ROS project

Linda RAHAOUI
Master ISI
Student number 28606582

Yang LI
Master ISI
Student number 21200035

***Abstract*—This report summarizes our project on controlling a Turtlebot 3 burger mobile robot in different environments, both in simulation and in the real world. We will describe our approach to three distinct challenges, including line following, navigating through a narrow corridor, and operating in a cluttered environment. We provide a detailed account of our methodology for each challenge and present our results, including a performance characterization of the ROS (Robot Operating System) architecture.**

I. Introduction

This report outlines our project of controlling a Turtlebot 3 burger mobile robot in challenging environments, both simulated and real. Our goal was to develop sensorimotor capabilities for the robot to navigate from a starting position to a goal position while completing various tasks such as line following (Challenge 1), navigating through a corridor (Challenge 2), and operating in a cluttered environment (Challenge 3). Our ROS architecture, which utilized multiple nodes to achieve our objectives, is detailed in section II.A. We present our methodology for each challenge in section II.B and evaluate the performances of algorithms in section III. Finally, we conclude by discussing the strengths and limitations of our approach.

II. Presentation of the ROS architecture

In this section, we will present the ROS architecture that we built to solve the objectives mentioned in the introduction. Our architecture consists of several nodes that work together to enable the Turtlebot to navigate through the challenging environment and complete the three tasks.

A. A short overview

Our architecture includes several nodes that communicate with each other through topics. The nodes used are **Red_line_detector**, **Robot_controller**, **Line_following**, **Obstacle_avoidance**, **wall_following** and **Arena_mvt**.

Figure 1 provides a detailed topology diagram of our system, showing the connection between each node through topics and messages. The nodes in our architecture perform specific functions, such as detecting lines and avoiding obstacles.

We used images from a simulated/real camera for Challenge 1, laser scans from a simulated/real LDS for Challenge 2, and both sensors together for Challenge 3. In addition, we connected each challenge to allow the robot to navigate through the entire map.

The **Red_line_detector** node subscribes to the "/camera/image/compressed" topic to receive images captured by the robot's camera in gazebo, and publishes "Bool" messages on the "/red_line_detected" topic. The **Robot_controller** node subscribes to the "/red_line_detected" topic to receive information about detected red lines and to perform appropriate tasks.

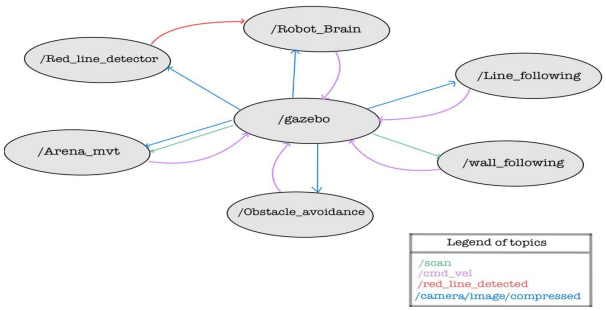

Fig. 1. Architecture of project.

Topics	Type of message
/scan	sensor_msgs/LaserScan
/cmd_vel	geometry_msgs/Twist
/red_line_detected	std_msgs/Bool
/camera/image/compressed	sensor_msgs/CompressedImage

Fig. 2. Type of messages used

The **Robot_controller** node is responsible for controlling the robot and launching appropriate nodes for different tasks depending on the "/red line detected" topic. It subscribes to the "/camera/image/compressed" topic to detect blue stop signs and control the robot's movement accordingly. The node also publishes velocity commands on the cmd_vel topic for smoother transitions between challenges.

The **Line_following**: node allows the robot to follow white and yellow lines on the floor. It subscribes to the "/camera/image/compressed" topic to receive images and publishes commands on the "/cmd_vel" topic for the robot's movement.

The **Obstacle_avoidance** node allows the robot to avoid obstacles while staying on the track. It subscribes to the "/camera/image/compressed" topic to receive images and publishes velocity commands on the "/cmd_vel" topic based on the detected colors and moments of the masks.

The **wall_following** node navigate through narrow corridors, keeping its trajectory as close as possible to the center between the two walls by using PID controller. It subscribes to the "/scan" topic to receive laser scan data and publishes

velocity commands on the ”/cmd_vel” topic based on the difference in distances between the left and right sides.

The **Arena_mvt** node allows the robot to operate in a cluttered environment. The robot moves towards the green targets while avoiding obstacles and staying within the boundaries defined by a blue line. It subscribes to the ”/scan” topic and ”/camera/image” topic to receive laser scan data and images from the camera. Then this node publishes velocity commands on the ”/cmd_vel” topic based on these information.

