

6) SOFTWARE - Description

The software is explained in detailed in chapter Part F - Localization System - Software explanation

UNIT E-1

BUILDING - Placing Watchtowers in a city

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Assembled Watchtowers.

Results: A Watchtower system ready to be used.

Contents

1.1. Placement of the watchtowers

This is a picture of the current Autolab in ETHZ. It will give you a quick approximate of how to place your watchtowers and how many you should put.

Figure 1.1. Picture of the Zurich ETH Autolab.

There's only two general rule of putting Watchtowers in a city.

- First, make sure that the field of views of Watchtowers do cover the whole city.
- Second, there should be enough overlapping between field of view between Watchtowers.

Below is a more synthetic view of the watchtower placement in ETHZ. The ratio of watchtower to road tiles is around 2/3.

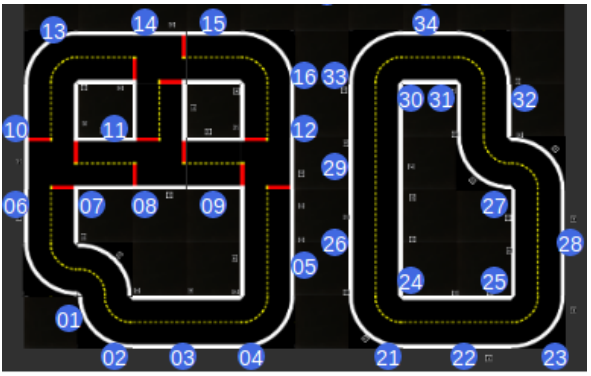


Figure 1.2. Overview of the autolab and of the watchtower placement. Each number shows the position of a watchtower. Watchtowers number 01 to 16 are on the left loop, while watchtower 21 to 35 are on the right loop.

Below is the field of view of watchtowers. Please keep in mind that they cover approximately 3 tiles each (even more in some cases), but that on the edge of the image, despite rectification, the image is distorted and thus the apriltag detection might give inaccurate results. This is why overlapping field of view are important. The more the better.

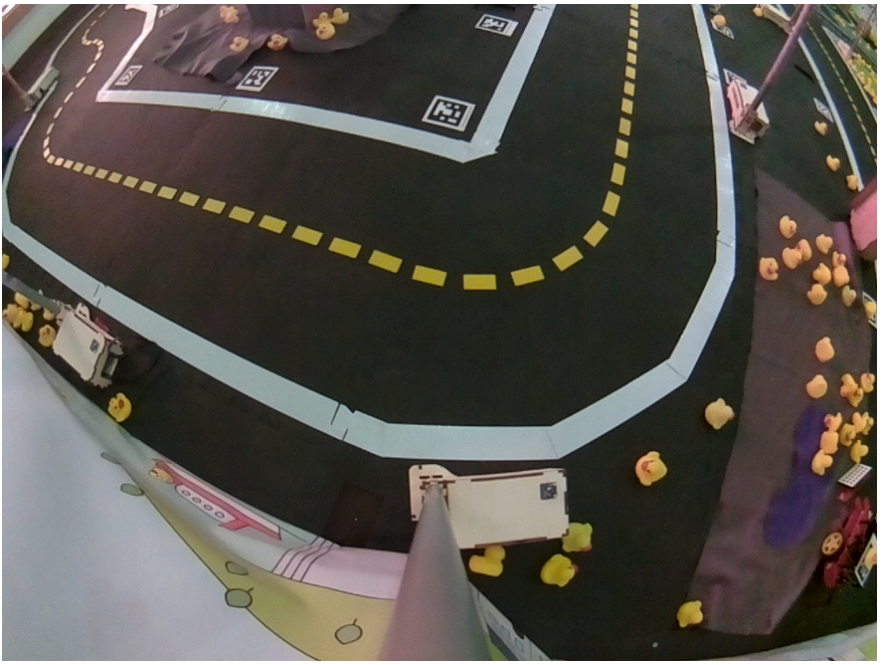


Figure 1.3. View from a watchtower in a corner. It covers approximately 3 tiles.

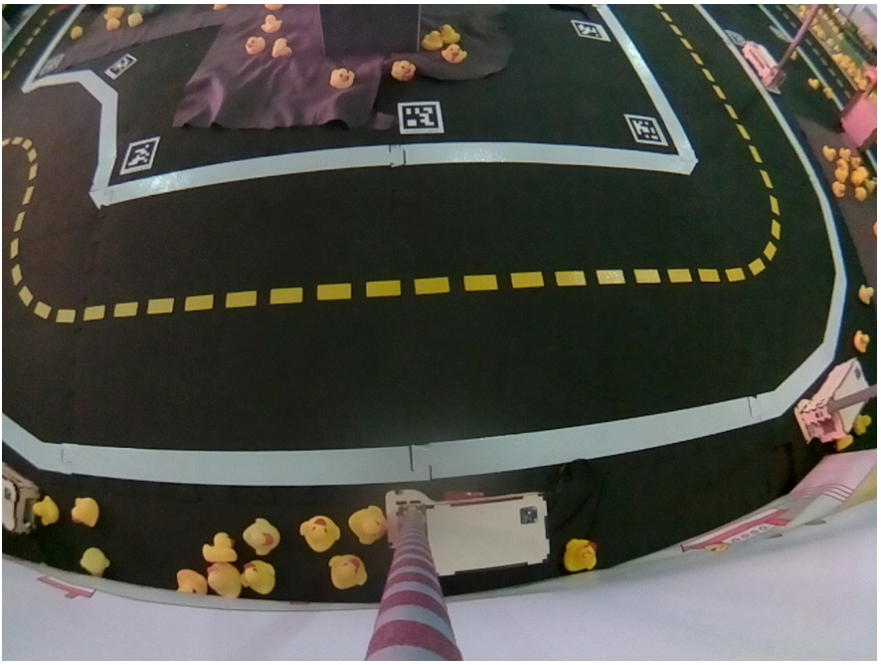


Figure 1.4. View from a watchtower on a straight line. It covers approximately 3 tiles.

1.2. Connection of the watchtowers

Beside the placement of Watchtowers, they should be connected via ethernet cables. At ETH, the Watchtowers are first connected to switches, that are connected to our router. At ETHZ, all of the Autolab is on a stage, and we pull the cables directly from underneath it, to keep the track clean from cables.

