ROS project

Viviane BAO
Master SAR
Student number 3800857

Hao YUAN
Master SAR
Student number 21117163

I. INTRODUCTION

In the context of this project, we have been tasked with controlling the trajectory of a Turtlebot3 burger robot using the open source 3D environment simulator Gazebo. The robot must complete three challenges: lane following, movement in a corridor, and navigation in a cluttered environment. Our focus is on the navigation system of a robot, which utilizes images from a simulated/real camera to detect and follow specific lines, as well as a laser scan from a simulated/real LDS sensor to detect and avoid certain obstacles.
The purpose of the report is to analyze the ROS architecture and its performance.

II. PRESENTATION OF THE ROS ARCHITECTURE

A. A short overview

This section presents the ROS architecture that we have developed. Our architecture is centered around a navigation node that can control the robot's movement based on received instructions and environment detected by sensors, such as a camera and a laser scanner. Our ROS architecture consists of two nodes: `path_tracker` and `PathTracker_after_corridor`.

The `path_tracker` subscribes to two topics: `/camera/image and /scan`, which provide data from the camera and laser scanner respectively. The `/camera/image` topic publishes messages of type `sensor_msgs/Image`,which represent images acquired from the camera. The /scan topic publishes messages of type `sensor_msgs/LaserScan`, which represent the range readings acquired from the laser scanner.
The node also publishes to the `/cmd_vel` topic, which controls the movement of the robot. The messages published to this topic are of type `geometry_msgs/Twist` which contain linear and angular velocity components.

The `path_tracker` subscribes to the `/camera/image` topic and converts the image data to the OpenCV format.OpenCV (Open Source Computer Vision Library) is a popular open-source computer vision library that provides a wide range of image and video processing functions. By converting the image data to the OpenCV format, the `path_tracker` node can use the various image processing functions provided by the library to extract relevant information from the image, such as lane following, obstacles, or corridor.

The `PathTracker_after_corridor` node applies the same logic used for lane following in the first node.

Overall, our ROS architecture is designed to enable a mobile robot to navigate autonomously through a complex environment, avoiding obstacles and following corridors.

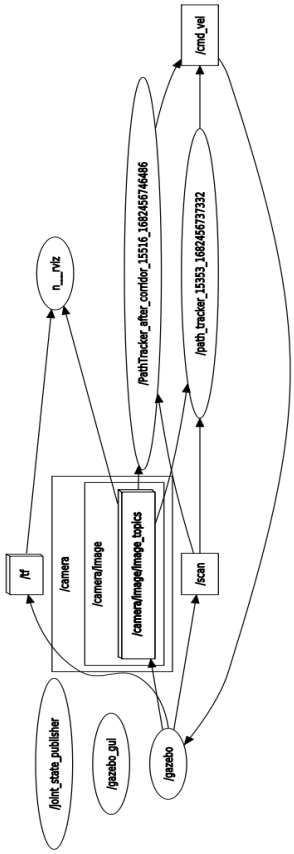


Fig. 1. Our ROS Architecture

B. Algorithmic structure of the architecture

In this section, we will elaborate on the interplay between the different nodes in our system architecture. The robot is equipped with three separate controllers for different modes of operation: normal mode, obstacle avoidance mode, and corridor mode. Depending on the situation and the data received from sensors, the robot can switch between these modes to navigate through its environment effectively.

Normal mode is the default mode when the robot does not detect any obstacles or corridors.
The `path_tracker` node subscribes to the `/camera/image` topic. The received image is converted to the OpenCV format, and a region of interest is selected. The image is then binarized and the centroid coordinates of the binary image are calculated using the cv2.moments function. The error is computed as the difference between the x-coordinate of the centroid and the center of the image. This error is used as input to the PID controller. The controller generates control commands that adjust the robot's angular velocity to maintain its position within the designated range. Finally, these commands are published to the `/cmd_vel` topic, allowing for precise adjustments to the robot's trajectory.

When an obstacle is detected in the robot's path, it switches to obstacle avoidance mode (within a certain range of the robot, based on the readings from the laser scanner). The robot uses its obstacle detection controller to identify the obstacle and navigate around it before returning to normal mode. The node monitors the data within 118 degrees in front of the robot at all times. If there is data less than 0.2, the robot enters obstacle avoidance mode. There are three behaviors for the robot in obstacle avoidance mode. The robot first generates three data: front distance data, left distance data, and right distance data. If the front distance data is less than 0.25, the node will publish twist.angular.z = -1 to `/cmd_vel`, which will cause the robot to turn right. If the front distance data is greater than 0.25 and the left distance data is greater than the right distance data, the node will publish messages to make the robot turn left. If the front distance data is greater than 0.25 and the left distance data is less than the right distance data, the robot will turn right.

PID controller is used to adjust the angular velocity of the robot, which helps to smoothly and accurately navigate around obstacles. The controller calculates the error between the desired angle and the actual angle of the robot, and uses this error to adjust the angular velocity.

