Czech Technical University in Prague
Faculty of Electrical Engineering

Technical Report

Driving with signs

Laboratory of Robotics

by

Oleh Borys, Mark Horpynych

Prague, Czech Republic
May 26, 2024

Contents

List of Figures

Abstract

This report presents the development and implementation of a navigation system for a robot tasked with following a course marked by checkpoints consisting of colored poles. The primary goals of this project were:

- To generalize the solution, making it easily adaptable to different tasks
- To enable the robot to adapt to varying light conditions
- To create a simulation/visualization component for easier development

To achieve these goals, we programmed the robot to recognize specific color patterns of the poles and make directional decisions accordingly. Obstacle detection and avoidance mechanisms were integrated to ensure smooth navigation. Our system was tested and successfully guided the robot through the required checkpoints, demonstrating reliable obstacle avoidance. Additionally, the robot achieved the bonus task of stopping at the final checkpoint marked by two green poles. These results validate the effectiveness of our navigation algorithm, its adaptability to different conditions, and the usefulness of the simulation/visualization component in facilitating development. You can access a complete Python implementation for the solution in our GitHub repository [1]. You can see a diagram for our solution in Fig. 1.

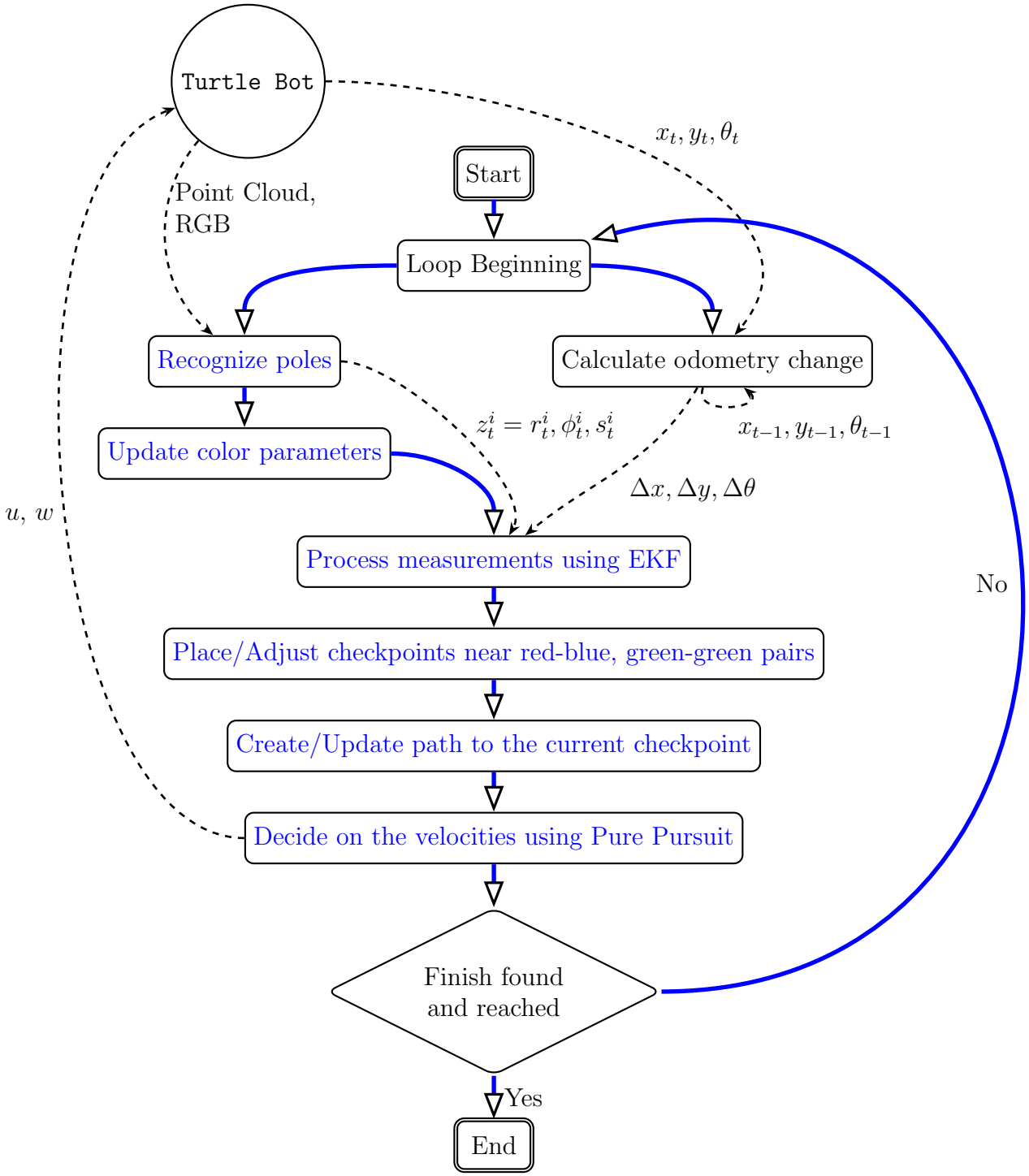


Figure 1: Diagram of the solution

Chapter 1

Introduction

We have a robot vacuum cleaner **TurtleBot 2** that is a platform designed for research and education. The main part is the **Kobuki** base, which provides basic sensors (bumper, wheel drop, etc), digital and analog inputs, digital outputs, and actuators. In addition to the Kobuki sensors, the TurtleBot 2 has a Kinect-like **RGBD sensor** 640x480 [2]. This robot has to complete the task requirements listed below.

Task requirements:

- The robot navigates a course marked by checkpoints made of poles. Each pole is a paper tube, 50mm in diameter and 350mm high, and they come in blue, red, and green. Checkpoints consist of two poles of different colors, spaced 50mm apart (the shortest distance between the sides of the poles).
- If a blue pole is on the left and a red pole is on the right, the robot turns right. If a red pole is on the left and a blue pole is on the right, the robot turns left.
- The robot can be oriented in any direction at the start. The first checkpoint is always the closest to the robot's center. Subsequent checkpoints are the closest ones in the given sector.
- Obstacles, made of poles in various colors, may be placed on the course. These poles stand alone (with a minimum center-to-center distance of 100mm) or side by side but never match the color pairs of checkpoints. Obstacles will not be placed within 500mm in front of a checkpoint or in the direction of travel. Obstacles will not block the view of the next checkpoint from the robot's position in front of a checkpoint.
- The robot must reach at least the fifth checkpoint within a 6-minute time limit.
- Bonus Task:
  The robot completes the task if it stops at the checkpoint with two green poles. This checkpoint is placed last on the course.

Please note that this list is not complete. All further details about the task can be found in [3].

Figure 1.1: A robot with checkpoints and an obstacle in the environment

Chapter 2

Implementation

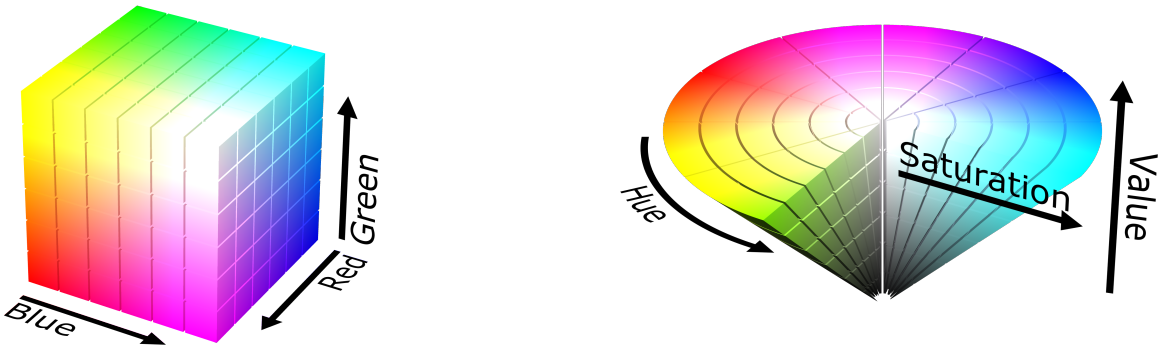
2.1 Color matching

The input for a Vision part is **RGB image Point Cloud image**, and the output should be **a list of poles with their color, distance, and angle from a robot**. It means that should be found a way to:

1. Distinct blue, green, and red colors reliably
2. Recognise a pole geometry with a certain color on the image
3. Find the distance and angle from a robot for every pole
4. Implement adaptive color recognition algorithm

For the first task, the **Adaptive Diagram Analysis** method is used consisting of image analysis and taking further action based on the results.

The aim is to recognize **blue, green, and red color** on the poles shown in Fig. 2.2. There are two color models available:



(a) RGB color model

(b) HSV color model

Figure 2.1: RGB and HSV color models

Parameters for RGB and HSV color models are the following:

RGB model	Value range	HSV model	Value range
Red	0 - 255	Hue	0 - 180
Green	0 - 255	Saturation	0 - 255
Blue	0 - 255	Value	0 - 255

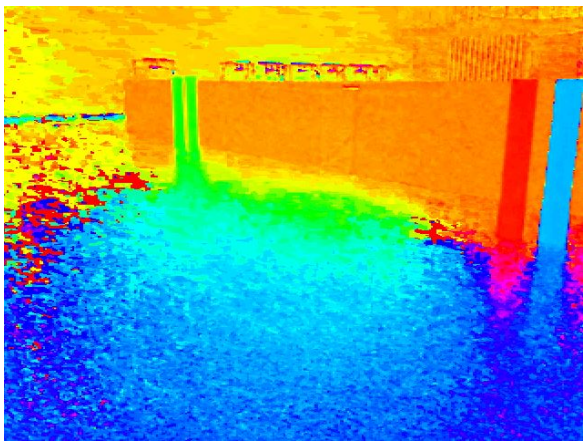
Table 2.1: Parameters for RGB and HSV color models

The **HSV color model** was chosen as it would give more control over colors to be recognized. Nevertheless, a Value parameter was not used at all.

