Classical Robotics Architectures using Duckietown
Contents

Part A - Perception fundamentals ................................................................. 3
  o Unit A-1 - Preliminaries ........................................................................ 4
  o Unit A-2 - Learning materials ............................................................... 5
  o Unit A-3 - Exercises ............................................................................ 6
  o Unit A-4 - Exercise: Augmented Reality .............................................. 7

Part B - Localization .................................................................................. 16
  o Unit B-1 - Preliminaries ........................................................................ 17
  o Unit B-2 - Learning materials ............................................................... 19
  o Unit B-3 - Exercises - lane pose estimation ........................................ 25

Part C - Modeling and control .................................................................. 27
  o Unit C-1 - Preliminaries ........................................................................ 28
  o Unit C-2 - Learning materials ............................................................... 29
  o Unit C-3 - Exercise: Control ................................................................. 30

Part D - Planning fundamentals ............................................................... 40
  o Unit D-1 - Preliminaries ........................................................................ 41
  o Unit D-2 - Learning materials ............................................................... 42
  o Unit D-3 - Exercises ............................................................................ 43
PART A
Perception fundamentals

Part about perception.
UNIT A-1

Preliminaries

Preliminaries...
UNIT A-2
Learning materials

Learning materials
UNIT A-3
Exercises
UNIT A-4
Exercise: Augmented Reality

The goal of this exercise is to familiarize yourself in developing functionalities in the framework of a pre-existing pipeline. In particular the focus is in the perception pipeline, where you will implement a computer graphics algorithm.

**KNOWLEDGE AND ACTIVITY GRAPH**

- **Requires:** Camera calibration
- **Requires:** Docker basics
- **Requires:** ROS basics
- **Requires:** Knowledge of the software architecture on a Duckiebot
- **Results:** Skills on how to develop new code as part of the Duckietown framework
- **Results:** Insights into a computer graphics pipeline.

### 4.1. Introduction

During lectures, we explained one direction of the image pipeline:

![Image pipeline](image.png)

**Figure 4.1**

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:

![Inverse pipeline](inverse.png)

**Figure 4.2**

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

### 4.2. Instructions

- Ensure that you have already done intrinsics and extrinsics camera calibration of your robot.
- Create a package called `augmented_reality` with functionalities specified below in Section 4.3 - Specification of `augmented_reality`. 
Then verify the results in the following 3 situations.

1) Situation 1: Calibration pattern
   - Put the robot in the middle of the calibration pattern.
   - Run the node `augmented_reality` with map file `calibration_pattern.yaml`.
   (Adjust the position of your Duckiebot until you get a decent match of reality and augmented reality.)

2) Situation 2: Lane
   - Put the robot in the middle of a lane.
   - Run the node `augmented_reality` with map file `lane.yaml`.
   (Adjust the position of your Duckiebot until you get a decent match of reality and augmented reality.)

3) Situation 3: Intersection
   - Put the robot at a stop line at a 4-way intersection in Duckietown.
   - Run the node `augmented_reality` with map file `intersection_4way.yaml`.
   (Adjust the position of your Duckiebot until you get a decent match of reality and augmented reality.)

4.3. Specification of `augmented_reality`

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

```
$ roslaunch augmented_reality augmented_reality.launch map_file:=map file robot_name:=robot name
```

where `map file` is a YAML file containing the map (specified in Section 4.4 - Specification of the map).

The package structure must be the one provided by the Duckietown template-ros, in addition, create a map folder 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, using the map file given in the roslaunch command.
3. Listen to the image topic `/robot name/camera_node/image/compressed`.
4. Read each image, project the map features onto the image, and then write the resulting image to the topic `/robot name/node_name/map file basename/image/compressed`

where `map file basename` is the basename of the file without the extension.

Create a ROS node called `augmented_reality_node`, which imports an `Augmenter`
class, from an augmented_reality module. The 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 adds the segments from the map files to the image.

In the ROS node, you just need a callback that uses the above specified class to modify the input image, so:

1. Implement a method called `callback` that writes the augmented image to the appropriate topic.

### 4.4. Specification of the map

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

The format is similar to any data structure for 3D computer graphics, with a few changes:

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

Here is an example of the file contents, hopefully self-explanatory.

The following map file describes 3 points, and two lines.