Then we have one PC connected via ethernet to the router, for faster and more reliable communication with the watchtowers.

1) Recommended network setup

Since all Watchtowers might send data at the same time, we recommend you have a strong network setup:

The best, and currently used in the ETHZ Autolab, is a switch with 1 Gb/s ports for each watchtower, with 2 ports at 10 Gb/s, that are connected to a central computer (which in turn needs a 10 Gb/s network card).

2) Power management

Last but not least, these Watchtowers needs *power*. Remember to prepare your USB chargers and cables that support 2.4A output.

> Note: EXPERIMENTAL : If you want to get less cables around, the possibility of using POE (power over ethernet) exists. With a additional hat, you can power the pi directly by the ethernet cable (provided that you switch provides the service as well).

UNIT E-2

BUILDING - Apriltags specifications

KNOWLEDGE AND ACTIVITY GRAPH

Requires: A fully operational Duckietown (unknown ref opmanual_duckietown/duckietowns)

previous **warning** next (6 of 9) index

warning

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

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

Location not known more precisely.

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

, compliant autobots and a map of the Autolab.

Results: The city is ready to be used for localization

Next: The two localization demos: offline and online

Contents

2.1. Why do we need ground Apritags

Watchtowers can move around very easily. Collisions with duckiebots and humans are happening all the time. To create a robust way of having a global localization in the autolab, we decide not to fix the Watchtowers.

Instead, we fix ground apriltags on the ground, and we measure and record their positions in the map. This provides enough ground truth for the localization system.

2.2. Making the ground Apriltags

As a reminder, the Apriltags already have predetermined sets of usecase. The ranges of tags are specified in Table 2.1.

For localization, **the ground Apriltags are in range 300 to 399**.

Note: For the Autobots themselves, the tags are in range 400 to 439.

TABLE 2.1. APRILTAG ID RANGES

Purpose	Size	Family	ID Range
Traffic signs	6.5cm x 6.5cm	36h11	1-199
Traffic lights	6.5cm x 6.5cm	36h11	200-299
Localization	6.5cm x 6.5cm	36h11	300-399
Autobots	6.5cm x 6.5cm	36h11	400-439
Street Name Signs	6.5cm x 6.5cm	36h11	440-587

To print the ones you need, you can find them here: pdf

2.3. Placing the ground Apriltags and the Watchtowers

At this point, you have already decided where to place your Watchtowers.

Now, you need to place the ground apriltags. Once they are printed, place them with double sided tape respecting the following convention:

- The ground Apriltags are outside of roads
- The ground Apriltags are near the side of the road, and the name of the tag is oriented toward the road (see figure).
- The ground Apriltag's orientation is a multiple of 45°, meaning there is only eights possible orientations.
- The only ground Apriltags not oriented in multiple of 90° are the ones on the outside of curve lanes. (see figure)

> Note: Try to put at least two ground Apriltag in the field of view of each watchtower (see Figure 1.4), and try to have Apriltags that are seen by multiple Watchtowers at the same time. This will improve the robustness of the localization graph.

2.4. Generating the map in duckietown-world

You should already have a map as explained in Unit B-4 - Autolab map. If not, go back and do it, as it will be necessary for the rest.

2.5. Adding the ground Apriltags localization to your map

1) Before measuring

This is the important part of the Apriltag specifications. You need to make sure that:

- The ground Apriltags are indeed 6.5cm x 6.5cm. If they are not, the localization system will get flawed data and thus will be useless.
- The ground Apriltags are very well fixed to the ground (using tape and nails). Once they are measured they should not move at all.
- The ground Apriltags angles are multiples of 45°. This will make the following much easier.

2) Measuring

- The measure of the ground Apriltags needs to be very precise. You should have a meter with millimeter precision.
- In the map you created before, **the origin is the bottom left corner**. Remember this as it is *important*.
- The tiles have an internal and an external border, because of the interlocking slots. In the following, as for the map, take **the inside bottom left as reference for a tile**. (see Figure 2.1)
- Each apriltag placement will be measured relatively to the tile it is on, from the above described origin.
- Always measure the center of the April tag itself.

How to measure:

In the following, you will be asked 5 numbers for each Apriltag:

- the x coordinate of the tile : it is the number (starting at 0 at the first bottom left origin tile) of tile along the x axis (bottom axis)
- the y coordinate of the tile : same as for x, it starts at 0
- the x measure of the Apriltag on the tile : you can get it by blocking your meter in the interior left edge and measuring (in meters) the distance from the interior edge to the center of the Apriltag.
- the y measure of the Apriltag on the tile : you can get it by blocking your meter in the interior bottom edge and measuring (in meters) the distance from the interior edge to the center of the Apriltag.
- the orientation of the apriltag (in degrees) : it is 0 if the Apriltag's name is aligned normally with the x axis (eg readable from the "bottom" of the map). Then it is defined with the trigonometric convention (counter-clockwise). This should normally always be in multiples of 45°.

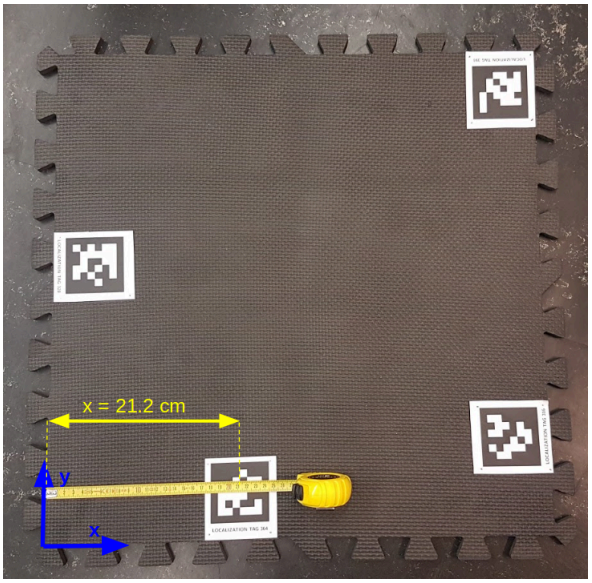


Figure 2.1. We measure from the inside band from the bottom left inner corner, defined as the origin of the tile

3) Filling the map in

Once you are sure of your positioning of the Apriltags, you can start measuring them. To do so:

- Open a terminal inside the duckietown-world repository, as you did to create your map. Your map should still be in the `src/duckietown_world/data/gd1/maps` folder.

