

# Robotics Project I

Marco B. GABRIEL, Group 12

May 24, 2021

IN.2022 Robotics 2021, BSc Course, 2nd Sem.

University of Fribourg

marco.gabriel@unifr.ch

## Abstract

This paper explores the possibilities and limitations of e-puck robots and their sensors for different behaviours.

To gather data, we conducted multiple experiments on different e-pucks in various situations, including sensing obstacles using the proximity sensors, following a black line, taking images with the camera, communicating between robots, and using the microphone to detect from which direction a sound is originating from.

By using the mentioned sensors we were able to successfully implement multiple variations of Braitenberg behaviours, color recognition and even determine the direction from which a sound is originating from, provided the sound source is close to the robot.

We conclude that the e-pucks are capable to be used for complex behaviors, although the sensors do pose some limitations to behaviors that rely on accurate sensory data like using the microphones to determine the position of a sound source.

**Keywords:** e-puck, robot, Braitenberg, behaviour, infra-red sensor, camera, ground sensor, microphone, line-following, wall-following, PID controller, LOVER, EXPLORER, alternating LOVER, simultaneous search, alternative search, communication, robots, color recognition, robust, vehicle, proximity sensor

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                            | <b>2</b>  |
| <b>2</b> | <b>Sensors</b>                                 | <b>3</b>  |
| 2.1      | Proximity infra-red sensors . . . . .          | 3         |
| 2.2      | Infra-red ground sensor . . . . .              | 4         |
| 2.3      | Camera . . . . .                               | 6         |
| 2.4      | Additional sensor analysis . . . . .           | 10        |
| <b>3</b> | <b>Behaviours</b>                              | <b>13</b> |
| 3.1      | Braitenberg vehicle . . . . .                  | 13        |
| 3.1.1    | LOVER . . . . .                                | 13        |
| 3.1.2    | EXPLORER . . . . .                             | 14        |
| 3.1.3    | LOVER/EXPLORER Finite State Machine . . . . .  | 14        |
| 3.2      | Line-following . . . . .                       | 16        |
| 3.2.1    | Simple Braitenberg line-follower . . . . .     | 16        |
| 3.2.2    | Robust non Braitenberg line-follower . . . . . | 17        |
| 3.2.3    | Robust Braitenberg line-follower . . . . .     | 18        |
| 3.3      | Wall-following . . . . .                       | 18        |
| 3.4      | Color recognition . . . . .                    | 20        |
| 3.5      | Multi-robot coordination . . . . .             | 21        |
| <b>4</b> | <b>Conclusion</b>                              | <b>23</b> |
|          | <b>Appendix</b>                                | <b>25</b> |
|          | Appendix A Experimental Results . . . . .      | 25        |
|          | Appendix B Source Code . . . . .               | 31        |

# Chapter 1

## Introduction

Robots are machines, that use sensors to get information about their environment and algorithms to respond autonomously to the environment in a mechanical manner. They have been used for research and to automate work for decades, and are still used today, and probably also in the future.

During the Robotics Course, we deal with the same topics that are also important in actual robotics research and in the industry. We get to work with different sensors of the e-puck robots and implement algorithms which use the gathered information about the environment to act accordingly without the need for human intervention. The goal of this project is mainly to prepare us for the second project, which constitutes the final exam.

Our work covers the proximity sensors, with which we implemented the LOVER and EXPLORER Braitenberg behaviours that succeeded in moving towards and moving away from objects respectively in a robust and efficient manner. Combining the two behaviours allowed us to create an alternating LOVER behaviour, which switches to EXPLORER once it reached an equilibrium distance from an object. After conducting experiments using the e-puck's ground sensors, we were able to allow the e-puck to recognise black lines on the floor and follow them by using modified Braitenberg and non-Braitenberg behaviours.

We were introduced to the concept of PID controllers and implemented a wall-following behavior using one. At the end, we worked with the camera and microphone of the e-puck and tested multi-robot communication. We succeeded to differentiate different colored objects in the e-puck's field of view and to some degree determine the location of a sound source. Using communication, we created an algorithm which controls multiple robots with the same controller and coordinates the robots running a modified version of alternating LOVER to either act simultaneously or alternating.

This paper covers first the workings of the different sensors and explains the implementations of the different behaviours in the second half.

## Chapter 2

# Sensors

### 2.1 Proximity infra-red sensors

According to our experiments, there are two major differences between the proximity graphs in reality (Figure 2.1) and in Webots (Figure 2.2). Firstly, the real-world data contains a considerable amount of noise, whereas the sensor data in Webots is practically noise free. This is apparent when considering the differences between the measurements of the right and left sensors. In theory, they should be the same value, but approximately 10% of measurements are clearly an error, since there are differences of more than 20% between the measurements of the right and left sensor.

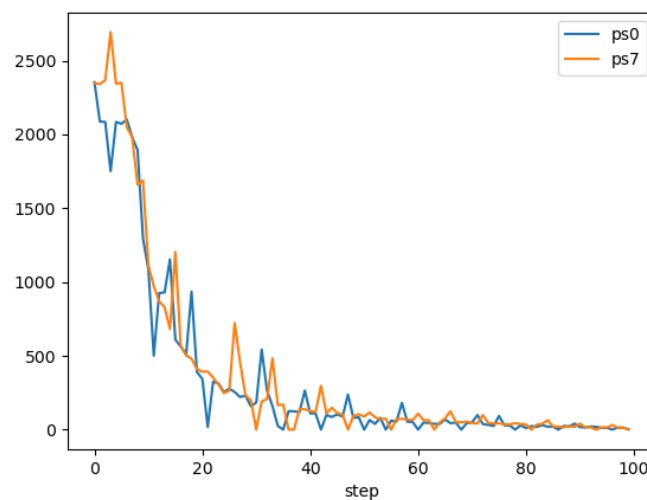


Figure 2.1: Proximity sensors in reality

Secondly, there is a difference in absolute values between the real and the simulated world. The highest possible measurement (e.g. the shortest measurable distance) is 2400 in the real world, but only 700 in Webots. The trend of the real world having a higher value continues to larger distances.

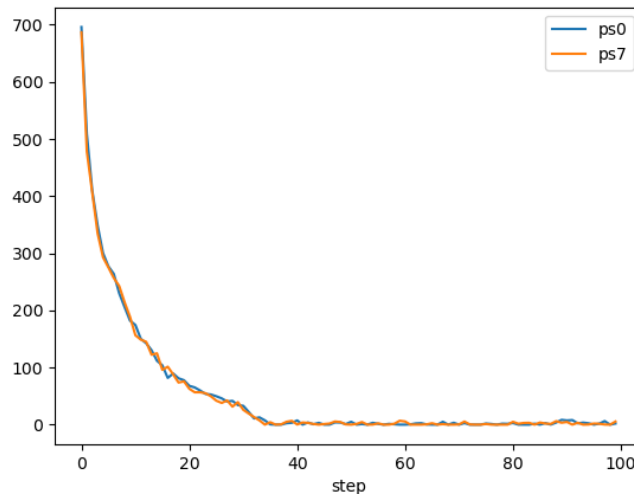


Figure 2.2: Proximity sensors in Webots

We conducted these measurements once in Webots and once in reality. For the measurements in reality, we placed robot 4211 in the middle of our robot arena. After making sure that there were no obstacles close to the robot, we started our program on the laptop controlling the robot. As soon as the robot finished calibrating its sensors, we placed an obstacle right in front of it. The robot then proceeded to drive backwards, away from the obstacle, while obtaining the sensor measurements.

For the experiment in Webots, we used the same procedure as in reality.

## 2.2 Infra-red ground sensor

The setup for the ground sensor response experiment was the following: A paper with a black line lies in the middle of the robot arena, robot 4207 is placed approximately 10cm away from the line, facing it under an angle of either 90 or 45 degrees (the robots left side being closer to the line at 45 degrees). We then started the program which makes the robot drive forwards in a straight line while collecting data from the ground sensors. We stopped the program some seconds after the robot drove over the black line.

The result of the perpendicular movement, visualised in Figure 2.3, shows that the three sensors do not detect the line at exact the same moment, but with up to 10 steps difference. This is probably due to the robot not exactly moving in a straight line and us not placing it at a 90 degree angle perfectly.

Additionally, the different sensors measure different brightness values when encountering the line. The left sensor (gs0) records the highest brightness with about 400, followed by the right sensor (gs2) and the center sensor with values of 300 and 250 respectively. We suspect two reasons for this difference: Firstly, unlike the proximity sensors, the ground sensors are not

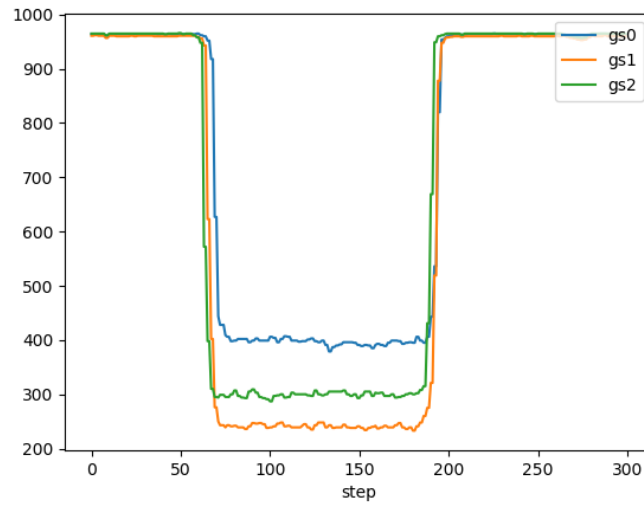


Figure 2.3: Ground sensor measurements when crossing a line perpendicularly

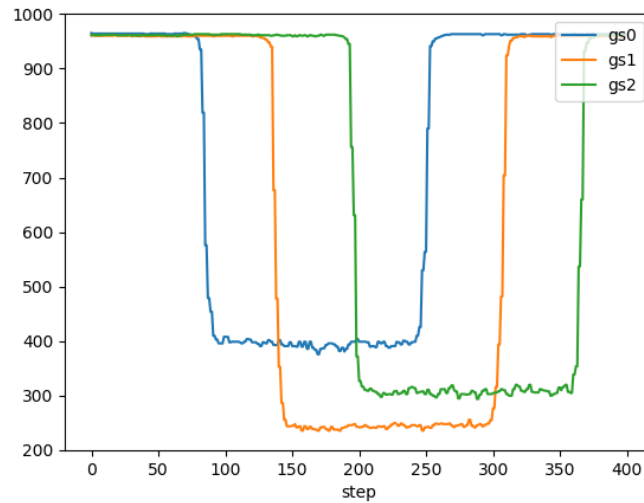


Figure 2.4: Ground sensor measurements when crossing a line diagonally

calibrated at the beginning, so this could lead to different values being measured. Secondly, There could have been light pollution from the sunlight entering through the windows. The sunlight would illuminate one side of the robot much brighter than the other, causing the left and right sensors to read different values. The center sensor is shadowed most by the robot itself, which could explain why the center sensor reads the lowest brightness. However, further research is needed before we can state a justified reason.