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

1) Reference frame specification

The reference frame specifications are defined as follows:

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

(Other image frames will be introduced later, such as the world and tile reference frame, which need the knowledge of the location of the robot.)
2) Color specification

RGB colors are written as:

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

where the RGB values are between 0 and 1.

Moreover, we support the following strings:

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

4.5. “Map” files

1) hud.yaml

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

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

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

2) calibration_pattern.yaml

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

We want something like this:

```yaml
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
```
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:
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]
4.6. 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
```

For other functionalities (i.e. loading calibration files), it could make sense to spend some time in looking into the existing Duckietown code.
PART B
Localization

Contents

Unit B-1 - Preliminaries .................................................................17
Unit B-2 - Learning materials........................................................19
Unit B-3 - Exercises - lane pose estimation..................................25
UNIT B-1
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.

![rviz screenshot](image)

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:
then:

```bash
$ docker run -it --net=host -e VEHICLE_NAME=DUCKIEBOT_HOSTNAME --env="DISPLAY" --volume="$HOME/.Xauthority:/root/.Xauthority:rw" duckietown/rviz-cra2:daffy-amd64 /bin/bash
```

and also:

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

finally we can launch the application:

```bash
$ 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 rosparsams

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

```bash
$ dts start_gui_tools DUCKIEBOT_NAME
```

1) Listing the parameters:

```bash
$ rospam list
```

2) Getting the parameters:

```bash
$ rospam get PARAMETER_NAME
```

3) Setting the parameters:

```bash
$ rospam set PARAMETER_NAME VALUE
```
UNIT B-2
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.
  - $\phi$ (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 $(d, \phi)$ 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 $(d, \phi)$ is represented as a matrix, holding $d$ on one axis and $\phi$ on the other. This means that the space of $(d, \phi)$ 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 $\phi$ 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 \((d, \phi)\) tuple is extracted from geometric knowledge of the lane. Each segment’s extracted tuple \((d, \phi)\) 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 \((d, \phi)\).

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

    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.
UNIT B-3
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 |
| Requires: Docker basics |
| Requires: ROS basics |
| Requires: Knowledge of the software architecture on a Duckiebot |
| 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 Unit C-3 - Working with logs.

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, plotting on the y-axis \((d, \phi)\) 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, \(\text{lane\_filter\_node}\) 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 \(/\text{DUCKIEBOT\_NAME}/\text{lane\_filter\_node/matrix\_mesh\_size}\) can be dynamically adjusted.

**Exercise 3. Choosing the best matrix size.**

While running the exercise-provided lane following, play with \(\text{matrix\_mesh\_size}\), and record different rosbags for the topic lane_pose (one for each value of \(\text{matrix\_mesh\_size}\)).

Write a custom Python script to analyze the frequency of the topic \(/\text{DUCKIEBOT\_NAME}/\text{lane\_filter\_node/lane\_pose}\) for each bag (should be the same as last exercise). Plot the relationship between \(\text{matrix\_mesh\_size}\) on one axis and the the mean and standard deviation of the frequency of the \(\text{lane\_pose}\) 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 \(/\text{DUCKIEBOT\_NAME}/\text{lane\_filter\_node/lane\_offset}\) and the provided snippet provided code snippet is sufficient to complete this task. Coding is not necessary for this exercise.
PART C
Modeling and control

Part about modeling and control.
UNIT C-1

Preliminaries

Preliminaries...
UNIT C-2
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.

**KNOWLEDGE AND ACTIVITY GRAPH**

| Requires: Terminal Basics |
| Requires: Docker Basics |
| Requires: Control Theory Basics |
| Results: Ability to implement a controller on a real robot. |

**Contents**

Section 3.1 - Overview .................................................................30
Section 3.2 - PI control .................................................................30
Section 3.3 - Linear Quadratic Regulator ........................................36

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 approach represents 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 Fig. 3.2 one can derive a continuous-time nonlinear model for the Duckiebot. Considering the state $\dot{x}(t) = [d \ \varphi]^T$, one can write $\dot{x} = \begin{bmatrix} u \cdot \sin(\varphi) \\ \omega \end{bmatrix}$. 
After linearization around the operation point \( \bar{\mathbf{x}}_0 = [0 \ 0]^T \) (if you do not remember linearization, have a look at chapter 5.4 in [1]), one has
\[
\dot{\mathbf{x}}(t) = \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t).
\]
Furthermore, you are provided the output model
\[
y(t) = [6 \ 1] \dot{\mathbf{x}}(t).
\]
Using the linearized version of the model, you can compute the transfer function of the system (if you do not remember how, have a look at chapter 8 in [1])
\[
P(s) = \frac{s + 6v}{s^2}.
\]
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
\[
u(t) = k_p e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau = k_p e(t) + k_I \int_0^t e(\tau) d\tau
\]
In frequency domain, this corresponds to
\[
U(s) = C(s)E(s),
\]
with
\[
C(s) = k_p + \frac{k_I}{s}
\]

**Exercise 5. 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 \( 4.2 \text{ rad/s} \).

- Next, find an integral gain \( k_I \) such that \( L(s) \) has a gain margin of approximately \( -25.6 \text{ dB} \approx 0.053 \). (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 6. 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 10 - Implement a LQR onwards. Start by pulling the image and running the container with the following command:

\[
\begin{align*}
\text{docker} & \quad -H \quad \text{DUCKIEBOT NAME}.local \quad \text{run} \\
& \quad --name \quad \text{dt-core-CRA3} \quad -v \\
& \quad /data:/data \quad --privileged \quad --network=host \quad -it \quad \text{hosnerm/dt-core:CRA3-template} \\
& \quad /bin/bash
\end{align*}
\]

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://hostname.local:9000/#/containers)) and jump into it using the following command:

\[
\begin{align*}
\text{docker} & \quad -H \quad \text{DUCKIEBOT NAME}.local \quad \text{attach} \\
& \quad \text{dt-core-CRA3}
\end{align*}
\]

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 \( \omega \).

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 easily 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. TODO: ADD REFERENCE 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 \([d_{i+1}, \phi_{i+1}]\). If these values grow too abruptly, the camera loses sight of the lines and the estimation of the
output $y$ is not possible any longer. By increasing $k_s$ in controller-1.py, check how much you can reduce the sampling rate until 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 ($\#\text{measurement} \mod k_s \equiv 0$), 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 is 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 in. In the following exercise you will examine how the Duckiebot behaves if this delay between image acquisition and pose estimation changes.

**Exercise 7. 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: $P_d(s) = e^{-sT_d} P(s)$ 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 not destabilize 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 $k_d \ast T$ where $T$ is the sampling time. Again run your code with:
$\text{roslaunch duckietown_demos lane_following_exercise.launch}
\text{veh:=DUCKIEBOT\_NAME exercise\_name:=1}$

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

An 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.


In Fig. Figure 3.4, 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 $k_l$.

![Figure 3.4. A PI-controller with an anti-windup logic implemented, Feedback Systems from Aström and Murray, page 308.](image)

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 $u_{\text{sat}} = 2\text{rad/s}$.

Below you 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 men-
tioned saturation input value $u_{sat}$. Use it to extend your existing PI controller with an anti-windup scheme.

Furthermore you are given the parameter $k_i$ 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 Fig. Figure 3.4. As a first step, test the performance of the Duckiebot with the anti-windup term turned off (i.e., $k_i = 0$). 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 $k_i$ 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:**

```python
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 $\omega$ - 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

A linear quadratic regulator (LQR) is 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 9. 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:

\[
\dot{\tilde{x}} = A\tilde{x} + B u = \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix} \tilde{x} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \\
\tilde{y} = C\tilde{x} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tilde{x}
\]

With state vector \( \tilde{x} = [d, \, \varphi]^T \) 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 \( T_s \) using exact discretisation TODO: ADD REFERENCE and test your discretization using the provided Matlab-files. What do you observe? Add the found matrices in the template controller-2.py.

**Exercise 10. 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 Fig. 3.6:

![Block diagram of a state feedback control.](image)

Figure 3.6. 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).
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

$$u_{LQR}(t) = \arg\min_{u(t)} J_{LQR}(u(t)) = \arg\min_{u(t)} \int_0^\infty u^T R u + x^T Q x + 2x^T N u dt$$

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.

Last but not least, choosing $N=0$ is typical as it provides guarantees on performance and robustness.

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 different values. Can you describe it intuitively?

Exercise 11. 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? TODO: Missing the part where we explain that we are not dealing with a LQR, but with a LQ”G”. cioè
PART D

Planning fundamentals

Part about planning.
UNIT D-1
Preliminaries

Preliminaries...
UNIT D-2

Learning materials

Learning materials
UNIT D-3
Exercises

Exercises