Inside the `duckietown-world` repository, run the following command:

```
$ python3 src/apriltag_measure/measure_ground_apriltags.py
MAP_NAME
```

- Follow the instructions in the terminal : choose an Apriltag number, and fill in the 5 asked numbers, as described above.
- If an Apriltag was already filled in before (if you are changing your map for instance), you will be asked to confirm the overwriting of the positioning. As everything is versioned in github, you can always go back to find the previous positions if need be.
- If you try recording an Apriltag number than is not in the allocated range (300-399), the script will also ask to confirm.
- At the end, just confirm the saving. The resulting map will be where it was before, with now the Apriltag measures added to it.
- As described in Unit B-4 - Autolab map, you should recompile your map and visualize the apriltags on it (easy debug to find obvious mistakes).

UNIT E-3

DEMO - Localization

KNOWLEDGE AND ACTIVITY GRAPH

Requires: A fully operational Duckietown (unknown ref opmanual_duckietown/duckietowns)

previous **warning** next (7 of 9) index

warning

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

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

Location not known more precisely.

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

, compliant autobots, watchtowers and a system of ground April tags

Results: running offline or online localization in the Autolab

Next: Contribute to localization software

Contents

3.1. Setting up your master computer

The localization pipeline needs a **master** computer that will receive all information and process it. In order for everything to work, you need a kinetic roscore running at all times, and it needs to be run first.

```
$ docker run --name roscore --rm --net=host -dit duckietown/dt-ros-
commons:daffy-amd64 roscore
```

If this container is stopped at some point, then all the acquisition bridges (see below) need to be restarted, as they need connection to this rosmaster.

3.2. Setting visualization (optional)

To set up an rviz visualization, run first :

```
$ xhost +
```

Then, remembering the fork of duckietown-world on which your map is, and remembering the name of the map, run:

```
$ docker run -it --rm --net=host --env="DISPLAY" -e ROS_MASTER=COMPUTER_HOSTNAME -e ROS_MASTER_IP=COMPUTER_IP -e DUCKIETOWN_WORLD_FORK=YOUR_FORK -e MAP_NAME=YOUR_MAP duckietown/dt-autolab-rviz
```

This should first show just the map with the tiles. When the graph optimizer runs (later down), the position that are calculated will show on this visualization.

3.3. Setting up software on the watchtowers

In order to use the localization pipeline, you need to have two containers running on the watchtowers:

- duckiebot-interface (a special version)
- acquisition-bridge

1) The duckiebot-interface for watchtowers

The watchtowers need a slightly modified version of the duckiebot interface.

The following commands should be run on every watchower. If you named them with numbers (Watchtower01 to watchtowerXX), then you can easily make for loops in shell.

First, remove the duckiebot interface that is running:

```
$ docker -H hostname.local rm -f dt18_03_roscore_duckiebot-interface_1
```

Then, pull the custom image

```
$ docker -H hostname.local pull duckietown/dt-duckiebot-interface:daffy-arm32v7
```

Then, launch it:

```
$ docker -H hostname.local run --name duckiebot-interface --privileged -e ROBOT_TYPE=watchtower --restart unless-stopped -v /data:/data -dit --network=host duckietown/dt-duckiebot-interface:daffy-arm32v7
```

2) The acquisition-bridge for watchtowers

In order to get the images from all watchtowers to the same rosmaster (your computer), we launch an acquisition bridge, whose role for the watchtowers is to publish camera image and camera information, but only when movement is detected. This way, we only get the data that is needed for localization.

To run this, please run on all watchtowers:

```
$ docker -H hostname.local run --name acquisition-bridge --net-
work=host -e ROBOT_TYPE=watchtower -e LAB_ROS_MAS-
TER_IP=YOUR_ROS_MASTER_IP -dit duckietown/acquisition-
bridge:daffy-arm32v7
```

3) Advice

Always leave your roscore running, and always leave the duckiebot-interface and the acquisition-bridge of the watchtowers running. This will make all subsequent processes much faster to launch.

3.4. Setting up software on the duckiebots

The duckiebots require the same two containers, duckiebot-interface and acquisition-bridge.

1) The acquisition bridge for duckiebots

The default duckiebot-interface is good enough for the duckiebots, so we will just add the acquisition-bridge.

On all duckiebots please run:

```
$ docker -H hostname.local run --name acquisition-bridge --net-
work=host -v /data:/data -e LAB_ROS_MASTER_IP=YOUR_ROS_MASTER_IP
-dit duckietown/acquisition-bridge:daffy-arm32v7
```

3.5. Checking if the acquisition bridges work

On your PC, you should now be able to get the image stream of all connected duckiebots, using rqt_image_view.

If you have ubuntu18 with melodic, rqt_image_view might not show the images. If so, use:

```
$ dts start_gui_tools PC_name
```

Then run rqt_image_view from there.

3.6. Offline localization

The offline localization is offline in the meaning that you only get a trajectory of your duckiebots after the experiment is over. The process is the following:

- You record an experiment
- You process the bag you get to extract apriltag poses and odometry
- You run the graph optimizer on this intermediary result and it gives you the trajectory of all duckiebots in yaml files.

1) Recording the experiment

When you are ready to start an experiment, on your master PC, run rosbag:

```
$ rosbag record -a -O BAG_NAME.BAG
```

and stop it at the end of the experiment.

2) Processing the apriltags and odometry from the bag

First, you need to know where your bag is. The folder containing it is referred as `PATH_TO_BAG_FOLDER` in the following. We recommend you create new separate folders for each experiment (with date and/or sequence number).

```
$ docker run --name post_processor -dit --rm -e INPUT_BAG_PATH=/data/BAG_NAME.BAG -e OUTPUT_BAG_PATH=/data/processed_BAG_NAME.BAG -e ROS_MASTER_URI=http://YOUR_IP:11311 -v PATH_TO_BAG_FOLDER:/data duckietown/post-processor:daffy-amd64
```

When the container stops, then you should have a new bag called `processed_BAG_NAME.BAG` inside of your `PATH_TO_BAG_FOLDER`.

