Classical Robotics Architectures using Duckietown

Contents

Perception fundamentals

In this section you will have to use the knowledge on computer vision you acquired during your classes. First, you will get familiar with the basics by projecting points and lines onto an image. In the second stage, you will code your first augmented reality system running on a Duckiebot!

Contents

<div align="center">

Unit A-1

Preliminaries

</div>

1.1. Required steps

1) Camera calibration

In this exercise the only sensor that will be used is the camera. So, ensure that you have already done intrinsics and extrinsics camera calibration of your Duckiebot.

1.2. Workflow tips

In this exercise you will extensively use images and inevitably make some mistakes that you will have to fix. Debugging ROS packages that manipulate images can be very difficult only from the terminal output of your node and visualizing the image stream can be useful. Within the Duckietown infrastructure we have a very nice way to do that: use duckietown shell.

You can start a container connected to the ROS master on you Duckiebot with:

```
$ dts start_gui_tools DUCKIEBOT_NAME
```

Once inside this container you can call the rqt commands to visualize the state of the pipeline on you robot.

```
$ rqt_image_view

$ rqt_graph
```

Or you can also use all the ROS related commands like rostopic, rosparam, rosnode, etc..

```
$ rostopic ...

$ rosparam ...
```

Learning materials

2.1. Introduction

During lectures, we explained the standard workflow of an image perception pipeline:
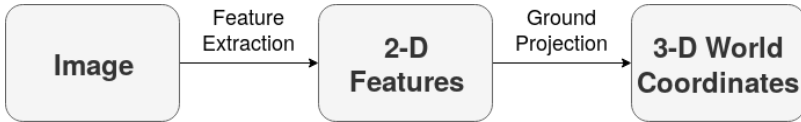
Figure 2.1

In this exercise, we are going to look at the pipeline in the opposite direction.

It is often said that:

> "The inverse of computer vision is computer graphics."

The inverse pipeline looks like this:
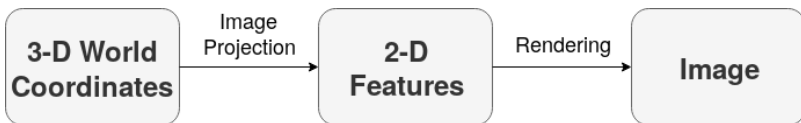
Figure 2.2

In simple words, instead of extracting information from our camera, we want to introduce some data in the imagery.

For this exercise concept like camera calibration, homography and projection matrices, image plane and world coordinates are essential. So be sure to have those in mind while you work your way through the exercises in the next sections.

Here is a quick reminder on what the homography matrix is and how it is obtained:

Planar Homography

- A *homography* is a projective mapping from one plane to another

- Example: Mapping points on the ground plane to the image plane

$$\mathbf{x}_i \simeq \mathbf{H}\mathbf{x}_w$$
$$\mathbf{H}^{-1}\mathbf{x}_i \simeq \mathbf{x}_w$$

Figure 2.3

key idea: $Z = 0$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \simeq \begin{bmatrix} f_x & s & c_x \\ & f_y & c_y \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^{\mathsf{T}} & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$
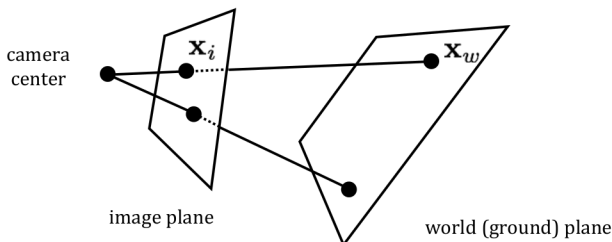
$$= \begin{bmatrix} f_x & s & c_x \\ & f_y & c_y \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{21} & r_{31} & t_1 \\ r_{12} & r_{22} & r_{32} & t_2 \\ r_{13} & r_{23} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} f_x & s & c_x \\ & f_y & c_y \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{21} & t_1 \\ r_{12} & r_{22} & t_2 \\ r_{13} & r_{23} & t_3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

homography matrix

$$= \begin{bmatrix} f_x & s & c_x \\ & f_y & c_y \\ & & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{21} & t_1 \\ r_{12} & r_{22} & t_2 \\ r_{13} & r_{23} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} h_{11} & h_{21} & h_{31} \\ h_{12} & h_{22} & h_{32} \\ h_{13} & h_{23} & h_{33} \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Figure 2.4

Basic Augmented Reality Exercise

The goal of this exercise is to familiarize yourself in developing functionalities in the framemork of a pre-existing pipeline. In particular, the focus is in the perception pipeline. You will implement a computer graphics algorithm that will be a part of it.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Camera calibration (unknown ref opmanual_duckiebot/camera-calib)

warning next (1 of 17) index

> warning

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

 > I do not know what is indicated by the link '#op-
manual_duckiebot/camera-calib'.
```

Location not known more precisely.

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

Requires: Docker basics (unknown ref duckietown-robotics-development/docker-basics)

previous **warning** next (2 of 17) index

> warning

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

 > I do not know what is indicated by the link
'#duckietown-robotics-development/docker-basics'.
```

Location not known more precisely.

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

Requires: ROS basics (unknown ref duckietown-robotics-development/sw-advanced)

previous **warning** next (3 of 17) index

> warning

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

 > I do not know what is indicated by the link
'#duckietown-robotics-development/sw-advanced'.
```

Location not known more precisely.

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

Requires: Knowledge of the software architecture on a Duckiebot (unknown ref duckietown-robotics-development/duckietown-code-structure)

previous **warning** next (4 of 17) index

> **warning**
>
> ```
> I will ignore this because it is an external link.
>
>  > I do not know what is indicated by the link
>  '#duckietown-robotics-development/duckietown-code-
>  structure'.
> ```
>
> Location not known more precisely.
>
> Created by function n/a in module n/a.

Results: Skills on how to develop new code as part of the Duckietown framework.

Results: Insights into a computer graphics pipeline.

3.1. Segments Projection Exercise

In this exercise you are asked to draw some segments on an image given a yaml file with their specification namely defining points coordinates and color of each segment. In order to do that you will have to create a package called `augmented_reality_basics` with the functionalities specified below in Section 3.2 - Specification of "augmented_reality_basics".

Then verify the results of your package in the following 3 scenarios.

1) Scenario 1: Calibration pattern

- Put the robot in the middle of the calibration pattern.
- Run the node `augmented_reality_basics` with map file `calibration_pattern.yaml`.

(Adjust the position of your Duckiebot until you get a decent match of reality and augmented reality.)

2) Scenario 2: Lane

- Put the robot on a tile, in the middle of a straight lane segment.
- Run the node `augmented_reality_basics` with map file `lane.yaml`.

(Adjust the position of your Duckiebot of your Duckiebot until you get a decent match of reality and augmented reality.)

3) Scenario 3: Intersection

- Put the robot at a stop line at a 4-way intersection in Duckietown.
- Run the node `augmented_reality_basics` with map file `intersection_4way.yaml`.

(Adjust the position of your Duckiebot until you get a decent match of reality and augmented reality.)

3.2. Specification of "augmented_reality_basics"

In this assignment you will be writing a ROS package to perform the augmented reality exercise. The program will be invoked with the following syntax:

```
$ roslaunch augmented_reality_basics augmented_reality_ba-
sics.launch map_file:=map file veh:="$VEHICLE_NAME"
```

where `map file` is a YAML file containing the map as specified in Section 3.3 - Map Specification. If you use a roslaunch in the `launch.sh` file remember to put `dt-exec` before each command.

The package structure *must* be the one provided by the Duckietown template-ros. In addition, create a map directory where you can store the map files.

Your program is supposed to do the following:

1. Load the intrinsic / extrinsic calibration parameters for the given robot.

2. Read the map file corresponding to the `map_file` parameter given in the roslaunch command above.

3. Subscribe to the image topic `/robot name/camera_node/image/compressed`.

4. When you receive an image, project the map features onto it, and then publish the result to the topic `/robot name/node_name/map file basename/image/compressed` where `map file basename` is the basename of the file without the `yaml` extension.

Create a ROS node called `augmented_reality_basics_node`, which imports an `Augmenter` class, from an `augmented_reality_basics` module. The `Augmenter` class should contain the following methods:

1. A method called `process_image` that undistorts raw images.

2. A method called `ground2pixel` that transforms points in ground coordinates (i.e. the robot reference frame) to pixels in the image.

3. A method called `render_segments` that plots the segments from the map files onto the image.

In the ROS node, you just need a callback on the camera image stream that uses the `Augmenter` class to modify the input image. Therefore, implement a method called `callback` that writes the augmented image to the appropriate topic.

> **Note:** As you will subscribe to the camera node's `camera_node/image/compressed` topic, you will need to run the `dt-duckiebot-interface` container alongside your own container.

3.3. Map Specification

The map file contains a 3D polygon, defined as a list of points and a list of segments that join those points.

The format is similar to any data structure for 3D computer graphics. Additionally, we have these two specifics:

1. Points are referred to by name.

2. It is possible to specify a reference frame for each point. (This will help make this into a general tool for debugging various types of problems).

Here is an example of the file contents, which is hopefully self-explanatory. The following map file describes three points, and two lines.

