# Our ROS project

Nemanja MAKSIMOVIC
*Master M1 SAR*
Student number 28706774

Reda KARA ZAITRI
*Master M1 SAR*
Student number 21212631

***Abstract*—This is our report about the ROS project. In this article, we present and explain the ROS architecture used to address the requested challenges. We also justify our choices and clarify the role of each element in the architecture.**

## I. Introduction

During this project, we implemented a code that enables the Turtlebot3 to navigate autonomously in a challenging environment. This was done in simulation using Gazebo and in reality, with the goal of navigating from the starting point to the endpoint without external interaction. The navigation had to take into account:

- Images obtained from a simulated/real camera to detect and follow lines
- Laser scans obtained from a simulated/real Lidar scan (LDS) to detect and avoid obstacles
- Both sensors to navigate in a challenging environment where both are required together

We established a ROS architecture built from different nodes that communicate through various topics with different messages to face the following challenges:

- Line following
- Obstacle avoidance
- Navigation through a corridor
- Navigation between pairs of pillars

## II. Presentation of the ROS Architecture

### A. A short overview

Until now we have two nodes working separately, the "Line follower" that addresses the first three challenges (line following, obstacle avoidance, corridor navigation). The second node addresses the challenge of navigating between pairs of pillars. They are both subscribed to the "/camera/image" and the "scan" topics to receive images from the robot's camera for the first topic, and the second one allows them to detect obstacles using the LDS data. They are also both publishing to the "/cmd_vel" topic to make the robot move according to the processed data. Our system is implemented using Python 3, OpenCV, Numpy and Rospy.

### B. Algorithmic structure of the architecture

The first node's code is organized into a class with several interconnected methods, each responsible for a specific part of the system:

- **Image callback**: This method retrieves images from the robot's camera. We use the cv_bridge module to convert the captured images into a format compatible with OpenCV. The images are cropped and then processed to detect objects based on color (detect_color, detect_red). We define HSV color ranges to detect different colors and use line detection algorithms to identify patterns therefore, the lines on both sides of the road. This method also determines which control methode to use based on the current challenge.
- **Scan callback**: This method retrieves Lidar data as an array of 365 elements to detect obstacles around the robot and their distances. It extracts relevant Lidar data and determines the direction of the nearest obstacle. Thresholding methods are used to identify significant obstacles.
- **Movement control methods**: These methods use camera and Lidar information to send movement commands to the robot. Each challenge is associated with a specific method.

### C. Attributes Usage

Several attributes store important data and configuration parameters, controlling the detection system's behavior. Key attributes include:

- **white_hsv, yellow_hsv**: HSV ranges for detecting white and yellow colors, used in color detection functions.
- **orange_bgr, beige_bgr**: BGR values for drawing circles around detected points for visualization.
- **d_obstacle, left_obstacle, right_obstacle**: Boolean attributes indicating obstacle detection in front, left, or right of the robot.
- **last_detection**: Stores the last line detection, indicating left or right, used for navigation adjustments.
- **low_H, low_S, low_V, high_H, high_S, high_V**: HSV ranges for detecting yellow lines.
- **default_left, default_right**: Default positions of left and right lines in the image, used as reference points.
- **bridge, image_sub, cmd_vel_pub**: ROS communication objects for image and velocity command topics.
- **reversed**: Indicates if the line sides are reversed, adjusting the line following behavior.

### D. Topics Usage

The system uses several ROS topics for communication:

- **/camera/image** (sensor_msgs/Image): Publishes images captured by the robot's camera, triggering the image callback function.

- **/scan** (sensor_msgs/LaserScan): Provides Lidar data as laser scans, triggering the scan callback function.
- **/cmd_vel** (geometry_msgs/Twist): Publishes movement commands for the robot, specifying linear and angular velocities.

### *E. Movement Control Methods*

For each challenge, a specific method controls the robot's movement. The robot detects red lines to switch between challenges, using the appropriate control method based on the current challenge:

- **controlor**: For line following, calculates horizontal and vertical differences between the line center and image center, adjusting linear and angular velocities accordingly.
- **determine_move_direction**: For obstacle avoidance, changes angular velocity based on obstacle direction. If no obstacles are detected, it uses the line following method.
- **determine_move_direction_corridor**: For corridor navigation, determines optimal movement direction based on distances to obstacles on each side, considering distant obstacles detected by the Lidar.


Fig. 1. Example of line following.

### *F. Important Parameters*

We use `rospy.get_param` to retrieve parameters from the ROS parameter server, allowing easy configuration via launch files:

- **/speed_linear, /speed_angular**: Control linear and angular speeds, crucial for balancing speed and trajectory accuracy.
- **/angle_detection**: Determines the Lidar detection angle, focusing on obstacles within a specific angle range.
- **/distance_limit**: Sets the distance limit for obstacle detection, defining when an obstacle is considered in the robot's path.

TABLE I
OPTIMAL PARAMETER VALUES

| Parameter | Value |
|---|---|
| speed_linear | 0.1 m/s |
| speed_angular | 0.5 rad/s |
| angle_detection | 50 degrees |
| distance_limit | 0.31 m |

## III. SECOND NODE: DOORS

### *A. Architecture and Functionality*

The second node addresses the challenge of navigating between pairs of pillars. Key methods include:

- **pose_callback**: Extracts the robot's orientation from odometry data, converting quaternions to Euler angles.
- **image_callback**: Converts received images to OpenCV format, detects color contours in the region of interest, and identifies potential obstacles based on contour size.
- **rotate**: Rotates the robot by a specified angle and angular speed, publishing rotation commands until completion.
- **scan_callback**: Processes Lidar data to detect pillars, guiding the robot through predefined steps for obstacle avoidance, pillar detection, and alignment.

