ROS Project: TurtleBot 3 Navigation and Control

Oumaima HABKI
Master SAR
28610792

Meili HELBERT
Master SAR
28706059

Juyeon KIM
Master SAR
28611849

***Abstract*—This report illustrates the various steps followed in order to control a Turtlebot3 burger mobile robot. It explains the choices made in ROS that allowed the complete navigation of the robot on the designated path and through the different challenges. Furthermore, the report sheds the light on the parameters that evaluate the performance of the robot and the quality of the written ROS code.**

I. Introduction

ROS - Robot Operating System is an open-source framework that helps developers build and reuse code between robotics applications by relying on a specific architecture using nodes, topics, services, etc.

This project relies on this framework to control the navigation of a Turtlebot3 burger mobile robot on 3 different challenges : line following, corridor and doors.
The Turtlebot3 burger mobile robot is equiped with a real camera that displays images from the environment allowing the detection and the following of the lines that limit the path, as well as a LiDAR that detects obstacles to avoid.

The aim of the project is to write a cohesive code that relies on a set of nodes that perform the computation, various topics that enable asynchronous communication between the streaming data and services that allow the synchronous request/response interactions.

This report starts with a description of the Turtlebot3 then provides a thorough explanation of the ROS architecture of the solutions. Furthermore, it lists out different ways of linking the 3 challenges of the project. Finally, it carries out a performance analysis based on various parameters namely speed, precision, etc.

II. ROS Architecture solution

A. Challenge1 - Line following and Obstacle avoidance

This first challenge is divided into 2 parts : Line following and Obstacle avoidance. Our approach involves the creation of three nodes : *line_detector, line_follower* and *obstacle_avoider*.

In our case, the *line_detector* node subscribes to the robot's camera topic and performs image processing (masking and centroid calculation for each colour) using the OpenCV library. It then publishes a LinePosition message on the /line_position topic to which the *line_follower* node is subscribed (Figure 1). A LinePosition message informs of which color has been detected and the position of the colour's centroid.
The `LinePosition.msg` is structured as follows:

```
# LinePosition.msg

# Define the centroid for the yellow line
float32 yellow_x
float32 yellow_y
bool yellow_detected

# Define the centroid for the white line
float32 white_x
float32 white_y
bool white_detected

# Define the centroid for the green line
float32 green_x
float32 green_y
bool green_detected

# Define the centroid for the red line
float32 red_x
float32 red_y
bool red_detected
```

Each field in the message represents the X and Y coordinates of the centroid of a line detected by the robot, with a boolean flag indicating whether the line was detected. This information is crucial for determining the commands needed to follow the lines accurately.

When the two track lines have been detected by the *line_detector* node, the *line_follower* ensures that the robot is oriented in the correct direction (i.e. the yellow line on the left and the white line on the right) and that it advances at the optimal linear and angular velocity, which are defined to compensate for the error. The error is calculated as the difference between the midpoint of the centroids of the two track lines and the centre of the image.
In the event that only one line is detected, the robot will rotate in order to locate the missing line. If no lines are identified, the robot will perform a similar rotation in order to search for any of the lines on the track.
In the specific instance of an intersection, the robot assumes that it is not adequately oriented, as the position of the yellow and white lines is incorrect. Consequently, it maneuvers as if it were searching for a missing line.
Based on the information received via the /line_position topic, the *line_follower* node can publish the appropriate velocity values on the /cmd_vel topic.

At the end of the first challenge, the robot is confronted with a series of three obstacles positioned along the track. In order to avoid these obstacles, we have integrated the obstacle avoidance logic into the line following logic, which was previously developed. The *obstacle_avoider* node subscribes to the /scan topic, where the LIDAR of the robot publishes its data, and publishes to the /obstacle_avoidance_active topic to inform on wheteher there is an obstacle being avoided or not (Figure 1).