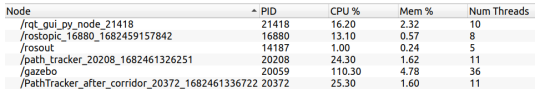
When navigating through a corridor, the robot switches to corridor mode. The robot utilizes its corridor following controller to keep it centered within the corridor. If an obstacle is detected within the corridor, the robot switches to obstacle avoidance mode before reverting back to corridor mode once the obstacle has been successfully avoided (in our case, there is no obstacles in the corridor). Whether the robot enters corridor mode is determined by radar data. If the left and right distance data of the robot are both less than 0.3 and the front distance data is greater than 0.35 and less than 2, the robot enters corridor mode.

In corridor mode, the robot adjusts its operating parameters by using the distance values on the left and right sides to keep the robot on the centerline of the corridor. There are four behaviors for the robot in the corridor: if the front distance data is less than 0.6 and the left distance data is greater than the right distance data, the node will publish twist.angular.z = 0.07 to `/cmd_vel`, which will cause the robot to turn left. If the front distance data is less than 0.6 and the left distance data is less than the right distance data, the node will publish twist.angular.z = -0.07 to `/cmd_vel`, which will cause the robot to turn right. If the front distance data is greater than 0.6 and the left distance data is greater than the right distance data, the node will publish twist.angular.z = 0.025 to `/cmd_vel`, which will cause the robot to turn counterclockwise at a slow and steady pace. It is important to note that increasing the value of twist.angular.z to a higher value could lead to stability issues or even tipping over if the speed is too high.

III. Performance characterization

We will now proceed to evaluate the performance of our ROS architecture for this section.
This evaluation will be based on the analysis of the:
-Response time: refers to the time taken by a ROS system to respond to a request. In our case, when we execute the Python file after launching the launch file, there is a delay before the robot starts moving.To solve this problem, we can adjust ROS parameetr such as the frequency at which messages are published (in our code, we instruct ROS to publish messages at a rate of 10Hz).
-Stability : refers to the ROS system's ability to maintain reliable and consistent performance over time. In our case, repeated launch of files results in proper functioning without any errors or unexpected behavior.
-Execution time : refers to the time required to perform a specific task. In our case ,the robot takes 5 minutes and 18 seconds (real time) to complete the lane following, obstacles (linear velocity of 0.05 m/s), and corridor (linear velocity of 0.03 m/s). We used this command : rostopic echo `/cmd_vel/linear/x` to display the current values of the linear velocity on the x axis of the robot. We set the robot's speed to 0.05 m/s to enable it to detect obstacles and avoid them (If we increase the linear velocity, the robot may behave unexpectedly, like deviating from the path or reversing when it is between two obstacles). In the corridor, the linear velocity of the robot is set to 0.03m/s to ensure better safety (to prevent the robot from crashing into the wall). So, these low but reasonable velocities are needed to succeed challenges.
-And resource consumption: a well-designed ROS architecture should use resources (memory, processor, etc) in an optimal way. We used the command: `rosrun rqt_top rqt_top` to display a real time view of the ROS computation graph and

the CPU and memory usage of each node. Here is a screenshot of the data we collected:



Node	PID	CPU %	Mem %	Num Threads
/rqt_gui_py_node_21418	21418	16.20	2.32	10
/rostopic_16880_1682459157842	16880	13.10	0.57	8
/rosout	14187	1.00	0.24	5
/path_tracker_20208_1682461326251	20208	24.30	1.62	11
/gazebo	20059	110.30	4.78	36
/PathTracker_after_corridor_20372_1682461336722	20372	25.30	1.60	11

Fig. 2. resource consumption of our ROS architecture

Moreover, the robot's behavior during obstacle avoidance is imperfect (the movement is not smooth). Similarly, it may happen that the robot straddles the lane or is not perfectly centered at the end of the corridor. These mentioned problems can affect the quality of the performance of the ROS architecture. To fix all the problems we mentionend, we can maybe try different algorithms for obstacle avoidance to achieve smoother movements and better performance or we can further reduce the linear velocity of the robot to improve its accuracy, or we can improve the camera or perception system of the robot, it will help to increase the performance of our ROS architecture.
One more problem is that we were not able to create the transition between the end of the corridor and the rest of the course. If we use the same code for this new section, the robot is unable to detect the sudden deviation from the trajectory and may not follow the indicated path.

IV. Conclusion and perspectives

The integration of sensor data from a camera and laser scanner, combined with the use of multiple controllers, enables the robot to navigate complex environments autonomously. It can follow lanes, avoid obstacles, and stay on track while traversing through corridors. This approach ensures that the robot can navigate effectively while also avoiding potential hazards.

If we had more time, we could have found a way to create a smooth transition between the end of the corridor and the continuation of the path. We could have also improved its obstacle avoidance behavior. All of these changes could improve the robot's movement's fluency.

Moreover, We could have also completed the last challenge (navigation in a cluttered environment).
We have considered using SLAM (Simultaneous Localization and Mapping) algorithm, it can bu used for robot navigation and localization in unknown environments. By utilizing sensor data, such as cameras and LiDARs, the algorithm is capable of constructing a map in real time and enabling autonomous navigation of robots. This allows the robot to update its own position and map information in real time, which can help it more accurately plan paths, avoid obstacles, and perform tasks in unfamiliar environments.
Unfortunately, we spent a lot of time fine-tuning the various parameters to allow the robot to walk properly for the first two challenges, therefore, we didn't have enough time to address the final challenge.

This project is a great example of the potential of ROS in robotics applications and a valuable resource to develop similar systems.