3) Launching the graph optimizer

Remember from Unit B-4 - Autolab map that you created a map. Now is the time to remember on which fork you pushed it (the default is `duckietown`), and what name you gave to your map. The map file needs to be in the same folder as the rest of the maps. They are respectively the YOUR_FORK_NAME and YOUR_MAP_NAME arguments in the following command line.

To run localization, execute:

```
$ docker run --rm  -e  ATMSGS_BAG=/data/processed_BAG_NAME.BAG -e OUTPUT_DIR=/data -e ROS_MASTER=YOUR_HOSTNAME -e ROS_MASTER_IP=YOUR_IP --name graph_optimizer -v PATH_TO_BAG_FOLDER:/data -e DUCKIETOWN_WORLD_FORK=YOUR_FORK_NAME -e MAP_NAME=YOUR_MAP_NAME duckietown/cslam-graphoptimizer:daffy-amd64
```

The poses can then be visualized in Rviz as the optimization advance.

The trajectories will be stored in the folder `PATH_TO_BAG_FOLDER`.

3.7. Online Localization

> **Note:** This is highly experimental, as up until now the processing power required to run localization online is really heavy. The goal of the current development is to make the process affordable for a single computer

Online localization is the idea of running an experiment and getting (with a reasonable delay) the localization and path of each duckiebot The processing bottle neck is on the processing of the April tags from the watchtower images.

Normally, at this point, you should have a duckiebot-interface and a acquisition bridge on each device (duckiebot and watchtower).

1) Online processing

For each Watchtower that is running do on your computer :

```
$ docker run --name apriltag_processor_WATCHTOWER_NUMBER --net-
work=host -dit --rm  -e ROS_MASTER_URI=http://YOUR_IP:11311 -e
ACQ_DEVICE_NAME=WATCHTOWER_NAME duckietown/apriltag-proces-
sor:daffy-amd64
```

Where `WATCHTOWER_NUMBER` is just 01 to XX and `WATCHTOWER_NAME` is the hostname of the Watchtower (usually it is `watchtowerXX`).

For each Autobot that is running do on your computer :

```
$ docker run --name odometry_processor_AUTOBOT_NUMBER --net-
work=host -dit --rm  -e ACQ_ROS_MASTER_URI_SERVER_IP=YOUR_IP -e
ACQ_DEVICE_NAME=AUTOBOT_NAME duckietown/wheel-odometry-proces-
sor:daffy-amd64
```

Where `AUTOBOT_NUMBER` is just 01 to XX and `AUTOBOT_NAME` is the hostname of the Autobot (usually it is `autobotXX`).

> **Warning:** The processing of apriltags is very heavy. Putting more than 4 processors on a computer is very risky. What you can do is use other computers that are on the same network. Launch exactly the same command and be sure to leave the IP of the designated master computer.

2) Localization

Once the online processing is started (or even before), run:

```
$ docker run --rm -e OUTPUT_DIR=/data -e ROS_MASTER=YOUR_HOST-
NAME -e ROS_MASTER_IP=YOUR_IP --net=host --name graph_optimizer
-v PATH_TO_RESULT_FOLDER:/data -e DUCKI-
ETOWN_WORLD_FORK=YOUR_FORK_NAME -e MAP_NAME=YOUR_MAP_NAME
duckietown/cslam-graphoptimizer:daffy-amd64
```

The `PATH_TO_RESULT_FOLDER` folder is the one where the results will be saved in yaml

files at the end of the experiment, when you CTRL+C the above command to finish.

3.8. Visualize the trajectories

After the localization is done (either offline or online), you can visualize the trajectory of each Autobot superimposed to your map using a jupyter notebook in the duckietown-world repository. Similarly to Unit B-4 - Autolab map, launch `jupiter notebook` and open your browser. Navigate to `notebooks` and open the notebook `65-Localization-ShowTrajectory`. Change the values in the first block of this notebook to reflect the name of your map, the location of your trajectory files within the file system, and the name of the Autobot to show.

Run all the cells, the last one will produce a picture of your map with the location of the Autobot at time `t=0` and a slider to adjust the time.

Part F
Localization System - Software explanation

Knowledge and activity graph

> **Requires:** The code for localization (in dt-env-developer)
>
> **Results:** Better knowledge of the inner working of the current localization pipeline
>
> **Next:** Contribute to make it better

Contents

Goals and specifications of the localization system

For all Autobots in the Autolab:

- 10 Hz rate pose trajectory (position and orientation)
- 1 cm accuracy
- 5 degree accuracy (less important maybe)

Outline of the code structure

> **Note:** In all the following, an agent will refer indifferently to a Watchtower or an Autobot.

The pipeline is roughly the following:

- Every agent has a camera flow of image that is either recorded to be used later or directly sent to a processor
- The processor (can be the agent itself or a central powerful computer) takes each image and extract, for each found Apriltag, the relative pose between the camera frame and the Apriltag frame, with the Apriltag ID and the time stamp of the image. This result is a stamped Apriltag transform.
- This set of transforms is sent to a graph builder, that uses all those to create a big graph in G2O, a graph optimization library.
- The graph builder does a lot of work (described below) to recreate and synchronize the time dependency of a moving Autobot (which, contrary to Watchtowers, cannot be represented as one vertex in the graph, but rather as successive vertices with unique

time stamps).

- The graph is regularly optimized, and the trajectory of each Autobot is extracted.

The global frame is fixed thanks to the measured ground Apriltags, that are fed into the graph at initialization, and set as fixed points of the graph.

UNIT F-1

Input to the system

Contents

1.1. Images from the Autobots

The Autobots already use a camera feed that is medium quality on the normal pipeline (640x480). All we do here is use the same image stream.

1.2. Images from the Watchtowers

For the Watchtowers it is a different story. As it is not used (yet) for anything else, we decided for the below reason to use a better image resolution (1296x972) and a higher shutter speed:

- The cameras are high, so the relative size of the Apriltags is small, so higher resolution mean better detection of the Apriltags (especially on the edges of the image)
- The Autobots are (mostly) always moving, and this creates blur with the default shutter speed. With a higher shutter speed, the blur is decreased, and the resulting loss of light is barely noticeable.

1.3. Odometry data from the Autobots

Here there are (at least) two options, but the result must be the same: What is needed is an odometry stream input.

An odometry input is a sequence of relative transform from position at time `t` to position at time `t+dt` of the Autobot, with `dt` being the sampling time.