The obstacle avoidance logic is as follows. If an object is detected in the front range at a distance below the minimum distance defined (min_distance = 0.25m), the robot will consider it an obstacle. Once an obstacle has been identified, the robot will assess the available space and determine whether to rotate towards the left or right side. Additionally, a security system has been implemented. Should the robot reach the critical minimum distance (critical_min_distance = 0.17m) within the front safety range, it will move backwards in order to avoid hitting the obstacle.
Once no further obstacles have been detected, the *obstacle_avoider* node transmits a message to the obstacle_avoidance_active topic, indicating that obstacle avoidance logic is no longer active. The *line_follower* node, which subscribes to the obstacle_avoidance_active topic, can then take over (Figure 1).

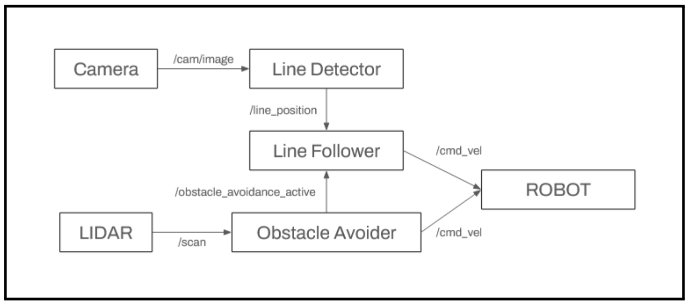

Figure 1: Interactions between nodes in challenge 1

We defined the ranges as shown in Figure 2:

- The **front range (from -25° to 25°)**: allows the LIDAR to detect an obstacle fast enough for the robot to stop in front of it, but without detecting the objects on the side of the track (signs...).
- The **left range (from -90° to -25°)** and the **right range (from 25° to 90°)**: allow the robot to analyse its surroundings when an obstacle is detected, and to determine on which side there's more space to avoid the obstacle.
- The **front safety range (from -55° to 55°)**: prevents the robot from hitting an obstacle with the wheels.

B. Challenge2 - Corridor

The second challenge consists of a U shaped Corridor and one node only was used for this purpose : *corridor*.

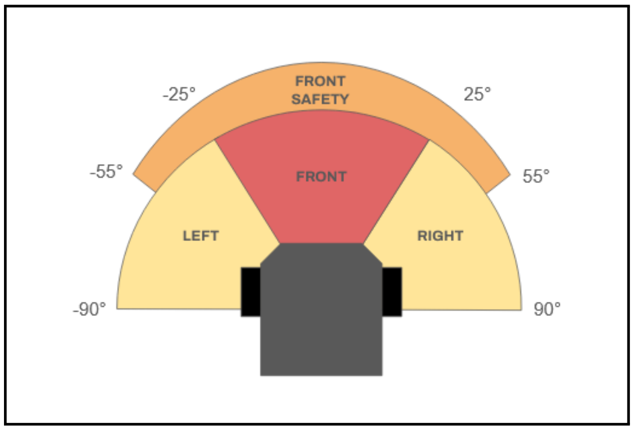

Figure 2: Representation of the ranges used in the obstacle avoidance logic

This part of the project only uses the information provided by the LiDAR and particularly the LDS-01 that is fixed on the Turtlebot3 burger mobile robot.

The challenges - related to this part of the project were mainly :

- Setting the best angular and linear speed that will allow a smooth navigation.
- Finding the most accurate distance between the robot and the wall that the robot shouldn't go beyond. This will allow the robot to navigate through the corridor without touching the walls while turning.
- Figuring out the most appropriate range for the values of the laserscan.

The node logic relies on creating a CorridorNavigator class which aim is to constantly compare the distance of the robot to the right and left wall and make a decision about whether to turn right or left. This logic can compare to a certain extent to the *obstacle_avoider* node used in challenge 1.

