Lab 5: Maze Challenge

January 9th, 2022

Award & Ceremony From all submissions, we will select the three best solutions. The selection will not solely be based on speed, but we will also consider other criteria, such as creativity, complexity and coding style. We will then present the performance in the maze during the Q&A session **on February 3rd**.

The winner will be decided by a live popularity voting of all participants, and your names will be added to a name plate on the trophy and displayed permanently at our institute.

Part I: The dreaded color sensor

In previous labs, you have already experienced that the detection quality of the Lejos color sensor is poor. Although it is a low-cost RGB sensor, there is still room for improvement as we have so far only relied on Lejos' built-in color detection algorithm.

In the maze you have to distinguish between the four colors RED, GREEN, BLUE and YELLOW, according to the colored sheets you have been given in the previous exercises. Therefore, you should implement your own color detection algorithm which is tailored to the given conditions, so that the detection reliability and range are improved.

To help you with this task, we carried out some preliminary analysis on the raw RGB data and provide a set of examples/tutorials for you[1]:

- Use Lowpass filter for noisy input data (LowpassFilter Introduction)

[1]Make sure to run the command `git pull upstream main` in your repo to get the materials.

- Java Lowpass Implementation (AbstractFilter Tutorial)
- An example (Figure 1a) of the data analysis for the filtered RGB intensity values of our four given color sheets.

Your custom color detection algorithm should only consider/distinguish between the colors of the sheets that are provided to you during the labs:

- RED (magenta/red-ish)
- GREEN
- BLUE
- YELLOW
- all other colors are considered "NONE"

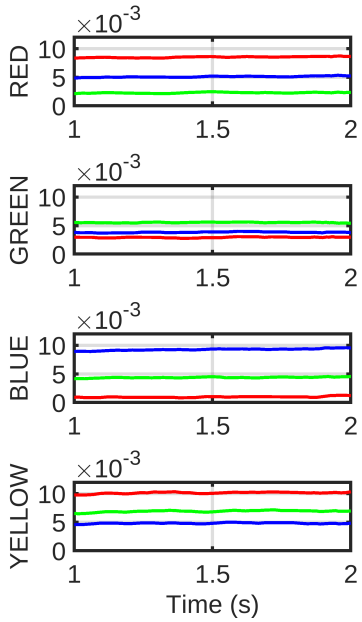
The final outcome of this task is a test program named `ColorDetectionTest.java` that runs your custom color detection in an infinite loop. The current distance and detected color should be printed on the LCD so that the performance of your color detector can be evaluated on the real robot.

Hints:

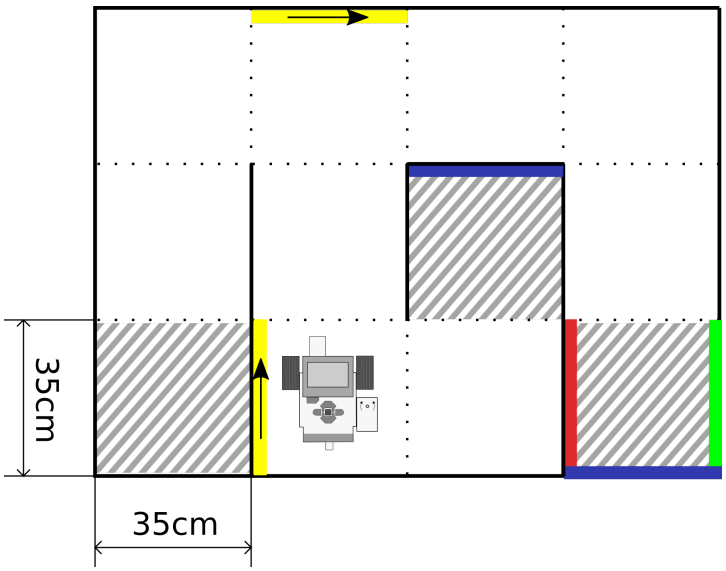
- You will have to access the RGB mode of the color sensor in order to get the raw RGB values (normalized between 0 and 1).
- You can start by implementing a simple algorithm that detects the colors `RED`, `GREEN`, and `BLUE` based on the largest component of the RGB values. To detect the color `YELLOW` and differentiate with `RED`, you need to identify the second largest RGB component.
- Test your algorithm for all four color sheets at distances between 5 – 10 cm for an initial analysis.
- You need to modify your algorithm such that it does not misdetect colors in the vicinity of the maze walls. For this purpose, it is recommended to apply a low-pass filter to the RGB signals.
- In the maze challenge (see next section) you should optimize your color detector so that it detects the colors "GREEN" and "YELLOW" robustly. It may sometimes recognize walls as "RED" or "BLUE" color.
- Also think of further enhancements to improve the robustness of the color detector, such as adding thresholds for the minimum RGB intensity or the operating range (coupling with the ultrasonic distance).

Part II: Escape from the maze

In the final challenge, we will place your robot in a maze, e.g. as shown in Figure 1b, however, the final maze will not be revealed. The goal is to find a target wall (exit) with a specified color within the maze.