Basically, **Hue** corresponds to the dominant length of the lightwave, **Saturation** corresponds to the color richness, and **Value** to its brightness, [4].

After we get the RGB image from a robot, we convert the image to the HSV color model and divide it into three channels: Hue, Saturation, and Value. Here is an example of how the image looks like:

(a) Original image



(b) Hue channel

(c) Saturation channel

(d) Value channel

Figure 2.2: The original image and its HSV channels

A blue and red pole is highly saturated, although a green one is not. That is what should be taken into account. Also, poles can be distinguished based on their hue values.

Let's analyze the color spectrum of the environment where a robot is supposed to be used. The idea is to take a **Hue-Saturation diagram** shown in 2.3, and then put a **Frequency Black-Transparent gradient mask** on it, which would show the frequency of color occurrence with certain hue and saturation on the current image. In other words, a black color would mean that the color with this hue and saturation does not appear at all on the image, while a transparent one would mean that it appears very often.

Also, we found out that this mask should be represented in a logarithmic scale, as otherwise, almost all of the diagram would be black due to the frequent occurrence of dark colors. This newly created diagram is shown in Fig. 2.4.

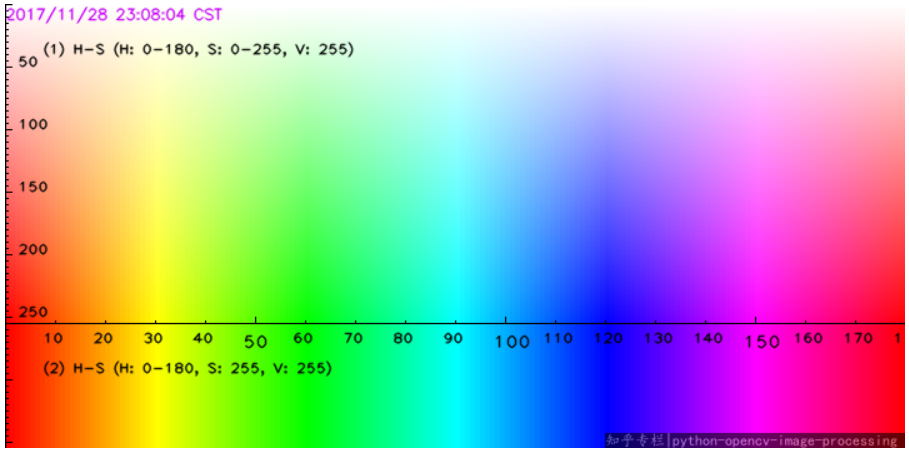


Figure 2.3: Hue-Saturation general diagram

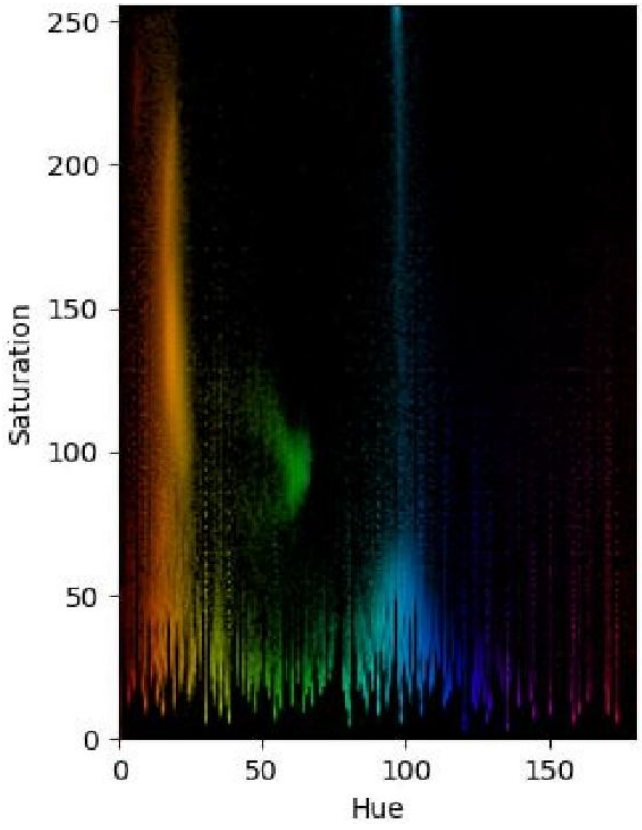


Figure 2.4: HS diagram after applying Frequency Black-Transparent gradient mask with Logarithmic scale

We defined the following color hue values defined:

$$\begin{aligned} \text{blue}_{\text{hue range}} &= [80, 130] \\ \text{green}_{\text{hue range}} &= [50, 65] \\ \text{red}_{\text{hue range}} &= [0, 4] \end{aligned} \tag{1}$$

They correspond to the blue, green, and red areas in Fig. 2.3. All images processed in this way give as a similar result as in 2.4 with the distinction that if there are colorful poles in the photo, this is reflected in the diagram by the corresponding peaks. The majority of objects are slightly saturated (what could be considered as a background). Nevertheless, some peaks in desired hue intervals appear to be our desired poles. For example, there is clearly a peak for a hue value of 100 with a saturation of over 230 for blue and a hue value of 50 with a saturation of over 80 for green.

The next sections describe how a **main hue value, hue deviation, and saturation threshold** is found for every desired color.

2.1.1 Saturation thresholds

As we can see from [5], color saturation varies greatly depending on the lighting in the space. Let's have a look at how saturated colors in the environment are. For this purpose, the Saturation-Frequency histogram was created, Fig. 2.5.

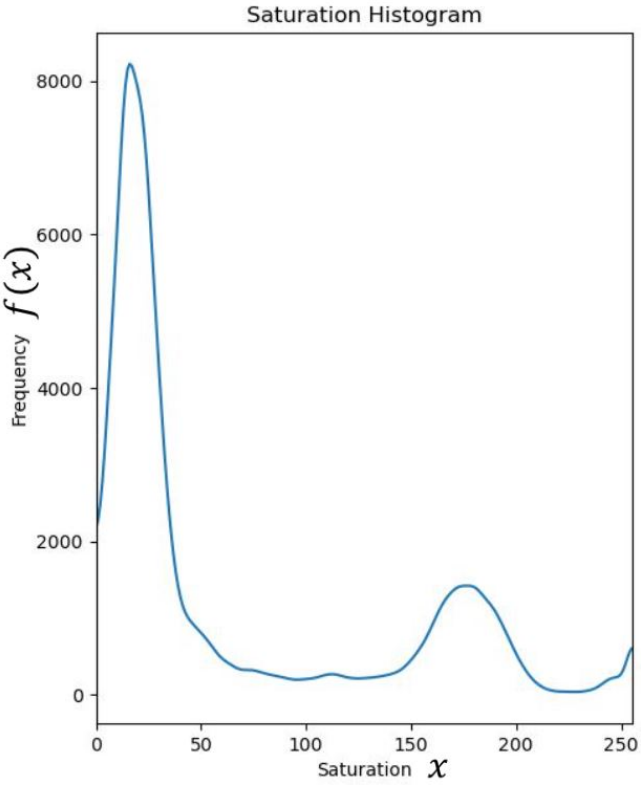


Figure 2.5: Saturation-Frequency histogram of the specific image

We applied a **Gaussian smoothing** to smooth a histogram with a $\sigma = 3$ using `gaussian_filter1d()` function from `scipy.ndimage` library. After comparing the histogram data for a large number of photos with the same light level, we made a conclusion that this is an approximate representation of our environment. For another light level, the histograms are also similar to each other and reproduce a similar pattern but shifted. Thus, if the methods we apply to recognize the poles used in this histogram, they would be universal for almost all images.

Basically, we use thresholding methods proposed in [4]. Our goal is to find a saturation threshold for blue, green, and red colors. Empirical evidence shows that corresponding thresholds can be found using these formulas:

$$\begin{aligned} \text{blue}_{\text{threshold}} &= \underset{x \in [45,235]}{\text{argmin}}\ f(x) \\ \text{green}_{\text{threshold}} &= \underset{x \in [45,235]}{\text{argmin}}\ f(x), \\ &\text{where } f(x) \text{ has the first local min at } x \\ \text{red}_{\text{threshold}} &= \underset{x \in [45,235]}{\text{argmin}}\ f(x) - \text{red\_saturation\_offset} \end{aligned} \tag{2}$$

The x and $f(x)$ correspond to the saturation and frequency respectively in Fig. 2.5. The $x \in [45, 235]$ part means, that the edges of the ranges are cut off due to its peaks. A high peak at the beginning indicates that there are many dark pixels (what we could refer to as the background). The offset for a red color was found empirically and is equal to red_saturation_offset = 50. Fig. 2.6. illustrates approximate positions of saturation thresholds for each color.