```
points:
    # define three named points: center, left, right
    center: [axle, [0, 0, 0]] # [reference frame, coordinates]
    left: [axle, [0.5, 0.1, 0]]
    right: [axle, [0.5, -0.1, 0]]
segments:
- points: [center, left]
  color: [rgb, [1, 0, 0]]
- points: [center, right]
  color: [rgb, [1, 0, 0]]
```

1) Reference frame specification

The reference frames are defined as follows:

- `axle` : center of the wheels axle; coordinates are in 3D.
- `camera` : camera frame; coordinates are in 3D.
- `image01` : a reference frame in which `(0,0)` is top left, and `(1,1)` is bottom right of the image; coordinates are 2D.

(Other reference frames will be introduced later, such as the `world` and `tile` frames, which will also need the pose of the robot.)

2) Color specification

RGB colors are written as:

```
[rgb, [ R , G , B ]]
```

where the RGB values are between 0 and 1. Alternatively, you can use one of the following strings defining some popular colours:

- `red` , equivalent to `[rgb, [1,0,0]]` ;
- `green` , equivalent to `[rgb, [0,1,0]]` ;
- `blue` , equivalent to `[rgb, [0,0,1]]` ;
- `yellow` , equivalent to `[rgb, [1,1,0]]` ;
- `magenta` , equivalent to `[rgb, [1,0,1]]` ;
- `cyan` , equivalent to `[rgb, [0,1,1]]` ;
- `white` , equivalent to `[rgb, [1,1,1]]` ;
- `black` , equivalent to `[rgb, [0,0,0]]` .

3.4. Map specification files

1) `hud.yaml`

This pattern serves as a simple test that we can draw lines in image coordinates:

```
points:
    TL: [image01, [0, 0]]
    TR: [image01, [0, 1]]
    BR: [image01, [1, 1]]
    BL: [image01, [1, 0]]
segments:
- points: [TL, TR]
  color: red
- points: [TR, BR]
  color: green
- points: [BR, BL]
  color: blue
- points: [BL, TL]
  color: yellow
```

The expected result is to put a border around the image: red on the top, green on the right, blue on the bottom, yellow on the left.

2) `calibration_pattern.yaml`

This pattern is based off the checkerboard calibration target used in estimating the intrinsic and extrinsic camera parameters:

```
points:
    TL: [axle, [0.315, 0.093, 0]]
    TR: [axle, [0.315, -0.093, 0]]
    BR: [axle, [0.191, -0.093, 0]]
    BL: [axle, [0.191, 0.093, 0]]
segments:
- points: [TL, TR]
  color: red
- points: [TR, BR]
  color: green
- points: [BR, BL]
  color: blue
- points: [BL, TL]
  color: yellow
```

The expected result is to put a border around the inside corners of the checkerboard: red on the top, green on the right, blue on the bottom, yellow on the left, like below.

Figure 3.1

3) `lane.yaml`
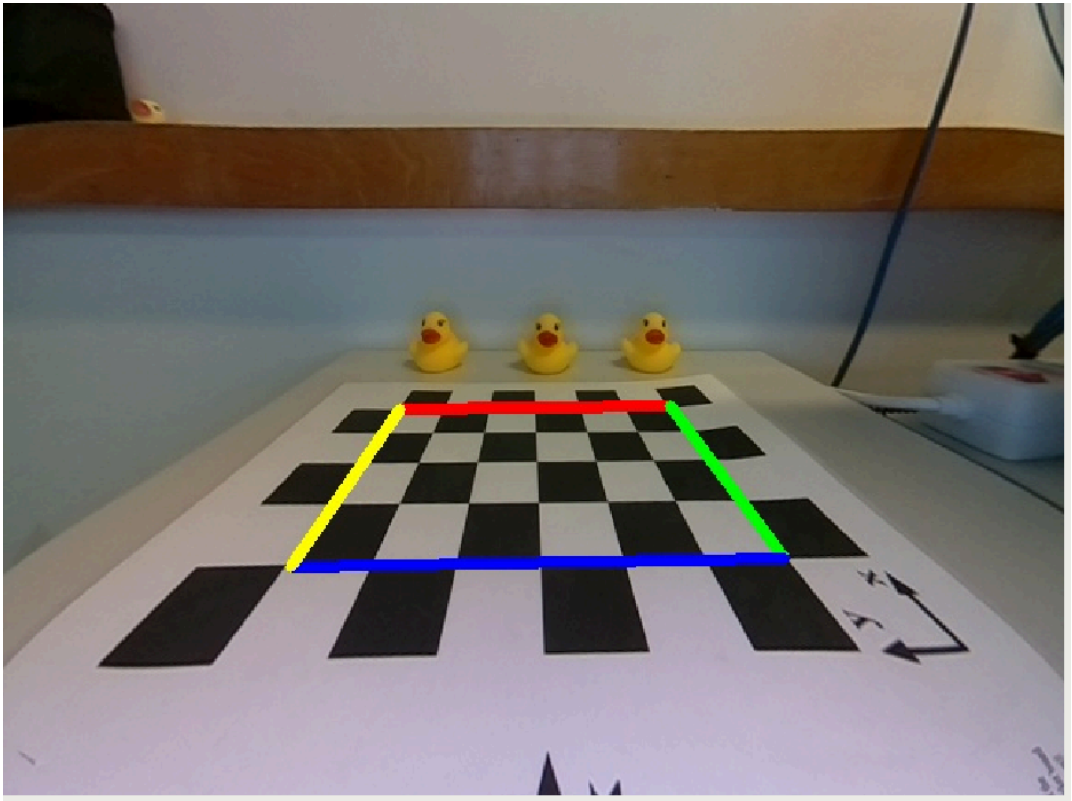
We want something like this:

```
|   |          |  · |              |   |
|   |          |  · |              |   |
|   |          |  · |              |   |
|   |          |  · |              |   |
|   |          |  · |              |   |
|   |          |  · |              |   |
W   W          Y    Y              W   W
1   2          3    4              5   6
```

Then we have:

```
points:
     p1: [axle, [0.15, 0.2794, 0]]
     q1: [axle, [0.6096, 0.2794, 0]]
     p2: [axle, [0.15, 0.2286, 0]]
     q2: [axle, [0.6096, 0.2286, 0]]
     p3: [axle, [0.15, 0.0127, 0]]
     q3: [axle, [0.6096, 0.0127, 0]]
     p4: [axle, [0.15, -0.0127, 0]]
     q4: [axle, [0.6096, -0.0127, 0]]
     p5: [axle, [0.15, -0.2286, 0]]
     q5: [axle, [0.6096, -0.2286, 0]]
     p6: [axle, [0.15, -0.2794, 0]]
     q6: [axle, [0.6096, -0.2794, 0]]
segments:
 - points: [p1, q1]
   color: white
 - points: [p2, q2]
   color: white
 - points: [p3, q3]
   color: yellow
 - points: [p4, q4]
   color: yellow
 - points: [p5, q5]
   color: white
 - points: [p6, q6]
   color: white
```
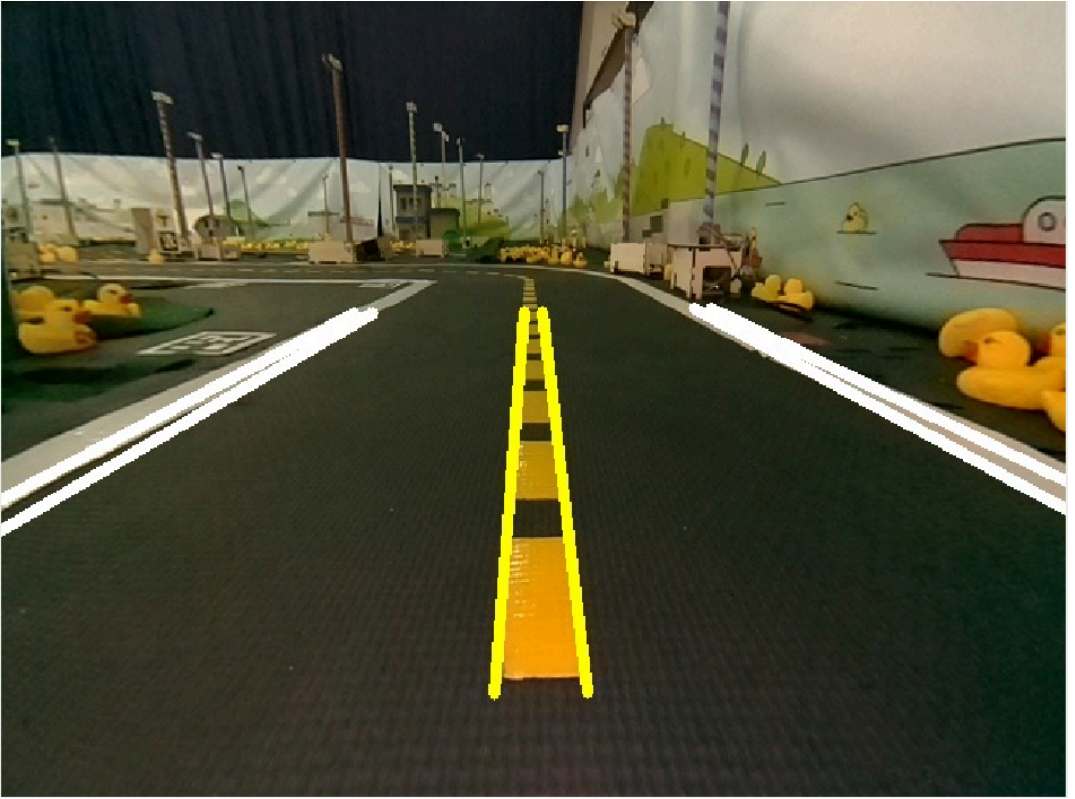
Expected output:

Figure 3.2

4) `intersection_4way.yaml`

```
points:
    NL1: [axle, [0.247, 0.295, 0]]
    NL2: [axle, [0.347, 0.301, 0]]
    NL3: [axle, [0.218, 0.256, 0]]
    NL4: [axle, [0.363, 0.251, 0]]
    NL5: [axle, [0.400, 0.287, 0]]
    NL6: [axle, [0.409, 0.513,   0]]
    NL7: [axle, [0.360, 0.314, 0]]
    NL8: [axle, [0.366, 0.456, 0]]
    NC1: [axle, [0.372, 0.007, 0]]
    NC2: [axle, [0.145, 0.008, 0]]
    NC3: [axle, [0.374, -0.0216, 0]]
    NC4: [axle, [0.146, -0.0180, 0]]
    NR1: [axle, [0.209, -0.234, 0]]
    NR2: [axle, [0.349, -0.237, 0]]
    NR3: [axle, [0.242, -0.276, 0]]
    NR4: [axle, [0.319, -0.274, 0]]
    NR5: [axle, [0.402, -0.283, 0]]
    NR6: [axle, [0.401, -0.479, 0]]
    NR7: [axle, [0.352,  -0.415, 0]]
    NR8: [axle, [0.352, -0.303, 0]]
    CL1: [axle, [0.586, 0.261, 0]]
    CL2: [axle, [0.595, 0.632, 0]]
    CL3: [axle, [0.618, 0.251, 0]]
    CL4: [axle, [0.637, 0.662, 0]]
    CR1: [axle, [0.565, -0.253, 0]]
    CR2: [axle, [0.567, -0.607, 0]]
    CR3: [axle, [0.610, -0.262, 0]]
    CR4: [axle, [0.611, -0.641, 0]]
    FL1: [axle, [0.781, 0.718, 0]]
    FL2: [axle, [0.763, 0.253, 0]]
    FL3: [axle, [0.863, 0.192, 0]]
    FL4: [axle, [1.185, 0.172, 0]]
    FL5: [axle, [0.842, 0.718, 0]]
    FL6: [axle, [0.875, 0.271,   0]]
    FL7: [axle, [0.879, 0.234, 0]]
    FL8: [axle, [1.180, 0.209, 0]]
    FC1: [axle, [0.823, 0.0162, 0]]
    FC2: [axle, [1.172, 0.00117, 0]]
    FC3: [axle, [0.845, -0.0100, 0]]
    FC4: [axle, [1.215, -0.0181, 0]]
    FR1: [axle, [0.764, -0.695, 0]]
    FR2: [axle, [0.768, -0.263, 0]]
    FR3: [axle, [0.810, -0.202, 0]]
    FR4: [axle, [1.203, -0.196, 0]]
    FR5: [axle, [0.795, -0.702, 0]]
    FR6: [axle, [0.803, -0.291, 0]]
    FR7: [axle, [0.832, -0.240, 0]]
    FR8: [axle, [1.210, -0.245, 0]]
segments:
- points: [NL1, NL2]
  color: white
- points: [NL3, NL4]
```

3.5. Suggestions

Start by using the file `hud.yaml`. To visualize it, you do not need the calibration data. It will be helpful to make sure that you can do the easy parts of the exercise: loading the map, and drawing the lines.

To write the segments you can use this function:

```python
def draw_segment(self, image, pt_x, pt_y, color):
    defined_colors = {
        'red': ['rgb', [1, 0, 0]],
        'green': ['rgb', [0, 1, 0]],
        'blue': ['rgb', [0, 0, 1]],
        'yellow': ['rgb', [1, 1, 0]],
        'magenta': ['rgb', [1, 0 , 1]],
        'cyan': ['rgb', [0, 1, 1]],
        'white': ['rgb', [1, 1, 1]],
        'black': ['rgb', [0, 0, 0]]}
    _color_type, [r, g, b] = defined_colors[color]
    cv2.line(image, (pt_x[0], pt_y[0]), (pt_x[1], pt_y[1]), (b * 255, g
* 255, r * 255), 5)
    return image
```

To read a generic YAML file you can use this function:

```python
def readYamlFile(self,fname):
    """
    Reads the YAML file in the path specified by 'fname'.
    E.G. :
        the calibration file is located in : `/data/config/calibrations/
filename/DUCKIEBOT_NAME.yaml`
    """
    with open(fname, 'r') as in_file:
        try:
            yaml_dict = yaml.load(in_file)
            return yaml_dict
        except yaml.YAMLError as exc:
            self.log("YAML syntax error. File: %s fname. Exc: %s"
                    %(fname, exc), type='fatal')
            rospy.signal_shutdown()
            return
```

For other functionalities (i.e. loading the calibration files), we recommend that you invest some time in looking into the existing Duckietown code. You can find some helpful functions and methods there.

Advanced Augmented Reality Exercise

The goal of this exercise is to put your skills in computer graphics to the test by projecting a complex 3D model on an AprilTag at an arbitrary position.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Camera calibration (unknown ref opmanual_duckiebot/camera-calib)

previous **warning** next (5 of 17) index

warning

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

 > I do not know what is indicated by the link '#op-
manual_duckiebot/camera-calib'.
```

Location not known more precisely.

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

Requires: Docker basics (unknown ref duckietown-robotics-development/docker-basics)

previous **warning** next (6 of 17) index

warning

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

 > I do not know what is indicated by the link
'#duckietown-robotics-development/docker-basics'.
```

Location not known more precisely.

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

Requires: ROS basics (unknown ref duckietown-robotics-development/sw-advanced)

previous **warning** next (7 of 17) index

warning

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

 > I do not know what is indicated by the link
'#duckietown-robotics-development/sw-advanced'.
```

Location not known more precisely.

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

Requires: Knowledge of the software architecture on a Duckiebot (unknown ref duckietown-robotics-development/duckietown-code-structure)

previous **warning** next (8 of 17) index

warning

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

 > I do not know what is indicated by the link
 '#duckietown-robotics-development/duckietown-code-
 structure'.
```

Location not known more precisely.

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

Requires: Basic Augmented Reality Exercise

Results: Advanced skills on how to manipulate transformations in Computer Graphics.

Results: Insights into the computer graphics pipeline.

4.1. 3D Model Projection Exercise

In this exercise you are asked to render a complete 3D model of a duckie on an image. The duckie model is provided as an `.obj` file. Similarly to many augmented reality games the model needs to be projected on an easily recognizable flat pattern. In this exercise you will use an AprilTag for this purpose. You should have received some traffic signs with AprilTags with the your Duckiebot box. They should look something like:

Traffic Signs

Figure 4.1

In Duckietown, AprilTags are recognized and managed through the lib-dt-apriltags Python package. Check it out to see how to use it!

In order to solve the exercise you will have to create a package called `augmented_reality_apriltag` with the functionalities specified in Unit A-3 - Basic Augmented Reality Exercise.

4.2. Instructions

1.   This exercise package structure will be based on the one provided by the AprilTag Template. Inside there you will find everything you need like the 3D model and the provided files.

2.   In this exercise you will have to use the AprilTag library so check that it has been added in the `dependencies-py3.txt` as `dt-apriltags`.

3.   We provided you a file called `renderClass.py`. Inside you will find the `Renderer`

class which allows you to draw a 3D `.obj` model onto an image. If you are curious about how this happens, the code inside this file is a modified version of Pygame OB-JFileLoader. The provided Renderer class contains the method `render(img, projection_matrix)`, where `img` is the image you want to project the model onto and `projection_matrix` is the 3x4 matrix that transforms the 3D model coordinates to the April-Tag reference system allowing you to project it. The constructor of an instance of the `Renderer` class requires the 3D model as input. Keep the 3D model in a directory with the path `src/models`. You can use the code below to correctly initialize an instance of the `Renderer` class:

```
# Import class from file.
from renderClass import Renderer

rospack = rospkg.RosPack()

# Initialize an instance of Renderer giving the model in input.
self.renderer = Renderer(rospack.get_path('YOUR PACKAGE NAME')+'/src/
models/duckie.obj')
```

Please refrain from changing the `renderClass.py` file. It has been tested and any change might lead to unexpected errors and problems that will not be supported.

1. You will also get a function to load the calibration parameters of your Duckiebot camera which should be in the node python file.

2. Conversely to the exercise in the previous section, here you are not asked to rectify the image to reduce the delay of this node. You are nonetheless invited to try this yourself and see how accuracy and speed are affected.

4.3. Exercise Structure

Looking at the project as a whole may make it seem more difficult than it really is. But we will follow the latin proverb *divide et impera* (*divide and rule*). In other words, we will brake down the big monolithic problem into a few smaller and manageable tasks.

First, let's focus on what we are trying to achieve: to project a 3D model onto an image such that the position and orientation of the model match the position and orientation of an AprilTag. The break-down of this task looks like that:

1. Detect the AprilTag and extract its reference frame.

2. Estimate the homography matrix: determine the transformation from the reference frame of the AprilTag (2D) to the reference frame of the target image (2D). This transformation is the homography matrix. For this step and the previous one, have a look at the lib-dt-apriltags repository, you will find it quite useful.

3. Derive the transformation from the reference frame of the AprilTag to the target image reference frame: if we want to project a 3D model placed on top of the reference surface to the target image, we need to extend the previous transformation to handle cases were the height of the point to project is different from zero. This can be achieved with some knowledge about coordinate system transformations and a bit of algebra. To know more about homography and orthonormal basis give a look here.

4. Project our 3D model in the image (pixel space) and draw it: you can use the provided `Renderer` class for this. This class has a method called `render` which gets an input `img` of type InputOutputArray and `projection_matrix` (3x4 floating-point matrix). The `render` method colors the polygons formed by the vertices of the `.obj` file.

The expected outcome of this exercise should something like this:

Figure 4.2

4.4. Parameter Tweaking

As you might have seen the AprilTag detection adds some delay to the pipeline. However, you can try to change some parameters for a better accuracy and speed trade-off. Try to play around with the parameters `nthreads` and `quad_decimate` of the `dt_apriltags.Detector` class. What do they change? Does the speed improve? What about the stability and accuracy of the detector?

PART B
Localization

Contents

Preliminaries

1.1. Required steps

1) Run the exercise

Run the exercise container:

```
$ docker -H DUCKIEBOT_NAME.local run --name lane_following_cra2
--net host -v /data:/data duckietown/lane-following-cra2:daffy
```

This container runs an extended version of the lane following demo from `dt-core`. It includes additional parameters which are important for this exercise.

2) Run rviz

`rviz` (ROS visualization) is a 3D visualizer for displaying sensor data and state information from ROS. More on information can be found in the official ROS wiki

For this exercise `rviz` will be helpful for displaying sensor messages from the Duckiebot. By selecting the appropriate topic we can output desired information.
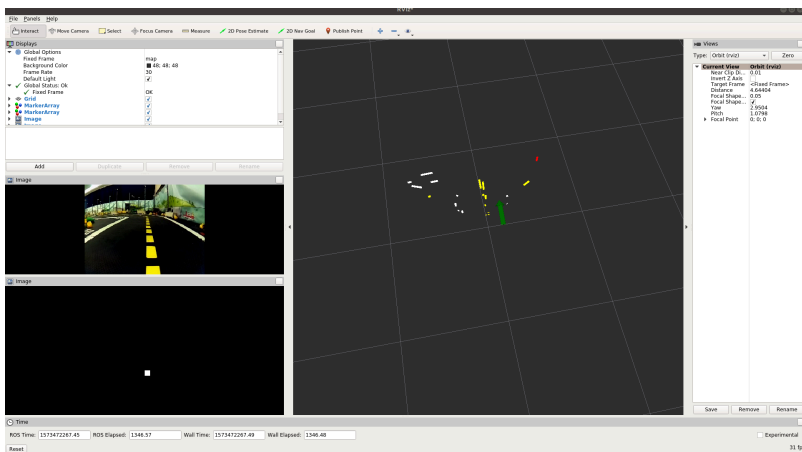

Figure 1.1

First, make sure that your display can be accessed from a container. Run:

```
$ xhost +local:root
```

Note: When you are done with the exercise, you should run the reverse command in order to secure your screen access again:

```
$ xhost -local:root
```

To start `rviz` run the following container:

```
$ docker run -it --net=host -e VEHICLE_NAME=DUCKIEBOT_HOSTNAME
--env="DISPLAY" --volume="$HOME/.Xauthority:/root/.Xauthority:rw"
duckietown/rviz-cra2:daffy /bin/bash
```

then:

```
$ export ROS_MASTER_URI="http://DUCKIEBOT_IP:11311"
```

and also:

```
$ export ROS_IP=DUCKIEBOT_IP
```

finally we can launch the application:

```
$ rviz
```

After starting `rviz` we need to add the required topics we want to inspect

- `/ DUCKIEBOT_NAME /duckiebot_visualizer/segment_list_markers`
- `/ DUCKIEBOT_NAME /lane_filter_node/belief_img`
- `/ DUCKIEBOT_NAME /lane_pose_visualizer_node/lane_pose_markers`

After adding these 3 topics, `rviz` should show the output as in the figure above.

3) Change rosparams

The following functions will be useful to change the dynamic parameters in the exercises:

```
$ dts start_gui_tools DUCKIEBOT_NAME
```

1) Listing the parameters:

```
$ rosparam list
```

2) Getting the parameters:

```
$ rosparam get PARAMETER_NAME
```

3) Setting the parameters:

```
$ rosparam set PARAMETER_NAME VALUE
```

Learning materials

The goal of this material is to get familiar with the pipeline that extracts lane localization from the image stream. This is the base of the Lane Following demo.
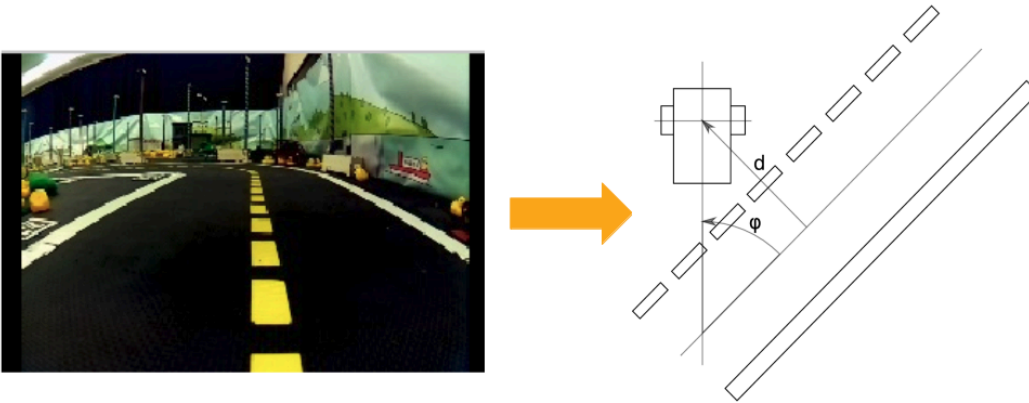

Figure 2.1. From camera image to lane pose.

2.1. Overview of the pipeline

Determining its own position in the lane is essential for any Duckiebot to drive safely in Duckietown. In the following section we will go step by step through the various steps of the image pipeline: from image to lane pose estimation.

Figure 2.2 shows the two most important parts of the localization: the *line detector* and the *lane filter*, and where they stand in the complete image to control pipeline. The control aspect will be the focus of the next set of exercises. We will focus here only on the two above-mentioned parts.
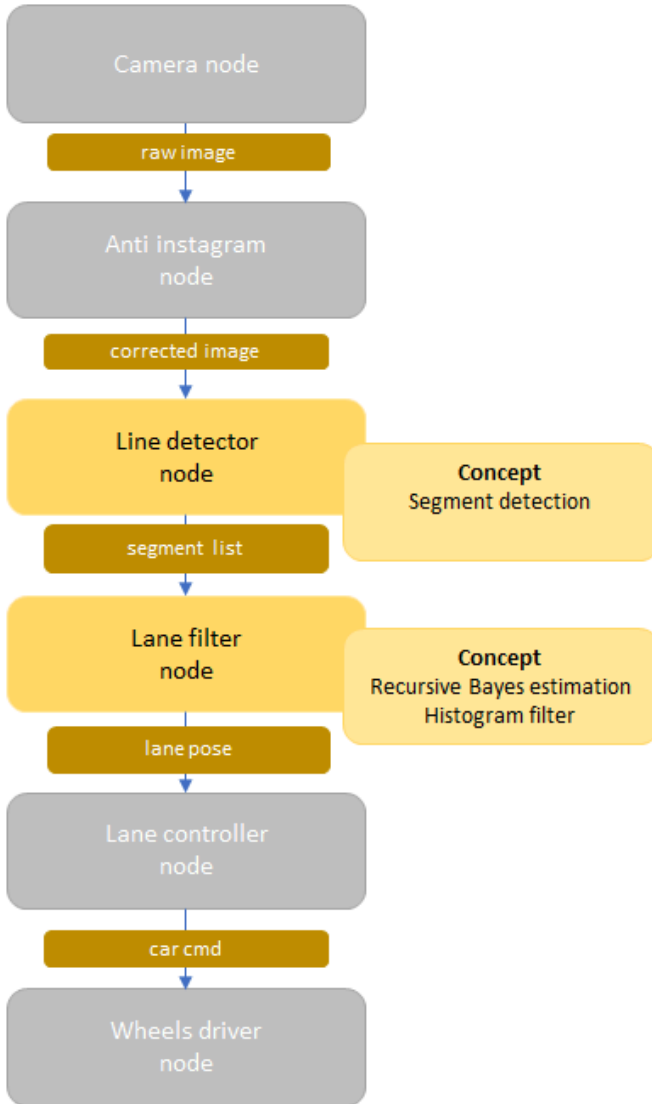
Figure 2.2. The two steps of focus are highlighted in yellow.

2.2. Line detector node

1) Role of the node

The line detector node is responsible for detecting lines in the field of view of the Duckiebot. As the color of the lines provides localization information, we are also interested in clustering them into three different colors: red, white and yellow.

2) ROS interfacing of the node

The line detector node subscribes to:

- The corrected image stream

The line detector node publishes:

- Segment list (type: `SegmentList.msg`) is an array which saves all segments (type: `Segment.msg`) found in the image. A segment consists of color `(red, yellow, white)` and 2D vector `(startpoint, endpoint)`.

3) Relevant part of the code

We won't go too much into the details of the code, but the most important bits are here:

Snippet of the main function:

```python
def processImage_(self, image_msg):
    ...
    white = self.detector_used.detectLines('white')
    yellow = self.detector_used.detectLines('yellow')
    red = self.detector_used.detectLines('red')
    ...max] = self.filter.getEstimate()
...
```

Snippet of the `detectLines` function:

```python
class LineDetectorHSV(dtu.Configurable, LineDetectorInterface):
    ...
    def detectLines(self, color):
        with dtu.timeit_clock('_colorFilter'):
            bw, edge_color = self._colorFilter(color)
        with dtu.timeit_clock('_HoughLine'):
            lines = self._HoughLine(edge_color)
        with dtu.timeit_clock('_findNormal'):
            centers, normals = self._findNormal(bw, lines)
        return Detections(lines=lines, normals=normals,
                          area=bw, centers=centers)
    ...
```

In a nutshell, the code first filters the image pixels by color, then uses a Hough line detector from OpenCV, and extract the normals to the detected lines. The most important part is executed in the Hough detector. Find the file here if you want to read more.

4) The focus of the exercise

Over all the parameters we could choose to play with here, we decided to focus on the number of segments that this node will output to the next one:

- If it gives too few segments, the localization will be imprecise but quick
- If it gives too many segments, the localization will be on average more accurate, but also slower to compute

There is a `segment_max_threshold` parameter that allows the user to limit the number of segments that are sent. The parameter limits the maximum number of segments for

each of the colors individually. Setting it for example to 10 will yield an output of 10 yellow, 10 white and 10 red segments. Exercise 1 - Choosing the best number of segments (frequency) will give you the opportunity to play with it and see the effects of the trade-off.

2.3. Lane filter node

1) Role of the node

The lane filter node is responsible for estimating the position of the Duckiebot with respect to the center of the driving lane.

2) ROS interfacing of the node

The lane filter node subscribes to:

- The segment list from the line detector node

The lane filter node publishes:

- Lane pose (type: `duckietown_msgs/lane_pose`): is struct with the following parameters which are currently in use:
- d (`float32`) the lateral offset, where $d = 0$ is the middle of the right lane.
- ϕ (`float32`) the angle from the center of the lane to the orientation of the Duckiebot.

> **Note:** When the Duckiebot is perfectly aligned in the center of its lane, facing forward, this estimation should be $(d = 0.0, \phi = 0.0)$

3) Bayes filter

To track the estimated pose (d, ϕ) of the Duckiebot in the lane, we use a Bayes filter. As usual, it relies on the predict and update steps.

Let's focus on the update step, as the predict step is simply applying the model of the dynamics on the belief.

In this node, the estimation of (d, ϕ) is represented as a matrix, holding d on one axis and ϕ on the other. This means that the space of (d, ϕ) is discretized. The discretization step is controlled by the `matrix_mesh_size` parameter. The bigger the discretization is, the rougher the estimates will be. The smaller the discretization is, the finer the estimates will be.

But since the minimum and maximum values of both d and ϕ are constant, the size of the matrix increases when the discretization step becomes smaller. In Exercise 3 - Choosing the best matrix size, you will have to play with this parameter to understand the trade-off between the granularity of the estimation and the computation time.

Snippet of the bayes filter:

```python
def processSegments(self,segment_list_msg):
    ...
    #(v and w come from car_cmd)
    self.filter.predict(dt=dt, v=v, w=w)

    #input: line segments from line detector
    #output: belief matrix
    self.filter.update(segment_list_msg.segments)

    #input: belief matrix
    #output: maximal d and phi from belief matrix
    [d_max, phi_max] = self.filter.getEstimate()
    ...
```

4) The histogram filter (for the update step)

Each 2D white and yellow segment is projected onto the Duckiebot reference frame. Then the corresponding (d, ϕ) tuple is extracted from geometric knowledge of the lane.

Each segment's extracted tuple (d, ϕ) casts a vote in the measurement likelihood histogram matrix, as mentioned above. This matrix can be then displayed as an image stream.

One would hope that the majority of the segments will vote to the same area of the histogram. With this matrix, the belief matrix is updated.

Then, the maximum is extracted from the updated belief matrix. The maximum's coordinates give us the best estimate of the tuple (d, ϕ).

Snippet of the the generation of votes for the histogram filter:

```python
#Generation of votes for the the histogram filter
def generateVote(self, segment):
    p1 = np.array([segment.points[0].x, segment.points[0].y])
    p2 = np.array([segment.points[1].x, segment.points[1].y])
    t_hat = (p2 - p1) / np.linalg.norm(p2 - p1)

    n_hat = np.array([-t_hat[1], t_hat[0]])
    d1 = np.inner(n_hat, p1)
    d2 = np.inner(n_hat, p2)
    l1 = np.inner(t_hat, p1)
    l2 = np.inner(t_hat, p2)
    if (l1 < 0):
        l1 = -l1
    if (l2 < 0):
        l2 = -l2

    l_i = (l1 + l2) / 2
    d_i = (d1 + d2) / 2
    phi_i = np.arcsin(t_hat[1])
    if segment.color == segment.WHITE:  # right lane is white
      if(p1[0] > p2[0]):  # right edge of white lane
          d_i = d_i - self.linewidth_white
      else:  # left edge of white lane
          d_i = - d_i
          phi_i = -phi_i
      d_i = d_i - self.lanewidth
    elif segment.color == segment.YELLOW:  # left lane is yellow
      if (p2[0] > p1[0]):  # left edge of yellow lane
          d_i = d_i - self.linewidth_yellow
          phi_i = -phi_i
      else:  # right edge of white lane
          d_i = -d_i
      d_i = - d_i

      weight = 1
      d_i += self.center_lane_offset

      return d_i, phi_i, l_i, weight
```

For more about this part of the code, go here.

Exercises - lane pose estimation

The goal of this exercises is to play with existing parameters to understand the different trade-offs mentioned in Unit B-2 - Learning materials.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Camera calibration **(unknown ref opmanual_duckiebot/camera-calib)**

previous **warning** next (9 of 17) index

warning

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

 > I do not know what is indicated by the link '#op-
manual_duckiebot/camera-calib'.
```

Location not known more precisely.

