Classical Robotics Architectures using Duckietown
Contents

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

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

Part C - Modeling and control ................................................................. 27
   ○ Unit C-1 - Preliminaries ....................................................................... 28
   ○ Unit C-2 - Learning materials ............................................................... 29
   ○ Unit C-3 - Exercises ........................................................................... 30

Part D - Planning fundamentals ............................................................... 31
   ○ Unit D-1 - Preliminaries ....................................................................... 32
   ○ Unit D-2 - Learning materials ............................................................... 33
   ○ Unit D-3 - Exercises ........................................................................... 34
PART A
Perception fundamentals

Part about perception.
UNIT A-1
Preliminaries

Preliminaries...
UNIT A-2
Learning materials

Learning materials
UNIT A-3

Exercises

Exercises
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:

<table>
<thead>
<tr>
<th>Image</th>
<th>Feature Extraction</th>
<th>2-D Features</th>
<th>Ground Projection</th>
<th>3-D World Coordinates</th>
</tr>
</thead>
</table>

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:

<table>
<thead>
<tr>
<th>3-D World Coordinates</th>
<th>Image Projection</th>
<th>2-D Features</th>
<th>Rendering</th>
<th>Image</th>
</tr>
</thead>
</table>

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`. 
Exercise: 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.

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

1) Reference frame specification

The reference frames are defined as follows:

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

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

RGB colors are written as:

\[[\text{rgb}, [R, G, B]]\]

where the RGB values are between 0 and 1.

Moreover, we support the following strings:

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

4.5. “Map” files

1) `hud.yaml`

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

```yaml
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.

Figure 4.3

3) `lane.yaml`

We want something like this:
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 4.4

4) intersection_4way.yaml
Exercise: Augmented Reality

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.

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-amd64 /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
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.
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 maximum number of segments 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 2 - 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 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.lwidth_white
        else:  # left edge of white lane
            d_i = -d_i
            phi_i = -phi_i
            d_i = d_i - self.lwidth
    elif segment.color == segment.YELLOW:  # left lane is yellow
        if (p2[0] > p1[0]):  # left edge of yellow lane
            d_i = d_i - self.lwidth_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 maximum number of segments.**

While running the exercise-provided lane following, play with `/DUCKIEBOT_NAME/line_detector_node/segment_max_threshold`, and record different ROS bags (one 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 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` 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, estimate the stability of the estimation and find out the minimal number for `segment_max_threshold` that keeps a stable estimation.
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 2. Choosing the best matrix size.**

While running the exercise-provided lane following, play with `matrix_mesh_size`, and record different ROS bags (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.

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 3. Driving the English style.**

The task is to make the Duckiebot drive on the left side of the road. An appropriate parameter in the provided code snippet is sufficient to complete this task.
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

Exercises
PART D
Planning fundamentals

Part about planning.
 UNIT D-1

Preliminaries

Preliminaries...
UNIT D-2
Learning materials

Learning materials
UNIT D-3

Exercises