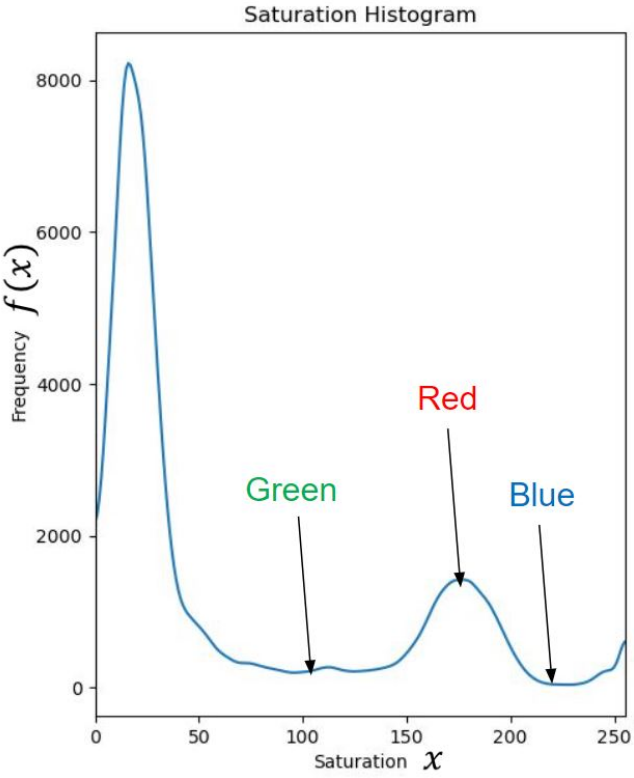
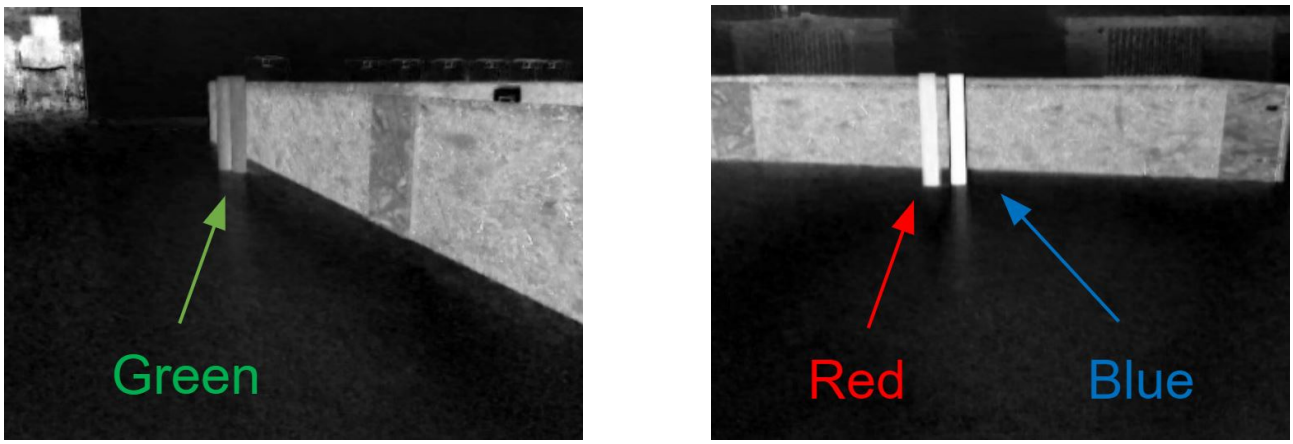


Figure 2.6: Saturation-Frequency histogram of the specific image with saturation thresholds for the corresponding colors

The reasons why we have chosen mentioned formulas are an observed environment and pole saturations. For a green pole, the saturation threshold is right after the saturation of the background. That is why we calculate the first local *argmin* on the histogram using an inversion of the current histogram and applying `find_peaks` function from `scipy.signal` library. But here arises a problem with a similarity to the saturation of wooden walls that were in the environment, Fig. 2.7. On the other hand, blue and red are among the most saturated objects, so the saturation thresholds for them were shifted much more to the right.



(a) A green pole

(b) A red and blue pole

Figure 2.7: Saturation of the image with a pole present

In a space illuminated uniformly, these thresholds would correspond approximately to the following values:

$$\text{blue}_{\text{threshold}} = 220, \qquad \text{green}_{\text{threshold}} = 90, \qquad \text{red}_{\text{threshold}} = 170 \tag{3}$$

2.1.2 Hue thresholds

Now with known saturation thresholds for each color from the previous section, we can find the hue value for every color. The normalized Hue-Frequency histogram is created, then smoothed using Gaussian smoothing, and all its peaks are considered to be peaks if they are above min peak = 2%. Every hue peak is checked whether its average saturation value on the image is higher than a saturation threshold for a certain color range mentioned in (1).

After that peaks are considered valid peaks and their weighted average with a weight as a frequency is calculated. This final value is the main hue value for the color hue range. For the majority of math operations, we use either `Numpy` or `math` library.

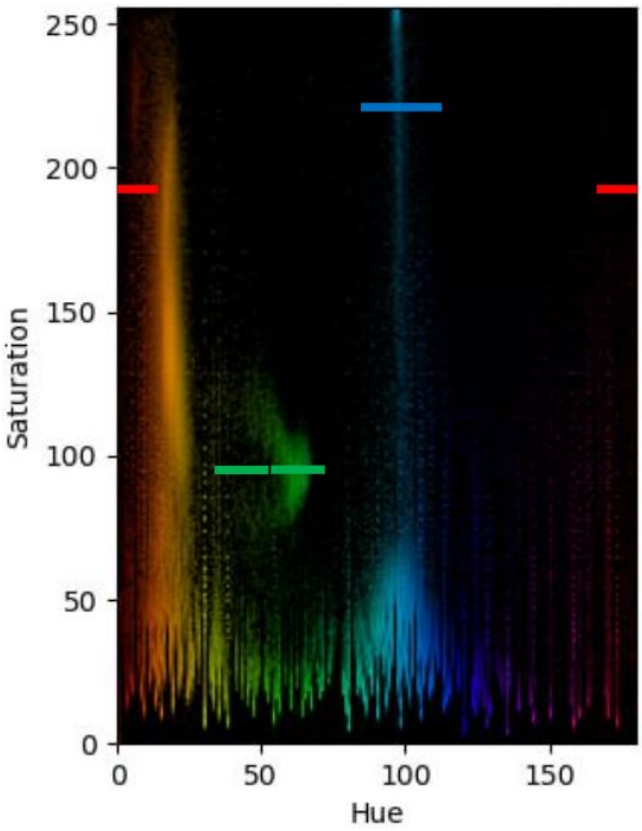


Figure 2.8: HS Frequency Brightness Color Space with bars corresponding to hue deviations and saturation thresholds

If there are no valid peaks, we calculate a mean value between argmin and argmax for the corresponding color ranges. Fig. 2.8 shows the HS Frequency Brightness Color Space after applying saturation thresholds and finding main hue values for each color.

Now all we have to do is to find a hue deviation for each color because a pole does not have the same hue value over its surface, [4]. Assuming that hue value distribution acts like a **Normal Distribution**, Fig. 2.9, the hue deviation value is calculated as the standard deviation of the [main hue - deviation range, main hue + deviation range], where deviation range is set to

$$\text{deviation range} = 50 \tag{4}$$

Do not confuse deviation range with hue deviation value, as the first stands for the range of color where the hue deviation value for a certain color will be calculated, while the second is the actual final hue deviation value for this color.

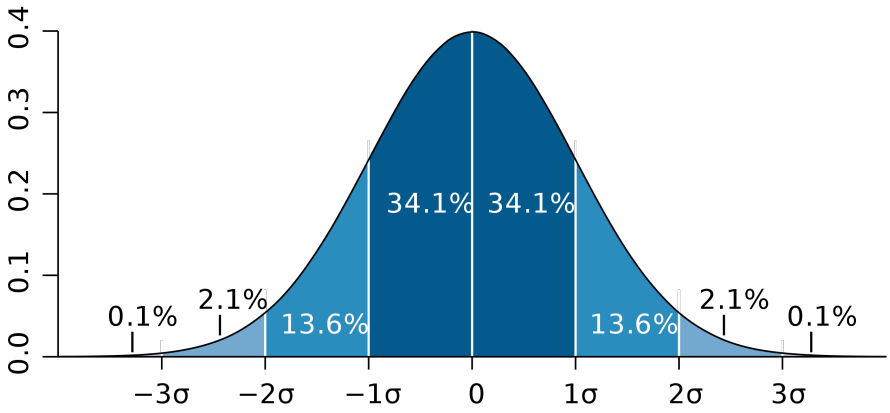


Figure 2.9: Standart Distribution diagram

In a space illuminated uniformly, final color parameters would correspond approximately to the following values:

Color name	Hue value	Hue Deviation value	Saturation value
Blue	114	14	220
Green	61	28	90
Red	2	8	170

Table 2.2: Approximate final color parameters in a space illuminated uniformly

2.2 Pole recognition

With the given color parameters for blue, green, and red colors, now the target moves to pole recognition of the corresponding color. In the pictures, they look like rectangles, so we will recognize them as rectangles. Let's look at the image example in Fig. 2.10.