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

Requires: Docker basics **(unknown ref duckietown-robotics-development/docker-basics)**

previous **warning** next (10 of 17) index

warning

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

 > I do not know what is indicated by the link
'#duckietown-robotics-development/docker-basics'.
```

Location not known more precisely.

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

Requires: ROS basics **(unknown ref duckietown-robotics-development/sw-advanced)**

previous **warning** next (11 of 17) index

warning

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

 > I do not know what is indicated by the link
'#duckietown-robotics-development/sw-advanced'.
```

Location not known more precisely.

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

Requires: Knowledge of the software architecture on a Duckiebot **(unknown ref duckietown-robotics-development/duckietown-code-structure)**

previous **warning** next (12 of 17) index

warning

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

 > I do not know what is indicated by the link
 '#duckietown-robotics-development/duckietown-code-
 structure'.
```

Location not known more precisely.

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

Results: Understand the trade-offs when dealing with image processing parameters

Results: Insights into the image pipeline of a Duckiebot.

3.1. Task 1: Line detector exercise

As previously introduced, the `line_detector_node` detects white, yellow and red segments. The more segments we get, the more accurate we expect the lane filter to be, but also the more resources we need for computation of the pose estimate (memory as well as CPU usage). This is a trade-off between accuracy and computational efficiency. The goal of this exercise is to analyze this trade-off by determining the relationship between the number of segments processed and the quality and frequency of pose estimates that are being computed.

For this task the parameter `/ DUCKIEBOT_NAME /line_detector_node/segment_max_threshold` can be dynamically adjusted.

Exercise 1. Choosing the best number of segments (frequency).
Put the Duckiebot in the city and let it drive one whole loop with the exercise-provided lane following. For every whole loop use a different parameter `/ DUCKIEBOT_NAME /line_detector_node/segment_max_threshold` and record a rosbag of `lane_pose` for each value of `segment_max_threshold`. You should know how to do that from (unknown ref duckietown-robotics-development/ros-logs)

previous **warning** next (13 of 17) index
warning

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

 > I do not know what is indicated by the link
 '#duckietown-robotics-development/ros-logs'.
```

Location not known more precisely.

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

.

Write a custom Python script to analyze the publishing_frequency of the topic `/ DUCKIEBOT_NAME /lane_filter_node/lane_pose` for each bag. Plot the relationship between `segment_max_threshold` on one axis and the mean and standard deviation of the lane_pose publishing frequency on the other axis. Provide at least 4

points on the plot. Include a point with a very high `segment_max_threshold` to virtually allow all segments to be computed.

Frequency isn't the only relevant metric. Using one segment per color will give fast computation but very noisy and unstable estimation. Using the `rviz` tool that you launched before, you can analyze the stability of the lane_pose.

Exercise 2. Choosing the best number of segments (stability).

Create a graph, ploting on the y-axis (d, ϕ) against time on the x-axis for each of the loops from the previously recorded rosbags.

3.2. Task 2: Lane pose exercise

As outlined in the introduction section, `lane_filter_node` estimates the Duckiebot's desired pose by means of recursive Bayes estimation. The sizes of the belief/likelihood matrices are adjustable parameters. We are interested in analyzing the effect of various matrix sizes on the precision/standard deviation of the lane pose estimation.

For this task the parameter `/ DUCKIEBOT_NAME /lane_filter_node/matrix_mesh_size` can be dynamically adjusted.

Exercise 3. Choosing the best matrix size.

While running the exercise-provided lane following, play with `matrix_mesh_size`, and record different rosbags for the topic lane_pose (one for each value of `matrix_mesh_size`).

Write a custom Python script to analyze the frequency of the topic `/ DUCKIEBOT_NAME /lane_filter_node/lane_pose` for each bag (should be the same as last exercise). Plot the relationship between `matrix_mesh_size` on one axis and the the mean and standard deviation of the frequency of the `lane_pose` topic on the other axis. Provide at least 4 points on the plot.

Warning: sometimes, when dynamically changing the parameters, errors might occur since the matrix size might be changing during computation of the segments. In the occurrence of such a problem, you can restart the node or set the previous value of the mesh and then retry.

3.3. Task 3: English driver

One of our brave Duckiebots wanted to make a visit to a fellow Duckiebot at the London Science Museum in Great Britain (yup, must be *really* brave to go right before Brexit :X). However, it needs to adhere to the local driving rules. Therefore you will have to help it learn to drive on the left side of the road.

Exercise 4. Driving the English style.

The task is to make the Duckiebot drive on the left side of the road. The parameter `/ DUCKIEBOT_NAME /lane_filter_node/lane_offset` and the provided snippet

provided code snippet is sufficient to complete this task. Coding is not necessary for this exercise.

Exercises - state estimation and sensor fusion

In this exercise you will learn how to create estimators with both proprioceptive and exteroceptive sensors and how to manipulate the frame transformations tree to easily fuse these estimates.

<div align="center">

KNOWLEDGE AND ACTIVITY GRAPH

</div>

Requires: Odometry with Wheel Encoders (unknown ref duckietown-robotics-development/odometry-modeling)

previous **warning** next (14 of 17) index

warning

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

 > I do not know what is indicated by the link
'#duckietown-robotics-development/odometry-modeling'.
```

Location not known more precisely.

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

Requires: Modeling the Duckiebot and Representations (unknown ref duckietown-robotics-development/representations-modeling)

previous **warning** next (15 of 17) index

warning

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

 > I do not know what is indicated by the link
'#duckietown-robotics-development/representations-
modeling'.
```

Location not known more precisely.

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

Requires: Advanced Augmented Reality Exercise

Results: Understand how to build a TF tree in ROS and visualize it in RViz.

Results: Be able to create a multi-package Dockerized ROS workspace, and deploy it on the Duckiebot.

Results: Experience with working with ROS Timers and Services

Results: Create a wheel-encoder based estimator. Understand the benefits and drawbacks of such an estimator.

Results: Create an Apriltag-based estimator. Understand the benefits and drawbacks of this estimator.

Results: Create a sensor fusion node that fuses estimates from the above individual

> estimators.

The overall goal of this exercise is to create a state estimation system that localizes the Duckiebot in global coordinates. So far we have only seen how this can be done locally, relative to the lane. This has two major limitations. First, we don't know how far we have travelled in the lane, as we only can get estimates on our lateral offset from the center of the lane. Second, we don't know if we are in a straight or turning lane segment. In this exercise, however, we will work towards localizing the robot in a global coordinate frame.

To be able to perform such a complex task, we will have to use all sensors onboard the robot, namely the camera and the wheel encoders. We will achieve our localization goal in several incremental steps. First, we will develop an estimator using only wheel encoders. Then, we will develop an estimator using only camera information via the Apriltag detection library. Finally, we will fuse these estimates so that we get the best of both worlds.

Exercise 5. Encoder localization package.

The first package you'll create in your Dockerized ROS workspace is one that will contain a node that publishes a `TransformStamped` message at 30Hz with the following fields:

- `frame_id`: `map`;
- `child_frame_id`: `encoder_baselink`;
- `stamp`: timestamp of the last wheel encoder tick;
- `transform`: a 2D pose of the robot baselink (see Figure 4.2 for a definition of this frame).

The node will also have to broadcast the TF `map`-`encoder_baselink`.

Deliverables:

- A screen recording similar to this Figure 4.1 where you drive the robot in a loop and try to "close the loop" (by driving to the point where you started from). Make sure that the video contains the camera images as well as the TF tree in Rviz. Use a visible landmark as the origin of the map frame.

- A link to a Github repository containing a package called `encoder_localization`.

Hints:

- For the estimates to be in a global frame, you will have to provide an initial pose estimate.

- Use `tf.TransformBroadcaster()` to broadcast a TF which can be visualized in RViz. Note that you can pass the exact same message as the one that your publisher uses.

- For the kinematic model of the Duckiebot, you'll have to load the following parameters from the kinematics calibration file: `baseline`, `radius`.

- To open RViz, simply run `dts start_gui_tools hostname.local` and run `rviz` from inside the container. Note that if you keep the container running you can save your RViz configuration file so that when you reopen it, it automatically displays your topics of interest.

- Rviz uses the following color-code convention for frame axes: red for the x-axis, green for the y-axis, and blue for the z-axis.

- To publish messages at a fixed frequency, consider using a ROS Timer. It works very similar to a subscriber except for the fact that the callback get called after a fixed duration of time.

- This link contains many useful methods from the `tf` library that allow you to switch between representations (e.g. expressing an euler yaw angle as a quaternion).
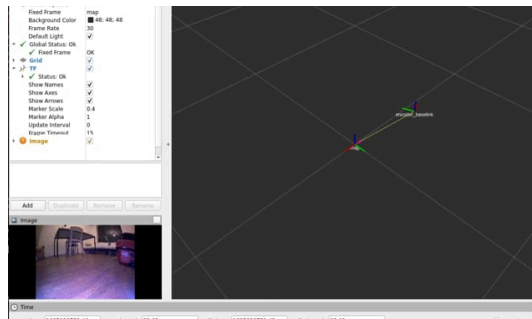

Figure 4.1. Example video deliverable for the encoder localization package
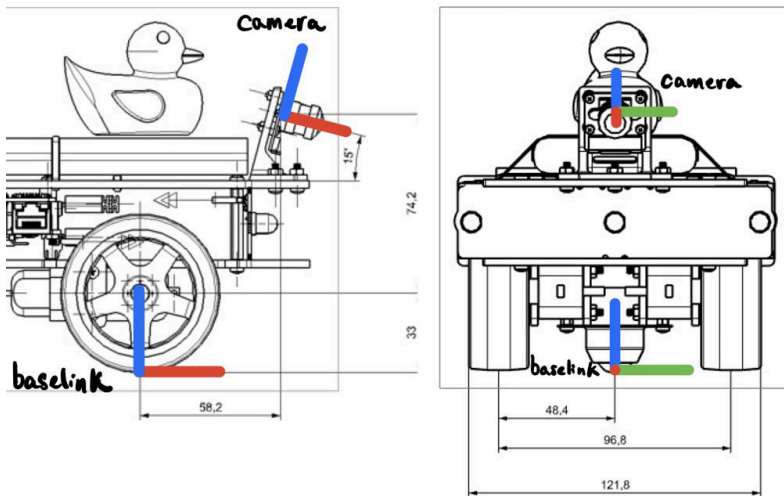

Figure 4.2. Position of the baselink and camera frames on the robot

As you have probably realized, some of the advantages of this localization system is that as long as the robot is moving, the wheel encoders will provide information about the state of the robot, at a high rate and with little delay. However, the pose of the robot is an integration of the wheel encoder measurements, meaning that it is subject to drift as any inaccuracies in measurement get propagated through time. When driving aggressively or through a slippery surface these inaccuracies are amplified (try this for yourself!).