The currently available options are :

- Wheel command odometry : From the command input of the wheel we construct the odometry
- Visual odometry : From the image stream of the duckiebot we construct odometry

Let's discuss the pros and cons of each method.

1) The wheel command odometry

Note : Right now, this is the used method

The pros:

- Very easy and fast to compute
- Can be done in real time

- The data is small to store and to transmit over network
- Is compliant with the dynamics model (no Y or Z velocity)
- Can use any identified model of the Autobot (gain-trim, kinematic, dynamic)

The cons:

- It is an input based odometry : very small worth
- Doesn't account for slipping (if the autobot is stuck but the wheels spin, the odometry continues to believe in forward movement)
- Has only been used with the simple and inaccurate gain-trim model

2) The visual odometry

Note : Right now, this is not used, but it would be preferable in the long term

The pros:

- Relies on actual output data
- Will easily account for slipping, as the image won't change
- Many libraries exist that can be used

The cons:

- Very slow to process
- Impossible to run real time on an Autobot
- Does not take dynamic model into account, which can generate noise on Y and Z velocities (but to be fair this could be then be ignored)
- No library has actually been extensively tested on Autobots.

UNIT F-2

Input processing

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Knowing the type of input to the system

Results: Knowing how this input is processed for the localization graph

Contents

2.1. Motivation

From Unit F-1 - Input to the system, we now know that we have many streams of image input, from both Watchtowers and Autobots. These images contain various Apriltags, that are placed and registered in the city as described in Unit E-2 - BUILDING - Apriltags specifications.

These Apriltags need to be processed in order to feed the associated transforms to the graph builder. This means that on every image, for every Apriltag on the image, a transform is computed that links the `camera_frame` to the `apriltag_frame`, keeping in memory the name of the agent (Watchtower or Autobot) that detected the Apriltag, as well as the Apriltag unique ID, and the timestamp at which the image was taken. This package of information makes up what will be called everywhere **stamped transform**.

We also have the wheel commands streams of each Autobots. These need to be processed into **stamped transform** as well, as those are the only thing we want to feed the localization graph with.

Note: **Notation**: This parts receives **images** and **wheel commands** and outputs **stamped transforms** to the ROS Listener.

2.2. Apriltag detection strategies

Apriltag detection is very computationally expensive, and different strategies can be used:

- **Offline acquisition** : each agent just records the images, then they are gathered and processed later, offline, without any time requirement. This is the easy, but unsatisfactory way. This also means that the localization can only be used *a posteriori*, so it can not be used for real time decision making.
- **Online acquisition** : the images are processed during the experiment, and the graph

and localization is done (with some delay) online. This is much *harder* to do as processing images is costly.

For **online acquisition**, three strategies can be used:

- Each agent tries to do the processing directly onboard and just sends the stamped transforms to a central ROS master that will do the graph
- Each agent directly sends the images and the central ROS master does the processing for every agent (this implies having a (or many) good computer)
- A mix of the two is also possible (the Autobots send their images, the Watchtowers process them)

As explained in Unit E-3 - DEMO - Localization, in both cases the Watchtowers and Autobots use the acquisition bridge(Pointer to beta/draft material that was removed - acquisition-bridge)

previous **warning** next (8 of 9) index

warning

`This link points to beta/draft material removed`

Location not known more precisely.

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

to send their image streams to a central computer. The Watchtowers only send images when movements are detected, to reduce the number of images to process and record.

For offline acquisition, all we need to do is record a rosbag on this computer.

For online acquisition, the stream of data needs to be used directly by an apriltag extractor.

2.3. Apriltag extraction

No matter how (or on which device) we get the image streams, we need to process them to get the stamped transforms of each Apriltag in each image.

1) Offline case

For the *offline* case, where speed is not relevant, we get a rosbag from the recording. We feed this bag to a `post processor`, which code is in part 08 of the cslam repository.

This code will run apriltag extraction as well as odometry processing and it will export all the corresponding stamped transforms to a new bag.

2) Online case

For the *online* case, we cannot just use one container to do all the extraction. The current strategy is to instantiate one apriltag processor per Watchtower. Each processor is one container that only listens to the image topic of the Watchtower it was assigned to, and outputs the processed stamped transforms. The code is in part 04 of the cslam repository. It is mainly exactly the same process as int the post processing container.

2.4. Odometry processing (wheel odometry)

In the wheel odometry case, we can listen to the wheel command topic or get it from a rosbag (same process for offline/online as for the Apriltag detection).

But no matter the way we get the data, here is how we process it:

- An odometry message is received with timestamp `t1`.
- The next one is received with timestamp `t2 > t1` .
- The wheel velocities are considered constant between `t1` and `t2`, with values from the odometry message at time `t1`.
- From the differential model of the Autobot, linear velocity `V_l` and angular velocity `Omega` can be computed
- From those constant velocities, the transform between `t1` and `t2` can be computed
- The transform is stamped with `t1` and with the Autobot's ID, which makes it a stamped transform, that is then sent to the localization graph.

1) Future work

Right now, the acquisition bridge(Pointer to beta/draft material that was removed - acquisition-bridge)

previous **warning** (9 of 9) index

warning

```
This link points to beta/draft material removed
```

Location not known more precisely.

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

for Autobots sends the wheel commands, but future work could imply using more advanced modeling and system identification of the Autobots to output more accurate linear velocity `V_l` and angular velocity `Omega`. Then the above algorithm would just skip the phase where it transform the wheel commands into `V_l` and `Omega`.

UNIT F-3

The ROS listener

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Knowing what the processed input to the localization system is. ()[#localization-input-processing].

Results: Knowing the role of the ROS listener (layer 1) of the localization system does.

Contents

3.1. The place of the layer in the bigger picture

As stated in Part F - Localization System - Software explanation, the localization graph optimizer works in layers:

- 1) The **ROS listener**, that receives the transforms data from the apriltag extraction and odometry
- 2) The **Resampler**, that filter the data inside the graph
- 3) The **Duckietown Graph Builder**, that creates and manage the graph
- 4) The **g2o graph builder**, a custom wrapper around the g2o library.

This part of the docs focuses on **layer 1, the ROS listener**.