Here is an overview of the structure of the *corridor* node:

Attributs	Methods
self.linear_speed	self.scan_callback
self.angular_speed	self.move
self.min_distance	self.run
self.max_distance	-
self.state	-
self.completed_mission	-

linear_speed, *angular_speed*, *min_distance* and *max_distance* were all set empirically.

scan_callback() - continuously retrieves the values scanned by the LiDAR of the robot. The LDS-01 has an omnidirectional coverage which means that it perceives its surroundings over a wide range of angles, particularly, over a 360-degree field of view.
In order to use just enough data to navigate through the corridor, it was decided to retrieve only 50 elements per side

as the corridor is rather narrow, as follows :

- *left_distances* - stores the 50 first elements of the ranges array, thus representing measurements to the left of the robot's position.
- *right_distances* - stores the 50 last elements of the ranges array, thus representing measurements to the right of the robot's position.

This allows the analysis of the environnment on both sides of the robot separately then only keeps the minimum non-infinite value in the left ranges and the right ranges as the distance to respectively the left and the right wall.

Moving on to the robot state-based navigation logic :

It relies on two states `follow_left_wall` set as default state and `follow_right_wall` otherwise.

- If *left_distance* is greater than *max_distance*, it is interpreted as a too close obstacle thus the robot turns right.
- If *right_distance* is greater than *max_distance*, it is interpreted as a too close obstacle thus the robot turns left.
- If both *left_distances* and *right_distances* are greater or equal to *self.max_distance*, this indicates automatically that there aren't any obstacles anymore and the robot stops completely setting both linear and angular velocities to zero.

=> Overall, this state-based logic effectively guides the robot through the U-shaped corridor by following either the right or the left wall while avoiding obstacles.

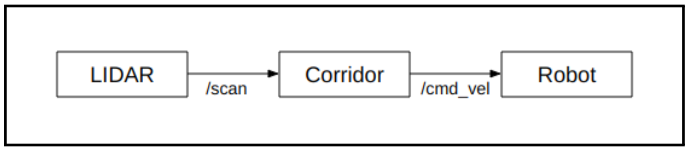

Figure 3: Overall structure of the *corridor* node of challenge 2

C. Challenge3 - Doors

The third challenge involves the robot navigating through randomly positioned doors, each consisting of a pair of cylindrical objects colored green, red, and blue. The Door class serves as a facilitator for detecting colored objects, avoiding obstacles, and making decisions. Navigation primarily relies on vision, wherein the robot moves towards a target point determined by the image processing node. Laser scan information is used to avoid collisions with the doors. The decision to move to the next door is based on the condition of not detecting the current target door color, indicating proximity to the door.

The image processing function receives images from the camera sensor and converts them to OpenCV format. The process of detecting doors involves identifying contours of a specified color, such as blue initially. Contours that are considered noise or irrelevant, such as those from a blue-colored lane, are filtered out using a threshold of contour area. This ensures that only door-like contours, which are

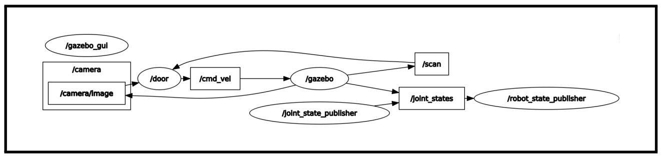

Figure 4: Overall structure of the */door* node of challenge 3

typically rectangular or cylindrical shapes, are retained. The target point for navigation is calculated from the centroids of these contours or from a single bottle. The robot then moves towards this target point until it no longer detects contours of sufficient size, indicating that it has reached the target door.