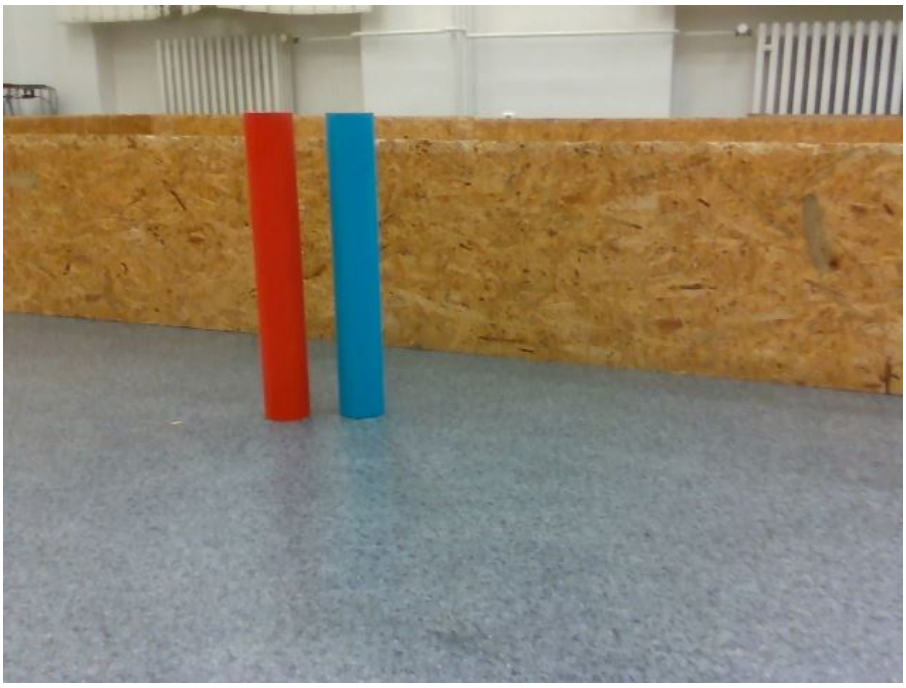


Figure 2.10: Original RGB image

2.2.1 Hue and saturation masks

Let's say we want to recognize a red pole in the image. For this purpose, the **Hue and Saturation mask** is created using `Numpy` arrays, [4]. They are combined after in a **Logical AND** condition: if a pixel is white in both masks, it is white in the **Combined mask**, Fig. 2.11, 2.12, 2.13.

Figure 2.11: Hue mask of red pole

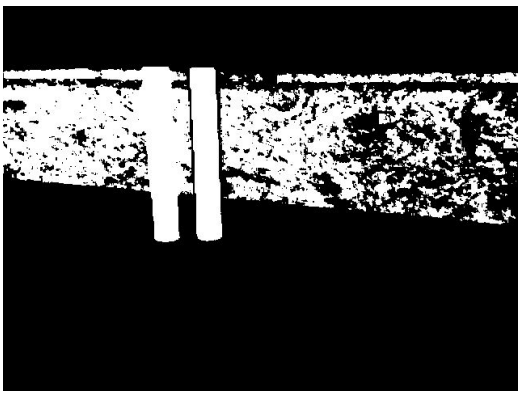


Figure 2.12: Saturation mask of red pole

Figure 2.13: Combined mask of red pole

As we can see, we achieved the task, although there are some unwanted tiny dots on the right side of the image highlighted with a red rectangle. It happens due to the inaccuracy of the Hue and Saturation mask. In this case, we could ignore that, but sometimes they create entire areas of white pixels that do not correspond to poles.

2.2.2 Rectangle Labeling and recognition

To remove white false areas, **mask area labeling** is used, [4]. The core of the implementation is `connectedComponentsWithStats()` function from `OpenCV` with arguments

connectivity = 4,
Itype = cv2.CV_32S.

If we process a Combined mask in Fig. 2.13 using the mentioned function and apply **aspect ratio and minimal area limiting conditions**, we can get a Labels mask shown in Fig. 2.14.

Figure 2.14: Masked labels on image

The aspect ratio is calculated as $\frac{\text{label height}}{\text{label width}}$ proportion and the acceptable range is [2, 8]. The acceptable range for the pole area is [2200, 70000] pixels, which would correspond to the pole area at a distance of [0.15, 1.5] meters from a robot camera to a pole according to empirical evidence.

Now we must recognize a rectangular shape on the mask and find out its parameters, such as:

1. Area
2. Center coordinates
3. Width
4. Height
5. Angle of rotation in an XY plane
6. Box points
7. Major points
8. Shape approximation
9. Distance
10. Position angle from a robot
11. Color

First of all, the Gaussian blur is applied on a label mask, Fig. 2.15, using `GaussianBlur()` function from `OpenCV` library with arguments

ksize = (7, 7),
sigmaX = 0.

Figure 2.15: Label mask with Gaussian Blur applied

After that, `findContours()` from `OpenCV`, [4], is used with arguments

mode = cv2.RETR_TREE,
method = cv2.CHAIN_APPROX_SIMPLE.

Then, for every contour found, the following lines are applied:

Python

```
peri = cv2.arcLength(cnt, True)
approx = cv2.approxPolyDP(cnt, peri * epsilon, True)
area = cv2.contourArea(approx)
num_vertices = len(approx)
x, y, width, height = cv2.boundingRect(approx)
```

Code 2.1: Parameters found for each contour

where epsilon = 0.02. The approximation of the polygon is given as an output. Further implementation is focused on limit conditions checking, such as:

- Is a polygon self-intersected?
- Is a number of vertices in a range [4, 6] inclusive?

Empirical evidence showed that this range of number of vertices is optimal and the most robust.

2.2.3 Pole distance and angle calculations

The next step is to calculate major rectangle points (Center, Top, Bottom, Left, Right, Top-Left, Top-Right, Bottom-Left, Bottom-Right) that are placed not on the edges of the rectangle, but with an offset from its edges. These points will be used to find a pole color and a distance from a robot.

We use a Point Cloud image to find the distance from a robot to a pole, [6]. The color corresponds to the distance to a point from the camera, the warmer the color, the closer the point is, Fig. 2.16.

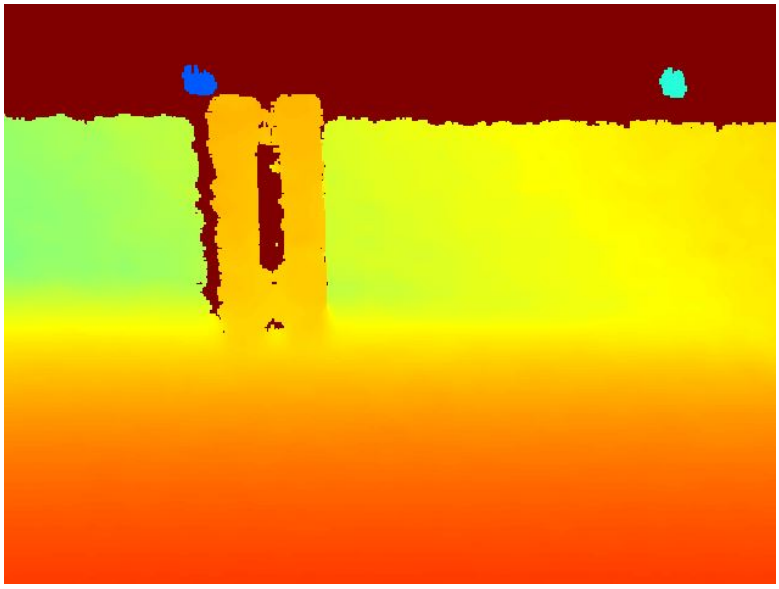


Figure 2.16: Colorful Point Cloud image

Only Center, Top, and Bottom major points are used for a final distance value. For each of these points, we take the distance from a Z axis of the Point Cloud and a horizontal position relative to a robot from a Y axis.

Also, such things as a camera offset and elevation are taken into account, see the section 2.5. After that, taking into account the pole radius of $2.5cm$, we calculate the ϕ angle on the horizontal axis of the pole position relative to the robot by the formula

$$\phi = \arcsin \frac{Z}{Y} \tag{5}$$

where Z is a Z component, Y is a component Y of the Point Cloud for a certain pixel.

In addition, all error checks for the arcsin, NaN, None values, and other errors are implemented. Furthermore, there is a checking whether one of the three distance points shows a very different distance value from two others. This difference threshold is set to be 10 cm. If so, this point is removed from a list with distance values. After that, we calculate a mean value between the remaining points. The output is the distance value z for a certain rectangle.

If for some reason the distance or the angle was not obtained, the pole is not included in the final list of recognized poles.

2.2.4 Final limit conditions and pole drawing

Finally, the last limiting conditions are imposed on the recognized poles:

- Is the pole aspect ratio in a specified range?
- Is the pole distance from a robot in a specified range?
- Is the pole area in a specified range?

For multiple distance ranges, the appropriate aspect ratio and area ranges are applied, which effectively prevents the recognition of false poles in the environment.
Every recognized pole is drawn on a RGB image using methods from `OpenCV`, Fig. 2.17.