A common way of getting rid of drift in mobile robots is to either use absolute position measurements (with GPS, for instance) or to use fixed landmarks in the world as reference. This is how we humans also navigate the environment. Since there is no GPS on-board the Duckiebot, we will have to use the latter approach. This is what we will

explore in the next exercise, where our landmarks will be traffic signs with Apriltags (see Figure 4.3).

Figure 4.3. Example traffic sign to be used for Apriltag localization. In order to accurately construct your TF tree, please measure the height indicated

Exercise 6. Apriltag localization package.

In this package you'll have to place a node that subscribes to `/camera_node/image/compressed` and publishes a `TransformStamped` message (if an image with an Apriltag has been received) with the following fields:

- `frame_id`: `map`;
- `child_frame_id`: `at_baselink`;
- `stamp`: timestamp of the last image received;
- `transform`: a 3D pose of the robot baselink (see Figure 4.2 for a definition of this frame).

This node will also have to broadcast the following TFs: `map`-`apriltag`, `apriltag`-`camera`, `camera`-`at_baselink`. Make sure that when you place the Apriltag in front of the robot you get something that looks roughly like Figure 4.5. This will make it easier to fuse this pose with the pose from the encoders.

Deliverables:

- A screen recording similar to the one in Figure 4.6 where you move the Apriltag in front of the robot from one side of the Field-of-View (FOV) to the other. Make sure that the video contains the camera images as well as the TF tree in Rviz.

- Instead of directly using compressed images, rectify them before passing them to the Apriltag detector. You will see that this significantly improves the accuracy of the detector but at the cost of significant delay. Provide a screen recording similar to this Figure 4.7 where you move the Apriltag in front of the robot from one side of the FOV to the other. Make sure that the video contains the camera images as well as the TF tree in Rviz. You should observe that the `map` and `baselink` are now in the same plane (or

at least much closer to it than with compressed images).

• (Bonus) Offload the computation of the rectified image to the GPU of the Jetson Nano so that the improved accuracy can be obtained without significant delay.

• A link to your Github repository containing a package called `at_localization`.

Hints:

• A frame cannot have two parents. If you try to broadcast the following TFs: `map`-`baselink` and `camera`-`baselink`, you will see that you'll only be able to see the individual frames in RViz.

• To rectify images, use the following `cv2` methods:

```python
newCameraMatrix = cv2.getOptimalNewCameraMatrix(cameraMatrix,
                                                distCoeffs,
                                                (640, 480),
                                                1.0)
cv2.initUndistortRectifyMap(cameraMatrix,
                            distCoeffs,
                            np.eye(3),
                            newCameraMatrix,
                            (640, 480),
                            cv2.CV_32FC1)
cv2.remap(compressed_image, map1, map2, cv2.INTER_LINEAR)
```

• To use the Apriltag detector, consult this link. You can extract the pose of the Apriltag in the camera frame with `tag.pose_R` (rotation matrix) and `tag.pose_t` (translation vector). Keep in mind that the coordinate frame convention (see Figure 4.4) is different than the one you are supposed to use in the deliverable!

• To broadcast static transforms, use

```python
self.static_tf_br = tf2_ros.StaticTransformBroadcaster()
```

To calculate the static transform between the `map` and `apriltag` frames (

$$T_{MA}$$

) use Figure 4.3 as reference. To calculate the static transform between the `baselink` and `camera` frames, you can use Figure 4.2 as reference, or you can try to use the homography matrix obtained during extrinsic calibration.

• To convert between frames, we recommend that you use 4x4 transformation matrices. An example of such a matrix is

$$T_{AB} = \begin{bmatrix} R_{AB} & {}_A\vec{r}_{AB} \\ \vec{0} & 1 \end{bmatrix}$$

which transforms a vector in frame B to frame A. During your manipulations, keep in mind that

$$T_{BA} = (T_{AB})^{-1}$$

and

$$T_{AB} = T_{AC}T_{CB}$$

for any frames
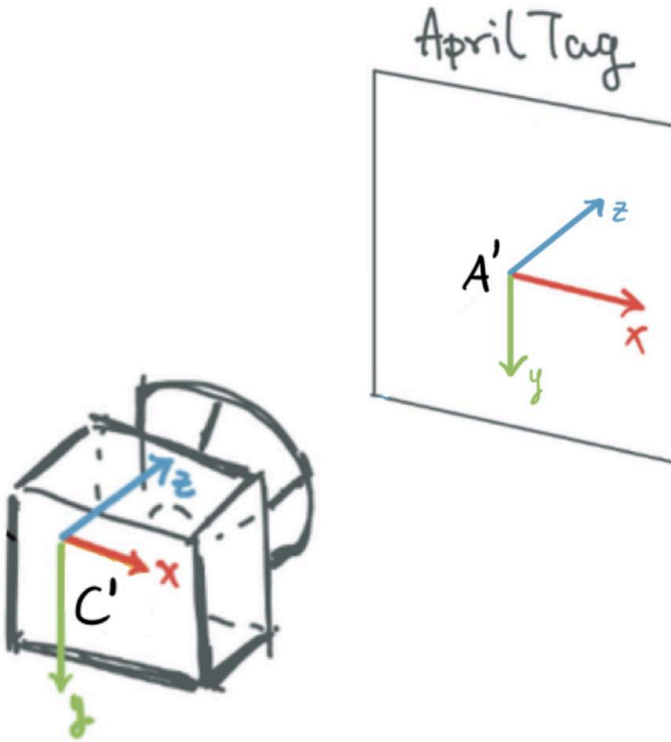
$$A, B$$

and

$$C$$

.

Figure 4.4. Frame convention used by Apriltag library when returning pose.
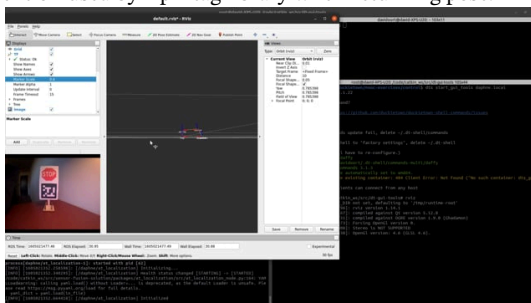

Figure 4.5. Goal TF tree for the AT localization package with rectified images and the robot facing the Apriltag
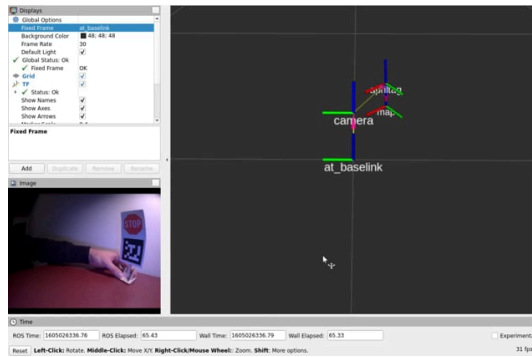
Figure 4.6. Example video deliverable for the AT localization package with compressed images
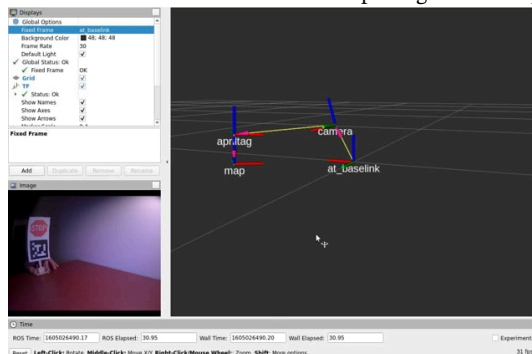

Figure 4.7. Example video deliverable for the AT localization package with rectified images

The detected Apriltag can provide accurate pose estimates with respect to landmarks but at the cost of a significant delay and a low frequency. Moreover, one cannot continuously rely on such estimates since the Apriltag could go out of sight. To combat this, we can fuse the estimates of the Apriltag and the wheel encoders for continuous, accurate and robust localization. This is the goal of the next exercise.

Exercise 7. Fused localization package.

In this package you'll have to place a node that publishes a `TransformStamped` message with the following fields:

- `frame_id`: `map`;
- `child_frame_id`: `fused_baselink`;
- `stamp`: current time,
- `transform`: a 2D pose of the robot baselink.

The node will also have to broadcast the TF `map` - `fused_baselink`. As a minimum, your fusion node should have the following behaviour:

• The first time the Apriltag becomes visible, you have to calibrate the `encoder_baselink` frame/estimate to match exactly the Apriltag pose. This should be done with a ROS Service provided by and `encoder_localization_node` (server) which `fused_localization_node` (client) can call. If you need an example of a Duckietown package that defines a similar service, check this out (pay a special attention to the `CMakeLists.txt` and `package.xml` files).

• Publish the robot pose using the Apriltag estimate (when available) projected on the

ground plane (recall that this node publishes 2D poses).

• If the Apriltag is not visible, use the encoder estimates, starting from the last April-tag pose received (there should be no pose jump if the Apriltag goes out of sight).

• If the Apriltag becomes visible again, switch back to using Apriltag estimates (a jump in fused pose is allowed).

• You don't have to handle the delay or the variance of the individual estimates to complete the exercise, but you are more than welcome to!

Deliverables:

• A screen recording similar to this Figure 4.8, where you drive the robot in a loop around the Apriltag with the virtual joystick. Start the recording with the Apriltag not visible, so that you validate that your calibration service is working and when it does bacome visible all the frames align. You should end the trajectory where you started it (feel free to use a marker on the ground). Make sure that the video contains the camera images as well as the TF tree in Rviz.

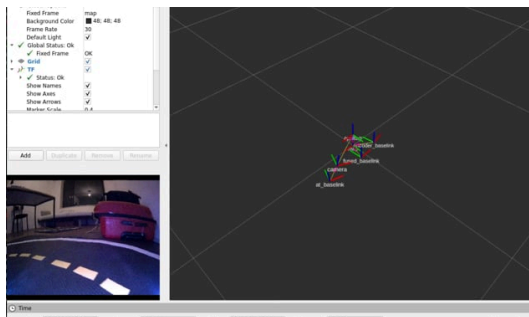• A link to a github repository containing a package called `fused_localization`

Figure 4.8. Example video deliverable for the fused localization package with rectified images and slow driving

Modeling and control

In the last chapter of this book you have learned how the Duckiebot can localize itself in the lane. In this chapter, you are going to learn how to leverage this knowledge to implement different control algorithms which enable the Duckiebot to keep itself in the lane and you will be introduced to a range of details that need to be addressed when controlling a real system.