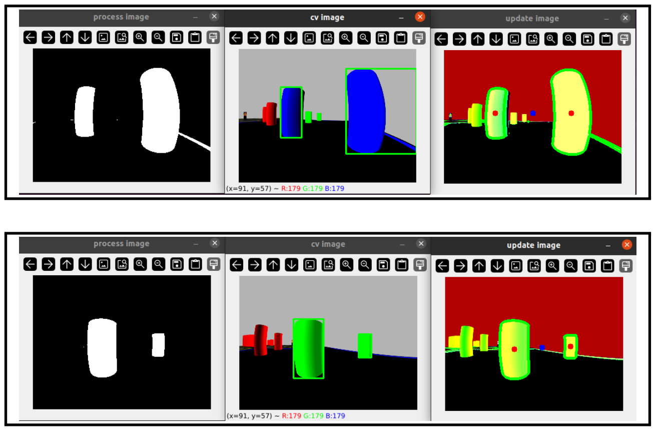

Figure 5: Output of image processing: mask, bounding box, contours and centroids

In this report, the Projet2024.world serves as a test case, where the sequence of doors is as follows: blue, green, red. Future development involves generalizing parameters, such as assigning colors to numerically labeled doors, to enhance the adaptability and scalability of the system.

D. Assembly of the 3 challenges

In order to level up the project, the transition between the 3 different challenges should be automatic and rather than launching each challenge seperately using seperate launch files, a single launch file should be used. For this purpose, 2 solutions were developed :

The first solution for transitioning between Challenge 1 and 2 is as follows. The *corridor* node subscribes to the /line_position and /obstacle_avoidance_active topics. In the absence of both lines and obstacles, the *corridor* node publishes to the /corridor_active topic, informing other nodes that the robot is applying the corridor logic.
However, this corridor logic was developed for a single use before it shuts down. Thus, when launching the simulation in Gazebo with the aforementioned nodes, we encountered an issue with Gazebo taking time to be launched properly. This resulted in the corridor logic starting and shutting down prematurely due to the robot failing to detect any lines before

the Gazebo's environment appears.
To address this issue, a new service, /start_corridor, was created. This service is activated when the */line_detector* node first detects lines. It then sends a service to the /corridor node, which can begin checking the conditions for the presence of a corridor. This ensures that the corridor does not begin until the first challenge has been started (Figure 6).

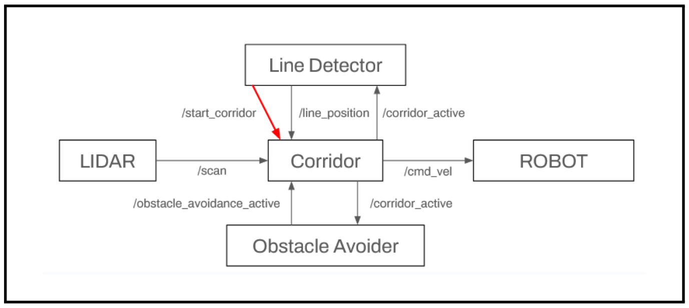

Figure 6: Interactions between nodes during the transition from challenge 1 to challenge 2

The development of this solution appeared to be the most appropriate for the real robot, given that the track's lines of the real environment were still visible in the corridor. As explained in the performance analysis section, this solution is not the most robust. Consequently, a second solution was developed.

The second solution consists of creating a single node that would activate and deactivate the appropriate nodes depending on the path of which challenge the robot is navigating through.
The *navigation_controller* node was set for this purpose.On the simulation, each challenge starts at pre-defined and known coordinates. Here is a breakdown of the structure of the node:

- The node subscribes to the */robot_pose* topic to recieve the robot's position information.
- The node uses 5 publishers to dynamically activate and deactivate each functionnality as follows :

Publisher	Topic
line_following_pub	/line_following/activate
line_detection_pub	/line_detection/activate
obstacle_avoider_pub	obstacle_avoider/activate
corridor_pub	corridor/activate
doors_pub	doors/activate

- The node uses 3 checkpoints based on the coordinates of the starting point of each challenge as follows :

Checkpoint	x_coordinate	y_coordinate
1	0.87	-1.74
2	1	1.5
3	-1.70	-0.2

=> The z_coordinate does not vary.
=> Here is the strcuture of the logic of the node :

Challenge n°	Publisher
1	✓line_following_pub ✓line_detection_pub ✓obstacle_avoider_pub × corridor_pub × doors_pub
2	× line_following_pub × line_detection_pub × obstacle_avoider_pub ✓corridor_pub × doors_pub
3	× line_following_pub × line_detection_pub × obstacle_avoider_pub × corridor_pub ✓doors_pub

III. Performance Analysis

A. *Assembly of challenges*

In order to test the robustness of our first solution to transition from challenge 1 to challenge 2, we launched our launch file 10 times in a row.