(a) Filtered RGB samples measured at a distance of 7.8 cm under office light conditions. Each row shows the result of one of the four given color sheets.



(b) Example of a possible maze configuration. Here, yellow walls indicate the correct direction (hint walls), i.e. in this example the robot should be programmed to turn right when looking at a yellow wall. At the final dead-end there will be three colored walls, of which the green colored wall indicates the exit.

Figure 1: Illustration of the Maze Challenge: Part I (Fig. 1a) and Part II (Fig. 1b.)

Challenge Specifications & Requirements

Maze Specifications:

- The maze is built in a grid system, where each tile is $35\,\mathrm{cm} \times 35\,\mathrm{cm}$ in dimension.
- The maze consists of maximum 4×4 tiles.
- The robot cannot leave the maze at any location.
- The maze does not contain loops.
- The target wall will be one of the three walls in a dead end. As an example, the tiles with striped background in Fig. 1b are candidates for a final wall.
- It can be assumed that the path that takes the robot to the goal contains a hint wall at each position where the maze branches in two possible directions. Note, that this also includes the start position, as shown in the example maze (Fig. 1b).

Task Requirements:

- **Mazebot Initialization:**
  - Before starting the mission, the Mazebot needs to know the color of the exit (target wall), as well as the color and direction of the hint wall. To configure the mission during initialization, implement a **start menu**, where:
    1. The **target color** is selected from: RED, GREEN, BLUE or YELLOW.
    2. The **hint color** is selected from: RED, GREEN, BLUE or YELLOW.
    3. The **hint direction** is selected from: LEFT or RIGHT.

  - Consider that the start position is unknown, however, you can assume that your robot will be placed approximately in the center of a tile at start.
- Most likely, all four colors will be used in the maze, but to further simplify the task, the following rules apply:
  - Target and hint wall colors are reduced to **YELLOW or GREEN**.
  - If a hint color is detected (either **YELLOW or GREEN**), the Mazebot may immediately turn to the hint direction **LEFT or RIGHT** and continue its mission.
  - The final dead end has three colored walls, where the target wall will be either **YELLOW or GREEN** (i.e. if hint=yellow, then target=green) and the other two colors (**BLUE and RED**) serve as a distraction.
- The moment the robot starts moving, it should produce a beep, and once the robot has found the target wall, it should produce another beep. We will measure the time between the beeps to enforce the time limit (3 minutes, tentative).
- During the mission, the current distance and the detected color must be displayed on the LCD.
- At the final dead-end, you are required to check all three walls before turning to the correct target color wall and stopping the mission indicated by a beep sound.
- As usual it is required to submit the final program such that its functionality can be verified by simulator testing: Prepare the `MainSimulated.java` such that it runs your program with the test maze provided via GitLab (similar to Fig. 1b).
- In addition, you should also provide your `ColorDetectionTest.java` program from the previous part. So if for whatever reason we can't evaluate your color detection algorithm with your final program (e.g. because the final target walls were not reached within time), we can still evaluate your custom color detector separately.

Hints

- Implement the start up menu at the end, and for the development, define the mission parameters, e.g. by using constants:
  - `HINT_COLOR = "YELLOW"`
  - `HINT_DIRECTION = "RIGHT"`
  - `EXIT_COLOR = "GREEN"`
- Also, the start up menu is not required to work in the simulator. So you can provide an option to set the mission parameters via the menu or via hard-coded definitions for simulator tests.

Remarks

- If during the mission, the robot beeps in front of walls other than the final target, this will result in a penalty.
- If the robot beeps at the final target, before it has checked all three walls/colors or before/without turning to the correct exit, this will result in a penalty.

Bonus points

- **Submission Deadline:** Upload your solution to GitLab until **Januar 29th, 23:59h CET**.

In this lab, you can earn up to **6 bonus points**. A complete submission includes:

- The source code for the maze traversal program,
- a UML class diagram,
- a short video presentation (max. 3 min) describing the main ideas and concepts of your customized solution,
- a text file containing the group members' names and matriculation numbers.

The UML class diagram should show all classes and interfaces you implemented for your solution. Classes and interfaces that were provided from `LeJOS` or from the `Java` standard library can be drawn as empty classes, i.e., by omitting the variables and methods. You can ignore the classes that are only relevant for simulation. Include your UML class diagram as jpg, png or pdf file in your repository.

You are required to record a short **max. three minute video**, in which you explain your solution. Tell us first, how your algorithm for traversing the maze works conceptually (ca. 1 minute), then explain the source code and what problems you are still facing, or what could still be improved. It is important, that **both team mates** participate in this video and have an equal amount of talking time. We reserve the right to adjust the points of a member if he or she is not actively participating. We will create group folders in StudIP (which can only be accessed by members of this group). Please upload your video there. Note, that the video files need to be smaller than 100MB.

Grading Criteria:

- basic functionality (evaluated in simulator): **up to 2 points**
- coding style, documentation, video presentation and UML diagram: **up to 2 points**
- real-world results: **up to 2 points**
  - solving the Maze Challenge within the time limit (3 min),
  - detecting colors correctly (hint color, target/wall colors at the exit).

—Good luck and have fun—