Smart Lane/Line Follower

Robotics 2022
Group E

Marc Salvadó, Wonbin Song, Elie Youssef

Abstract
In this project we build a line/lane following robot with obstacle detection and avoidance capabilities. In particular, the robot moves along a line, and whenever an obstacle appears in front of the robot, it recognizes the obstacle and is able to change to another line. The task is extended to lane following. Specifically, the robot moves between two lines and is also able to change to another lane once it encounters an obstacle. We show that the tasks are achieved by applying the classical image processing techniques with relatively simple algorithms.

1 Introduction

Self-driving is among the most popular applications of computer vision and neural networks in general [1, 2]. Motivated by recent advances in self-driving car technologies, our goal of the project is: given a robot with a mounted camera, we would like the robot to navigate using a line or between lanes drawn on the floor, and along the way, avoid any obstacles it faces, switching lines/lanes to continue its path.

- Line Following: The robot moves along a line drawn on the floor
- Lane Following: The robot moves between 2 lines drawn on the floor
- Obstacle Detection: The robot recognizes an object (in this case a red swivel chair) as an obstacle
- Obstacle Avoidance: The robot moves away from the detected obstacle, by changing lines/lanes

2 Approaches

We used OpenCV [3] to create masks to "see" the lines and the obstacles. After which, using OpenCV's implemented methods, namely Moments, [4] , we could get the position of these obstacles and lines, and move our robot accordingly.

Line/Lane and Obstacle Detection : To detect the lines and obstacles on the floor, we first convert the received image's color space from RGB to HSV [5]. We then threshold the HSV image for a range of white color (line) and a range of red color (obstacle). And then we extract the lines and the obstacles, separately.

In both *line* and *lane following*, we apply a proportional controller, setting the error *err* as follows:

$$err = goal - \frac{w}{2},$$

where w is the width of the screen, so that $\frac{w}{2}$ stands for the current position; and *goal* depends on whether we are using *line* or *lane following*, and is defined here below:

Line Following : The *goal* is defined as the center of mass of the white blobs (contiguous collection of similarly colored pixels) representing the lines, after masking irrelevant parts of the screen so that it focuses on a single line. If no white blobs are detected, the controller spins to find some.

The linear speed is 0.1 m/s when a line is detected and 0 m/s otherwise.

Lane Following : The *goal* is defined differently depending on whether (I) white blobs are detected on both sides of the screen (left half, right half), (II) in only one of the halves, or (III) no blobs are detected.

(I) If white blobs are detected on both halves of the screen, then the *goal* is defined as:

$$goal := \frac{x_L + x_R}{2},$$

where x_L stands for the rightmost detected white point on the left half of the screen, and x_R for the leftmost on the right half. This way, by supposing that we are always inside the correct lane, all the lines from the sides (standing for other lanes, etc.) will be ignored.

(II) If white blobs are only detected in one of the halves, then the *goal* is defined as:

$$goal := x_L + \frac{w}{2} \implies err = x_L$$

if the white blobs are detected on the left half (following the previous definitions of x_L and w), and

$$goal := x_R - \frac{w}{2}; \implies err = x_R - w$$

if the white blobs are detected on the right half. The idea of this approach is to turn away from the detected line and detect the "other" line delimiting the lane.

(III) If no white blobs are detected, then the robot moves straight ($goal = \frac{w}{2} \implies err = 0$).

We start with an initial linear speed of 0.1 m/s, and we increase it by 0.001 m/s at each iteration (with period 1/60 s), with a limit of 1 m/s, while two lines are detected;

however, when less than 2 lines are detected, or an obstacle is detected and during the corresponding maneuvers to avoid it, the speed is reset to 0.1 m/s.

In addition, at the start of the simulation, the robot knows on which of the two lanes it is ('left', 'right').

The way in which the lines are detected is the following: using a 2D NumPy array, we obtain the color of some rows and some columns of a relevant area of the processed image. This is done separately for the left and the right halves of the image. Then, the rightmost white point on the left half and the leftmost white point on the right half form the aforementioned x_L and x_R (if one is missing, the *goal* variable changes as indicated above).