### *B. Attributes Usage*

Several attributes store information and states necessary for autonomous navigation:

- **speed_linear, speed_angular, angle_detection, distance_limit_bottle**: Parameters controlling robot speed, Lidar detection angle, and distance limit for obstacles.

TABLE II
OPTIMAL PARAMETER VALUES FOR DOORS NODE

| Parameter | Value |
|---|---|
| speed_linear | 0.1 m/s |
| speed_angular | 0.4 rad/s |
| angle_detection | 90 degrees |
| distance_limit_bottle | 0.15 m |

- **first_step, second_step, third_step, fourth_step**: Boolean attributes indicating the robot's progress through navigation steps.
- **obstacles**: List of booleans indicating detected obstacles' colors in different camera regions.
- **accomplished**: Integer representing the number of completed navigation stages.
- **pos_z**: Stores the robot's orientation from odometry data, used for controlling rotation during navigation.
- **first_bottle, t1**: Used for detecting the first pillar and recording the initial rotation angle.

### *C. Navigation Steps*

The robot follows several steps to navigate between pillar pairs, completing three pairs to validate the challenge. The steps are:

- **First Step**: Triggered upon initial obstacle detection, adjusts movement to approach the first pillar, recording the current orientation angle.
- **Second Step**: Once close to the first pillar, rotates to find the second pillar of the same color.
- **Third Step**: Advances towards the second pillar while maintaining a safe distance, recording the orientation angle.
- **Fourth Step**: Aligns the robot between the two pillars based on recorded orientation angles.
- **Fifth Step**: Advances and adjusts trajectory if too close to a pillar, completing the step when no obstacles are detected nearby.

## IV. PERFORMANCE CHARACTERIZATION

To prove their effectiveness we tried to measure performance for each challenge.

- challenge 2: For obstacle avoidance, we can verify that the robot maintains a safe distance from the objects it detects in its environment. To check performance, we

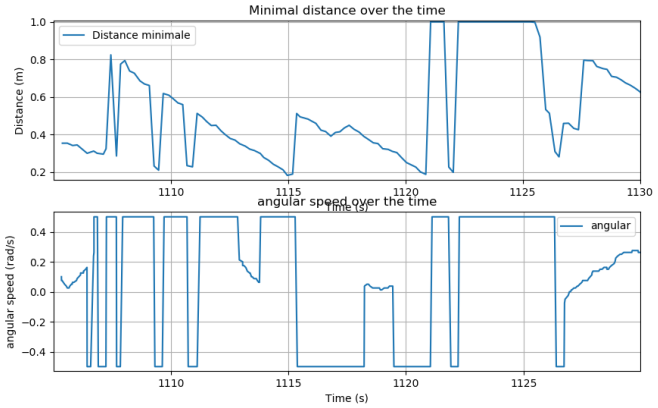


Fig. 2. challenge 2.

imposed a limit distance to obstacles of 0.31 m. We can observe on the first graph the distance of the closest obstacle to the robot, and on the second graph we can observe the angular speed of the robot. We note that the robot will manage to respect a safety distance of at least 0.2 meters, which is still lower than the imposed distance limit. We also note that when the distance to the nearest obstacle is less than the imposed limited distance, the robot will suddenly change its angular speed in order to avoid the obstacle.

- challenge 3: For corridor navigation, you can observe the distance between each part of the wall and see how the robot reacts. We see that when the difference between the distances to the walls will be too great. The robot will readjust its trajectory.
- challenge 4: To measure the performance in navigating between the pillars, we can estimate the angular value the robot is supposed to have while passing between the

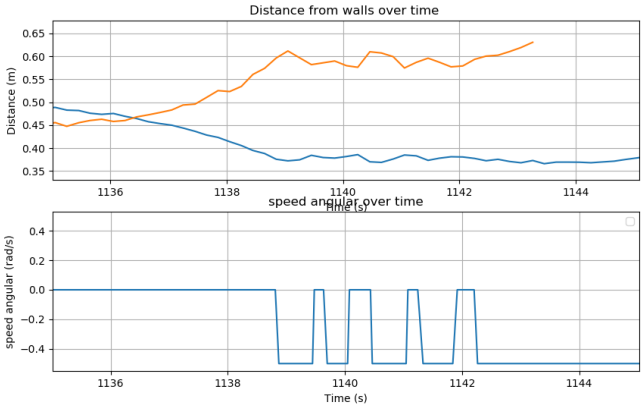


Fig. 3. challenge 3.

pillars and measure the actual value during the simulation. From this, we can calculate the relative error.

TABLE III
PERFORMANCE OF CHALLENGE 4

| Pillar pair number | estimated value (rad) | actual value (rad) | relative error (%) |
|---|---|---|---|
| First | $\frac{3\pi}{2}$ | 4.55 | 3.5 |
| Second | 0 | 0.17 | 2.6 |
| Third | $\frac{\pi}{2}$ | 1.80 | 12.7 |

We see that the relative error is quite small. To improve performance, we can always change the parameter value until we find a better configuration.

## V. CONCLUSION

This project was an invaluable experience, presenting numerous challenges that required extensive research, debugging, and effective communication to ensure we stayed aligned with our objectives. Although time constraints limited our implementation, future work could involve more complex algorithms. For instance, enhancing line following by creating a navigable zone based on camera and LDS data to optimize the robot's trajectory. Additionally, using a Kinect camera to map pillar positions by color and employing a controller based on odometry data would refine the robot's navigation in the last challenge. These improvements could be explored further in our future careers.

## ACKNOWLEDGMENT

We thank our professors and colleagues for their support and guidance throughout this project.