Furthermore, there is a small amount of noise in the data when the sensors are not above a white surface. The noise is not exceeding an absolute change in value of plus or minus 20.

Regarding the measurements of a diagonal movement over the line, visualised in Figure 2.4, there is a clear and consistent delay between the different sensors detecting the line. This is to

be expected, since the robot encounters the line at an angle of 45 degrees. The left sensor (ds0) is the first to detect the line, followed by the center (ds1) and right sensor (ds2).

The brightness values are similar to the values of the former experiment. The left sensor records again the highest, whereas the center sensor records the lowest brightness value. The reasons are probably the same as discussed in the previous experiment, even though the robot was oriented 45 degrees differently.

## 2.3 Camera

We conducted the Camera measurements by placing robot number 210 in the center of the robot arena and placing different objects into it's field of view. These objects are red, green and blue blocks and another e-puck. Lastly, we let the robot record the white arena walls. To analyse the recorded data, we plotted it as histograms, which show the number of pixels with a given brightness value for each color.

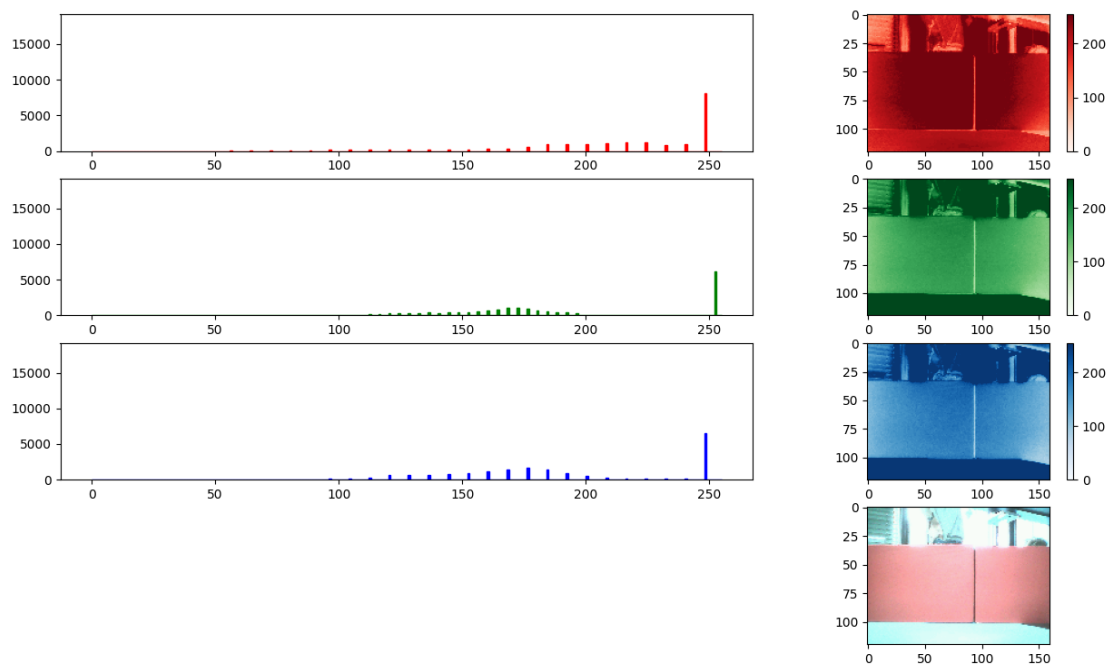


Figure 2.5: Histogram of an image of a red object

Regarding the histograms of the colored blocks, I conclude that the object's color manifests in the histograms in a consistent manner. The distribution of brightness values of the color channel corresponding to the object's color is significantly higher (e.g. shifted to the right) compared to the other color channels. This is especially apparent in Figure 2.5 and Figure 4.1 (in Appendix) which represent the images of a red and blue object respectively. The same effect can be seen for the green object (Figure 2.6), but it is less obvious for this color. The reason for this difference is, that the green channel has a resolution of 6 bits, whereas the other channels only have 5 bits. This reasoning is based on a code excerpt of the `unifr_api_epuck` library which



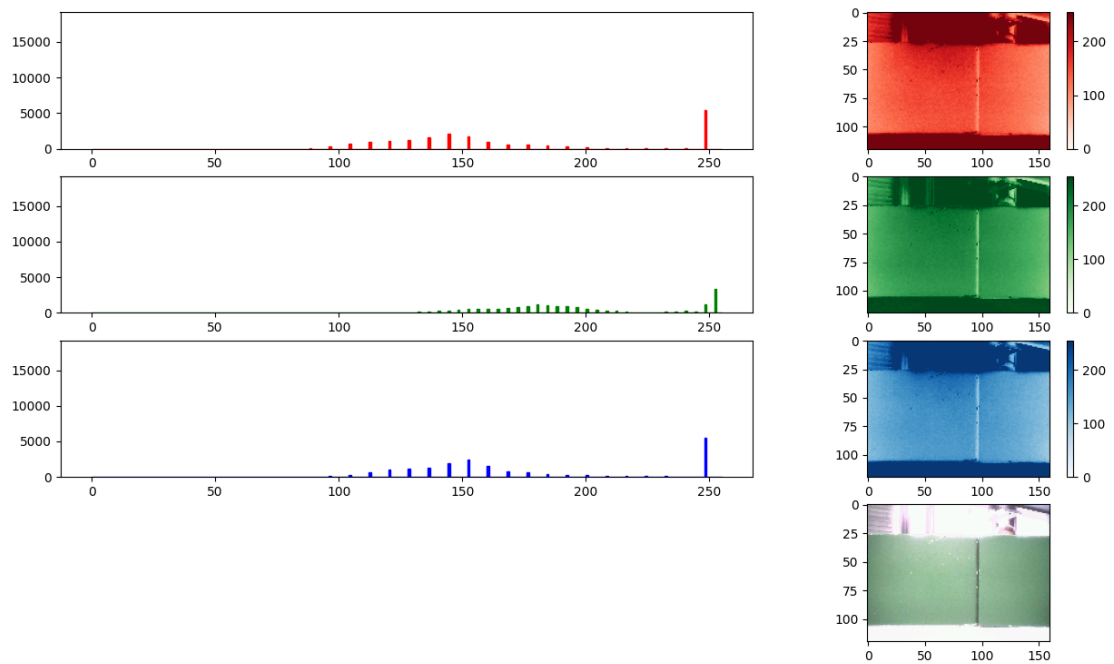


Figure 2.6: Histogram of an image of a green object

can be seen in Figure 4.7 in the Appendix. This fact leads to the graph having less height, because there are more independent bars. The spike at the end of the scale is caused by the white floor of the arena and the bright background above the colored objects.

The histogram of a white arena wall (Figure 2.7) looks symmetrical across the different color channels. The majority of pixels has a value at about 255, which is the maximum value it can get, because it is stored in an 8 bit integer. This indicates that the image is overexposed by the camera, which is actually a bad sign, since this could limit the robots ability to distinguish colors in very bright environments or bright colors, because there is not enough contrast in the data.

Analysing the histogram of the image of another e-puck (Figure 2.8) leads to an interesting finding. The image contains different colors with different brightnesses, caused by the different colors and materials of the e-puck. For that reason, the graphs look flat and nicely distributed, with the exception of a peak at the high end, which is caused by the bright white ring on top of the e-puck.

To remove the spike at the high end of the scale, a letterbox function can be used, which only considers the pixels that are inside a broad, but narrow rectangle located in the middle of the image. The resulting graphs are much cleaner and make it easier to recognise colors. Two examples for this are the plots of the red (Figure 2.9) and the blue letterbox (Figure 2.10). The letterbox containing the red object still features pixels with the maximum value, whereas the letterbox of the blue object does not. This might be an indication that the red color is more vibrant/exact than the blue color and therefore resonates better with the cameras sensor, but this is just a detail.