Overall, our architecture (figure 1) demonstrates our ability to master ROS concepts and use them in an actual project. The next section will focus on performance characterization of our ROS architecture.

B. Algorithmic structure of the architecture

The launch file is run and the **Red_line_detector** node detects the first red line. It sends a True message to the red_line_detected topic. The **Robot_controller** node receives the True message on the topic and calls the callback function. The value of the counter variable (cpt) is decremented to 5. Each time a new red line is detected, **Red_line_detector** sends True on the topic and in the **Robot_controller** node cpt is decremented, then the processing is done as follows:

When cpt is equal to 5, the **Robot_controller** node launches the **Line_following** node. **Line_following** is subscribed to the /camera/image/compressed topic, where **gazebo** node publishes images from the camera. For each image received, **Line_following** uses masks to detect the colors of the left and right lines (yellow and white), and based on the moment of these masks, it publishes the appropriate linear and angular velocities on the /cmd_vel topic.

When cpt is equal to 4, nothing happens as the robot continues to follow the lines.

When cpt is equal to 3, the **Robot_controller** node stops the **Line_following** node and the robot movement. It then launches the **Obstacle_avoidance** node. The **Obstacle_avoidance** node is subscribed to the /camera/image/compressed topic. For each new image published by **gazebo** topic, **Obstacle_avoidance** checks if an obstacle (red) and the lines (yellow and white) are detected in the image. Based on what is detected, **Obstacle_avoidance** publishes the required velocities on the /cmd_vel topic. For example, if both white and red are detected, and the moment of the white mask is greater than the moment of the red mask, **Obstacle_avoidance** publishes the commands to turn left. If the moment of the red mask is greater than the moment of the white mask, **Obstacle_avoidance** publishes the commands to turn left or right based on the position of the obstacle.

Before being able to move on to the next step, the robot must stop obstacle avoidance and move towards the corridor. And the same process occurs when the robot has reached the end of the corridor. This is done thanks to detecting blue traffic signs made by the second image_callback of the **Robot_controller** node. Each time an image is received on the camera topic, a process is done to find out if a blue panel has been detected. The blue signs (see figure 3) will serve as stop signs. They will allow a transition between the movement in the corridor and the following lines. Depending on the detect panel and the status of the robot (cpt value equal to 3 or 2) the function SetStop is called and the robot will stop or move forward then another node will be run.

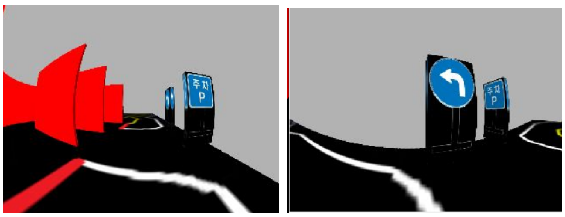

Fig. 3. Blue signs

When cpt is equal to 2, we start the **wall_following** node to cross the corridor. This node

- subscribes to laser scan data and extracts the range measurements for the left, right, and front regions of the robot by using the specified index ranges;
- then, removes infinite and non-numeric values from the measurements and calculates the average distances for the left and right sides. We define the error = left_distance - right_distance;
- decides whether the robot should go ahead (error $<$ 0.02) else adjust the direction
  - if error $<$ 0.02: the robot goes ahead
  - else: the robot adjusts the direction(the PID controller calculates angular velocity).

When cpt is equal to 1, the **Robot_controller** node stops the robot movement, stops the **Obstacle_avoidance** node, and relaunches the **Line_following** node. The process of step 3 is repeated.

When cpt is equal to 0, we start the **Arena_mvt**, the Turtlebot navigates through a cluttered environment using laser scan data for obstacle avoidance and camera image data for navigation towards a green target, while staying within the boundaries of a blue line. The node subscribes to topics for laser scan data (/scan) and camera image data (/camera/image/compressed), and determines whether to use laser data or image data to set the speed of the robot:

- if there are obstacles within a certain range($<$0.2), the robot moves away from them by dividing the front area into four parts for obstacle avoidance.
- else: image data is used, the node calculates moments to detect blue and green regions.
  - if there is no blue or green color detected in the entire image: the robot has reached the destination, it stops;
  - elif the green color occupies at least 10% of the image and there is a blue line at the destination: the robot moves forward, attempting to center the green region in the image. This condition is necessary because, besides the green pillars, there is also a blue line(the yellow circle in figure 4) at the exit of the

map. Without this condition, the robot would avoid the blue line and wouldn't be able to exit the map;
- elif blue color is detected in the bottom of the image: the robot adjusts its motion to avoid the blue region;
- elif only a green color is detected, the robot adjusts its motion to center the green region in the image in order to find the destination;
- else: the robot moves forward.

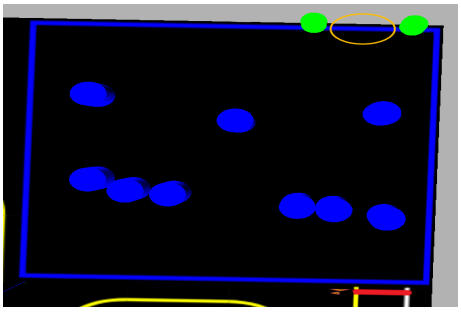

Fig. 4. The map of challenge 3.

III. Performance Characterization

In this section, we will analyze the performance of the architecture. We use the mean error E as an indicator, which measures the average deviation of the robot from the desired trajectory. It is calculated as the average difference between the desired trajectory and the actual trajectory followed by the robot.

$$E = \frac{1}{N}\sum_{i=1}^{N}|x_i - x_i'|, \tag{1}$$

where N is the total number of samples, x_i is the desired trajectory at sample i, and x_i' is the actual trajectory at sample i.

By plotting the error values over time, we can analyze the performance of the line following and corridor crossing algorithms, which provides insights into how well the robot is following the desired trajectory.

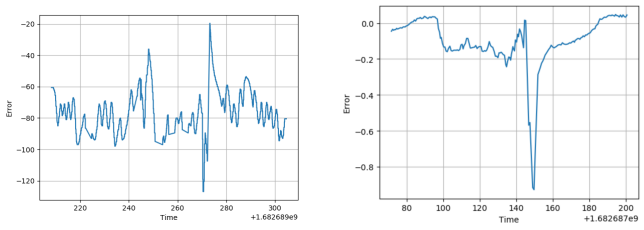

Fig. 5. The error over time for line following (left) and corridor crossing(right)

Figures 5 display the error over time for the line following and corridor crossing challenges, respectively. Both show relatively stable error values with small fluctuations and some peaks caused by sharp turns. The mean error for the line following is -76.2 pixels, indicating a significant deviation from the desired trajectory. Considering that the size of the image is 240x320 pixels, the observed error values are quite large. For the corridor crossing, it is -0.096m, indicating a relatively smaller deviation. These error values can be attributed to suboptimal tuning of the controller parameters.

Additionally, for challenge 3, the success rate of the system can also be considered as a performance indicator. We tested the same code on two computers. The older computer, which takes 12 minutes to complete the entire process, has a higher success rate of around 60 percent. On the other hand, the newer computer, which takes 4 and a half minutes to complete the entire process, has a lower success rate of around 30 percent.

IV. Conclusion and Perspectives

Based on our project on controlling a Turtlebot 3 burger mobile robot in different environments, we have successfully addressed three distinct challenges: line following, navigating through a narrow corridor, and operating in a cluttered environment.

However, our approach also had certain limitations. Firstly, the accuracy and robustness of our system were dependent on the quality of sensor data. In some cases, the robot's performance could be affected by challenging lighting conditions, occlusions, or noisy sensor readings. Future work could focus on applying better image processing techniques to handle more complex and dynamic environments.

Secondly, challenge 3 involves the use of both laser and camera sensors for obstacle avoidance and target detection in a cluttered environment. The proposed approach is to prioritize laser data when the robot is close to obstacles to ensure safe navigation, and to utilize the camera to identify the target and prevent the robot from going out of bounds. However, a potential issue arises when there are corners in the map where the robot is both close to obstacles and close to the blue line. In such cases, the robot's priority on using laser data may lead to the risk of going out of bounds.

To address this issue, we can consider the overall robot's position and orientation relative to the map, the system can determine if the robot is in a corner scenario. In such situations, instead of solely relying on laser data, the system can dynamically adjust the priority of the sensors based on the specific requirements of the task. We consider SLAM(Simultaneous localization and mapping) if we have more time.

Another issue is that the controller's performance can be further improved. We can achieve this by adjusting the parameters to enhance its behavior. Also, defining an effective region for image processing is crucial, as it directly impacts the robot's responsiveness. By selecting the upper part of the captured image, for example, the robot can react earlier to changes in the environment. The same principle applies if we choose the lower part of the image. Similarly, defining the laser ranges for the robot's front, left, and right distances is essential. These ranges directly influence the definition of error and the calculation of angular velocity. Therefore, optimizing the controller from this perspective can yield significant improvements.