This layer has the rode node. It can work either online or offline:

- Online : it has ros subscriber to `/pose_acquisition/poses` and `/pose_acquisition/odometry`, which respectively receive transforms from apriltag detection and odometry transforms
- Offline : it receives a rosbag containing the same type of information and plays it back to the same subscribers

The received stamped transforms all need to be **filtered and formatted** the same way and have consistent agent names between them (see example below).

Note: **Notation**: This layer receives **stamped transforms** and outputs **formatted transforms** to the resampler.

3.2. For stamped transforms from Apriltag detection

Upon receiving a stamped transform from an Apriltag detection, we needs to filter the stamped transform's following attributes:

- `header.frame_id` –> agent which detected the apriltag
- `header.tag_id` –> apriltag number

The `frame_id` will always be a duckiebot or a watchtower, for instance `autobot01` or `watchtower03`. But the `tag_id` will always be a number. We therefore need a list of apriltag attribution. For instance, the default attribution of apriltag 402 is `autobot03`, but the one for apriltag 53 is a traffic sign.

There are three possible case for a apriltag message:

- A watchtower sees an autobot, then `tag_id` is for instance 402
- A watchtower sees any other apriltag, then `tag_id` is not in the 4XX
- An Autobot sees any apriltag, they never see other autobots' apriltags

The main issue is then that autobot03 is referred as 402 when seen, but as itself when it send a transform. So, thanks to the list of attribution, in the first case, we change the tag_id to `autobot03`.

3.3. For odometry transforms

Upon receiving a odometry transform, we get the transform's following attributes:

- `header.frame_id`
- `header.child_frame_id`

The two are the same, since an odometry transform is just a transform between the autobot at time t1 and the autobot at time t2.

3.4. Frames of reference and associated transformations

Let's formalize some frame definitions:

The **autobot** has three important frames to consider:

- The `autobot_base` frame, located on the top red plate, center of the wheels, X forward, Y left and Z upward (in the driving direction). This is the frame that is considered for the graph.
- The `autobot_apriltag` frame, which is on top of the bot
- The `autobot_camera` frame, which is centered in the lens, Z forward, Y down, X right. And the camera is mounted on a 10 degrees stand. All three frames are attached by static transforms (meaning they don't change relative poses to one another).

The **watchtower** just has one frame, called `watchtower_camera`. It is the one of the camera, described as the one of the `autobot_camera`.

The rest of the **apriltags** have also one frame, called `apriltag_base`.

Since we want to consider the autobot only in its `autobot_base` frame, this means two things:

- The transforms from the watchtowers to the autobots are actually from `watchtow-`

`er_camera` to `autobot_apriltag`. They therefore need to be transferred to be `watchtower_camera` to `autobot_base`. The `autobot_apriltag` to `autobot_base` is a known transform that is applied to all such transforms.

- Similarly, the messages coming from autobots cameras are `autobot_camera` to `apriltag_base`, so we transform them to `autobot_base` to `apriltag_base` by using the know static transform `autobot_base` to `autobot_camera`. Note that in the first case, it is a right multiplication, and in the second a left multiplication (in SE3).

3.5. The importance of time stamps

Each odometry or apriltag stamped transform message comes with a timestamp. Since we want to track the movement in time of each autobot, those are very important to keep and transmit with the transform.

For things that don't move, e.g. the watchtowers and the apriltags that are not on autobots, we don't need to keep the timestamp. This only happens when the transform is from a `watchtower_camera` to an `apriltag_base` (which is not on an autobot).

3.6. To conclude

The ROS listener makes sure to transmit **formatted transforms** with the right frame ids (parent and child), to the right frames of reference (`autobot_base` for the autobots), with the right time stamps.

At the end of the pipeline, it receives back optimized estimates of the trajectories of the autobots and of the positions of the watchtowers. It then publishes them and stores them.

UNIT F-4

The Resampler

KNOWLEDGE AND ACTIVITY GRAPH

> **Requires:** Knowing the role of the ROS listener (layer 1) of the localization system does. Unit F-3 - The ROS listener
>
> **Results:** Knowing what the resampler does and why.

Contents

4.1. The place of the layer in the bigger picture

As stated in Part F - Localization System - Software explanation, the localization graph optimizer works in layers:

- 1) The **ROS listener**, that receives the transforms data from the apriltag extraction and odometry
- 2) The **Resampler**, that filter the data inside the graph
- 3) The **Duckietown Graph Builder**, that creates and manage the graph
- 4) The **g2o graph builder**, a custom wrapper around the g2o library.

This part of the docs focuses on **layer 2, the Resampler**.

> **Note: Notation**: This layer receives **formatted transforms** and outputs **resampled transforms** to the duckietown graph builder.

4.2. What is the resampler and why do we need it

1) The problem

The input of the resampler consists of multiple non synchronized streams of data from multiple agents. At one point in time, for one Autobot, there can be up to about 5 Watchtowers that see it, each with a ~20Hz stream. Adding to this the odometry stream of the Autobot (about 30Hz) and the image stream of the Autobot (about 30Hz as well), we can end up with a total of 160 different time stamps to give to the Autobot per seconds. This makes no sense, as we cant possibly output a trajectory with higher frame rate as the lowest frame rate of the sensors.

2) The solution

The resampler's goal is to generate a synchronized and regular stream of transforms for

the graph optimizer, which we call the **resampled transforms**. The process will allow the layer 3 of the system, the duckietown graph builder, to build a graph with less nodes, at a controlled rate.

4.3. Working principle

What the resampler does is:

- For each Autobot, it keeps the odometry transform history
- For each Watchtower, it keeps the transform history of each detected Autobot
- The transforms from Watchtowers to other apriltags are transmitted directly to the graph optimizer, as we don't keep their timestamps.

Then, at a regular interval (the default rate is 15Hz), each sensor is queried for a transform.

Let's call t_{query} the timestamp of the query.

1) For the Apriltag transforms

For each Watchtower, for each `AutobotXX` seen by Watchtower:

Let's call t_{prev} and t_{next} respectively the closest timestamps before and after t_{query} such that the Watchtower has transforms to `AutobotXX` at H_{prev} at t_{prev} and transform H_{next} at t_{next}.