Finally, when two lines are detected, the area in which the robot will seek for obstacles is only between them (when only one line is detected, only the area considered as "inner part of the lane" will be considered; and when no lines are detected, the whole x axis). This will prevent the robot from detecting an obstacle that lies on the adjacent lane or outside the circuit (in short, outside its own lane).

3 Experimental setup

Using CoppeliaSim, we have simulated a RoboMaster on a plane with lines on the floor as well as red chairs representing the obstacles; a picture of the circuit is shown in Figure 1. We used ROS2 Galactic to interact with the RoboMaster using the robomaster_ros package provided in the VM. This was used to control the movement and get the camera feed.

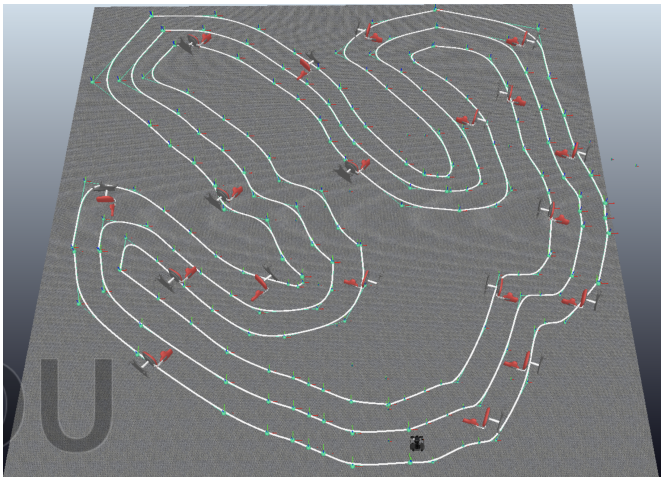

Figure 1. Map to test the ability of our robot to perform the lane following, line following and obstacle detection and avoidance tasks. It consists of a plane with lines on the floor and red chairs representing the obstacles.

4 Results

Lane following. The lane following task was achieved successfully. In Figure 2, which is a frame of a video showing the robot's performance in the circuit shown in Figure 1, one can see what happens during a 90°-closed curve: at some point, only one line is detected; then, the robot "moves away" from this detected line. As a consequence of this action, the robot is (usually) able to detect the two lines again, as shown in Figure 3, which corresponds to a frame a bit later from Figure 2.

When the lines are straight, the robot also follows the lane successfully, as shown in Figure 4. An instant later than the previous frame, the robot encounters an obstacle, so the obstacle-avoiding maneuver starts: the robot rotates towards the other lane (the robot knows, since the beginning, on which lane it is, and keeps track of the lane changes) as in Figure 5. Then, it moves forward for some time (150 iterations, at 1/60 s/it = 2.5 s), as shown in Figure 6, to finally come back to the *Detecting lines* state.

Finally, if the current lane is detected well, then the obstacles in the adjacent lane or outside the circuit will not fool the robot. In Figure 10, there is the screenshot of the processed image showing the detected lane (green) and the obstacles (red).

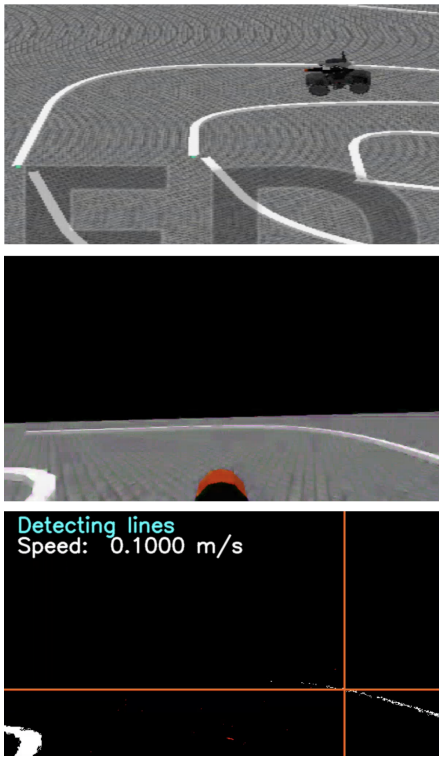

Figure 2. Plot showing the robot detecting the one line on a 90°-curve on the left. From top to bottom, the screenshots are the user view from CoppeliaSim, the image captured by the robot camera and the processed image to detect lines and obstacles (OpenCV). Since only 1 line is detected, it is highlighted in orange. During this process, the words "Detecting lines" are printed in light blue. The speed is always printed in white.

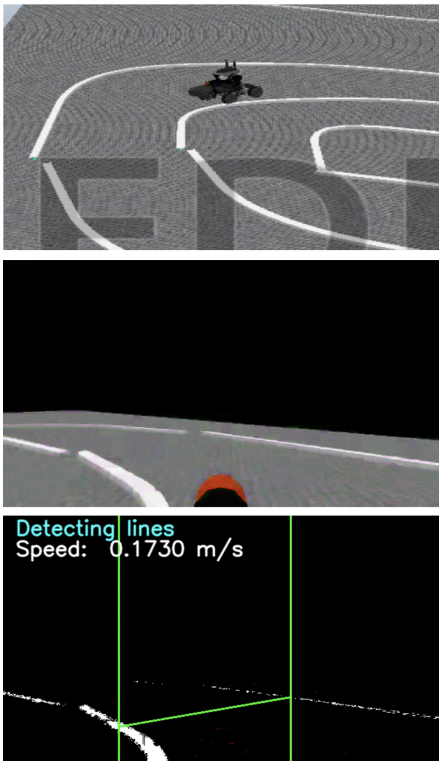

Figure 3. Plot showing the robot detecting the two lines (and its corresponding lane) on a 90°-curve on the left. Since 2 lines are detected, they are highlighted in green. During this process, the words “Detecting lines” are printed in light blue.

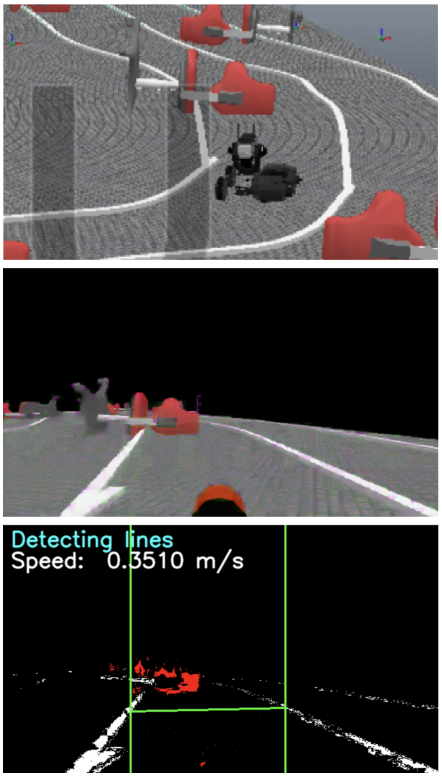

Figure 4. Plot showing the robot detecting the two lines (and its corresponding lane) on a straight lane. Since 2 lines are detected, they are highlighted in green. During this process, the words “Detecting lines” are printed in light blue.

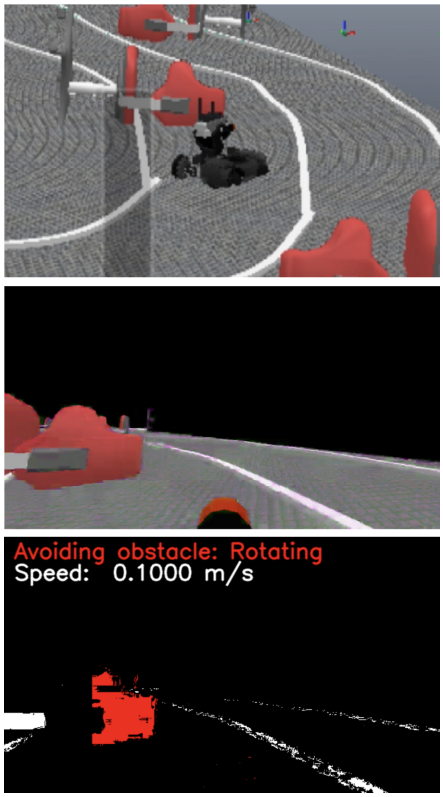

Figure 5. Plot showing the robot detecting an obstacle (red) and rotating towards the other lane in order to avoid it. During this process, the text “Detecting lines” (light blue) is replaced by “Avoiding obstacle: Rotating” (red).

Figure 6. Plot showing the robot advancing, after having detected a red obstacle and rotated towards the other lane, in order to change lanes. This process is performed for some time before trying to detect lines again. During this, the text “Avoiding obstacle: Rotating” (red) is replaced by “Avoiding obstacle: Advancing waiting” (yellow).

Line following. The line following task was also achieved successfully. In Figure 7, on the same map, the robot follows a straight line. In Figure 8, the robot appears avoiding an obstacle, just like in lane following. Then it moves ahead to look for a line again.

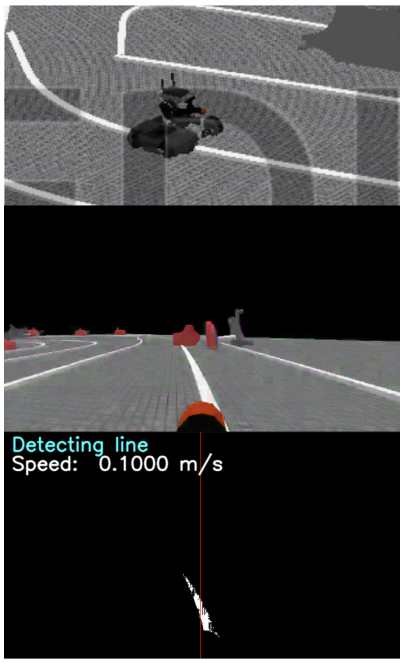

Figure 7. Plot showing the robot detecting a line. The axis of the center of mass is highlighted in red in the bottom picture. During this process, the words “Detecting line” are printed in light blue.

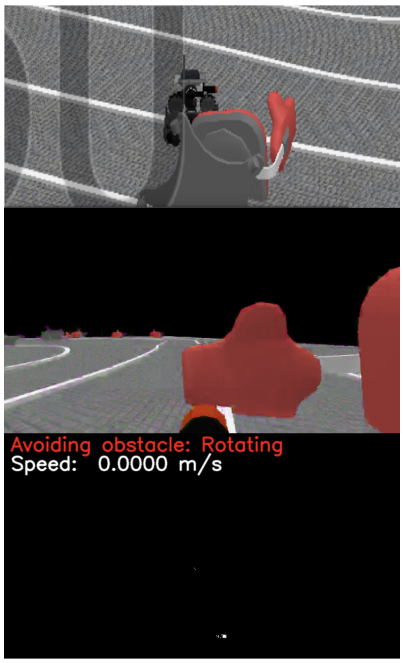

Figure 8. Plot showing the robot detecting an obstacle (red) and rotating towards the other line in order to avoid it. During this process, the text “Detecting lines” (light blue) is replaced by “Avoiding obstacle: Rotating” (red).

5 Limitations

Shared limitations

- The robot needs to know on which line/lane it starts.
- It only detects red objects as an obstacle, and an object of a different color is completely invisible to it. For example, when the chairs are lying in a position such that the black/gray part on its bottom is eclipsing its red parts, then it fails to identify it as an obstacle.