<div align="center">

Unit C-1

Preliminaries

</div>

Preliminaries...

Learning materials

Learning materials

UNIT C-3

Exercise: Control

In this exercise you will learn how to implement different control algorithms on a Duckiebot and gain intuition on a range of details that need to be addressed when controlling a real system.

<div align="center">

KNOWLEDGE AND ACTIVITY GRAPH

</div>

Requires: Terminal Basics (unknown ref duckietown-robotics-development/terminal-basics)

previous **warning** next (16 of 17) index

warning

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

 > I do not know what is indicated by the link
'#duckietown-robotics-development/terminal-basics'.
```

Location not known more precisely.

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

Requires: Docker Basics (unknown ref duckietown-robotics-development/docker-basics)

previous **warning** (17 of 17) index

warning

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

 > I do not know what is indicated by the link
'#duckietown-robotics-development/docker-basics'.
```

Location not known more precisely.

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

Requires: Control Theory Basics

Results: Ability to implement a controller on a real robot.

Contents

3.1. Overview

In the following exercise you will be asked to implement two different kinds of control algorithms to control the Duckiebot. In a first step, you will write a PI controller and gain some intuition on different factors that are important for the controller design

such as the discretization method, the sampling time, and the latency of the estimate. You will also learn what an anti-windup scheme is and how it can be useful on a real robot.

In a second step, you will implement a Linear Quadratic Regulator, or LQR for short. You then augment it by an integral part, making it a LQRI. This is a more high-level approach to the control problem. You will see how it is less intuitive but at the same time it brings certain advantages as you will see in the exercise.

3.2. PI control

1) Modeling

As you have learned, using Figure 3.1 one can derive a continuous-time nonlinear model for the Duckiebot. Considering the state $\vec{x}(t) = [d \quad \varphi]^{\mathsf{T}}$, one can write $\dot{\vec{x}} = \begin{bmatrix} v \cdot \sin(\varphi) \\ \omega \end{bmatrix}$. Where v is the linear velocity and ω the yaw rate of the Duckiebot.

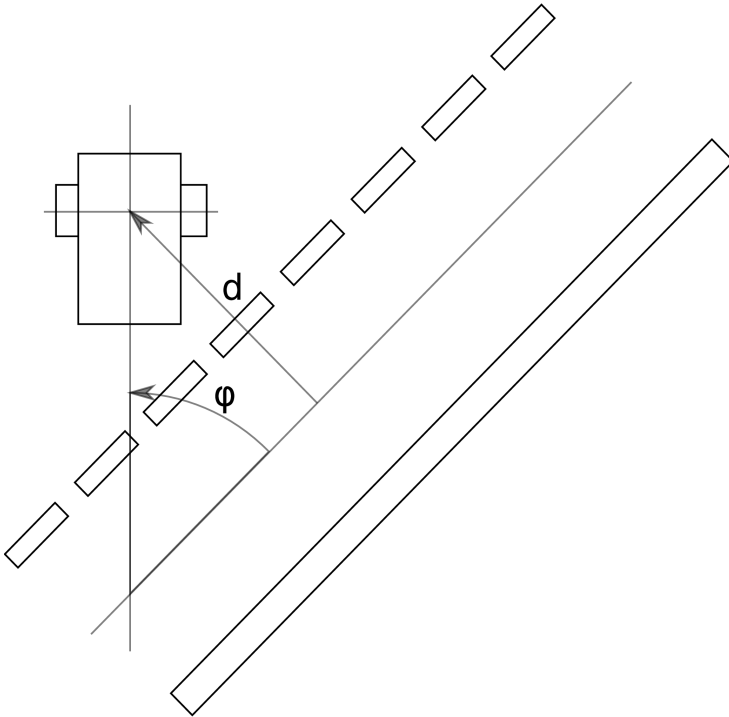
Figure 3.1. Top view of the Duckiebot on a road with its two states.

After linearization around the operation point $\vec{x}_e = [0 \quad 0]^{\mathsf{T}}$ (if you do not remember linearization, have a look at chapter 5.4 in [1]), one has $\dot{\vec{x}}(t) = \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t)$ where the input $u(t)$ is the desired yaw rate of the Duckiebot. Furthermore, you are provided the output model

$y(t) = [6 \quad 1] \vec{x}(t).$

Using the linearized version of the model, you can compute the transfer function of the system:

$$P(s) = \frac{s+6v}{s^2}.$$

If you do not remember how, have a look at chapter 8 in [1].

Consider now the error to be $e(t) = r(t) - y(t)$. Using a PI-controller (if you do not remember what a PI-controller is, have a look at chapter 10 in [1]), one can write

$$u(t) = k_\text{P}\left(e(t) + \frac{1}{T_I}\int_0^t e(\tau)d\tau\right) = k_\text{P}e(t) + k_\text{I}\int_0^t e(\tau)d\tau$$

with k_\text{P} being the proportional gain and k_\text{I} being the integral gain. In frequency domain, this corresponds to

$$U(s) = C(s)E(s),$$

with

$$C(s) = k_\text{P} + \frac{k_\text{I}}{s}$$

Exercise 8. Find the gains.

Using the above defined model of the Duckiebot and the structure for a PI controller, find the parameters for the proportional and integral gain of your PI controller such that the closed-loop system is stable. You can follow the steps below to do this:

- For the Duckiebot you are assuming a constant linear velocity $v = 0.22\text{m/s}$. Given this velocity and using a tool of your choice (for example the Duckiebot bodeplot tool), find a proportional gain k_P such that $L(s) = P(s)C(s)$ has a crossover frequency of approximately 4.2rad/s.

- Next, find an integral gain k_I such that $L(s)$ has a gain margin of approximately $-25.6dB \approx 0.053$. (this refers to a gain of the controller which is about 19 times higher than the critical minimal gain that is needed for stability).

The aforementioned numbers are needed in order to guarantee stability. You are free to play around with them and see for yourself how this impacts the behaviour of your Duckiebot.

2) Discretization

Now that you have found a continuous time controller, you need to discretize it in order to implement it on your Duckiebot. There are several ways of doing this. In the following exercise, you are asked to implement the designed PI controller in reality, using different discretization techniques.

Exercise 9. Discretization of a PI controller.

There is a template for this and the following exercises of this chapter. It is a Docker image of the `dt-core` with an additional folder called `CRA3`. This folder contains two controller templates, `controller-1.py` and `controller-2.py`. The first one will be used for the exercises with the PI controller and the other one for the implementation of the LQR(I) which you will do from exercise Exercise 13 - Implement a LQR

onwards. Start by pulling the image and running the container with the following command:

```
$ docker -H DUCKIEBOT_NAME.local run --name dt-core-CRA3 -v
/data:/data --privileged --network=host -it duckietown/dt-
core:CRA3-template /bin/bash
```

Note: in case you have to stop the container at any point (in case you take a break or the Duckiebot decides to crash and therefore makes you take a break), start the container again (for example by using the portainer interface (`http://host-name.local:9000/#/containers`)) and jump into it using the following command:

```
$ docker -H DUCKIEBOT_NAME.local attach dt-core-CRA3
```

The text editor *vim* is already installed inside the container such that you can change and adjust files within the container without having to rebuild the image every time you want to change something. If you are not familiar with vim, you can either read through this short beginners guide to vim or install another text editor of your choice.

Now use vim or your preferred text editor to open the file `controller-1.py` which can be found in the folder `CRA3`.

The file contains a template for your PI controller including input and output variables of the controller and several variables which will be used within this exercise. As inputs you will get the lane pose estimate of the Duckiebot and you will have to compute the output which is in the form of the yaw rate ω.

Familiarize yourself with the template and fill in the previously found values for the proportional and integral gain k_P and k_I.

Now you are ready to implement a PI controller using different discretization methods:

Assume constant sampling time

In a first attempt, you can use an approximation for your sampling time. The Duckiebot typically updates its lane pose estimate, i.e. where the Duckiebot thinks it is placed within the lane, at around 12 Hz. If you assume this sampling rate to be constant, you can discretize the PI controller you designed. Implement your PI controller under the assumption of a constant sampling in the file `controller-1.py`. When discretizing the system, choose Euler forward as the discretization technique (if you do not remember how, have a look at chapter 2.3 in [4]). You can run the controller you just designed by executing the following command:

```
$ roslaunch duckietown_demos lane_following_exercise.launch
veh:=DUCKIEBOT_NAME exercise_name:=1
```

Observe the behaviour of the Duckiebot. Does it perform well? What do you observe? Think about why this is the case.

Optional: repeat the above task using the Tustin discretization method. Do you observe any difference?

Assume a dynamic sampling time

Now you will use the actual time that has passed in between two lane pose estimates of the Duckiebot to discretize the system. The time between two lane pose estimates is already available to you in the template and is called `dt_last`. Adjust the discretization method (either Euler forward or Tustin) of your controller to account for the actual sampling time. After you adjusted your file, use the same command as above to test your controller. Observe the behaviour again, what differences do you notice? Why is that?

Assume a large sampling time

In the last exercise you implemented a discrete time controller and saw how slight variations in the sampling time can have an impact on the performance of the Duckiebot. You now want to further explore how the sampling time impacts the performance of the controller by increasing it and observing the outcome. For the following, consider Euler forward as the discretization technique. The model of a Duckiebot only works on a specific range of consequent states $[d_{i,i+1}, \varphi_{i,i+1}]$. If these values grow too abruptly, the camera loses sight of the lines and the estimation of the output y is not anymore possible. By increasing k_s in `controller-1.py`, check how much you can reduce the sampling rate before the system destabilizes. Notice that since your controller is discrete, you can only increase the sampling time T in discrete steps k_s where $T_{new} = k_s \cdot T$. This functionality is already implemented in the lane controller node for you. To reduce the sampling rate, the Duckiebot only handles every k_s-th measurement (#**measurements mod** $k_s \equiv 0$), and drops all the other measurements. Adjust the parameter k_s such that the Duckiebot becomes unstable. What is the approximate sampling time when the Duckiebot becomes unstable? Again run your code with:

```
$ roslaunch duckietown_demos lane_following_exercise.launch
veh:=DUCKIEBOT_NAME exercise_name:=1
```

After you have found a value for k_s that destabilizes your Duckiebot, try to improve the robustness of your controller against the smaller sampling rate and make it stable again. There are different ways to do this. Explain how you did it and why.