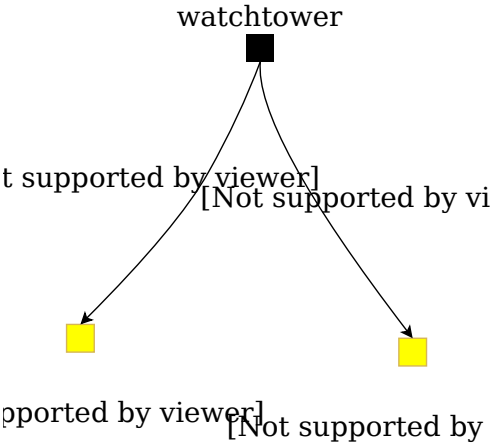


Figure 4.1. We have two transforms from the Watchtower to the Autobot, only at times t_{prev} and t_{next}

First, we compute from those two transforms the transform H_{movement} that is the movement of the Autobot from t_{prev} to t_{next}:

$$H_{\text{movement}} = H_{\text{next}} \cdot H_{\text{prev}}^{-1}$$

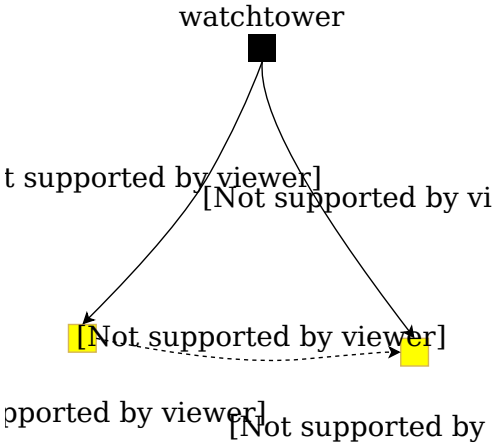


Figure 4.2. $H_{\text{movement}} = H_{\text{next}} \cdot H_{\text{prev}}^{-1}$

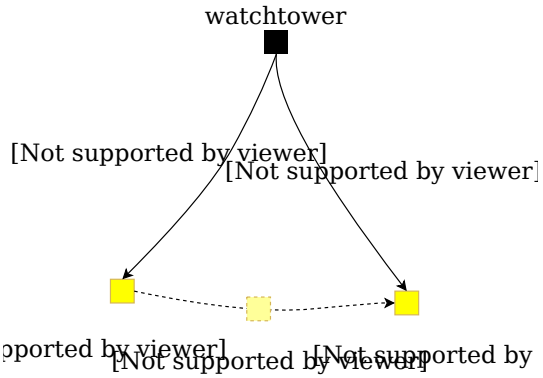


Figure 4.3. Extrapolating the Autobot's position at t_{query}

Then we “crop” this H_movement to only take the part happening from t_{prev} to t_{query}:

$$H_{\text{movement-cropped}} = \text{interpolation}(H_{\text{movement}}, t_{\text{query}})$$

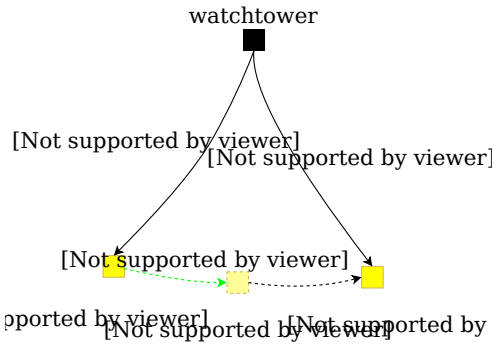


Figure 4.4. Taking only the first part of the transform

Then we left-multiply it by H_{prev} to finally get the transform from the camera frame to the duckiebot at time t_{query}:

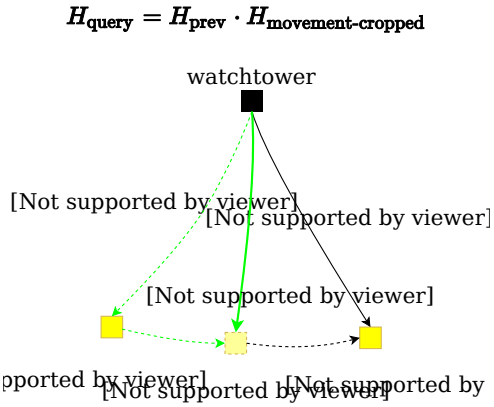


Figure 4.5. Creating H_{query} from the computed interpolation

This will give the best approximation of the transform at t_{query}. Of course, this is done only if t_{prev} and t_{next} exist and are close enough to t_{query}. We use the Lie Algebra of SE3 to compute the interpolation.

This is then the output H_{query} that is called resampled transform.

What this ensures is that if multiple Watchtowers see `AutobotXX` at the same time, their inputs will be synchronized and linked to the same node in the duckietown graph builder.

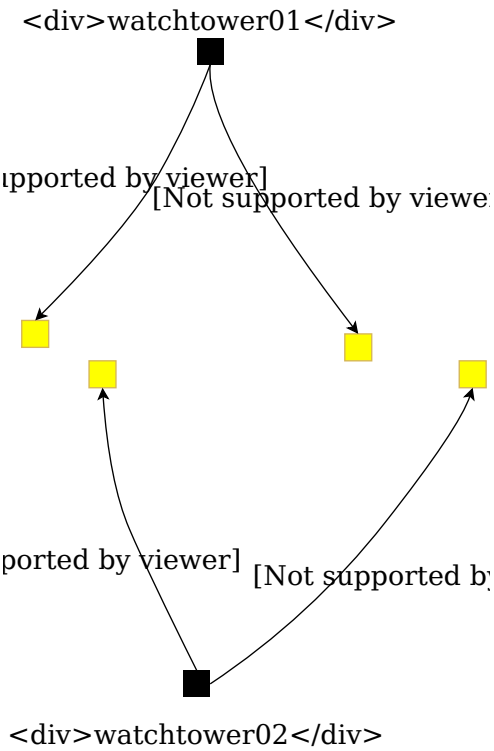


Figure 4.6. Without the resampler, all transforms are not synchronized, this will create 4 separated nodes in the graph