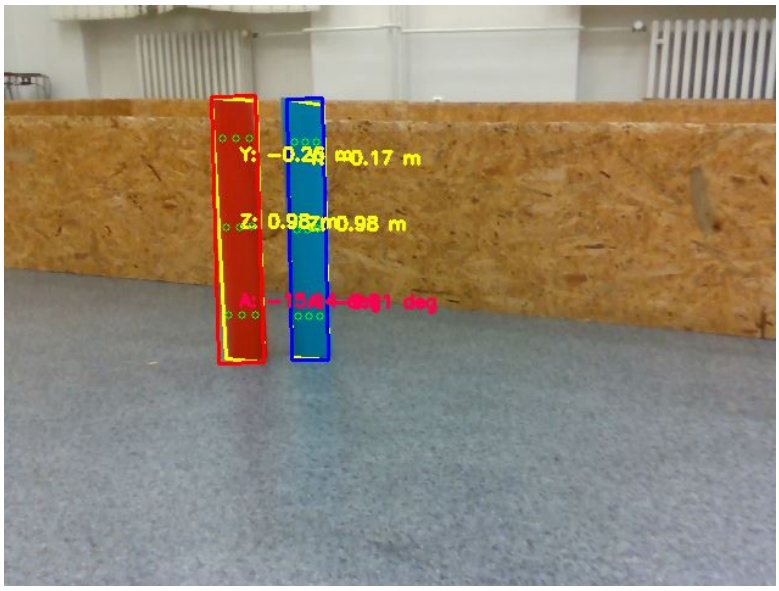


Figure 2.17: RGB image showing final recognized poles and their distances, angles, and colors

The final output is the following:

a list of poles z_t^i, where

$$z_t^i = [r_t^i, \phi_t^i, s_t^i] \tag{6}$$

r is a distance, ϕ is an angle, s is color of the i-th pole z at image t.

2.3 Color Calibration (not inc. in the final implementation)

The first idea was to implement a **Color Calibration** at the beginning of the ride. The robot had to turn around 12 times at $\frac{\pi}{6}$ intervals with pauses, take a photo during the pause, find the color parameters for each photo, then calculate their mean values, and use them as final color parameters for the rest of the ride. We implemented such an algorithm, but there were several reasons to abandon this idea and use a **Real-time Color Adaptation** instead, described in the section 2.4.

Using Color Calibration, the robot took about 15 seconds at the beginning to calibrate, which added time to the overall ride. And the solution would not be adaptive, which means that the robot would recognize objects only for the light conditions in which it was calibrated. These are the two main reasons why we moved to Real-time Color Adaptation.

2.4 Real-time Color Adaptation

The main motivation for the following algorithm is to make a robot adapt to different light conditions, even if they change during a ride. Also, it would mean that we don't require a Color Calibration at the beginning of the ride, meaning we would not waste any additional 15 seconds for such a process.

To implement this, we created special queues for hue, hue deviation, and saturation for each color using `deque` from `collections`. These queues can contain up to 4 items each, and every newly added element removes the earliest added one.

At the beginning, these queues are empty. At this time, the robot cannot recognize anything, while the number of elements in a queue for each parameter and color is not equal to 4. These first 4 color parameters are taken obviously from an environment using algorithms, described earlier. During a ride, the queues are updated every time a robot takes a photo.

But now if a robot recognizes a pole, the color parameters are retrieved directly from this pole and added to a queue instead of color parameters from an environment. In other words, a robot takes into account the last color values from an environment and actual color values from previously recognized poles during its ride with the second being a priority.

The algorithm calculates color values of already recognized poles at their major points and then averages them, giving the final color values for the pole color. The final color parameters are the mean values of 4 values in each queue.

If a robot does not recognize a pole, a mean value of maximum of 100 previous color measurements from the environment is added in a queue.

The main advantage of this solution is the ability to adapt to the light conditions of the environment and that a robot quickly achieves the appropriate color parameters at the beginning of the ride (approx. 3 seconds).

As an example, a robot can be launched in bright conditions, then the room will become dark, and it will adapt to the new values in a few seconds, Fig. 2.18, Fig. 2.19. And vice versa, after launching in a dark room, a robot adapts to brighter conditions.

(a) RGB image with detected poles (b) Combined mask for poles

Figure 2.18: Pole recognition in bright conditions

(a) RGB image with detected poles (b) Combined mask for poles

Figure 2.19: Pole recognition in dark conditions

2.5 Camera offset and elevation

- **Camera offset**
  An RGB and Point Cloud cameras are not in the center of the robot, but a little bit shifted to the sides on a horizontal axis, [6]. It means that the actual Y component of the pixel placed in the center of the Point Cloud is not aligned with a real value. The offset for the Y component was calculated empirically and is equal to -4 cm.

- **Camera elevation**
  Also, the camera is elevated to the height of the pole (approximately 35 cm), which means that it measures greater distances to the Top, Center, and Bottom major points than they actually are. The scheme is shown in Fig. 2.20,

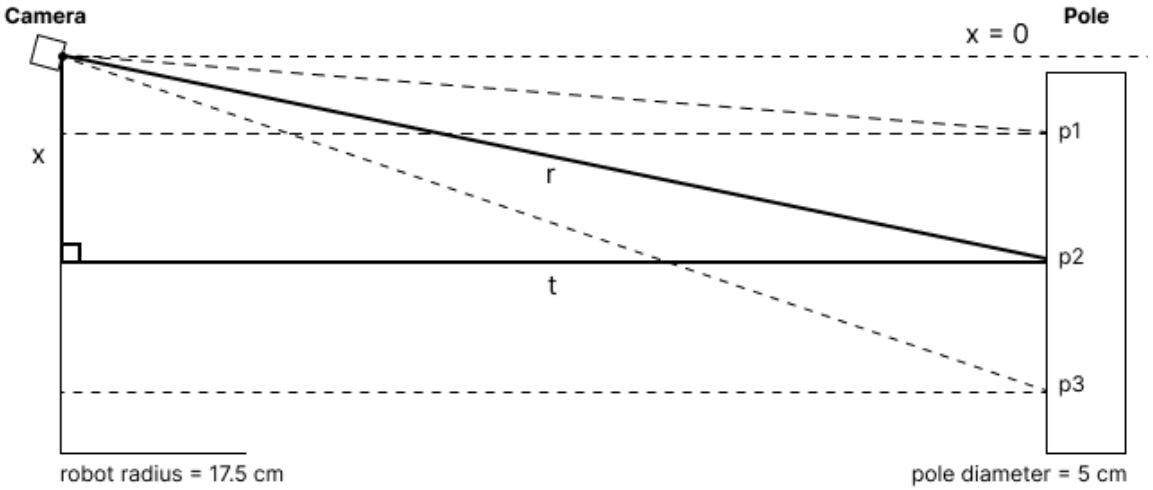


Figure 2.20: Elevated camera scheme for calculating true distance

where r is the measured distance to a major point, x is the point elevation relative to zero level ($x = 0$), t is the actual distance from a camera to a point. After that, we calculate the actual distance t using the Pythagorean theorem

$$t = \sqrt{r^2 - x^2} + \text{pole radius} \tag{7}$$

where a pole radius is $2,5$ cm.

After an actual distance t is found for the Top, Center, and Bottom major points, they are further processed as described in the section 2.2.3.

2.6 Extended Kalman Filter

To solve the problem of simultaneous localization and mapping we used an Extended Kalman filter. The core idea for implementation was taken from [7] the version of EKF with unknown correspondences. After some experiments, the prediction step of the algorithm was changed.

The robot didn't respond well to a velocity-based control model. For example, after putting in instructions of going at the speed of $1\,\mathrm{m\,s^{-1}}$ for $1\,\mathrm{s}$, the robot only moved around $0.89\,\mathrm{m}$, so the error was immense. We decided to use an odometry-based control system. Telling the robot

to go forward until the odometry showed 1 m had very good accuracy, and all the integration errors would be eliminated by the EKF anyway.

The prediction step (steps 4-6 in the book) was modified to use odometry changes from the last cycle $(\Delta x, \Delta y, \Delta\theta)$, to update the robot's state vector. Specifically, the state vector μ_t for the robot's position and orientation at time t is computed as:

$$\mu_{t,1:3} = \mu_{t-1,1:3} + \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{pmatrix}$$

The Jacobian used for updating the covariance matrix is given by:

$$G = \begin{pmatrix} 1 & 0 & -\Delta x \sin \mu_{\theta,t-1} - \Delta y \cos \mu_{\theta,t-1} \\ 0 & 1 & \Delta x \cos \mu_{\theta,t-1} - \Delta y \sin \mu_{\theta,t-1} \\ 0 & 0 & 1 \end{pmatrix}$$

The measurement step (steps 8-25) stayed as is, with the color of the pole used as signature s_t^i and being represented as a number, red - 0, blue - 1, green - 2.

Parameters used for EKF

- Covariance matrix for the robot position at time $t = 0$: $\Sigma_0 = 0.001 \cdot I$
- Covariance matrix extension for a new obstacle: $\Sigma = 100 \cdot I$ (not sure how correct this is)
- Process noise covariance: $R = \begin{pmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & 0 \\ 0 & 0 & \sigma_\theta^2 \end{pmatrix} = \begin{pmatrix} 0.00001 & 0 & 0 \\ 0 & 0.00001 & 0 \\ 0 & 0 & 0.00001 \end{pmatrix}$
- Measurement noise covariance: $Q = \begin{pmatrix} \sigma_r^2 & 0 & 0 \\ 0 & \sigma_\phi^2 & 0 \\ 0 & 0 & \sigma_s^2 \end{pmatrix} = \begin{pmatrix} 0.05 & 0 & 0 \\ 0 & 0.00872665 & 0 \\ 0 & 0 & 0.00001 \end{pmatrix}$, where 0.00872665 is half a degree.

Because EKF can't process negative measurements (obstacle not being in the place where it expects it) only obstacles that were measured enough times (for our case 3 times) were considered in the robot's program.

2.7 Checkpoint Placement

The environment is initialized with 4 checkpoints for the robot to make a full turn and find the closest pair of obstacles.

Every cycle, every pair of red-blue and green-green obstacles are processed. For close red-blue pairs the main checkpoint is calculated from their position and the position of the robot to arrive 5 cm before the poles considering the robot's radius. Another 5 checkpoints are added to continue the exploration after reaching the checkpoint. That is, to move to the side of the red pole, move 30 cm forward, and look left and right with the look around angle ($\frac{\pi}{8}$). After that, the robot turns forward again so that if it hasn't seen any obstacles, it will continue exploring in that direction. You can see a diagram from a simulation run on Fig. 2.21.

For close green-green pairs only one final checkpoint is added and the program stops adding any new checkpoints.

Because of the better approximation of the pole positions, the positions of checkpoints for pairs are adjusted every cycle. And if the current checkpoint moves away from the path for

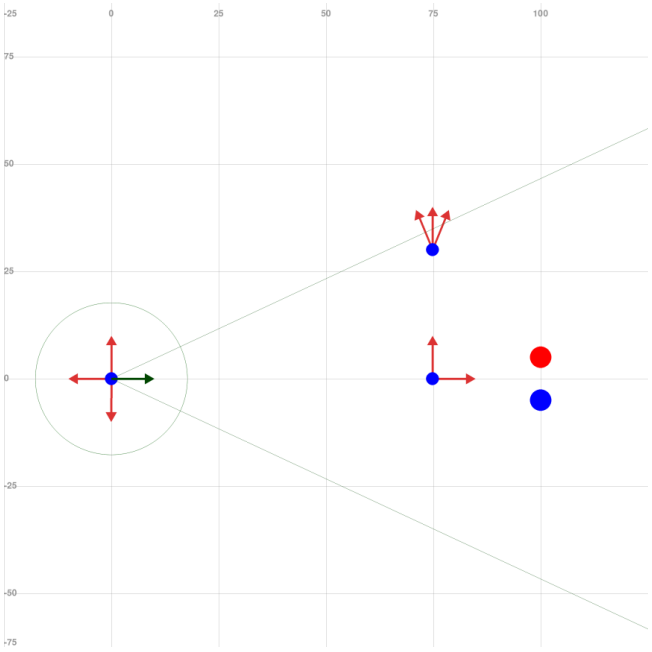


Figure 2.21: Checkpoint placement

more than 2 cm or the path intersects any obstacles, the whole path is generated again. Which isn't very efficient and we will talk about that in the Improvements 5 chapter.

If the robot doesn't have any checkpoints to go through and the finish isn't reached it adds a set of checkpoints for exploration in front of itself at a dynamic distance so that it doesn't set new checkpoints into obstacles. Fig. 2.22.

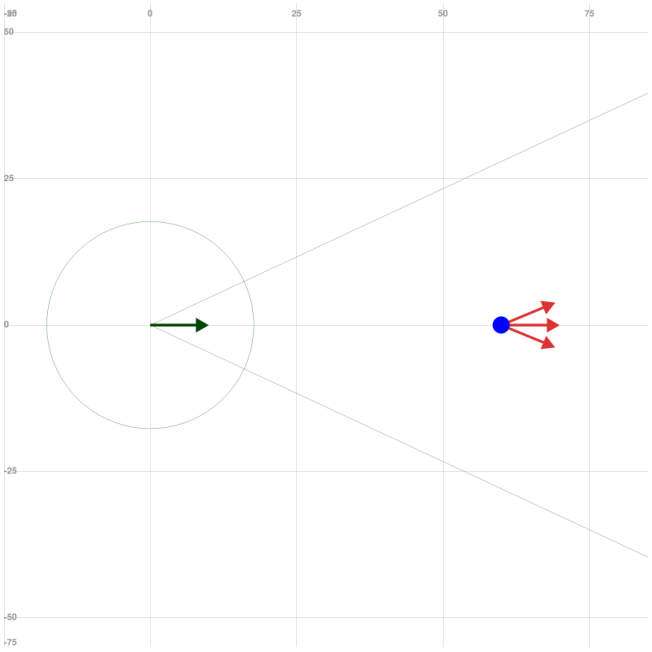


Figure 2.22: Exploration checkpoints

2.8 Path Creation

The path is generated either by drawing a straight line if possible, or using A* taken from [8] which uses a grid of cells size 3x3 cm. It is made to have a clearance around obstacles of 6 cm. After creation, the path goes through some post-processing steps described in subsections 2.8.1-2.8.3.

2.8.1 Simplification

Examples of path simplification: Fig. 2.23 - 2.24.

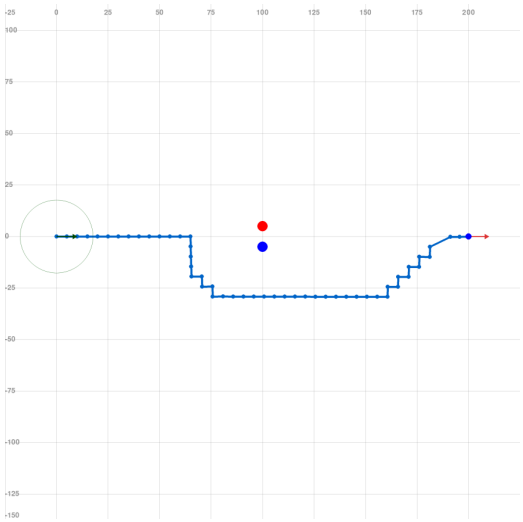


Figure 2.23: Path generated with A*

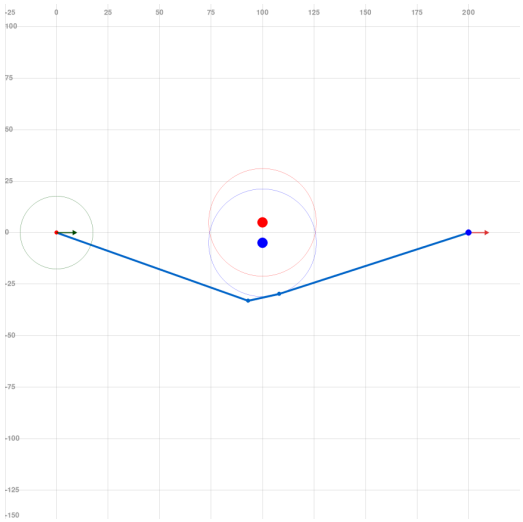


Figure 2.24: The path after being simplified from both directions

Simplification is done using this function:

Python

```python
def simplify_path(path):
    if len(path) == 2:
        return path
    simplified_path = [path[0]]
    i = 0
    while i < len(path) - 1:
        j = len(path) - 1
        while j > i + 1:
            if straight_path_exists(path[i], path[j]):
                i = j
                break
            j -= 1
        simplified_path.append(path[i])
        i += 1

    if simplified_path[-1] != path[-1]:
        simplified_path.append(path[-1])

    return simplified_path
```

Where `straight_path_exists()` checks if a straight line exists between two nodes that don't collide with any obstacles. In the implementation itself, there is also a second step after this one, the path is injected with nodes and simplified again but by going from the opposite direction so that it will be more symmetrical.

2.8.2 Node injection

Every 2 cm a node is injected through the path, so the next step of Bézier curve interpolation doesn't smooth out the path too much. This step also adds a node after the checkpoint for a more robust angle to the node at the end. Which you can see near the checkpoint in Fig. 2.25

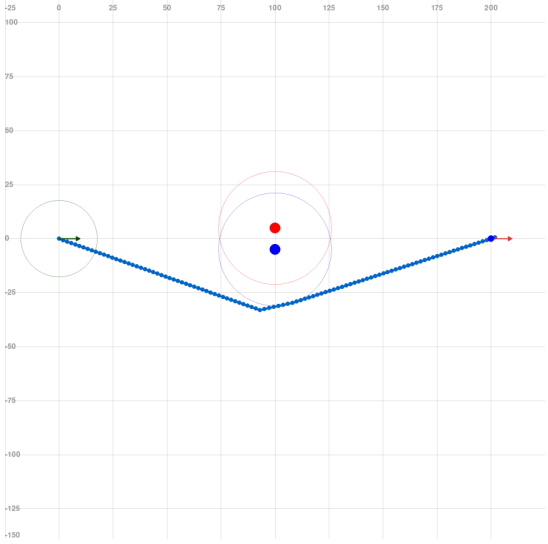


Figure 2.25: Path with injected nodes

2.8.3 Bézier Curve Interpolation

For smoother curvature change during path execution, Bézier Curve Interpolation is used.

Bézier Curve Interpolation is done using this function:

Python

```
def bezier_curve_interpolation(self, path, frequency):
    def bezier_curve(control_points, num_points):
        t = np.linspace(0, 1, num_points)
        n = len(control_points) - 1
        curve = np.zeros((num_points, 2))
        for i in range(num_points):
            point = np.zeros(2)
            for k in range(n + 1):
                binomial_coefficients = comb(n, k)
                term = binomial_coefficients * (t[i] ** k) * ((1 - t[i]) **
                    (n - k)) * np.array(
                     [control_points[k].x, control_points[k].y])
                point += term.reshape(2)
            curve[i] = point
        return [Node(x, y) for x, y in curve]

    total_length = self.calculate_path_length(path)
    num_points = int(total_length // frequency)

    smooth_path = bezier_curve(path, num_points)
    return smooth_path
```

The function will try to keep the frequency at the specified number (for us 2 cm) even though at curved parts the density will be higher than at straight parts. Fig. 2.26.

