Vision for Autonomous Robots
Lab Project 1: Image Calibration & Line Following

Wouter Bant (13176676)
Angelo Broere (13168215)
Gidi Ozery (14643618)
Valeria Sepicacchi (15187853)

Due: November 10th, 2024 CEST

1 Introduction

In this lab assignment, we design an edge detection pipeline, score edges based on heuristics, and given the proposed edge we investigate various ways of following the edge via a smooth trajectory. For edge detection, we explore ways with and without the canny edge detector. When determining the edge of interest, we look at various heuristics including the edge length and position relative to the previously proposed edge. For incorporating a smooth trajectory, we investigate using an exponential weighted average for the angular velocity and averaging multiple actions over batches of images to send a single move command to the robot. This move command is directed to the robot every 0.8 seconds.

2 Session 1: Camera Calibration

2.1 Topic List for Camera Calibration

To see all the topics we used:

```
ros2 topic list
```

To launch the camera calibration on ROS2, we used:

```
ros2 run camera_calibration cameracalibrator \
--size 8x5 \
--square 0.03 \
--ros-args -r /image:=/camera/image_raw
```

2.2 Camera Calibration Results

2.2.1 Intrinsic Camera Matrix

The calibration process yielded the following intrinsic camera matrix:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

$$K_{ours} = \begin{bmatrix} 290.46 & 0 & 312.90 \\ 0 & 290.37 & 203.015 \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

2.2.2 Camera Distortion Parameters

Using the default distortion model, we obtained the following distortion coefficients:

$$D = [-2.80 \times 10^{-1}, 6.43 \times 10^{-2}, -6.80 \times 10^{-5}, 1.97 \times 10^{-3}, 0.00] \tag{3}$$

So $x_{distorted} = x(1 - 0.279797r^2 + 0.064309r^4)$ and $y_{distorted} = y(1 - 0.279797r^2 + 0.064309r^4)$ with $r = \sqrt{x^2 + y^2}$ (here we didn't round the numbers).

2.2.3 Collected Images

Figure 1: Images used for camera calibration.

2.3 Advanced Calibration Implementation

We implemented an improved calibration algorithm using OpenCV's additional distortion models[1]. The different distortion models used in our approach are: Default, Rational and the Thin Prism model. The following matrices are the camera matrices calculated by our approach, using the different distortion models:

$$K_{default} = \begin{bmatrix} 273.65 & 0 & 319.01 \\ 0 & 273.55 & 203.12 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

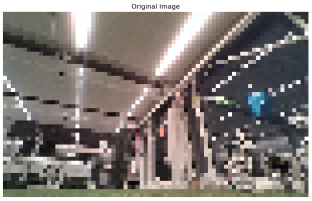
$$K_{rational} = \begin{bmatrix} 273.21 & 0 & 320.87 \\ 0 & 273.08 & 203.25 \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

$$K_{thin\_prism} = \begin{bmatrix} 274.62 & 0 & 305.28 \\ 0 & 274.71 & 192.29 \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

(a) Raw input image

(b) Undistorted default (using 5 distortion coefficients)

(c) Undistorted with rational model



(d) Undistorted with thin prism model

Figure 2: Comparison of different distortion models

[1] https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html

Coefficient	Default Model	Rational Model	Thin Prism Model
k_1	-0.28416531	-0.14005517	-0.29394566
k_2	0.09616325	-0.14634682	0.11084072
p_1	-0.00031899	-0.00050158	-0.00548283
p_2	0.00129471	0.00081934	-0.00508525
k_3	-0.01529113	0.00344207	-0.02123715
k_4	-	0.17342671	0
k_5	-	-0.26600062	0
k_6	-	-0.00599121	0
s_1	-	0	0.01992598
s_2	-	0	-0.00193286
s_3	-	0	0.0153437
s_4	-	0	-0.00206453
τ_x	-	0	-
τ_y	-	0	-
Re-projection error	2.6639...	2.8789...	0.0566...

