Robobo Line Follower

Ângela Cardoso
Faculdade de Engenharia da Universidade do Porto
Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
Email: angela.cardoso@fe.up.pt

***Abstract*—As a form of testing and learning the ROS framework implementation in the smartphone robot Robobo, a line following program was designed and implemented. The line is sensed with the camera on the phone, whose image is processed using OpenCV. A colored line on white background map was used to test the behavior of the robot and to manually tune the PID constants. The results show that the robot manages to follow the line while maintaining stability. However, due to the lack of quality from the camera image, the linear velocity of the robot must be low. Also, better PID constants can very likely be obtained through a more sophisticated tunning.**

***Index Terms*—Mobile robots, Reactive robots, Image Processing, PID control, Smartphone robots.**

1. Introduction

Robobo [1] is an educational robot developed by Mytechia. In terms of hardware it consists of a wheeled robotic base where an Android smartphone must be placed. The phone uses Bluetooth to communicate with the base and Wi-Fi to communicate with a computer, which is used to program the behavior of the robot.

Currently, there are three ways to program Robobo [2]. The simplest is block programming, which uses the Robobo Android application on the phone and a ScratchX [3] extension on a computer. One can also use native programming, with the Java language and the Robobo Framework. Finally, and of most interest to us, Robobo can be programmed with ROS [4], [5], using the Android application Robobo Developer. This application creates a ROS master node at the phone, to which a computer running ROS and using a set of defined custom messages connects, in order to control the behavior of the robot.

In order to test and get acquainted with implementation of the ROS framework in Robobo, a simple reactive robot was implemented. It uses the camera in the smartphone to detect a line of a previously determined color (black, red, green or blue) and then attempts to follow that line indefinitely.

The system developed barely makes use of memories or past experiences to inform its current decisions. However, in order to improve the robots stability, a Proportional–Integral–Derivative (PID) controller [6], [7] was used. This implies that at each moment, the previous error is known and used to compute the derivative term. A sum of all previous errors is also kept in the integral term. Therefore, this Robobo line follower is not purely reactive.

2. Architecture

2.1. Robot Design and Camera Position

As can be seen in Figure 1, the Robobo base possesses a smartphone holder. This holder is equipped with a pan movement, that allows it to rotate horizontally to the desired position, as well as a tilt movement, that can be use to rotate the phone vertically.

Figure 1. A Robobo robot.

The base also has two wheels on the front and a plastic caster at the back, which allow it to rotate in place. In order to move the robot, the speed of each wheel must be defined, thus determining both its linear and angular velocities.

Because we want to move the robot forward, that is with the wheels in the front, and we want to use the camera on the back of the smartphone to detect the line to follow, the phone holder must be rotated clockwise, so that it stays approximately 180 degrees rotated from the position shown in Figure 1.

The vertical position of the phone should be around 130 degrees, a little lower then the position shown in Figure 1. This way, even though we will be processing only the portion of the image closest to the robot, that image is wider

compared to a situation where the phone is lying down close to the base and the camera is directly facing the floor.

2.2. Maps

Two maps where used to test the line following Robobo. The first map is a simple curved black closed line in a white background, as shown in Figure 2.

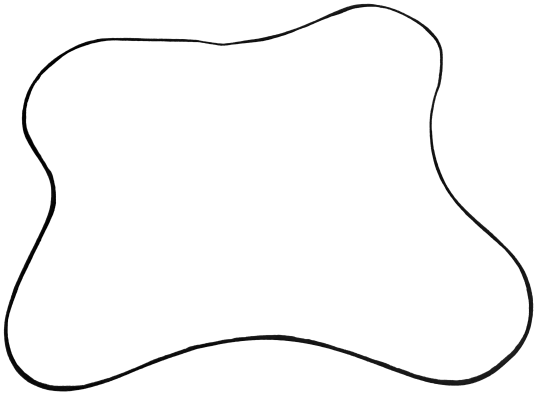

Figure 2. The black line test map.

The second map, shown in Figure 3, has three lines in colors the red, green and blue. The color that the robot follows is chosen by setting a run time argument: 0 for black, 1 for red, 2 for green and 3 for blue. Of course these colors, except perhaps in the black case, must be calibrated, but that must be done manually in the program code. An interesting future development would be to add a color calibration mode, that could then be used to track any color the user may desire, as long as it can be sufficiently separated from the background and other possible colors on the map.

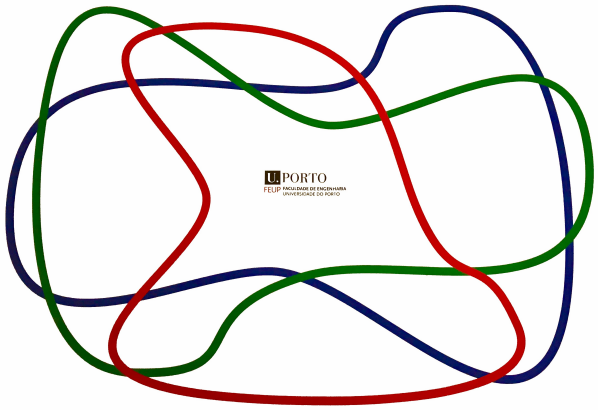

Figure 3. The colored lines test map.

2.3. Behavior Architecture

The robot follows a timed finite state machine architecture, according to the diagram in Figure 4. It is timed, because whenever a new camera image is received, which is determined by the frequency of the spin function in ROS, the state the robot is in may change, and these are the only times that may happen.

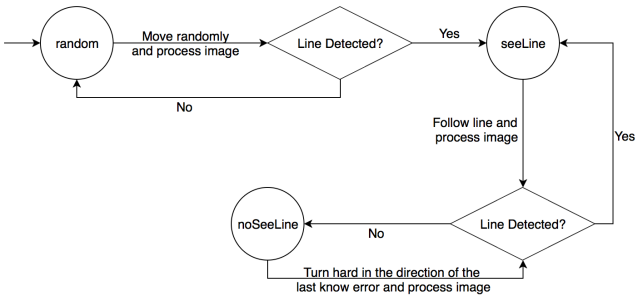

Figure 4. Timed finite state machine architecture.

The initial state of the robot is random, because until the robot detects a line to follow, it will move randomly. At the first camera image received where a line is detected the new state will be seeLine and the robot will never go back to moving randomly. In fact, if it is not in the seeLine state, it will be in the noSeeLine state, meaning that the processing of the previous camera image did not reveal a line. If that happens, the robot will take a sharp turn in the direction it was previously turning and try to detect the line again. Whenever the robot sees the line it will move in such a way as to stay on top of it and then process a new camera image.

There is no final state, because the robot will keep following the line forever, or at least attempting to do so, until the program is interrupted.

Because Robobo is an educational robot and the features of the smartphone are well explored in its programming applications, the changes in state to seeLine are accompanied by a happy face and an approving sound from the robot. Similarly, changes to the state noSeeLine are accompanied by a sad face and a disapproving sound. This makes it quite easy during runtime to note when the robot looses sight of the line and detects it again.

2.4. Algorithms

Essentially, three algorithms where used to program Robobo to follow a line: an image processing algorithm, that returns the normalized distance of the line to the center; a random movement algorithm, that the robot uses until it first finds a line; and a line following algorithm, that is used from the moment a line is first detected.

2.4.1. Camera image processing. Whenever a new camera image is received, see Figure 5, the program takes a series of steps until it determines a normalized error, that represents the distance of the line detected in the image to the center. This image processing makes use of the OpenCV library [8].

First, a region of interest (ROI) is selected, as shown in Figure 6. This is done in order to have better control

Figure 5. Camera image.

over which part of the image is analyzed, allowing for faster processing and better error calculation. The way it was implemented, the user is allowed to select the relative distance of the top of the ROI to the top of the image. This is a floating point number between 0.0 and 0.8, where 0.0 represents the very top of the image and 0.8 the bottom, because the ROI height is fixed at 20% of the image height.

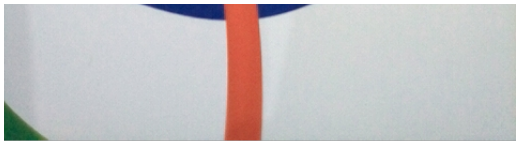

Figure 6. ROI with red, blue and green lines.

Once the ROI is selected, a mask that detects only the desired color is computed. Then, the inverse of this mask is replaced by white, so that the only non white pixels are those belonging to the line, see Figure 7.

Figure 7. ROI after mask with red line only.

This image is converted to gray scale, resulting in the image in Figure 8, then it is blurred, see Figure 9, and finally a threshold is computed, so that the line pixels become white and everything else is black, as in Figure 10

Figure 8. Gray scale ROI with line.

Figure 9. Blurred ROI with line.

Figure 10. Thresholded ROI with line.

In the final image processing steps, the contours of the image are computed, see Figure 11. In most scenarios a single contour will be computed, or none, if there is no line of the chosen color in the ROI. But in order to account for those cases where two or more lines of the same color or two parts of the same line are visible in the ROI, the most central contour is selected. In fact, for each contour, its moments [9] are computed and the centroid is then calculated as

$$\left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}\right).$$

The minimum distance of the centroids to the center of the ROI determines which contour is selected. The error corresponding to that contour is then given by

$$1 - 2 * \frac{m_{10}/m_{00}}{d},$$

where d represents the width of the ROI. This yields a floating point number between -1 and 1, where -1 means the line is in the far right of the ROI, 1 means it is in the far left and 0 means it is in the middle.

Figure 11. ROI with line and contours.

Figure 12 shows the ROI after threshold, with the line contours in green and the computed centroid in red. Since the line is slightly to the left of the center of the ROI, the corresponding error will be positive, but close to 0.

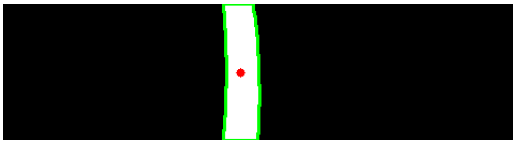

Figure 12. ROI with line, contours and centroid.

2.4.2. Initial random movement. Until the first time a line is detected in the camera image from the phone, the robot will move randomly. The implementation of this random movement is very straightforward, a simple movement command is sent to the robot, setting the speed of each wheel to a random integer between 1 and 5. When the next camera image is received if the processing still fails to reveal a line to follow, a new random movement command is sent to the robot.

2.4.3. Follow line movement. After a line first is detected the robot will attempt move along following that line. To do this in a stable manner, a PID controller was implemented. The result is that at each image processed, the angular velocity of the robot is given by

$$\omega = K_p * P + K_i * I + K_d * D,$$

where K_p, K_i and K_d are the proportional, integral and derivative constants defined by the user, and P, I and D are the proportional, integral and derivative terms, respectively. In this case, $P = e$, $I = I + e$ and $D = e - d$, where e is the current error obtained from the image processing and d is the error obtained in the previous image. When the robot is initialized, e, d and I are all set to 0.

Since the velocity for each wheel must be specified in any move command, the final line following robot movement is given by setting the left wheel speed to $v - \omega$ and the right wheel speed to $v + \omega$, where v is the linear velocity defined by the user.

If at some point during its movement the robot loses sight of the line, that is, the ROI of the current image has no line of the desired color, then a new artificial error is computed as $e = d/|d|$, which means that the error is -1 if the previous error was negative and 1 if it was positive. Given that the robot just lost sight of the line, which we assume is continuous, the previous error should not be 0, but in the unlikely chance it is, then the current error is also set to 0. After this artificial error is computed, movement proceeds as usual.

3. Results

Since there is no simulator to test Robobo and the line following program, non virtual tests had to be conducted. For these tests the maps in Figures 2 and 3 where printed in large white paper and the robot was set up those maps, as shown in Figure 13.

Because the random movement is not very relevant and is so simple, there is no real need to test it. Hence, for most tests, the robot was placed directly on top of the line it is supposed to follow. Also, in order to guarantee that the starting movements are simple, usually a straighter part of the line was chosen to place the robot.

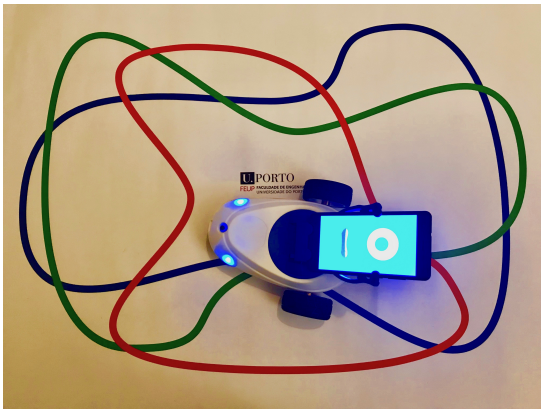

Figure 13. The colored lines map with Robobo.

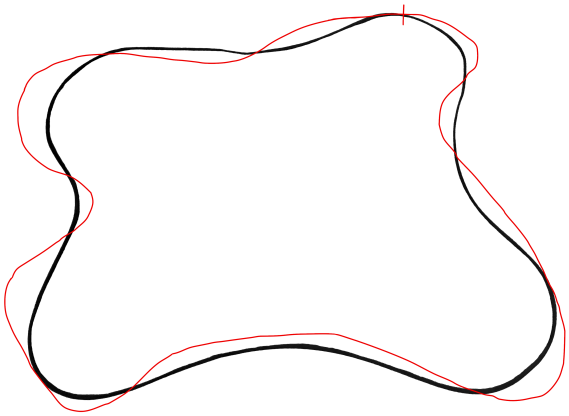

Figure 14. The black map and the line made by Robobo.