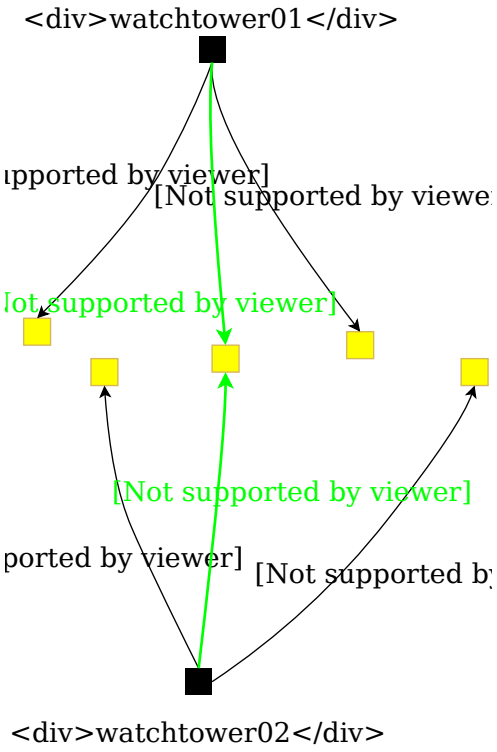


Figure 4.7. With the resampler, we query the same time for each watchtower, thus synchronizing the data

2) For the odometry transforms

For each Autobot, the sequence of odometry transforms are stored. As said before, for the odometry two time stamps are required as the transform is a transform of time, not of space. Hence, we store the previous queried time, called $t_{\text{query-prev}}$, and calculate the odometry transform between $t_{\text{query-prev}}$ and t_{query}.

This means that we need to get :

- t_{prev} : the time stamp closest before $t_{\text{query-prev}}$
- t_{next} : the time stamp closest after t_{query}
- $[t_1, t_2, ..., t_n]$, the `n` timestamps for of odometry messages between $t_{\text{query-prev}}$ and t_{query}

> Note: `n` depends on the difference between the query rate and the odometry rate. If the odometry rate is 30Hz and the query rate is 10Hz, then n will usually be 2. If both rate are comparable, `n` might be zero and the following algorithm just does the two first interpolations as one.

Then, we do two interpolations :

- The transform H_{prev} which is between $t_{\text{query-prev}}$ and t_1 (we need the odometry at time t_{prev} to compute it).

- The transform H_{final} which is between t_{n} and t_{query}

Then, we have the transforms $[H_{\text{1-to-2}}, H_{\text{2-to-3}}, \ldots, H_{\text{(n-1)-to-n}}]$ that are the `n-1` inner transforms. By direct multiplication, we have that the requested H_{query} is:

$$H_{\text{query}} = H_{\text{prev}} \cdot H_{\text{1-to-2}} \cdot H_{\text{2-to-3}} \cdot \ldots \cdot H_{\text{(n-1)-to-n}} \cdot H_{\text{final}}$$

This is then the output H_{query} that is called resampled transform.

UNIT F-5

The duckietown graph builder

KNOWLEDGE AND ACTIVITY GRAPH

Requires:

Results:

UNIT F-6

The g2o graph builder

KNOWLEDGE AND ACTIVITY GRAPH

Requires:

Results:

PART G
Autolab operation manual

KNOWLEDGE AND ACTIVITY GRAPH

Requires: A built Autolab, a computer connected to the same network as the watchtowers and Autobots

Results: Control of the Autolab by a user friendly web-interface, experiment automation and more

Contents

This manual guides you through the various procedures we use to operate an Autolab. We will also show you how to use the different scripts and tools we use to make it as easy as possible for you to run an Autolab.

UNIT G-1

Autolab control interface installation

KNOWLEDGE AND ACTIVITY GRAPH

Requires: put requirements here

Results: put result here

Next: put next steps here

Contents

1.1. Installing the compose interface

1) Setup of the workspace

It is recommended to create a folder somewhere you like, that we'll call `Autolab_control` . We will do everything from inside of that folder.

2) Launching the compose interface

The Autolab control interface is built with Compose, described as:

> *A CMS (Content Management System) platform written in PHP that provides functionalities for the fast development of web applications on Linux servers.*

To set it up, go into `Autolab_control` and do

```
$ git clone https://github.com/afdaniele/compose

$ docker pull afdaniele/compose:latest

$ docker run -itd -p 80:80 -v /FULL/PATH/TO/Autolab_control/compose/:/var/www/html/ afdaniele/compose:latest
```

This will start the \compose\ interface on your computer. The last command makes sure that whatever parameters are set within the interface will be saved, and reused as long as you keep launching the same command. If you run into any issue, visit the compose website.

Wait up to 10 seconds, then visit the page `http://SERVER_HOSTNAME/` in your browser. `SERVER_HOSTNAME` is the name of the computer you are using. Dont forget the `.local` . For example, on a computer named `autolab_computer_01` , go to `http://auto-`

`lab_computer_01.local/` .

3) Getting the Autolab Control package

First, follow the initial setup instructions for compose. You can skip step 1, and use the developer mode.

Then, navigate to the **package store** tab, where you'll see a list of installable packages.

In addition to any package you want, you should install :

- Autolab Control
- ROS
- Data

Once the list is selected, do not forget to actually click install at the top of the page.

4) Setting the paremeters in the compose page

In the compose webpage, go to the **settings** tab, and scroll to the Autolab Control part and fill it.

In the ROS section, set the port to `9090` .

1.2. Setting up the Rosbridge

In order for the web interface to communicate with the rest of the Autolab, you will need a Rosbridge running at all times.

To do so, go to `Autolab_control` and do

```
$ git clone git@github.com:duckietown/rosbridge_kinetic.git
$ cd rosbridge_kinetic
$ vim docker-compose.yaml
```

In `docker-compose.yaml` , modify the file to use your server's IP address. Then run

```
$ docker-compose up -d
```

1.3. Setting up the flask server

The flask server is where the control happens. It contains all the functions that are used by the user interface.

To run it, got to `Autolab_control` and do

```
$ git clone https://github.com/duckietown/autolab_con-
trol_flask_scripts
$ cd autolab_control_flask_scripts
$ git checkout daffy
$ dts devel build -a amd64
$ vim docker-compose.yaml
```

Modify the last volume in `docker-compose.yaml` and mount the right path to your fleet roster.

Then:

💻 `$ docker-compose up -d`

1.4. Using the interface

Now, everything should be setup. Go to the `autolab control` tab and enjoy.

UNIT G-2

Autolab control interface usage

KNOWLEDGE AND ACTIVITY GRAPH

Requires: put requirements here

Results: put result here

Next: put next steps here