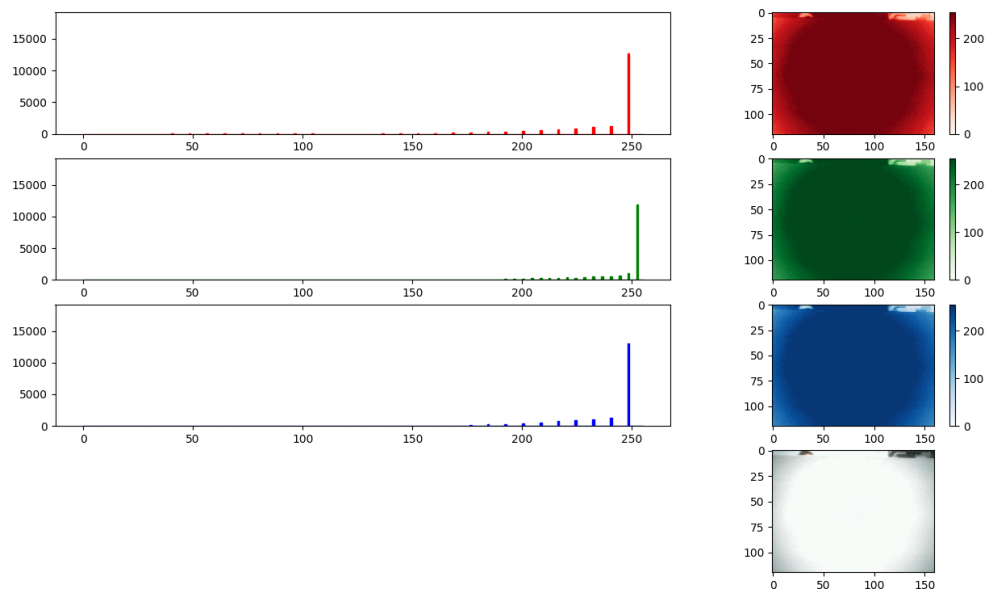


Figure 2.7: Histogram of an image of the arena wall

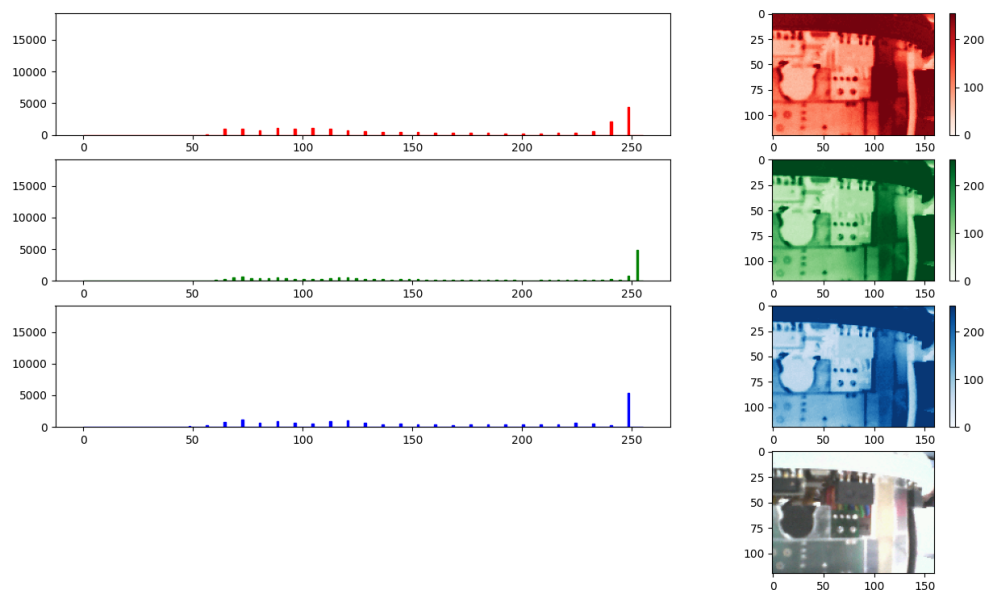


Figure 2.8: Histogram of an image of a another e-puck

The influence of the letterbox on the images of the arena wall and the e-puck are negligible, but can be seen in the Appendix in Figure 4.2 and 4.3 respectively.

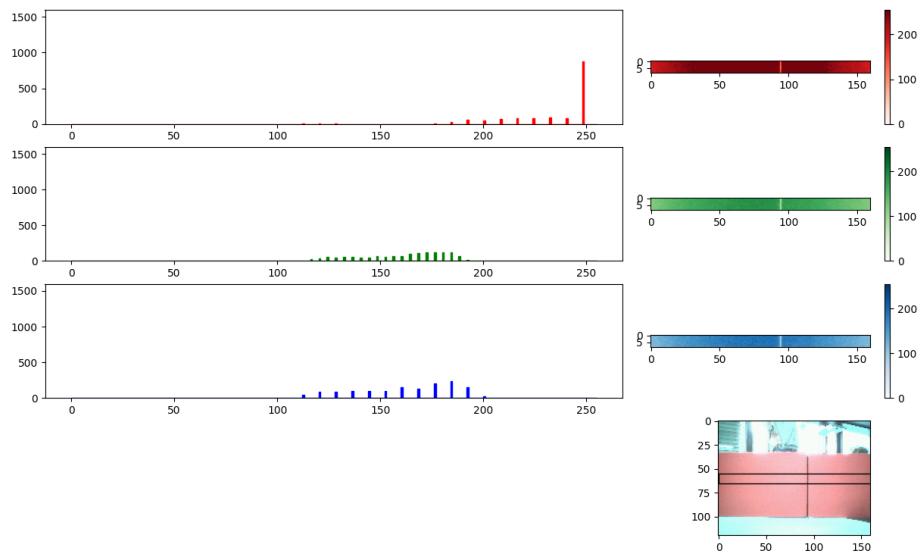


Figure 2.9: Histogram of the letterbox of an image of a red object

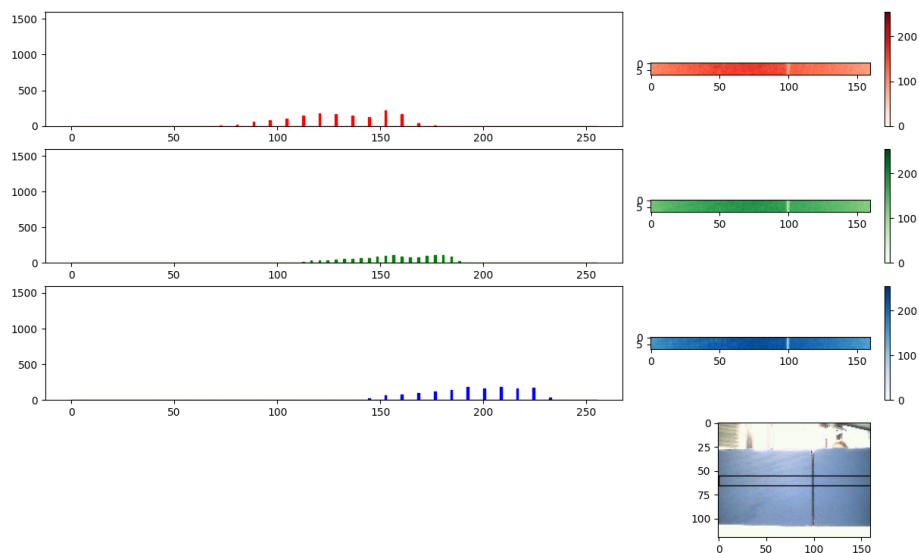


Figure 2.10: Histogram of the letterbox of an image of a blue object

## 2.4 Additional sensor analysis

The additional sensor of our choice is the microphone. Since the e-puck is equipped with 4 microphones, one on each side (mic0 = front, mic1 = right, mic2 = back, mic3 = left), we wanted to test if the e-puck can detect the location of a sound source. The microphones only measure the amplitude, but do not return any data on the pitch of the measured sound. The microphones do not return data with a high enough frequency to calculate the location of a sound source by using the delay between the exact moments when the microphones record the noise, caused by the limited speed of sound. So the only way to calculate the location of a sound source is to use differences in the measured amplitude.

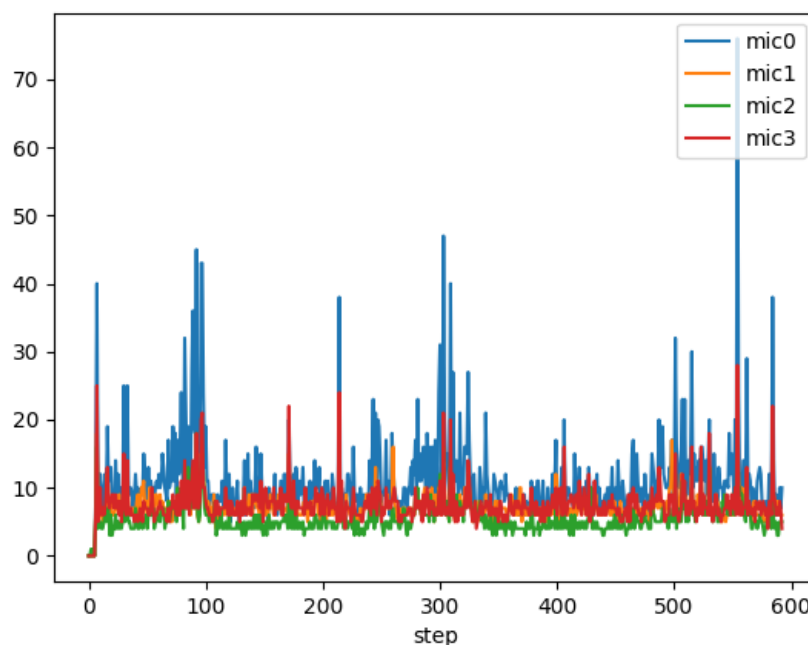


Figure 2.11: Plot of the amplitude of the different microphones when the robot is driving

To find an answer to our question, we placed robot 211 on the robot arena and let it turn around itself clockwise, while a smartphone plays a sine wave of 600Hz at maximum volume. This way, the distance between the e-puck and the sound source remains constant and does therefore not influence the measured amplitude. We conducted measurements with different distances between the e-puck and the sound source. The results are shown in Figure 2.12 and 2.13. We also made a plot when only the robot was driving, so we have an understanding of how much noise the e-puck creates itself, which can be seen in Figure 2.11.

Comparing Figure 2.11 and 2.12, I conclude that the noise made by the robot itself is not loud enough to interfere with the microphone measurements, because the robot's noise is mostly below 20, whereas the sound source creates values of 20 and above consistently, even if located 40cm away. Additionally, it is apparent that the front microphone (mic0) records much higher values than the other microphones. One explanation might be that the front microphone is mounted a bit lower than the others, so it is closer to the robot's noise. Yet, since the same microphone records the highest values also when the sound source is turned on, I suspect that the reason for the different values is that the microphones are not calibrated.

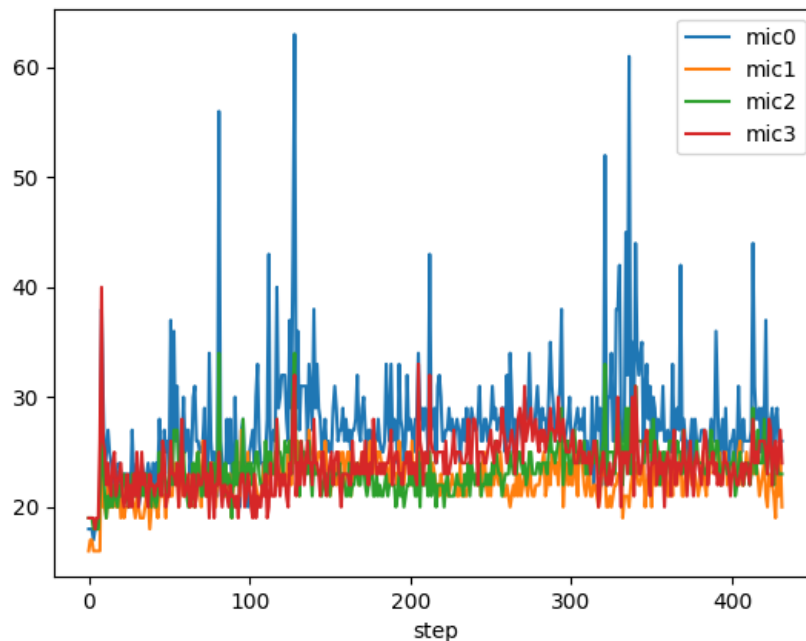


Figure 2.12: Plot of the amplitude of the different microphones when the sound source is far away from the e-puck (40cm away)

Regarding the graph of a far away sound source in Figure 2.12, it is not feasible to determine the sound source's location if the sound source is far away. The difference between the various microphones is too small and noise too big. However, looking at Figure 2.13 which shows the data when the sound source is close by (10cm), an interesting pattern shows. Because of the robot's turning movement, the graph of each microphone results in a wave, being highest when the corresponding microphone points in the direction of the sound source and lowest when the microphone is on the opposing side. So at every given moment, the approximate location of the sound source can be found out by taking the microphone with the highest reported value. In this case, we can conclude that the sound source was in front of the robot in the beginning, then on the left, back, right, and in front of the robot again. This conclusion is coherent with our experiment's setup.

The reason why it is not possible to determine the sound source's location if it is relatively far away is probably the nature of how sound works. The intensity of sound decreases with the inverse square of the distance. The formula is  $I = S/4\pi r^2$ , where  $S$  is the source intensity and  $I$  the intensity at a point  $r$  away from the source. This property is visualised in Figure 2.14 in the Appendix. Now it is apparent, that the difference between two microphones is bigger (e.g. the curve is steeper), when the sound source is closer. This means it is easier to detect a difference between two microphones if the sound source is close, and harder when the source is further away.

These results mean, that we can use the microphones to detect the location of sound sources only if the sound sources are relatively close (less than 20cm) to the robot. Additionally, we can of course use the microphones to detect the presence of sound in general, even if the sound source is far away.

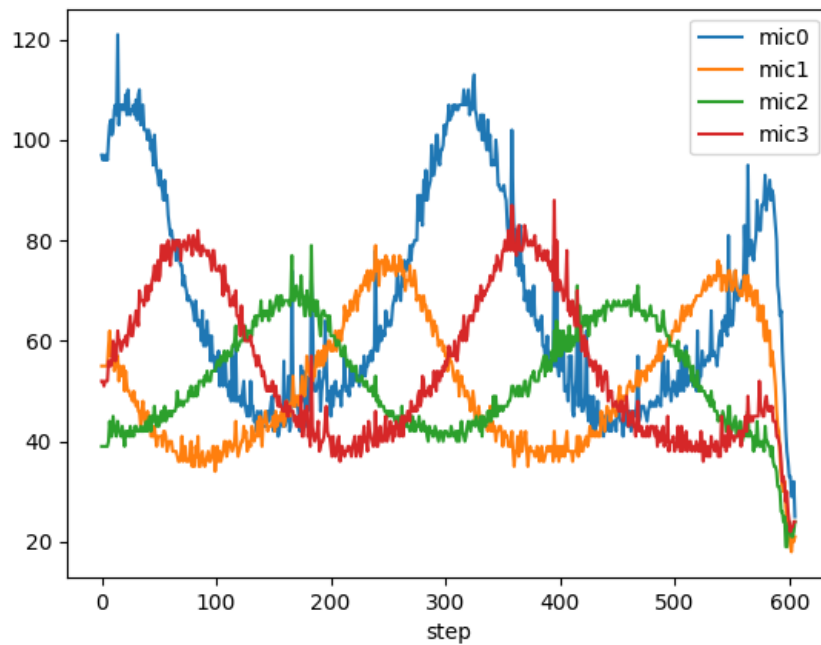


Figure 2.13: Plot of the amplitude of the different microphones when the sound source is close to the e-puck (10cm away)

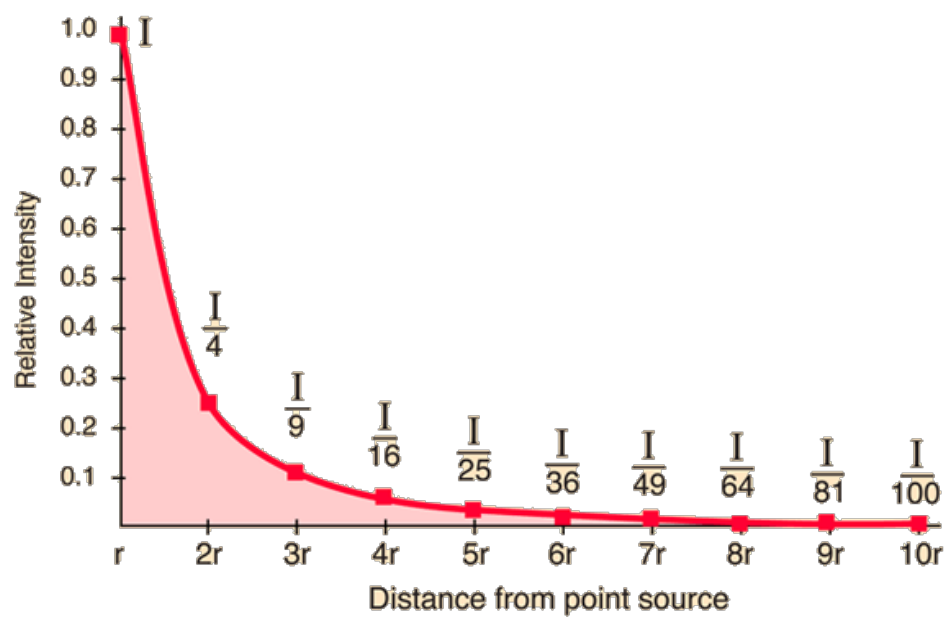


Figure 2.14: Graph showing the relative sound intensity by distance [1]

# Chapter 3

## Behaviours

### 3.1 Braitenberg vehicle

#### 3.1.1 LOVER

The formula for the left wheel's speed is shown in Figure 3.1. The formula for the right wheel is the same, except for using the other sensors.

```
1 NORM_SPEED = 3 #Max speed
2 MAX_PROX = 150 #Closest distance
3
4 prox_left = (d * prox_values[4] + c * prox_values[5] + b * prox_values[6] + a * prox_values[7]) / (a + b + c + d)
5 inv_speed_left = (NORM_SPEED * prox_left) / MAX_PROX
6 speed_left = NORM_SPEED - inv_speed_left
```

Figure 3.1: Formula for the left wheel's speed of Lover

The formula first calculates the weighted average of all proximity sensors of the same side that the wheel is on, at line 4. Using this average, the *inverse speed* is calculated by multiplying the max speed with the ratio of the *average proximity* over the *closest distance*. This means, the *inverse speed* is equal to the maximum speed, if there is an object as close as it should get, and close to zero if there is no object, or it is far away. The final speed is then calculated by subtracting the *inverse speed* from the max speed.

*To sum up, the wheels turn slower the closer an object is, which leads to the robot making turns towards objects and stopping once it is close enough.*

The weights in Figure 3.2 for the proximity formula resulted in the best behaviour.

```
1 a = 5 #forwards
2 b = 20 #diagonal forwards
3 c = 50 #side
4 d = 10 #backwards
```

Figure 3.2: Weights for the proximity sensors of Lover

I chose a relatively small value for  $a$  (the front), because the robot does not have to do a tight turn to reach an obstacle in front of it, so the impact on steering should be minimal.

For  $b$  and especially for  $c$ , I chose high values, since in order for the robot to reach an object located 90 degrees to it, the robot has to perform a tight turn. Otherwise it just drives straight and misses the object. To achieve this tight turn, the value needs to be heavily weighted.

Since the  $d$  sensors point to the back, I consider them not as useful as the others. The object detected at the back is either an object where the robot has already been (I consider driving back to the same object as an uninteresting behaviour), or is practically unreachable. The object is unreachable, because there is a blind spot between the sensors  $c$  and  $d$ . After sensing an object at  $d$ , the robot would initiate a turn which leads to the object being in the blind spot between  $c$  and  $d$ . This would lead to the robot driving away of the object, or never performing a turn tight enough to reach the object. This is why I chose a small value for  $d$ .

### 3.1.2 EXPLORER

The formula for the right wheel's speed is shown in Figure 3.3. The formula for the left wheel is the same, except for using the other sensors.

```

2 NORM_SPEED = 3 #Max speed
  MAX_PROX = 250 #Closest distance

4 proxR = (d*prox_values[4] + c*prox_values[5] + b*prox_values[6] + a*prox_values[7]) / (a+b+c+d)
  dsR = (NORM_SPEED * proxR) / MAX_PROX #inverse speed
6 speedR = NORM_SPEED - dsR

```

Figure 3.3: Formula for the right wheel's speed of Explorer

The formula first calculates the weighted average of all proximity sensors of the **opposite side** of the driven wheel on line 4. Using this average, the *inverse speed* is calculated by multiplying the max speed with the ratio of measured *average proximity* over the *closest distance*. This means, the *inverse speed* is equal to the maximum speed, if there is an object on the opposite side, and close to zero if there is no object on the opposite side. The final speed is then calculated by subtracting the *inverse speed* from the max speed.

*To sum up, the wheels turn slower, the closer an object on the **opposite side** of the robot is, which leads to the robot making turns away from objects.*

I achieved the best behaviour with the weights in Figure 3.4 for the proximity formula.

```

2 a = 4 #forwards
  b = 2 #diagonal forwards
  c = 1 #side
4 d = 1 #backwards

```

Figure 3.4: Weights for the proximity sensors of Explorer

I chose big values for the forward facing sensors such as  $a$  and  $b$ , because if there is an obstacle in front of the robot, it needs to react strongly to avoid a collision.

The side and rear facing sensors,  $c$  and  $d$ , do not seem important to me, since the robot is unlikely to drive into objects next to or behind it. Following this reasoning, I decided to weigh  $c$  and  $d$  less than  $a$  and  $b$ .

### 3.1.3 LOVER/EXPLORER Finite State Machine

As an overview of how the alternating lover works, Figure 3.5 depicts its behaviour as a Finite State Machine.

Detecting the equilibrium is done by checking whether the average of the four proximity sensors at the front is bigger than a specified value. In Figure 3.6,  $pv$  is the average of the proximity sensors, and  $EQ\_DIST$  is the specified threshold.



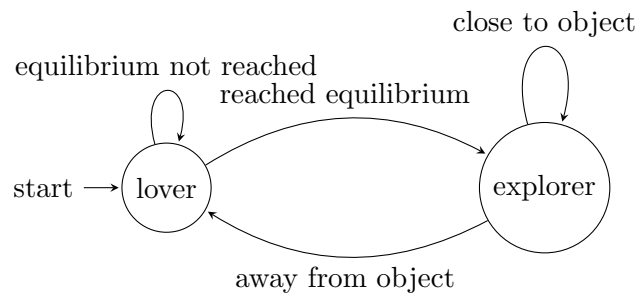


Figure 3.5: AFSM of alternating Lover

```

EQ_DIST = 80
2
3 def is_equilibrium(prox_values):
4     pv = (prox_values[0]+prox_values[7]+prox_values[6]+prox_values[1])/4
5     return EQ_DIST <= pv

```

Figure 3.6: Function to detect equilibrium

This function might fail if the object is too small, so the threshold is never reached, because not enough sensors are close enough to the object. Possible solutions might be to accept the situation as an equilibrium if only one sensor measures a high enough value, or we could introduce a timing function which detects long periods without changing proximity values and counts them as equilibria.

To fully implement alternating Lover, there is also the need for a function to detect when to switch back from the Explorer to the Lover state. We used the `is_away()` function shown in Figure 3.7. It returns `True` if the four proximity sensors at the front do not sense any obstacles. `CLEAR_DIST` should be set to as close to 0 as possible, but because of noise in the proximity data, I found 0.5 to work best.

```

1 CLEAR_DIST = 0.5
2
3 def is_away(prox_values):
4     pv = (prox_values[0]+prox_values[7]+prox_values[6]+prox_values[1])/4
5     return CLEAR_DIST >= pv

```

Figure 3.7: Function to detect when to switch back to Lover

The main loop of our implementation of alternating Lover is shown in Figure 3.8. At the start of the loop (line 3-12), the weights for the proximity sensors are declared depending on the current state (Lover or Explorer).

Next, the formulas for calculating the speeds are executed (line 14-20).

Then, depending on the current state, the LEDs are turned on or off, and the speeds of the motors are set either coupled linearly or cross coupled.

At the end, it is checked whether the state should be switched, e.g. whether the equilibrium is reached or the robot is far enough from any objects by using the previously mentioned functions `is_equilibrium()` and `is_away()`

```

1 while r.go_on():
2     prox_values = r.get_calibrate_prox()
3     if lover:
4         a = 1
5         b = 1
6         c = 2
7         d = 1
8     else:
9         a = 4
10        b = 2
11        c = 1
12        d = 1
13
14    proxR = (a*prox_values[0] + b*prox_values[1] + c*prox_values[2] + d*prox_values[3]) / (a+b+c+d)
15    dsR = (NORM_SPEED * proxR) / MAX_PROX
16    speedR = NORM_SPEED - dsR
17
18    proxL = (d*prox_values[4] + c*prox_values[5] + b*prox_values[6] + a*prox_values[7]) / (a+b+c+d)
19    dsL = (NORM_SPEED * proxL) / MAX_PROX
20    speedL = NORM_SPEED - dsL
21
22    if lover:
23        #for lover implementation --> linear coupling
24        r.enable_all_led()
25        r.set_speed(speedL, speedR)
26        #detect equilibrium
27        if is_equilibrium(prox_values):
28            lover = False
29
30    else:
31        #for explorer implementation --> cross coupling
32        r.disable_all_led()
33        r.set_speed(speedR, speedL)
34        if is_away(prox_values):
35            lover = True

```

Figure 3.8: Main loop of alternating Lover

We provide a video of our first alternating lover implementation: [Link to the video demonstration of alternating Lover with LEDs \[2\]](#)

Notably: The program that we were running while filming included a wrong implementation of Explorer. So that part of the video is not coherent with our current code, but the Lover part and the LEDs worked as intended.

## 3.2 Line-following

### 3.2.1 Simple Braitenberg line-follower

As a simple Braitenberg line-following implementation, we adapted the LOVER code to use the ground sensors instead of the proximity sensors. Additionally, we had to remove the line where it inverts the speed, because the sensor values have a different meaning. A high proximity value means that an object is close, but a high brightness value means that there is no black line. These changes are visualised in Figure 3.9.

```

1 while r.go_on():
2     ground_values = r.get_ground()
3     speed_right = NORM_SPEED * ground_values[GS_RIGHT] / MAX_VALUE
4     speed_left = NORM_SPEED * ground_values[GS_LEFT] / MAX_VALUE
5     r.set_speed(speed_left, speed_right)

```

Figure 3.9: LOVER implementation for line-following

However, this implementation fails on the rectangular line. When reaching a corner, the robot loses sight of the line and does not recognise the corner. I believe that this implementation would work, if the line were thinner, so the line would be in between the two sensors.

### 3.2.2 Robust non Braitenberg line-follower

For implementing a robust non Braitenberg model, we chose following concept as our main design: At the beginning, the ground sensor readings are stored in an array and rounded to either black (1) or white(0). This is done by the code in Figure 3.10.

```

1  gs_values = r.get_ground()
   gs = [1 if x < LINE else 0 for x in gs_values]#1=black, 0=white, [Left, Center, Right]

```

Figure 3.10: Storing the sensor readings in an array and making them binary

Then, we distinguish four cases. Two cases for when the robot is driving away from the line, e.g. the line is only detected in either the left or the right sensor, one case for if it detects no line, and one case for it to be perfectly on the line, e.g. 2 or more sensors detect the line.

```

2  if gs == [0, 0, 1]: #too far left, case 1
   pass
   elif gs == [1, 0, 0]: #too far right, case 2
   pass
   elif gs == [0, 0, 0]: #no line, case 3
   pass
   elif gs == [1, 1, 0] or gs == [0, 1, 1] or gs == [1, 1, 1] or gs == [0, 1, 0]:
   else: #good, on line, case 4
   pass

```

Figure 3.11: The 4 cases to act accordingly to

If the robot is too far off the line (case 1 & 2 in Figure 3.11) it turns to the side where it still detects the line. In case 4, the robot drives straight forward. In case 3, it behaves differently depending on whether it was already following a line or it did not yet encounter one.

```

1  if saw_line:
   if right: #turn right
3     speed_left = NORM_SPEED/3
     speed_right = -NORM_SPEED/3
   else: #turn left
5     speed_right = NORM_SPEED/3
     speed_left = -NORM_SPEED/3
7     counter+=1
   if counter >= SEARCH_COUNTER: #after approx. 135 deg
9     right = False
   if counter >= SEARCH_COUNTER*3:
11    saw_line = False
13    counter=0
    right = True
15 else:
17    speed_left = NORM_SPEED
    speed_right = NORM_SPEED

```

Figure 3.12: Inside case 3: how to act if there is no line detected

As shown in Figure 3.12, the robot just drives straight forward if it has not been following a line. If it did, however, the robot will first perform a slow turn to the right for approximately 135 degrees. This is an attempt to find the line again in case it did a corner to the right. If it does not find the line, it then does a slow turn to the left for twice the duration which should be more or less equivalent to a 135 degree turn to the left. Using this procedure, the robots succeeds in following even triangular corners. If it still does not find the line, it just starts driving in a straight line until it finds a black line.

We provide a video of our robust line follower implementation and the robust Braitenberg line follower: <https://www.youtube.com/watch?v=s1a9hyodkdo> [3]

### 3.2.3 Robust Braitenberg line-follower

For the robust Braitenberg line follower, we modified the EXPLORER implementation in the same way that we modified the LOVER implementation. We changed it to using the left and right ground sensor. Also, unlike in the simple Braitenberg line-follower, we kept the line with the inverse speed, because the explorer needs to consider the white floor as an object which it tries to drive away from. This part is shown in Figure 3.13.

```

1 ground_values = r.get_ground()
2 speed_right = NORM_SPEED * ground_values[GS_RIGHT] / MAX_VALUE
3 speed_left = NORM_SPEED * ground_values[GS_LEFT] / MAX_VALUE
4 speed_right = NORM_SPEED - speed_right
5 speed_left = NORM_SPEED - speed_left
6 r.set_speed(speed_right, speed_left)

```

Figure 3.13: Main part of the robust Braitenberg line follower

## 3.3 Wall-following

The Wall-following implementation uses a PID controller to define it's behaviour. A PID (Proportional (current), Integral (previous), Derivative (next)) is a method of controlling a process using a single variable, in our case the variable  $ds$  which is the difference between the speeds of the two wheels. The PID needs a feedback value, which is the distance to the wall.

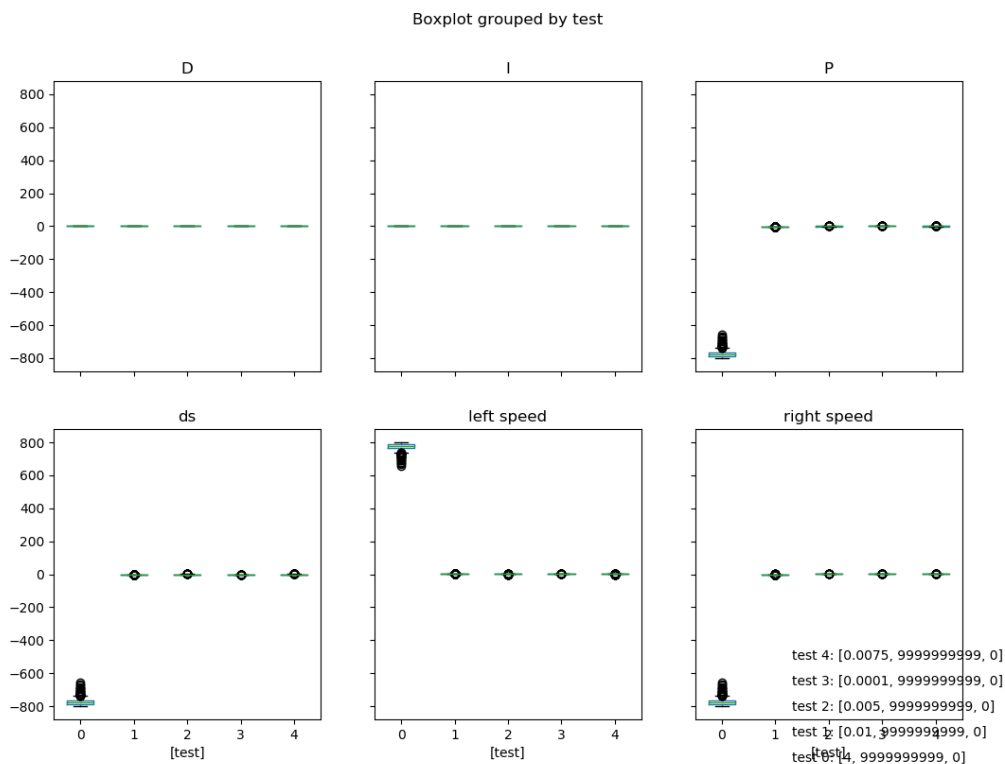


Figure 3.14: Plot of different K values

An optimal PID is defined as having a non volatile  $ds$  value. To optimise our parameters for the PID, we started experimenting with different values for  $k$  and plotted them in Figure 3.14. It was apparent, that small values (e.g. 0.01 of test 1-4) perform much better than high values ( $k=4$  of test 0). Consequently, we restricted our testing to small values and visualised our results in Figure 4.4 (Appendix). After analysing the charts and observing how the robot drove, we settled for  $k=0.0075$  as the best value. If  $k$  is too small, e.g.  $k=0.0001$ , the robot does not follow the wall, and if the value is too high it drives too volatile. But we did not notice major differences in values between 0.005 and 0.0075.

We tested different values for  $T_d$  and  $T_i$  in the same manner and found the values of  $T_i=1500$  and  $T_d=0.005$  as optimal. The plots on which we based our decision are Figure 4.5 and Figure 4.6 in the Appendix. We aimed for minimal scatter of the  $ds$  chart.

In general, our wall-following behaviour performs well in all reasonable situations. We did not encounter an unsatisfying behaviour during our tests.

```

# checks if we are away far enough from wall to drive straight
2   if (((ps[7] + ps[6] + ps[5] + ps[4]) / 4) <= nW) and (((ps[0] + ps[1] + ps[2] + ps[3]) / 4) <= nW):
4       left = False
       right = False
       led_off()
6       r.set_speed(2,2)

```

Figure 3.15: Drive straight ahead if there is no wall

Our implementation of the wall-follower is as follows: If it detects no wall, it drives straight ahead. The corresponding code is shown in Figure 3.15. It checks if there is a wall on the right or on the left side, and applies the PID to the according side's proximity sensors, and turns the according LEDs on. This is done by checking which side's average proximity value is higher, as shown in Figure 3.16. This leads to the robot following the wall that is closer to it, if there are walls on both sides.

```

# checks if right sensors detect an object (wall)
2   if (((ps[7] + ps[6] + ps[5] + ps[4]) / 4) < (ps[0] + ps[1] + ps[2] + ps[3]) / 4) and (((ps[0] + ps[1] + ps[2] + ps[3]) / 4) >= sW):
4       right = True #Sets side to right
       left = False
       led_right()

```

Figure 3.16: Detect if wall is on the right side

Finally, the PID is applied with the proximity data of the according side and returns the value for  $ds$ , which is then applied to the right and left speed. Our code for this part is written in Figure 3.17. Also, there is the clamping behavior to better handle corners of the wall.

```

1   proxR = (a * ps[0] + b * ps[1] + c * ps[2] + d * ps[3]) / (a+b+c+d);
       # compute PID response according to IR sensor value
3   ds = pid.compute(proxR, PID_WALL_TARGET);
       # make the robot turn towards the wall by default
5   ds += .05
       speedR = NORM_SPEED + ds
       speedL = NORM_SPEED - ds
7   # "clamping" function for corners
9   if abs(ds) > PID_MAX_DS :
       speedR = +ds
       speedL = -ds
11  r.set_speed(speedL, speedR)

```

Figure 3.17: Code which applies the PID

We provide a video of our wall-following implementation: [www.youtube.com/watch?v=exJq5PMYeWI](https://www.youtube.com/watch?v=exJq5PMYeWI) [4]

### 3.4 Color recognition

The basis of our implementation is the EXPLORER behaviour. We put the behaviour in a function called *explorer()* and call it in every iteration of the *while robot.go\_on()* loop. We then have to detect which color the robot sees. We do this by looking at the pixel in the middle of the image (coordinates  $x=80$ ,  $y=60$ ) after getting the current image from the camera. We only do the color analysis every 4 steps to make the robot move smoother, because getting the cameras data takes a long time. The code for this part can be read in Figure 3.18.

```

2 while robot.go_on():
    explorer(robot)#run the EXPLORER behaviour
    i+=1
4 if i%4:#only run the camera part every 4th step
    colors = np.array(robot.get_camera())
6    [r,g,b] = colors
    x=80
8    y=60

```

Figure 3.18: Part 1 of the color recognition

Afterwards, the actual color recognition is done. We do this by first checking if the contrast of the colors is high enough to be considered as a color, or if the robot only sees a tone of grey (e.g. white, grey or black). This part is shown in Figure 3.19. We consider the pixel to be a shade of grey if the difference between each of the color values is smaller than 15. For an example, (200, 190, 205) will be considered as grey, whereas (200, 190, 175) will be considered as not grey. If the pixel is grey, we turn off all LEDs, if not, we continue to decide which color it exactly is as shown in Figure 3.20.

```

2 #np.array values are uint8, so they will cause an underflow if not parsed to int()
    threshold_grey = 15
    grey=False
4 if abs(int(r[x][y])-int(g[x][y]))<threshold_grey:
    if abs(int(g[x][y])-int(b[x][y]))<threshold_grey:
6         if abs(int(b[x][y])-int(r[x][y]))<threshold_grey:
            robot.disable_all_led()
8             grey = True

```

Figure 3.19: Part 2 of the color recognition

To determine which color the pixel is, we just check which color has the highest value and set the LEDs accordingly. This means, (200, 190, 175) will be considered as red, because 200 is the highest number of all three.

It is worth mentioning, that the values returned by the camera are unsigned bytes, which means they can only represent values from 0 to 255. When performing calculations on them, like calculating the difference, the results can be negative leading to an underflow. To mitigate this problem we first cast the values to integers which support negative numbers.

We provide a video of our color recognition implementation: <https://youtu.be/HF-owC1sL9w>[5]

```

2      if not grey:
3          if (r[x][y]>g[x][y]and r[x][y]>b[x][y]): #red
4              robot.disable_all_led()
5              turn_all_rgb_on(100, 0, 0)
6              robot.enable_led(6)
7          elif (g[x][y]>r[x][y]and g[x][y]>b[x][y]): #green
8              robot.disable_all_led()
9              turn_all_rgb_on(0, 100, 0)
10             robot.enable_led(4)
11          elif (b[x][y]>r[x][y]and b[x][y]>g[x][y]): #blue
12              robot.disable_all_led()
13              turn_all_rgb_on(0, 0, 100)
14              robot.enable_led(2)

```

Figure 3.20: Part 3 of the color recognition

### 3.5 Multi-robot coordination

We implemented two different behaviours for multiple robots, simultaneous and alternative search. The behaviour of each robot running the simultaneous search implementation is visualised in Figure 3.21. All robots start in LOVER mode and wait for each other to reach the equilibrium before continuing with EXPLORER mode all together. The alternative search function a bit differently, namely once all robots wait in equilibrium, only one of them starts continuing to EXPLORER state. This leads to only one robot at a time being in EXPLORER state, while the others wait in equilibrium.

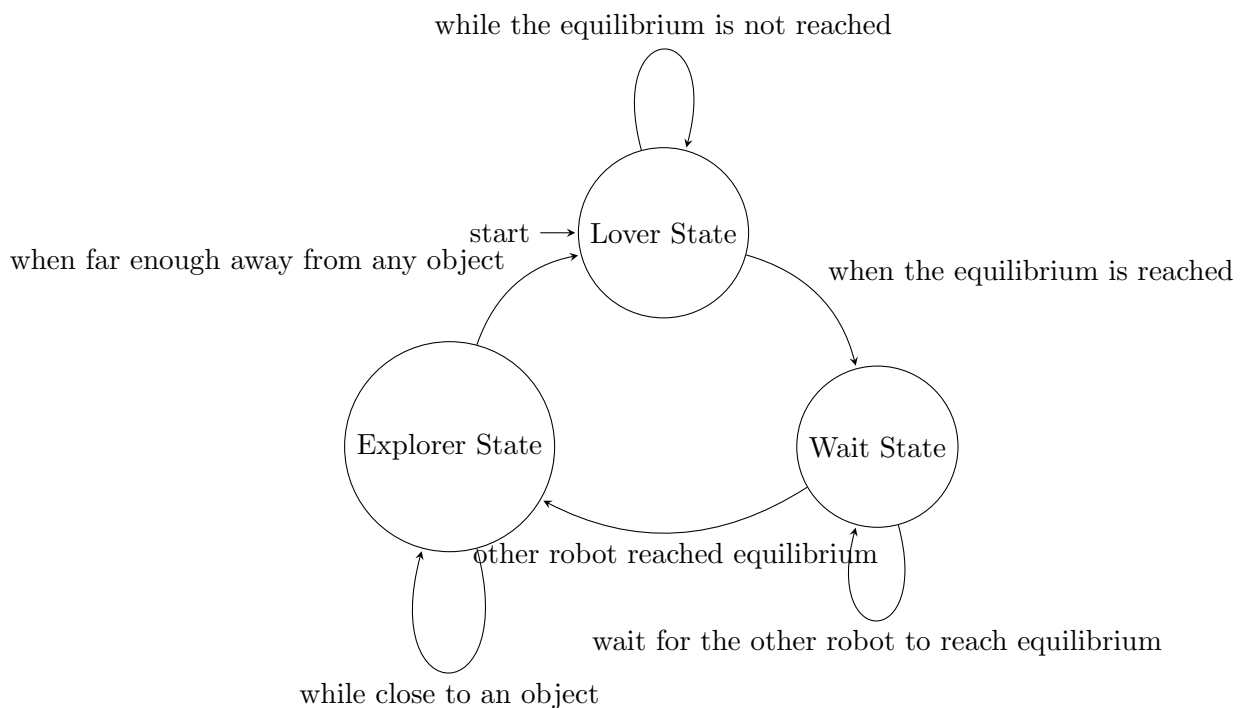


Figure 3.21: Simultaneous search final state machine for a 2 robot system

For both behaviours, we used the alternating lover implementation as a base and adapted it to our needs. To enable us to control multiple robots with the same controller without the use of a bash script, we used python's multiprocessing module to start multiple instances of the controller function *minion()*, one for each IP given as a command line argument. This part is

shown in Figure 3.22.

```

1 if __name__ == "__main__":
2     L=[] #get all IP addresses of the robots from the command line arguments
3     for i in sys.argv[1:]:
4         L.append(i)
5     manager = multiprocessing.Manager()
6     pl = []
7     for i in L: #start a sub process for every robot
8         p=multiprocessing.Process(target=minion, args=(i, i))
9         pl.append(p)
10        p.start()
11    for i in pl:
12        #waits for the sub processes to stop
13        i.join()

```

Figure 3.22: Use the multiprocessing module to start multiple instances of the e-puck controller

The necessary modifications for the simultaneous search are the following: Inside the *while r.go\_on()* loop, we added a state called *wait state* and made the state to change to wait state if the equilibrium is reached, as shown in Figure 3.23. In the wait state, the robot first sends a message to all other robots, saying that it reached the equilibrium, and then waits until it received a message from every other robot indicating that it reached the equilibrium. Once that happened, it continues to the EXPLORER state.

```

1 elif state == WAITSTATE:
2     #send "reached equilibrium" to all other robots
3     r.send_msg(REACHED)
4     waiting_robots = 1
5     #The robot waits for N 'reached'-signals before continuing with EXPLORER
6     while waiting_robots < N_robots:
7         msg = r.receive_msg()
8         if msg==REACHED:
9             waiting_robots += 1
10        #continue with EXPLORER
11        state = EXPLORER
12        counter_avoid = 0
13        r.disable_all_led()
14
15 elif state == LOVER and counter_obstacle > COUNT_LOVE:
16     state = WAITSTATE #switch to WAITSTATE after equilibrium is reached

```

Figure 3.23: Modification of alternating lover for simultaneous search

To get from the simultaneous search to the alternative search, only a minor modification shown in Figure 3.24 is needed. We add to the lover state that the robot receives and 'ignores' messages from other robots. That way, the robot only counts robots as waiting if they reached the equilibrium after itself. This leads to the robot that stops first, acting like usual in simultaneous search and start to run EXPLORER after all others stopped. But all other robots ignored the "reached equilibrium" message of the first robot and are therefore waiting for at least one more "reached equilibrium" message. The robot that reached equilibrium second, is waiting for one message, the third for two, etc. When the first robot, which is moving currently, reaches equilibrium again, it's message will make the second robot start moving.

```

1 if state == LOVER:
2     #ignore/delete messages from robots that stopped previously
3     msg = r.receive_msg()

```

Figure 3.24: Modification of simultaneous search for alternative search

We provide a video of our multi robot implementation: <https://youtu.be/HF-owC1sL9w?t=38>[5]



## Chapter 4

# Conclusion

This paper is intended to state the capabilities and limitations of the e-puck robots. We conclude based on our experiments, that the e-pucks are capable to be used with many different behaviours relying on sensory data. We succeeded in implementing the Braitenberg LOVER and EXPLORER behaviours as well as line- and wall-followers, color recognition and also communication between multiple robots. The e-puck's sensor provide mostly a pleasing accuracy, although there are some minor downfalls. These downfalls are the difference in reported brightness by the ground sensors and that the microphones are not nicely calibrated and not sensitive enough to enable detecting from which direction a sound is originating from, when the sound source is far away.

An unexpected fact that stood out to me was how little modifications some behaviours need to be changed to different behaviours that work just as robustly. This is especially true for the small changes to LOVER to get an EXPLORER behaviour, as well as how well a EXPLORER behaviour works for following a line, or what little had to be changed to get an alternative search algorithm from a simultaneous search algorithm.

It remains exciting what behaviours will be invented in the next project, the e-pucks definitely have potential for some amazing possibilities. Especially interesting is how well the sound localisation using the microphones will work in action, and whether it is needed or even possible to calibrate uncalibrateable sensors in post, e.g. after they made their measurements.

# Bibliography

- [1] *Graph of sound intensity by distance.* <http://hyperphysics.phy-astr.gsu.edu/hbase/Acoustic/invsqs.html> version 1 Last visited: 18.04.2021.
- [2] *Video of alternating Lover with LEDs.* [https://drive.google.com/file/d/1\\_N-3FaMlfVto-3zkSSeGvvdrfnfVadtV/view?usp=drivesdk](https://drive.google.com/file/d/1_N-3FaMlfVto-3zkSSeGvvdrfnfVadtV/view?usp=drivesdk) version 1 Last visited: 14.03.2021.
- [3] *Video of robust line follower.* <https://www.youtube.com/watch?v=s1a9hyodkdo> version 1 Last visited: 28.03.2021.
- [4] *Video of PID Wall Follower.* <https://www.youtube.com/watch?v=exJq5PMYeWI> version 1 Last visited: 28.03.2021.
- [5] *Video of Color recognition and simultaneous/alternative search.* <https://www.youtube.com/watch?v=HF-owC1sL9w> version 1 Last visited: 18.04.2021.

# Appendix

## Appendix A Experimental Results

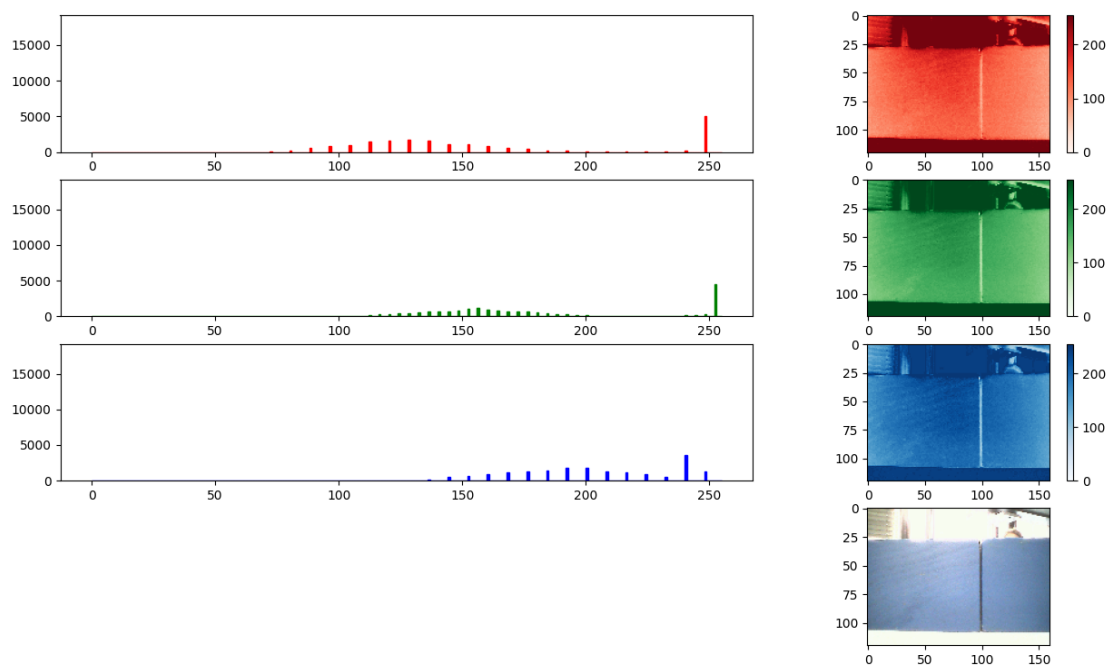


Figure 4.1: Histogram of an image of a blue object

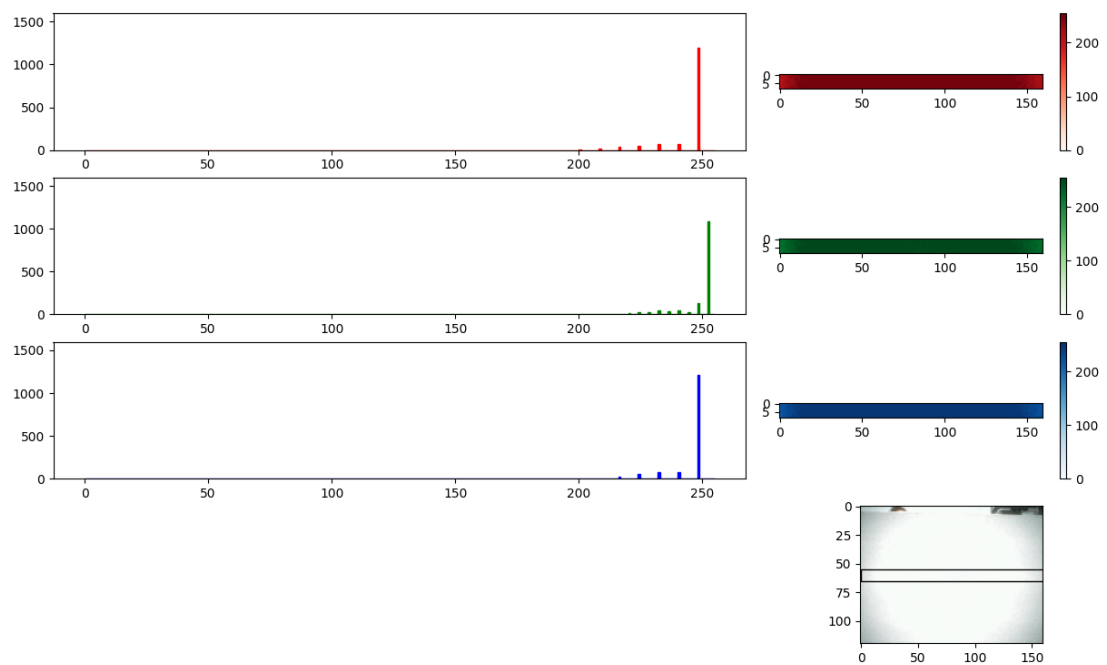


Figure 4.2: Histogram of the letterbox of an image of the arena wall.

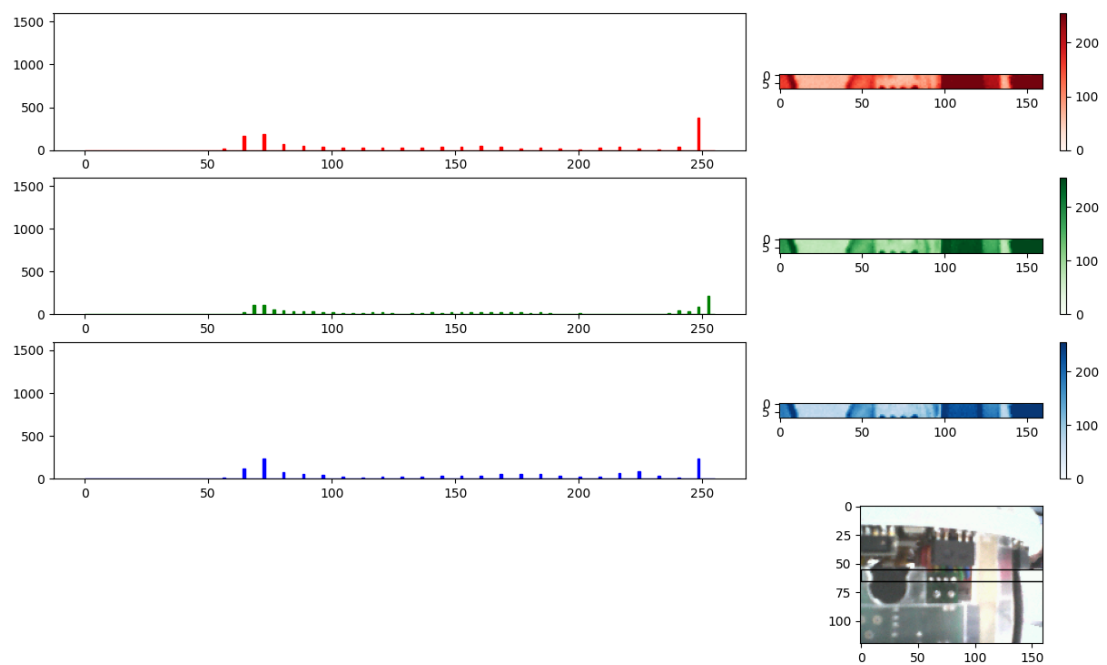


Figure 4.3: Histogram of the letterbox of an image of another e-puck.

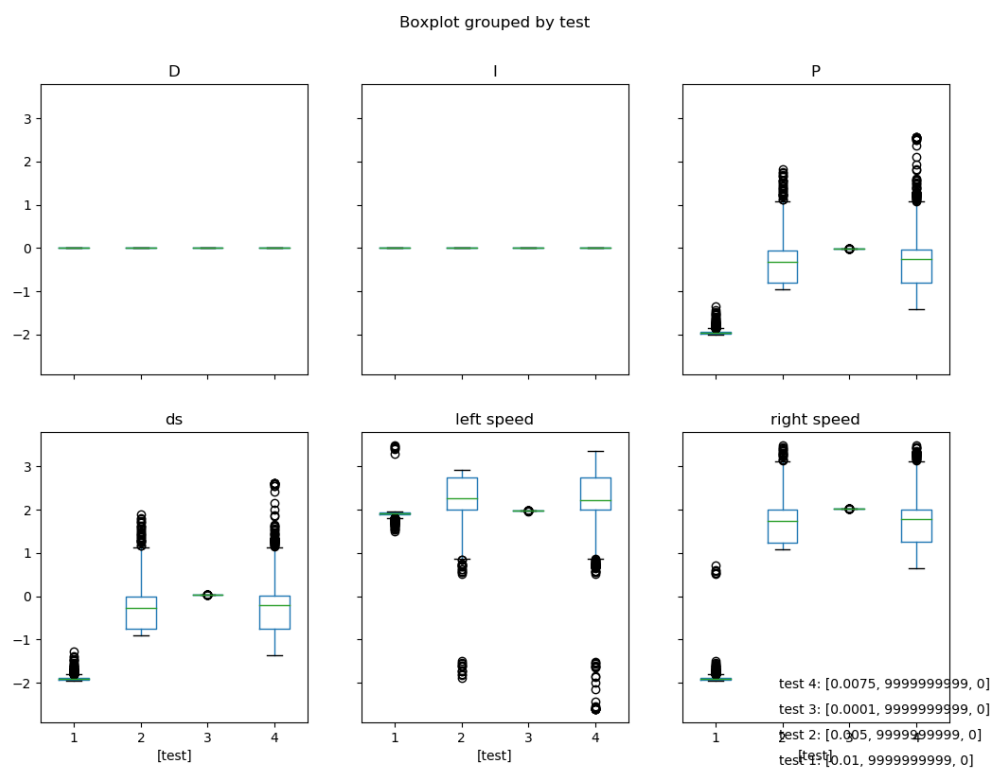


Figure 4.4: Plot of different K values

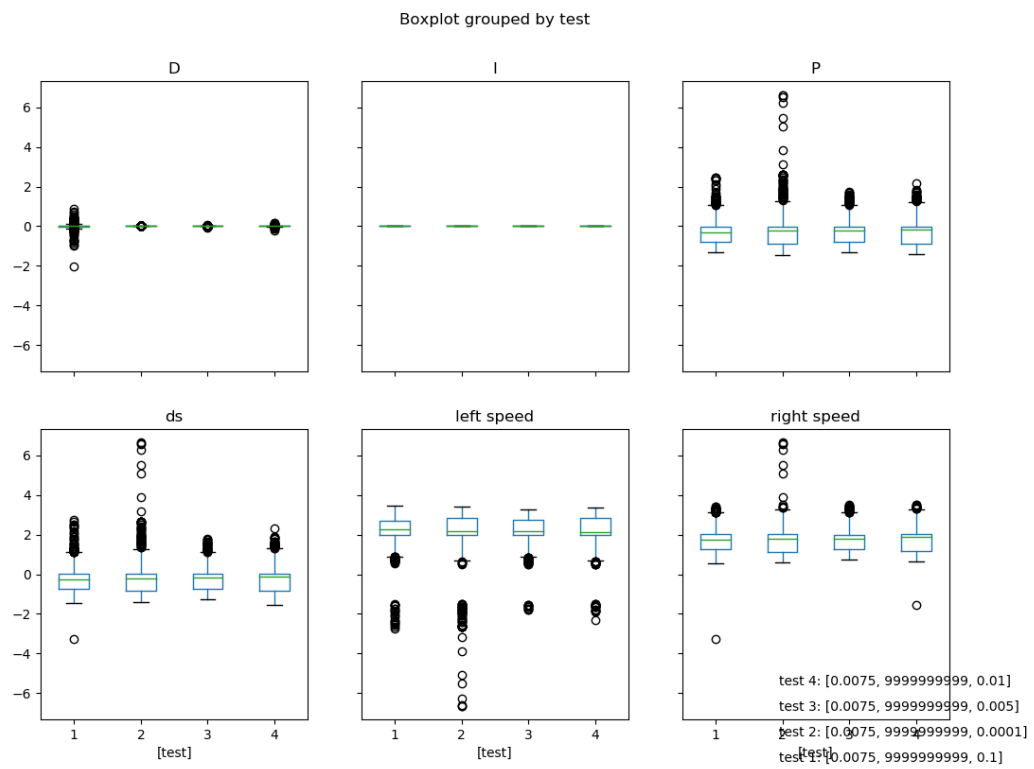
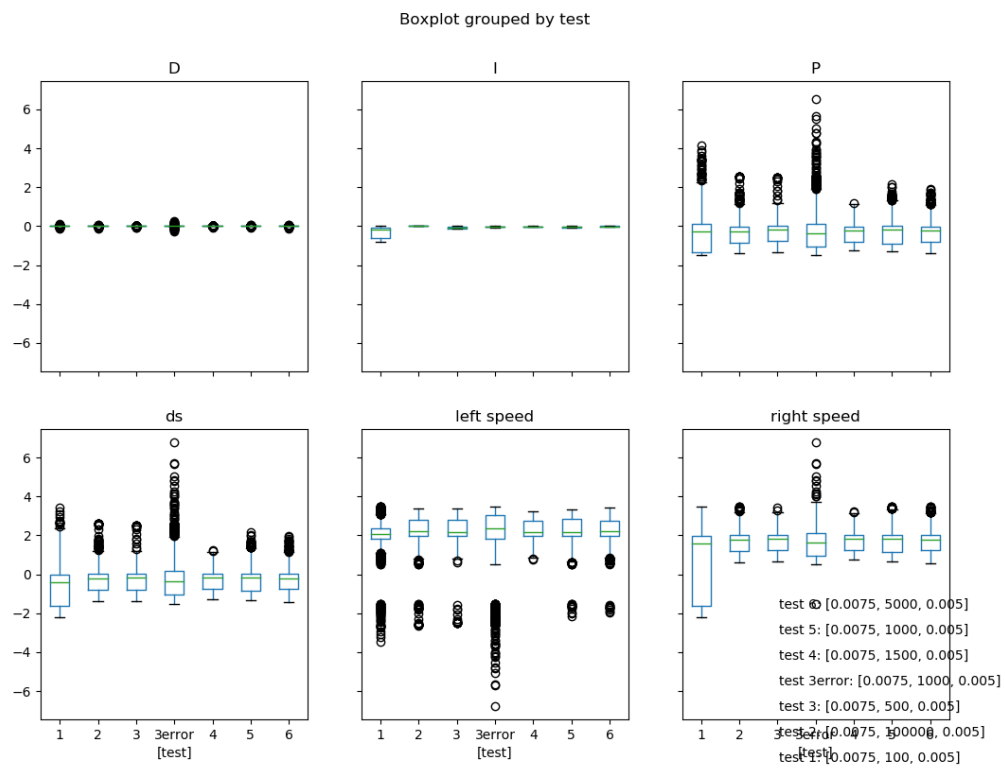


Figure 4.5: Plot of different Td values

Figure 4.6: Plot of different  $T_i$  values



## Appendix B Source Code

```
1 def get_camera(self):  
3     if self.__camera_updated:  
        if self.__my_filename_current_image:  
            self.__rgb565_to_bgr888()
```

Figure 4.7: Extract from epuck\_wifi.py of the unifr\_api\_epuck library which shows the bit resolution of the different color channels used by the camera