In an attempt to measure both the stability of the robot and how closely it is able to follow the line, a pencil was attached to the front of the robot so that it could mark the passage of the robot through the map. The result of this experiment, which ran over the black line map, can be observed in Figure 14. The red line is the line made by Robobo while following the black line. The place where the robot was placed on the line is also marked.

The linear velocity for this test was 8, the distance of the ROI was 0.8 (closest to the robot) and the PID constants where $K_p = 8, K_i = 0$ and $K_d = 8$. As shown in Figure 14, with these parameters the movement of the robot is quite stable, in fact, there is practically no overreach followed by subsequent corrections. The red line does diverge from the original black line, especially in harder turns, but this is to be expected. In any case, better results can most likely be achieved with further tuning of the PID constants.

4. Limitations

4.1. Initial robot position

If the robot is not properly placed on top of the line it should follow, then it may have trouble finding it. This happens when the random movement leads the robot to find the line in front of it but perpendicular to the robots orientation. In that case, the centroid of the contour of the line will be close to the center of the ROI, hence the computed error will be close to 0 and the robot will move forward, missing the line. Of course one could treat this special case separately in the image processing algorithm, but it would imply always testing for a case that should not happen once the robot starts following the line.

4.2. Camera image width

As mentioned above, placing the phone vertically, as the design of the Robobo implies, makes the camera image inadequate for line detection. The width of the image is simply too small, which makes the robot loose sight of the line every time it makes a semi-sharp turn.

In order to fix this, an artificial error is computed, which makes the robot turn harder in the direction it was already turning. In some cases this direction may be wrong, leading the robot to make a U-turn. Also, if the line is not spotted again, because the robot got to far away from it, the robot will just stay spinning circularly in the same spot.

4.3. Linear velocity

Due to the implementation of the `Robobo Developer` application, the camera image is already compressed when received in the line following program. This results in a very low quality image, especially if the robot is moving fast at the time the image is captured. Thus, faster linear velocities make the robot loose sight of the line more often and make its behavior more erratic, even after tunning the PID constants.

5. Conclusion

Although a line following robot is usually simple to implement, using the camera to sense the line poses a much more significant challenge, since it demands some knowledge of image processing. This was the most difficult, but also the most interesting and rewarding part of this project.

Learning to deal with the frequent loss of the line also provided a good experience, once this limitation was accepted and appropriate measures to counter it where taken. Indeed, it was only after this implementation that the robot managed to take a full lap following the line without getting lost or completely ignoring some curves.

Also difficult, especially because there is no simulator, was tunning the PID constants. But unfortunately, apart from the moment where the robot actually started following the line in stable way, this part was quite boring.

The main objective of testing and learning to use `ROS` to program `Robobo` was clearly achieved. But the final resulting robot is also quite riveting, it is fun to see and hear the robot as it follows the line, temporarily losing track of it and going sad, then finding it again and going happy.

Acknowledgment

The author would like to thank Fran Bellas, for technical help regarding `Robobo` as well as some hints on writing `C++` code, and Armando Sousa, for the colored map, an introduction to PID controllers and other advice.

References

[1] Mytechia, "Robobo," 2016, [Accessed 3-January-2018]. [Online]. Available: http://en.theroboboproject.com

[2] ——, "Robobo programming," 2016, [Accessed 3-January-2018]. [Online]. Available: https://bitbucket.org/mytechia/robobo-programming/wiki/Home

[3] L. K. G. at the MIT Media Lab, "Scratch," 2005, [Accessed 3-January-2018]. [Online]. Available: https://scratch.mit.edu

[4] O. S. R. Foundation, "ROS," 2007, [Accessed 3-January-2018]. [Online]. Available: http://www.ros.org

[5] J. M. O'Kane, *A Gentle Introduction to ROS*. Independently published, Oct. 2013. [Online]. Available: http://www.cse.sc.edu/~jokane/agitr/

[6] Wikipedia, "PID controller," 2002, [Accessed 3-January-2018]. [Online]. Available: https://en.wikipedia.org/wiki/PID_controller

[7] Enigmerald, "PID tutorials for line following," 2014, [Accessed 3-January-2018]. [Online]. Available: https://www.robotshop.com/letsmakerobots/pid-tutorials-line-following

[8] W. G. Intel and Itseez, "OpenCV," 1999, [Accessed 3-January-2018]. [Online]. Available: https://opencv.org

[9] Wikipedia, "Image moment," 2005, [Accessed 3-January-2018]. [Online]. Available: https://en.wikipedia.org/wiki/Image_moment