Table I: Results of launching 10 times the first solution to transition from challenge 1 to challenge 2

Attempt n°	Line Follower	Avoid Obstacle	Corridor
1	✓Intersection ✓Curves	✓Obstacles	× Enter corridor
2	✓Intersection ✓Curves	✓Obstacles	✓Enter corridor × Exit corridor
3	✓Intersection ✓Curves	✓Obstacle 1 × Obstacle 2	
4 ✓	✓Intersection ✓Curves	✓Obstacles	✓Enter corridor ✓Exit corridor
5 ✓	✓Intersection ✓Curves	✓Obstacles	✓Enter corridor ✓Exit corridor
6	✓Intersection ✓Curves	✓Obstacles 1 & 2 × Obstacle 3	
7 ✓	✓Intersection ✓Curves	✓Obstacles	✓Enter corridor ✓Exit corridor
8 ✓	✓Intersection ✓Curves	✓Obstacles	✓Enter corridor ✓Exit corridor
9 ✓	✓Intersection ✓Curves	✓Obstacles	✓Enter corridor ✓Exit corridor
10 ✓	✓Intersection ✓Curves	✓Obstacles 1 & 3 × Obstacle 2	✓Enter corridor ✓Exit corridor

From Table I, it can be observed that in 40% of cases, the robot was unable to reach challenge 3. To identify the underlying causes, a detailed analysis is required.

Firstly, the line following logic appears to be highly consistent. The robot was able to follow the lines successfully, both before and after the obstacles. Secondly, issues with the corridor are rare. Upon entering the corridor, the robot is usually able to exit and transition smoothly to following lines. However, on occasion, such as in attempt 2, the robot may finish the corridor and take the incorrect direction, resulting in its inability to find the lines of the track. This was the only instance in which the robot was unable to find its way after the corridor.

In attempt 1, the robot encountered an obstacle at the entrance of the corridor. From the information provided by the terminal, it can be inferred that the *corridor* node started running and was immediately shut down before Gazebo was opened. It appears that the /start_corridor service is not operational when Gazebo is used for the first time. When Gazebo is launched for the first time, it takes time before the environment of the challenges is displayed, which could explain this situation.

The most challenging aspect of the robot's navigation is obstacle avoidance. While the robot appears to be able to avoid the first obstacle, it frequently becomes stuck before obstacle 2 or 3 for various reasons. In the absence of lines or obstacles, the robot assumes it is in a corridor. It can also hit obstacles with its wheels despite the security system we implemented, or repeatedly attempt to locate a missing line that is out of the camera's reach.

In some instances, the robot's movements are quite rapid in the final turn before the corridor. This results in the corridor being identified as an obstacle. As evidenced by attempts 4, 5 and 8, the obstacle avoidance logic is capable of effectively replacing the corridor node.

The results of this experiment indicate that the primary challenge is the lack of robustness of the obstacle avoidance logic, which must be addressed if this project is to be continued.

B. The Real Time Factor and the rate

The Real Time Factor (RTF) measures how the simulation time compares to real-world time. It is mathematically the ratio of *Simulation Time* and *Real Time* and its value has impacted the performance of the robot both on simulation and in the real world.

The rate specifies how frequently a code should run. It is measured in (Hz). For instance, a rate of 10Hz corresponds to an execution of the code of 10 times per second.

Both parameters undoubtely interact. In fact, the following behaviors are observed :

- If the simulation cannot keep up with the desired rate due to computational limitations for example, the real-time factor drops below 1.
  => This is an indication that the simulation is lagging behind real-world time, which definitely affects the accuracy and the performance of the written code.
- Similarly, if the RTF > 1, this means that the simulation is running faster than the real-time.