3) Latency of the estimate

Until now, the delay which is present in the Duckiebot (the plant) has not been explicitly addressed. From the moment an image is recorded until the lane pose estimate is available, it takes roughly 85ms. This implies that you will never be able to act upon the exact state that your Duckiebot is observed to be in. In the following exercise you will examine how the Duckiebot behaves if this delay between image acquisition and pose estimation changes.

Exercise 10. Increasing the delay.

Stability - Theoretical

As you have already seen in the previous tasks, the time delay of 85ms does not destabilize your system. By using your calculations from Section 3.2 - PI control, you are indeed able to identify a maximal time delay such that your system is still stable in theory. This can be done by having a look at the transfer function of a time-delayed system: $P_d(s) = e^{-sT_d} P(s)$ with T_d being the time delay. An increase of T_d leads to a shift of the phase in negative direction. Therefore, T_d must not be larger than the phase margin of $L(s)$ (which was roughly **70°** in our case) to prevent destabilizing the system. Calculate the maximal T_d such that the system is still stable.

Stability - Practical

Before you can reach the theoretical limit you found in the previous task, the Duckiebot will most likely leave the road and the pose estimation will fail since the lines are not in the field of view of the camera anymore. In `controller-1.py`, increase the time delay gain k_d of the system until the Duckiebot cannot stay in the lane anymore. Notice that the time delay is implemented in discrete steps of $k_d * T$ where T is the sampling time. Again run your code with:

```
$ roslaunch duckietown_demos lane_following_exercise.launch
veh:=DUCKIEBOT_NAME exercise_name:=1
```

How big is the difference between the theoretical and the practical limit?
Optional: Check if using another discretization technique substantially changes these numbers.

---

4) Increase performance of your PI controller

The integral part in the controller comes with a drawback in a real system: Due to the fact that the motors on a Duckiebot can only run up to a specific speed, you are not able to perform unbounded high inputs demanded by the controller. If the Duckiebot cannot execute the commands which the controller demands, the difference between the demanded input and the executed input will remain and therefore be added on top of the demanded input which is already too high to be executed. This leads us to a situation in which the integral term can become very large. If you now reach your desired equilibrium point, the integrator will still have a large value, causing the Duckiebot to overshoot.
But behold, there is a solution to this problem! It is called anti-windup filter and will be examined in the next exercise.

Exercise 11. Effect of an Anti-Windup Filter.

In Figure 3.2, you can see a diagram of an anti-windup logic for a PI-controller. k_t determines how fast the integral is reset and is usually chosen in the order of k_I.
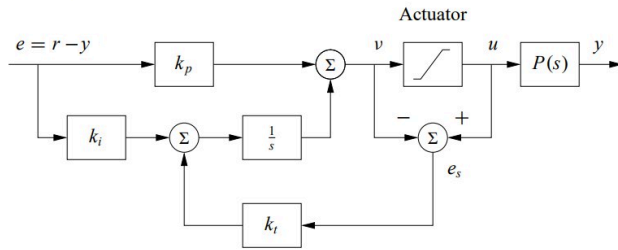
Figure 3.2. A PI-controller with an anti-windup logic implemented, Feedback Systems from Aström and Murray, page 308.

Typically, the actuator saturation (i.e. when it reaches its physical limit) can be measured. In our case, however, as there is no feedback on the wheels commands that are being executed, we will make an assumption. You will simulate a saturation of the motors at a value of $u_{sat} = 2\text{rad/s}$.

Below you can find a simple helper function that you can use to add an anti-windup to your existing PI controller. It takes an unbounded input and limits it to the mentioned saturation input value u_{sat}. Use it to extend your existing PI controller with an anti-windup scheme.

Furthermore you are given the parameter k_t in the file `controller-1.py`. It shall be used as a gain on the difference between the input u and the saturation input value u_{sat} which is fed back to the integrator part of the controller as it is shown in Figure 3.2. As a first step, test the performance of the Duckiebot with the anti-windup term turned off (i.e. $k_t = 0$). You will see that the performance is poor after curves. If you increase the integral gain k_I, you are even able to destabilize the system! In order to avoid destabilization and improve the performance of the system, set k_t to roughly the same value as k_I. Note the difference! You can run your code as before with:

```
$ roslaunch duckietown_demos lane_following_exercise.launch
veh:=DUCKIEBOT_NAME exercise_name:=1
```

Optional: With different values of k_P and k_I, one could improve the behaviour even more.

Template for saturation function:

```
def sat(self, u):
        if u > self.u_sat:
            return self.u_sat
        if u < -self.u_sat:
            return -self.u_sat
        return u
```

As you may have found out, for very aggressive controllers with an integral part and systems which saturate for relatively low inputs, the use of an anti-windup logic is necessary. In the case of a Duckiebot however, an anti-windup logic is only needed if

you want to introduce a limitation to the angular velocity ω - for example to simulate a real car (minimal turning radius).

By now you should have a nicely working controller to keep your Duckiebot in the lane, which is robust against a range of perturbations which arise from the real world. But is the solution that you found optimal and does it give us a guarantee on its stability? The answer to both of these questions is no. Also, you saw that the fact that your model is not exactly matching the reality can lead to a worse performance.

Therefore, it would be useful to have a control algorithm which does not depend heavily on the given model and gives us guarantees on its stability. In the last two exercise parts, you will look at a different controller which will help us solve the above mentioned problems; namely a Linear-Quadratic-Regulator (LQR).

3.3. Linear Quadratic Regulator (*Optional*)

A Linear Quadratic Regulator (LQR) is a a state feedback control approach which works by minimizing a cost function. This approach is especially suitable if we want to have some high-level tuning parameters where the cost can be traded off against the performance of the controller. Here, we typically refer to "cost" as the needed input $u(t)$ and "performance" as the reference tracking and robustness characteristics of the controller. In addition, LQR control works well even when no precise model is available as it is often the case in practical applications. This makes it a suitable controller for real world applications.

Exercise 12. Discretize the model.

As in the part above, you will start with the model of the Duckiebot. This time though you are going to discretize the system before creating a controller for it which will make updating the weights easier once you test your controllers on the real system. The continuous time model of a Duckiebot is: $\dot{\vec{x}} = A\vec{x} + Bu = \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix} \vec{x} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$

$\vec{y} = C\vec{x} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \vec{x}$ With state vector $\vec{x} = [d, \quad \varphi]^T$ and input $u = \omega$. Notice, that the matrix C is an identity matrix, which means that the states are directly mapped to the outputs. Discretize the system in terms of velocity v and the sampling time T using exact discretisation (if you do not remember how, have a look at chapter 1.4 in [5]) and test your discretization using the provided Matlab-files). What do you observe? Add the found matrices in the template `controller-2.py`.

Exercise 13. Implement a LQR.

To achieve a better lane following behaviour, a LQR can be implemented. The structure of a state feedback controller such as the LQR looks as in Figure 3.3:
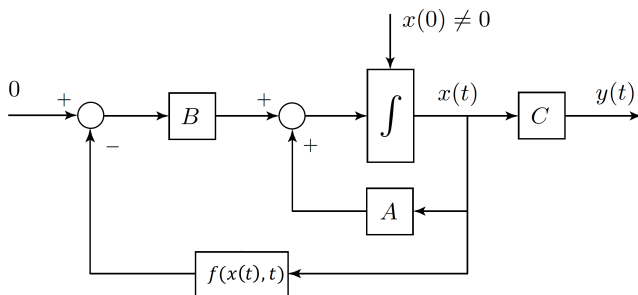
Figure 3.3. Block diagram of a state feedback control.

Because of limited computation resources, a steady-state (or *infinite horizon*) version of the LQR will be implemented. Because you are considering the discrete time model of the Duckiebot, the Discrete-time Algebraic Riccati Equation (DARE) has to be solved:

$$\Phi = A^T \Phi A - (A^T \Phi B)(R + B^T \Phi B)^{-1}(B^T \Phi A) + Q$$

To solve this equation use the Python control library (see Python control library documentation).

Note that what you will implement is not exactly a LQR controller. In fact, since you have state estimation, it would be preciser to talk about a LQG. However, the state estimation is not a proper gaussian filter, meaning that you are dealing with a LQGish.

A word on weighting

In general, it is a good idea to choose the weighting matrices to be diagonal, as this gives you the freedom of weighting every state individually. Also you should normalize your R and Q matrices. Choose the corresponding weights and tune them until you achieve a satisfying behaviour on the track. To find suitable parameters for the weighting matrices, keep in mind that we are finding our control input by minimizing a cost function of the form

$$u_{LQR}(t) = \underset{u(t)}{\text{argmin}} \ J_{LQR}(u(t)) = \underset{u(t)}{\text{argmin}} \int_0^\infty u^T R u + x^T Q x \ dt$$

So intuitively, one can note that a low weight on a certain state means that it has less of an impact when trying to minimize the overall cost function. A high weight means that we want to minimize this state more in order to minimize the overall function.

For example, if we give a low weight on the input $u(t)$, i.e. the weighting matrix R contains smaller values than the weighting matrix Q, the controller will care less about the input used and therefore converge to the desired reference faster.

Once you are ready, run your LQR with:

```
$ roslaunch duckietown_demos lane_following_exercise.launch
veh:=DUCKIEBOT_NAME exercise_name:=2
```

Explain what happens when you assign the entries in your weighting matrices dif-

ferent values. Can you describe it intuitively?

Exercise 14. Implement a LQRI.

The above controller should yield satisfactory results already. But you can even do better! As the LQR does not have any integrator action, a steady state error will persist. To eliminate this error, you will expand your continuous time state space system by an additional state which describes the integral of the distance d. The expanded system then looks as follows:

$$A = \begin{bmatrix} 0 & v & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad D = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Bonus question (*optional*): Why don't you also account for the integral state of the angle?

Now discretize the above system as before and extend the state space matrices and the weighting matrices in your existing code in `controller-2.py`. Run it again with

```
$ roslaunch duckietown_demos lane_following_exercise.launch
veh:=DUCKIEBOT_NAME exercise_name:=2
```

How does your controller perform now?

Planning fundamentals

Part about planning.

Preliminaries

Preliminaries...

UNIT D-2
Learning materials

Learning materials

Exercises

Exercises

[1] Karl Johan Astroem and Richard M. Murray. Feedback Systems . http://www.cds.caltech.edu/~murray/books/AM08/pdf/am08-complete_22Feb09.pdf, 2009.

[4] Gioele Zardini. Control Systems II. https://n.ethz.ch/~gzardini/controlsystems2/Theory%20and%20Hints/FS18/Skript/Skript_Control_Systems_II_V1.pdf, 2018.

[5] Andrea Carron. Signals and Systems. https://ethz.ch/content/dam/ethz/special-interest/mavt/dynamic-systems-n-control/idsc-dam/Lectures/Signals-and-Systems/2019/notes/lecture1.pdf, 2019.