Table 1: Distortion coefficients obtained with different distortion models

The quantitative and qualitative results show conflicting insights regarding the calibration quality in our opinion. The thin prism model has the lowest reconstruction error while showing good angle correction. However, the resolution is much lower making it too noisy to do reliable edge detection. Although the rational model has the highest re-projection error, the reconstruction is visually better than the default model when looking at the grass. However, the sides of the image are of lower resolution which for line detection was not a problem as we crop that part out. The default model performs reasonably well, although we notice still some distortion on the far left and right.

3 Session 2: Line Detection

3.1 Pipeline Design

Our line detection pipeline consists of the following steps:

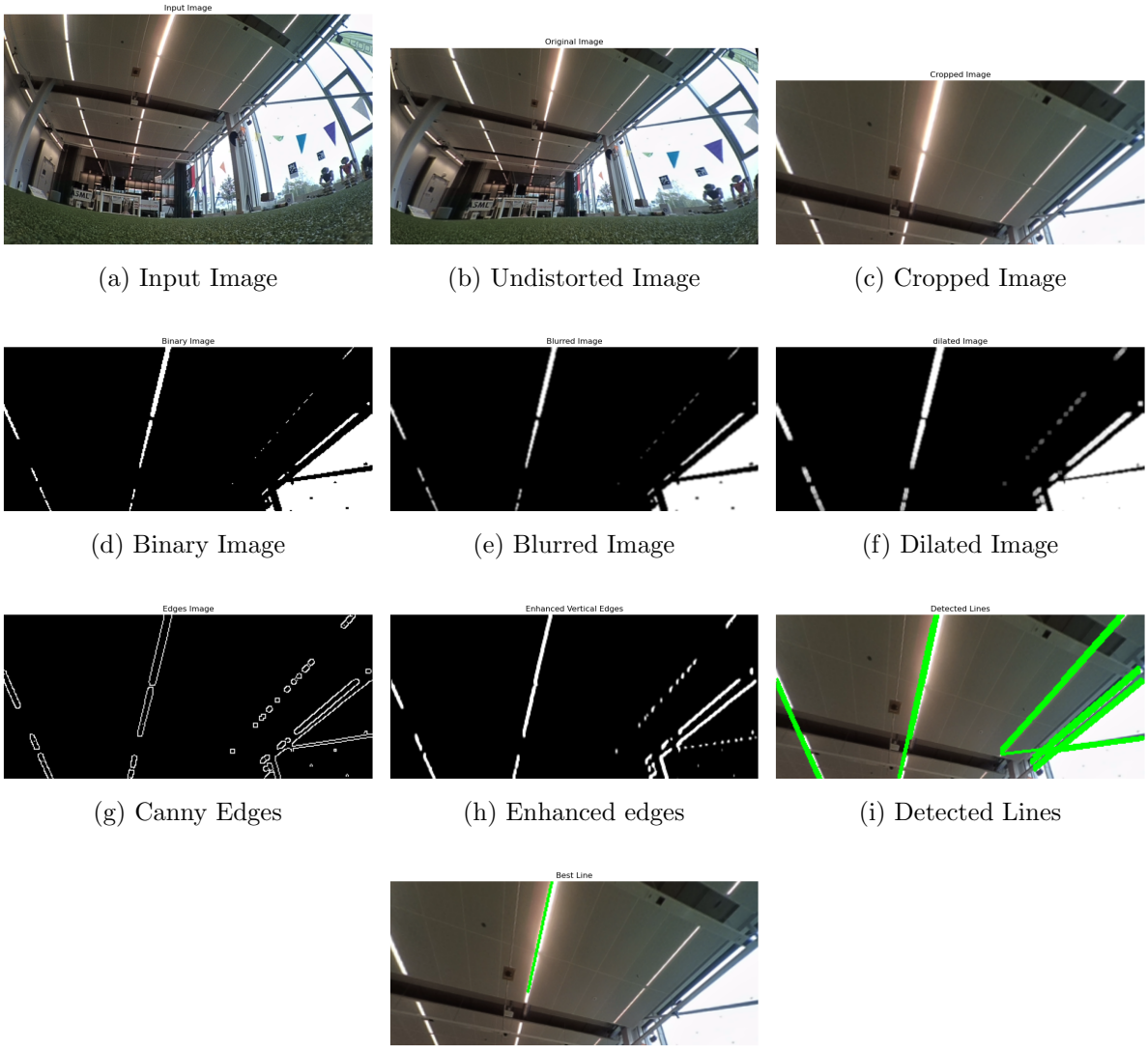
- Image Preprocessing:
  - Undistorted image with obtained transformation
  - Crop image
  - Blur image to remove noise
  - Dilate image to connect edges
- Edge detection with one of the following:
  - Canny edge detection with OpenCV implementation
  - With a Sobel kernel to find vertical-like edges
    $kernel = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])$
  - Canny edge detection followed by Sobel kernel
- Line detection with Hough transform (finding the best line is discussed later)

It is maybe easier to see it in code:

```
def pipeline(self, img):
    img = self.convert_to_cv2_image(img)
    if self.config.get("undistort_image"): img = self._undistort_image(img)
    img = self.crop_to_top_roi(img)
    img = self.crop_sides(img)
    img = self.convert_to_gray(img)
    img = self.binarize_image(img)
    if self.config.get("blur_image", False): img = self.blur_image(img)
```

```
    if self.config.get("dilate_image", False): img = self.dilate_image(img)
    if self.config.get("method") == "canny": edges = self.canny_edge_detection(img)
    elif self.config.get("method") == "sobel": edges = self.sobel_edge_detection(img)
    if self.config.get("enhance_vertical_edges"):
        edges = self.enhance_vertical_edges(img)
    lines = self.hough_transform(edges)
    best_line = self.get_best_line(lines)
    action = self.action(best_line)
    return action
```

Below is a typical line detection process depicted:



(a) Input Image (b) Undistorted Image (c) Cropped Image

(d) Binary Image (e) Blurred Image (f) Dilated Image

(g) Canny Edges (h) Enhanced edges (i) Detected Lines

(j) Best Line

Figure 3: Processing Steps in the Pipeline Example 1

The raw input image (a) is undistorted by the found default camera calibration parameters (b). This process makes sure that straight lines in the classroom are also seen as straight lines by the robot. The result is cropped according to (c). We tried various cropping sizes, and we noticed that retaining too much of the original image can distract the robot when selecting the best line. Occasionally, the line detection algorithm would also pick up lines on the floor. We introduce multiple heuristics for the robot when picking the line of interest. After cropping, we binarize (d) the images and find the ceiling light, followed by slight blurring (e) to remove some of the noise that can negatively affect the decision-making of the robot with line detection. However, in some cases, the blurring also makes it much harder to detect the actual lines. To compensate for this, we apply dilation (f) and then use the canny edge detector (g) to detect lines. However, as can be seen in Figure 3g, this often detects two parallel lines. To detect a singular line instead, we apply a Sobel kernel to enhance the edges. This results in these parallel lines often being merged into a singular line in the middle. We apply a Hough transform (i) to find candidate lines and score these to get the best line (j).

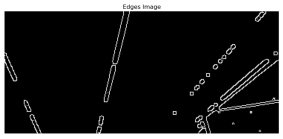
3.2 Canny Edge Detector and Our Edge Detector

For the implementation, we used the OpenCV library and found that a canny lower of 100 and a canny upper of 150 gave robust results:

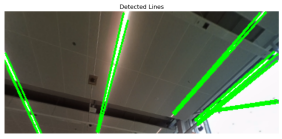
```
def canny_edge_detection(self, img):
    return cv2.Canny(img, self.config.get("canny_lower"),
                          self.config.get("canny_upper"))
```

As the pipeline images already showed, the Canny edge detection seems to be redundant in our pipeline and experiments showed that removing the Canny edge detection did not deteriorate performance. Hence, we simply removed it and replaced it with a vertical Sobel kernel instead. Below we show in images (b) and (c) the canny edges and detected lines. In (d) the enhanced canny edges are shown and in (e) the detected lines with the result. In (f) we apply the Sobel kernel directly to the dilated image and show the detected lines in (g).

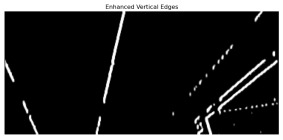
(a) Input image



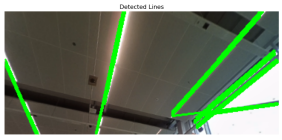
(b) Canny edges



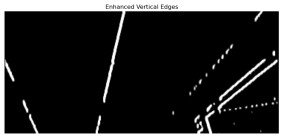
(c) Detected lines



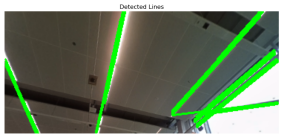
(d) Enhanced Canny edges



(e) Detected lines (enhanced edges)



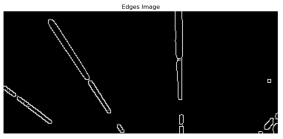
(f) Edges with Sobel kernel



(g) Detected lines (dilated edges)

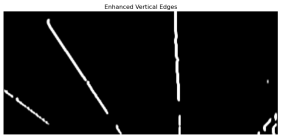
Figure 4: Example 1

(a) Input image



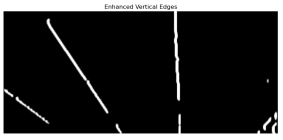
(b) Canny edges

(c) Detected lines



(d) Enhanced Canny edges

(e) Detected lines (enhanced edges)

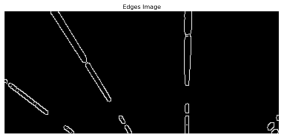


(f) Edges with Sobel kernel

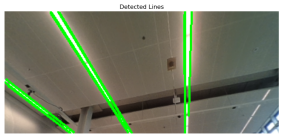
(g) Detected lines (dilated edges)

Figure 5: Example 2

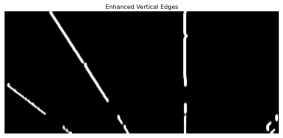
(a) Input image



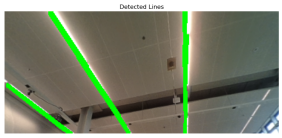
(b) Canny edges



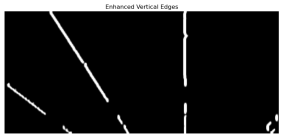
(c) Detected lines



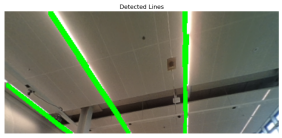
(d) Enhanced Canny edges



(e) Detected lines (enhanced edges)



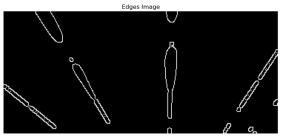
(f) Edges with Sobel kernel



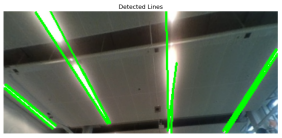
(g) Detected lines (dilated edges)

Figure 6: Example 3

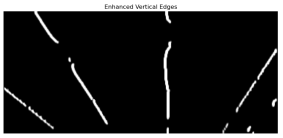
(a) Input image



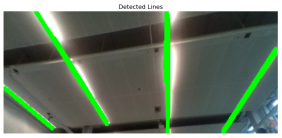
(b) Canny edges



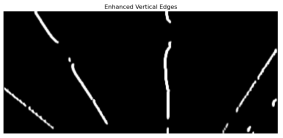
(c) Detected lines



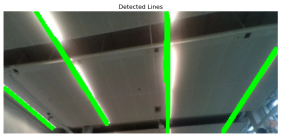
(d) Enhanced Canny edges



(e) Detected lines (enhanced edges)



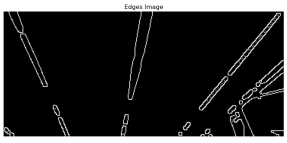
(f) Edges with Sobel kernel



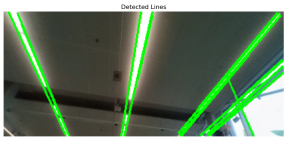
(g) Detected lines (dilated edges)

Figure 7: Example 4

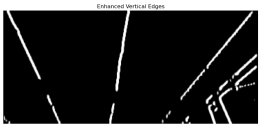
(a) Input image



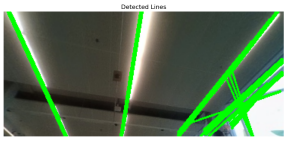
(b) Canny edges



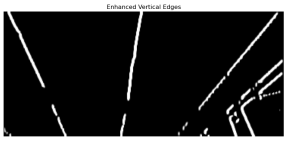
(c) Detected lines



(d) Enhanced Canny edges



(e) Detected lines when using enhanced Canny edges



(f) Edges with Sobel kernel on dilated image

(g) Detected lines when using edges on dilated image

Figure 8: Example 5

Why is our solution with a simple Sobel kernel not worse than edges detected by the Canny edge

detector:

- We are interested in somewhat vertical lines. This inductive bias is easily incorporated using a Sobel kernel for vertical edges. However, the Canny edge detector will also look for more horizontal edges. Still, in example 5 it can be seen, because the horizontal lines are pretty wide, that even the Sobel kernel keeps these lines. Further, sometimes there are 'bulbs' of light that can throw off the canny edge detector as in Figure 6 (c). Then Sobel kernel discards the right side of this bulb as it is not straight enough and produces more consistent lines over frames.

- As the dilated image essentially captures the entire width of the ceiling lights, the lines found by the Canny edge detector will detect the borders of the ceiling lights. For this assignment, we are interested in just following the ceiling lights itself and the Sobel kernel naturally finds some lines in between these boundaries. This is also the most significant difference in detected lines. See for example Figure 8 (c) canny detects two lines and for every frame, the best line is one of these two and not consistently one. In Figure 8(e) and (g) there are multiple lines detected but these are all parallel so on a frame-by-frame basis we can robustly steer the robot based on this as there are no sudden jumps. The same can be seen in Figures 4, 5, and most clearly in 6.

4 Session 3: Line Following

4.1 Deriving Movement from Best Line: Part I

First, we clarify how we reference angles: specifically, we refer to the angle at the bottom left of the chosen line. For example, in Figure 9, frame 39 has an angle of approximately 100 degrees, while frame 40 has an angle of around 80 degrees.

We take the proposed lines by the Hough transformation and determine the line of interest by picking the longest line among the candidates. Now we determine the angle of this line and adjust the robot's trajectory with approximately the same angle. Note that the robot is not encouraged to move under the line of interest, but will do so when it starts under the line. In code, this can be summarized as:

```
x1, y1, x2, y2 = best_line[0]
if x1 > x2:
    x1, y1, x2, y2 = x2, y2, x1, y1
line_angle = np.degrees(np.arctan2(y2 - y1, x2 - x1))
line_angle = (line_angle - 90) / 180 if line_angle > 0 else (90 + line_angle) / 180
line_angle *= 10
max_angular_speed = 1.0
new_ang = max(-max_angular_speed, min(max_angular_speed, line_angle))
```

To make the steering work properly, we multiplied the angle by 10 which we found experimentally. Also, we limit the absolute angular speed by 1 to prevent over-correction. We implemented a second implementation that allows for parallel driving with respect to the line. This is achieved by initially centering the closest point of the line. After a certain number of steps, or once the distance is sufficiently small and the line is sufficiently straight, we shift our focus to the point where the line intersects the bottom of the image:

```
x1, y1, x2, y2 = best_line[0]
if x1 > x2:
    x1, y1, x2, y2 = x2, y2, x1, y1
line_angle = np.degrees(np.arctan2(y2 - y1, x2 - x1))
line_angle = (line_angle - 90) / 180 if line_angle > 0 else (90 + line_angle) / 180
line_angle = abs(line_angle)
if self.config.get("horizon_center"):
    x = x1 if y1 > y2 else x2  # keep horizon point center
    if y1 > y2:
        x1, x2, y1, y2 = x1, x2, y2, y1
    # (y1 - y2) / (x1 - x2) = (y1 - height) / (x1 - x)
    x = x1 - (y1 - height) * (x1 - x2) / (y1 - y2)
    new_ang = (width / 2 - x) / (6 * width)
else:
    x = x2 if y1 > y2 else x1  # keep closest point center
    if x < width / 2:  # point is too far left
        new_ang = line_angle  # go to left to center it
```

```
    else:
        new_ang = -line_angle
new_ang *= 10
cur_distance = abs(x - width / 2)
if (cur_distance < 30 and line_angle * 180 < 10) or self.frame > 15:
    # we are now sufficiently under the line
    self.config["horizon_center"] = True  # start following the horizon
```

This second method resulted as expected in trajectories parallel with the light on the ceiling as opposed to following the seen angle of a line, which produces a somewhat straight trajectory but follows the angle as the robot sees it (so that the lines would intersect at some point at if they actually followed that path in the world view).

4.2 Deriving Movement from Best Line: Part II

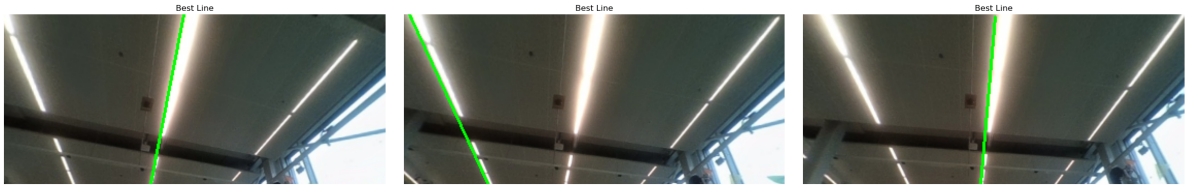
We found that the heuristic of choosing the longest line picks the correct line in approximately 95% of cases when placed somewhat parallel to one of the ceiling lights. However, the tracked line can occasionally switch as shown in Figure 9. Also, the percentage of following the correct line is much lower when placed under a heavy angle.

To remedy this, we instead take the line closest to the previously used line. Here we look at the x-value of the highest point of the line in the image. We compare this x-value with one of the previous best lines and choose the closest one. Given that our line detection pipeline is robust, the correct line is in all frames for every experiment. With this new heuristic, we are also always able to track the same line. For initialization, we set the previous x-value to the middle of the image, for the robot to start tracking the line that is closest to the middle of its initial view.

So now the code looks like:

```
def get_best_line(self, lines):
    best_line = None
    best_loss = float("inf")

    for line in lines:
        x1, y1, x2, y2 = line[0]
        length = np.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
        angle = abs(np.degrees(np.arctan2(y2 - y1, x2 - x1)))
        x = x1 if y1 < y2 else x2
        distance_to_previous_line = abs(x - self.last_x)
        if self.config.get("loss") == "distance":
            loss = distance_to_previous_line
        elif self.config.get("loss") == "length":
            loss = -length
        if loss < best_loss:
            best_loss = loss
            best_line = line
    x1, y1, x2, y2 = best_line[0]
    x = x1 if y1 < y2 else x2
    self.last_x = x
    return best_line
```



(a) Frame 39 (b) Frame 40 (c) Frame 41

Figure 9: Taking the longest line can cause the selected line to be different across frames. This was not a problem with the closest line picking heuristic.

4.3 Impact of Camera Calibration

(a) Undistorted image (b) Without distortion: turn right with 2.1 degrees (c) Distorted image (d) With distortion: turn right with 4.9 degrees

Figure 10: Impact of camera calibration on action

Figure 10 shows a typical difference in action with and without distortion. With camera calibration, the robot sees it is almost right under the line with the right angle and makes a small 2.1 degree adjustment. Without camera calibration, we notice an increase in angular adjustment from 2.1 degrees to 4.9 degrees. This is due to distortion of the camera by which the ceiling lights are slightly bent. Without camera calibration, the robot thus takes different actions. We observed that when centering the farthest point of the line the robot could still follow the line, however, the amount of swinging increased substantially. In contrast, using the other method of following the line's angle had a stronger behavioral impact. At times, the distortion caused the robot to want to drive off the track.

4.4 Optimization and Challenges

To counter swinging, we take an exponential weighted average of the angle. The idea behind this is that sudden high rotation angles become less likely, as they are averaged out. Figure 11 visualizes this where the rotation over time is modeled with $\cos(t)/(0.1t)$. From the Figure, it is clear that the peaks are lower and hence the robot is less likely to overshoot and therefore also doesn't need to oversteer later. Experimentally, we found that this approach is especially effective when in the initial position the robot is already aligned well with the line. To be robust in case the robot is further from the line, we use a weight of 0.5. In this case, only the last two angles have a significant impact on the new angle. The more closely the robot starts aligned with the line of interest, the better a lower alpha performs. With a smoothing of 0.5, we find that the robot swings less in all experiments.

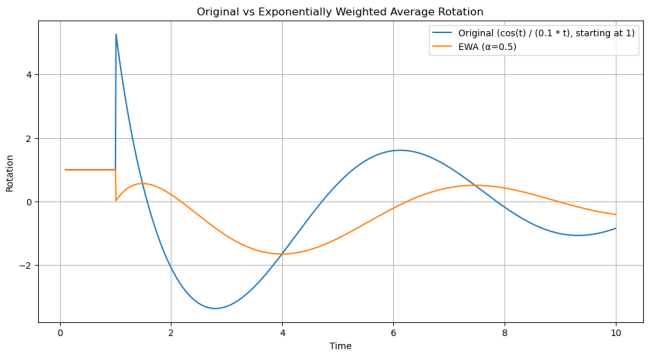
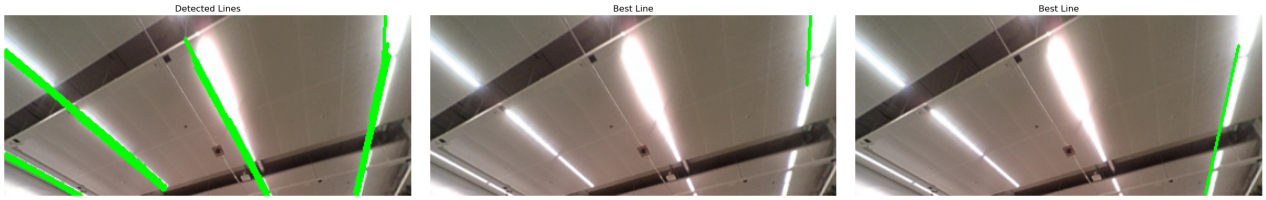


Figure 11: Exponential average can have dampening effect

We also found that it is not desirable to send a move command to the robot for every image it receives, as this can overwhelm the robot's processing capacity. To handle this, we keep a queue with the action based on the latest 5 images. Every 0.8 seconds we execute the average of these 5 actions. If one of the images leads to a significantly different action (likely incorrect due to a short line with an incorrect orientation, as shown in Figure 12 (b)), the impact on the robot is minimized as it is averaged. Consequently, the robot shows less swinging.



(a) Lines found by Hough detection (b) Line closest to the previous line (c) Close line that is long

Figure 12: Using only closest line vs also using line length

Another way to limit the impact of short lines is trying to avoid them in the selection process. To do this we combine the two line scoring methods we have: we look for lines close to the previously used line and from this candidate set, we choose the longest line. The positive impact of this is visualized in Figure 12 (c). Specifically, we add to the code from before:

```
if loss < best_loss:
    if best_line is not None and self.config.get("loss") == "distanceLength" and abs(best_x - x) < 20 and length < best_length:  # skip if the line is close but not longer
        continue
```

This adjustment additionally increased the robustness of actions (less swinging). The use of 20 pixels was determined experimentally.

5 Conclusion

In this lab assignment, we designed a pipeline that is able to successfully follow ceiling lights. We found that the best edge detection pipeline does not require the Canny edge detector. Furthermore, keeping the farthest visible point of the line of interest centered is the method that leads to the smoothest trajectory. We also implemented an algorithm that first centers the robot under a ceiling light before following the farthest visible point. We found that the best lines are used, when both the information about the length and relative position of the line compared to the previously proposed line is incorporated. To further decrease the amount of swinging, we use an exponential weighted average of angular velocities and average actions from batches of images to make a single move command every 0.8 seconds.