Lane following limitations

- Sharp curves pose an issue, confusing the robot. An example can be shown in Figure 9, a screenshot taken from a test video on the circuit, in which one can see (indicated in orange) that the robot is fooled by the inner lines of a narrow lane, and interprets it as the outer part of the lane .
- Lagging occurs on our laptops, which causes delayed reactions, ultimately causing failures. Lagging is mostly critical when rotating after detecting an obstacle: if the robot rotates too much —let’s say, 90°—, then it will cross the new lane’s external line perpendicularly. However, the algorithm is robust enough so that even if the robot is facing the aforementioned line by almost 90°(in favour of the correct direction), then the robot will redirect again to the correct orientation.
- The noise from the camera input sometimes impedes the correct functioning of the lines detection. This is usually inoffensive when there are no obstacles involved, since at most it can provoke a small maneuver that will probably be corrected at the following iteration. However, this is a serious issue when there are obstacles in the adjacent lane: if the line that divides the current lane from the adjacent one is not detected, then the robot will also consider the adjacency lane as part of its obstacle-detection area —see sketch in Figure 11—; if at that moment there is an obstacle on the adjacent lane, the robot will interpret it as in the current lane —unlike in Figure 10— and will perform an “avoiding maneuver” that will result in, first, turning towards the obstacle, and later, perpendicularly towards outside the circuit from the adjacent lane.
- Sometimes the robot moves out of the lane while performing a turn (see Figure 12).

Line following limitations

- Sharp curves are also problematic in line following; in this case, because the robot usually detects an adjacent line and changes lanes involuntarily.
- A situation occurs where obstacles are one after the other on both lines. This causes the robot to avoid the first obstacle by rotating away from it, however the rotation is then extended upon detecting the second

obstacle, even before clearing the first one, causing it to crash.

Figure 9. Screenshot (on CoppeliaSim) of a close, narrow curve that the robot usually fails to perform. When entering the curve, the robot detects only the outer line, so it turns left; but then, instead of detecting the two lines of the lane, it detects only the inner line (in orange), because, since the lane is so narrow, it lays on the left half of the image. This way, the robot interprets that the line in orange (inner) is actually the outer line of the lane —the actual outer line is not detected, because this helps ignore the adjacent lanes during turns—. As a result, the robot follows the trajectory portrayed by orange dots.

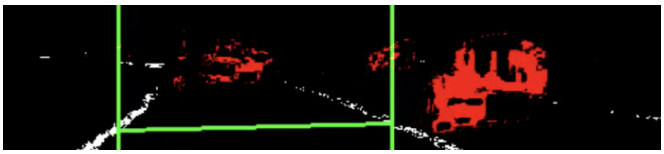

Figure 10. If the current lane is detected well, the obstacles in the adjacent lane will not fool the robot.

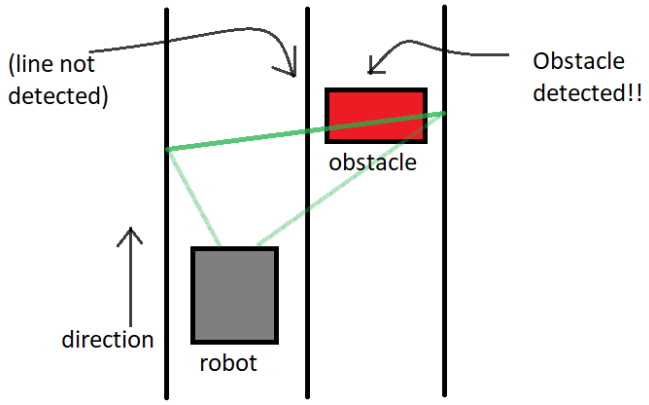

Figure 11. Sketch explaining the critical situation where noise impedes the detection of the middle line, causing the robot (even for an instant) to consider the whole road as a lane, and thus detecting an obstacle in it.

Figure 12. Screenshot showing that sometimes the robot, while making a turn, can exit partially its lane.

References

[1] Joel Janai, Fatma Güney, Aseem Behl, Andreas Geiger, et al. Computer vision for autonomous vehicles: Problems, datasets and state of the art. *Foundations and Trends® in Computer Graphics and Vision*, 12(1–3):1–308, 2020.

[2] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37 (3):362–386, 2020.

[3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[4] OpenCV. Image moments. `https://docs.opencv.org/4.x/d0/d49/tutorial_moments.html`, 2022.

[5] Wikipedia. HSL and HSV — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=HSL%20and%20HSV&oldid=1078569886`, 2022.