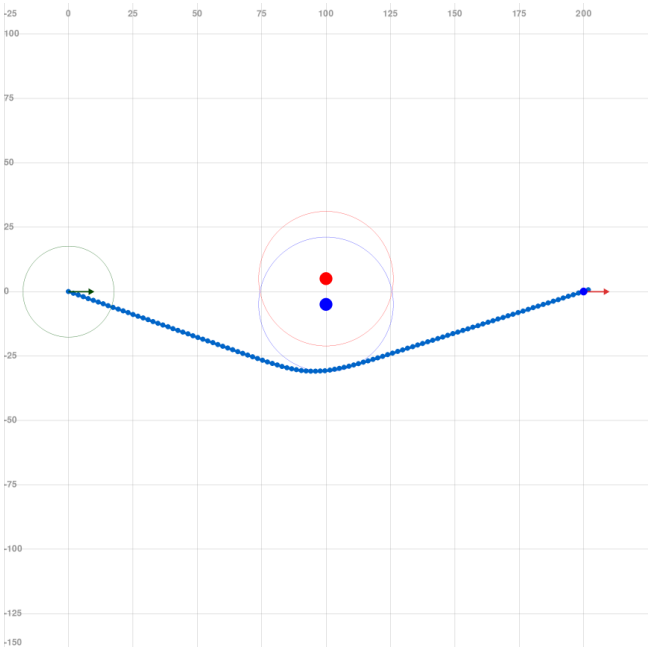


Figure 2.26: Path after Bézier Interpolation

With all of these steps, the finished path is still very accurate in terms of avoiding obstacles, and with very low peak curvature. You can see an example in Fig. 2.27.

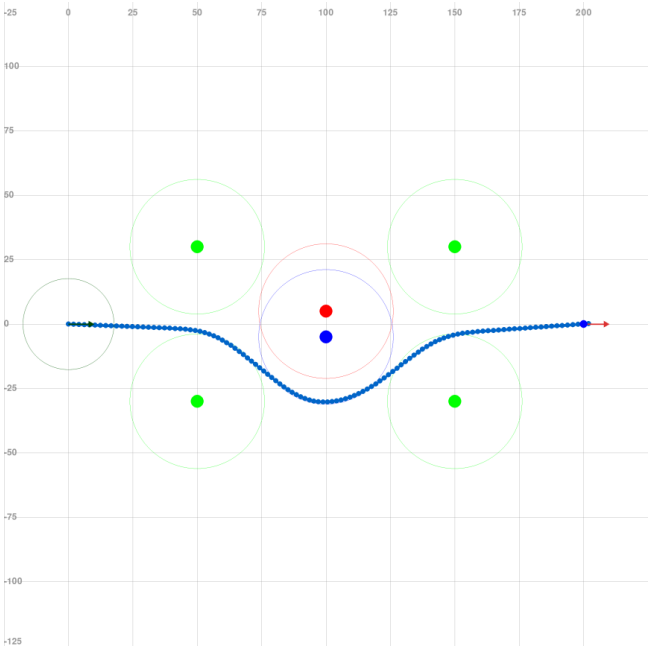


Figure 2.27: Path through a complicated obstacle pattern

2.9 Path Execution

The program is moving through the path using the Pure Pursuit algorithm, taken from [9], with certain changes. Another way of choosing the look-ahead distance was implemented. And unlike in that paper, our robot didn't have a predefined speed profile for the path so the linear speed was adjusted dynamically depending on the look-ahead distance.

2.9.1 Look-ahead Distance

The goal is to make the look-ahead distance shorter as the robot approaches a curvier part of the path, and longer if the path in front of it is straight. Here's our version of the algorithm that achieves that:

1. Find a point on the path that is closest to the robot.
2. Move it through the path, for each point calculate the angle difference it would make with the robot.
3. Calculate the allowed look-ahead distance for that angle difference using equation 2.1.
4. If the point is inside a new look-ahead distance set it as a new look-ahead point, otherwise, break the loop and use the previous point.

The equation for allowed look-ahead distance:

$$d = \begin{cases} d_{max} & \text{if } |\phi| < \alpha \\ \frac{d_{min} - d_{max}}{\frac{\pi}{2}^p} |\phi|^p + d_{max} & \text{if } \alpha \leq |\phi| \leq \frac{\pi}{2} \\ d_{min} & \text{if } |\phi| > \frac{\pi}{2} \end{cases} \tag{2.1}$$

where:

- ϕ is the difference that the point would make with the robot's orientation.
- d is the distance allowed for look-ahead point propagation.
- d_{max} is the maximum possible look-ahead distance. (0.4 m)
- d_{min} is the minimum possible look-ahead distance. (2.5 cm)
- p is the power for interpolation ($\frac{1}{10}$).
- α is the offset until which the distance stays at the maximum value. (0.02 rad)

Take note, that all of these values should be adjusted for your robot specifically, depending on its physical parameters. You can see the values that worked for us to the right of every variable. With all of that, we get a function that can be visualized in polar coordinates if we set $\theta = \phi$, $r = d$. We get the allowed look-ahead distance around the robot visualized, Fig. 2.28.

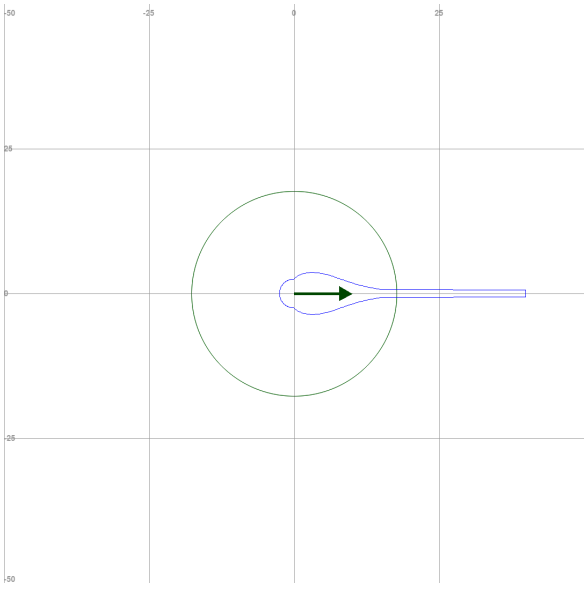


Figure 2.28: Allowed look-ahead distance depending on angle from the robot's orientation

You can see how the look-ahead distance looks in practice, when the robot is making a turn Fig. 2.29 and when it goes through a straight part of the path Fig. 2.30.

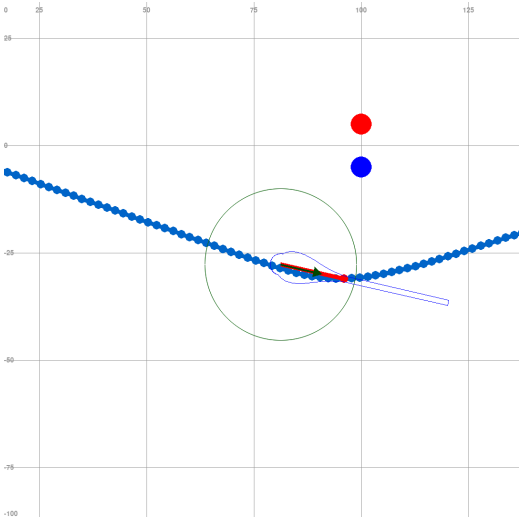


Figure 2.29: Curved part of the path

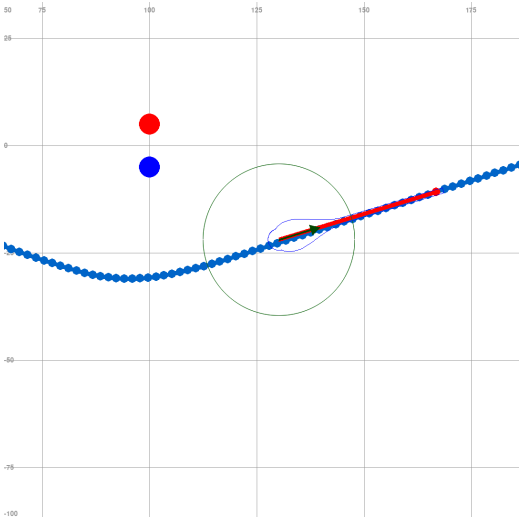


Figure 2.30: Straight part of the path

2.9.2 Linear Velocity

Function for the allowed look-ahead distance worked well, for having a good resolution of angular velocities, along the path. But we also wanted the robot to respond to the path curvature or the fact that it's ending, with its linear velocity. Making a linear relation between look-ahead distance and u caused the robot to slow down a lot while making turns. So another interpolation was used, which is described in equation 2.2. It makes the robot only slow down when the LD falls drastically, which you can see in Fig. 2.31. So this works great for arriving at checkpoints and going through turns. This could be further improved if we allow the robot to accelerate faster and then raise the power of the interpolation so it will move on average bigger speed.

The equation for calculating linear velocity from the look-ahead distance:

$$u = \begin{cases} u_{min} & \text{if } d < d_{min} \\ \frac{u_{min}-u_{max}}{(d_{min}-d_{max})^p}(d-d_{max})^p + u_{max} & \text{if } d_{min} \leq d \leq d_{max} \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

where:

- d is the look-ahead distance set by equation 2.1.
- d_{max} is the maximum possible look-ahead distance. (0.4 m)
- d_{min} is the minimum possible look-ahead distance. (2.5 cm)
- p is the power for interpolation. (3).
- u_{min} is the lowest allowed linear velocity. ($0.03\,\mathrm{m\,s^{-1}}$)
- u_{max} is the highest allowed linear velocity. ($0.5\,\mathrm{m\,s^{-1}}$)

You can see the values for our case on the right of the variable explanations. If we set $x = d$ and $y = u$, we get a graph in Fig. 2.31.

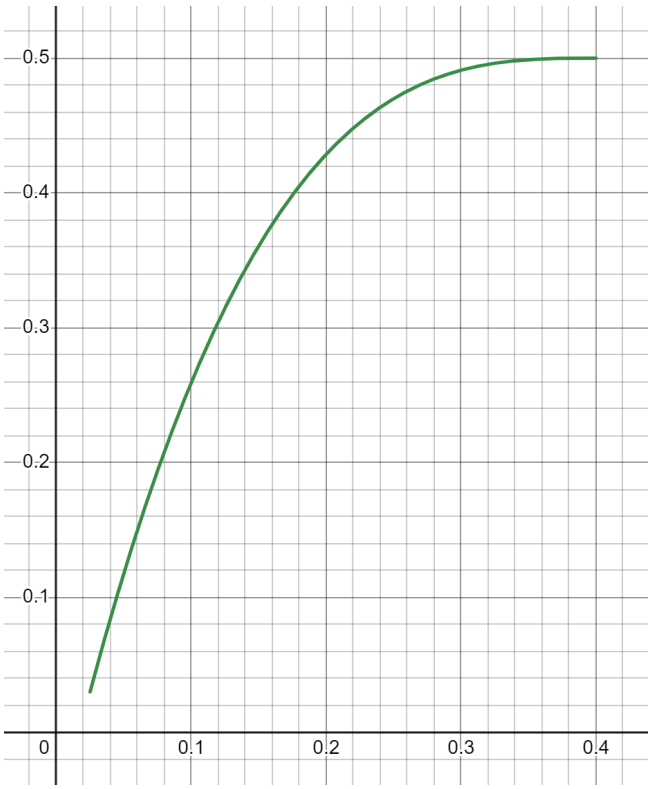


Figure 2.31: Relation between LD and u from equation 2.2

Increasing the power of interpolation would result in higher average speed, especially on turns. But if the robot doesn't have big enough acceleration, it will have trouble stopping before the checkpoint, and would often go past it.

After those two steps the angular velocity is calculated from linear velocity by the usual equation for Pure Pursuit which you can find either in the original paper it was described in [10] or in the paper we took it from [9].

Chapter 3

Simulation and Visualization

As you may have seen in the previous chapter, there also is a visualization written using `pygame`, to display the state-space when a real robot is moving, or to play a simulation where the movements and measurements are being computed from the virtual environment. You can see a demo of it on our Git Hub for the project. Here's one of the frames from that demo, Fig. 3.1.

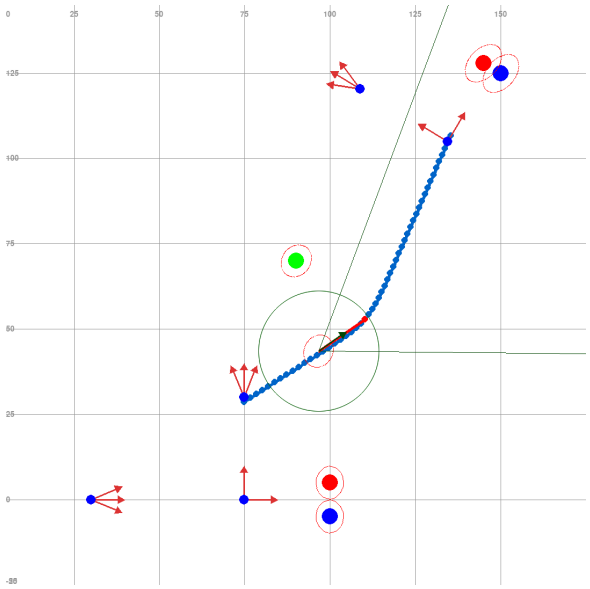


Figure 3.1: A frame from a simulation demo

- **Red Ellipses**. These are calculated from the covariance matrix and represent a 95% certainty that an object is inside that ellipse.
- **Checkpoints**. Blue circles with red arrows show the checkpoints along the robot's path. They do have an orientation, so when the robot arrives at them, it turns to match that orientation.
- **Path and Look-ahead Point**. Small blue circles represent the current path to the checkpoint, and a red dot on it with a line is the look-ahead point.
- **Robot**. You can't see it in this picture but there are two robots there, one is "real" for the simulation and the other is an estimate taken from the EKF.
- **Obstacles**. Big red, blue, and green circles are the poles that mark checkpoints or are just obstacles for the robot to avoid.

Chapter 4

Summary

This report details the successful development and implementation of a navigation system for a robot designed to follow a course marked by colored poles, fulfilling all requirements. The robot's environment mapping utilized the Extended Kalman Filter (EKF), and Adaptive Diagram analysis was used for pole color recognition. Traditional computer vision techniques identified pole shapes, while the system adapted to varying light conditions. Simulation and visualization components aided development. Path planning was achieved using the A* algorithm with Bézier curves interpolation, and path tracking employed the Pure Pursuit algorithm with adaptive look-ahead distance and linear velocity. The solution proved to be effective and robust, with room for future improvements.

Chapter 5

Possible Improvements

- **More precise definition of color parameters**
  Currently, the field recognition is quite accurate and works almost every time, although sometimes the robot is still unable to recognize them due to inaccurate saturation, key color value, or deviation. It would be possible to better select the parameters for the algorithm by testing the result or to implement more advanced algorithms than histograms.

- **Machine Learning for recognition of different objects**
  Instead of a limited solution focused only on pole recognition, the implementation of ML solution would be more flexible and capable of recognizing not only well-defined geometric shapes, but also other various objects and their placement.

- **More abstract checkpoint placement**
  We can't change the way the first checkpoint is placed in front of the two poles because the robot must go through it, but further exploration could be described more abstractly. As well as making the robot arrive on the axis perpendicular to the poles axis, so that the precision of the final position is better.

- **Better path planning algorithms**
  As was mentioned in the checkpoint placement section 2.7, the path is fully scraped if it's too far from the goal or if it collides with any obstacle. This could be fixed for example with the use of the D* Lite algorithm, which only adjusts the path when new obstacles are measured.

- **Binary occupancy grid**
  Using D* Lite would ask for having a binary occupancy grid, which would be a good solution considering we only use the depth camera to find out the distances to the poles, but we could use it to build a whole 2D or 3D map of the environment. But this approach would complicate the project tenfold, so only obstacles and their positions for now.

- **Pure Pursuit counterparts**
  Using Pure Pursuit does yield a lot of benefits, but the amount of constants and over-complications is huge. We were scared to use something like a PID regulator because that would need a lot of real-world experiments with the robot itself, or even better, writing a differential equation for its movement, but who knows, maybe it would have provided a better solution.

References

[1] GitHub. *GitHub Repository*. 2023. URL: https://github.com/Marchell0o0/LAR.

[2] P. Krsek. *Úvodní informace*. https://cw.fel.cvut.cz/wiki/_media/courses/b3b33lar/krsek_lar24_uvod_cs.pdf. 2023.

[3] Pavel Krsek et al. *Jízda podle směrovek*. 2023. URL: https://cw.fel.cvut.cz/wiki/courses/b3b33lar/tasks/signpost_cs.

[4] P. Krsek. *Metody zpracování dat z RGB kamery*. https://cw.fel.cvut.cz/wiki/_media/courses/b3b33lar/krsek_lar24_cs.pdf. 2023.

[5] Koen van de Sande, Theo Gevers, and Cees Snoek. "Evaluating Color Descriptors for Object and Scene Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.9 (2010), pp. 1582–1596. DOI: 10.1109/TPAMI.2009.154.

[6] V. Petrik. *Metody odhadování hloubky*. https://cw.fel.cvut.cz/wiki/_media/courses/b3b33lar/petrik_lar24_depth_en.pdf. 2023.

[7] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. Intelligent Robotics and Autonomous Agents series. MIT Press, 2005. ISBN: 9780262201629. URL: https://books.google.cz/books?id=2Zn6AQAAQBAJ.

[8] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[9] Team 1712. *Implementation of the Adaptive Pure Pursuit Controller*. Tech. rep. 2018. URL: https://www.chiefdelphi.com/t/paper-implementation-of-the-adaptive-pure-pursuit-controller/166552/1.

[10] R. Craig Coulter. "Implementation of the Pure Pursuit Path Tracking Algorithm". In: CMU-RI-TR-92-01 (Jan. 1992). URL: https://www.ri.cmu.edu/publications/implementation-of-the-pure-pursuit-path-tracking-algorithm/.