Thus, it is a priority to apply the following solutions for the best RTF :

- Adjust the rate using the mathematical formula of the RTF.
- Use a powerful hardware.

=> It is crucial to ensure that the least compromise is made between the rate and the real time factor in order to observe the most realistic and reliable behaviors.

C. The LiDAR

As precised above, the turtlebot3 burger mobile robot used for this project uses an LDS-01. It is an omnicoverage LiDAR that covers a field of view of 360°.

For performance considerations, it is important to specify the desired coverage. Hence, it is crucial to adapt the range of points collected depending on the caracteristics of the circuit such as the presence or the absence of obstacles and in the case of the U-shaped corridor, the width of the corridor is an important parameter to consider.

However, it is necessary to bare in mind that by processing a large number of laser scan measurements, the computational demand increases greatly.

D. The queue size

The queue size is an important parameter which value is chosen depending on the number of the desired outgoing messages that are stored on the buffer if they cannot be sent immediately.
For instance, queue size = 1 means that only one message is buffered at a time.

The queue size is an important performance regulator.
For a low queue size, latency is reduced because the latest message published is always processed, however, it is high likely that messages get lost if the following messages arrive faster.
For a high queue size, it is a no brainer that there are less chances for the messages to get lost, however, a delay may be introduced because all of the messages are processed in the order they are received.

Last but not least, a higher queue size always consums a larger storage.

IV. Conclusion

This project report has detailed the development and implementation of a control system for the Turtlebot3 burger mobile robot using the Robot Operating System (ROS). Through three distinct challenges—line following and obstacle avoidance, corridor navigation, and doors—we explored the complexities of autonomous navigation in a given environment.

In the first challenge, the robot's capacity to follow lines and avoid obstacles was tested through a combination of image processing and LiDAR data. The *line_detector* node successfully identified and tracked lines of various colours, while the *obstacle_avoider* node managed the detection and avoidance of obstacles. Despite some challenges in line detection at intersections and the complexity of obstacle avoidance, the robot demonstrated a robust capacity to navigate within defined parameters.

The second challenge, which involved corridor navigation, necessitated precise control of the robot's movement within a narrow pathway. The *corridor* node utilized LiDAR data to maintain appropriate distances from the corridor walls, ensuring smooth navigation. The state-based navigation logic, which dynamically switched between following the left or right wall, proved effective. However, occasional misinterpretations of the environment highlighted the need for further refinement in distinguishing between obstacles and corridor boundaries.

The third challenge was focused on the detection of doors and navigation through cylindrical objects of different colours. The integration of image processing techniques enabled the robot to accurately identify and move towards target doors. The system's performance in this challenge demonstrated the importance of reliable vision-based navigation, especially in environments with distinct, recognisable features.

Throughout these challenges, the project placed significant emphasis on the importance of seamless integration between different ROS nodes and the critical role of real-time data processing. The transition between challenges required careful coordination in order to ensure that the robot could adapt to new tasks without manual intervention. Performance analysis revealed that while the system performed well under controlled conditions, certain scenarios, such as the initial activation of nodes and the robot's behaviour in complex environments, presented opportunities for improvement. The Real Time Factor (RTF) analysis revealed the necessity for optimising the balance between simulation and real-world performance in order to enhance the robot's responsiveness and accuracy.

In conclusion, this project has demonstrated the feasibility of using ROS for developing an autonomous navigation system for the Turtlebot3 burger mobile robot. The challenges addressed and the solutions implemented provide a solid foundation for future work in enhancing the robot's capabilities. Future improvements could focus on refining the obstacle avoidance logic, improving the robustness of line detection, and optimising the transition mechanisms between different navigation tasks. By building on these insights, further advancements can be made towards achieving more sophisticated and reliable autonomous robotic systems.

V. References

1) https://github.com/judy3378/projet.git
2) https://docs.opencv.org/master/d6/d00/tutorial_py_root.html
3) https://docs.opencv.org/4.2.0/pages.html
4) https://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython