Simple Reactive Robot

Carlos Veríssimo
Department of Informatics Engineering
FEUP
Porto, Portugal
up201907716@up.pt

Miguel Amorim
Department of Informatics Engineering
FEUP
Porto, Portugal
up201907756@up.pt

Rafael Camelo
Department of Informatics Engineering
FEUP
Porto, Portugal
up201907729@up.pt

***Abstract*—In an era defined by dynamic and ever-evolving technological landscapes, the development of a REACTIVE (Real-time, Environment-Adaptive, and Contextually-Intelligent) robot under the aegis of the Robot Operating System (ROS) represents a pivotal step forward in the realm of robotics. This ambitious project aspires to create a robotic entity with unparalleled adaptability, reactivity, and intelligence in response to its environment. The simulation is performed in ROS, which runs the Python code and integrates it with the Gazebo simulation environment. The reactive robot is generated in a random start position. It can successfully follow the wall, avoid hitting, and stop in the desired end position.**

***Index Terms*—Robotics, Intelligent, ROS, Gazebo, Simulation, Sensor, Actuator, Navigation, Environment**

I. Introduction

As technology continues to advance, autonomous mobile robots are increasingly finding applications across various sectors, including corporate environments, industrial settings, healthcare facilities, educational institutions, agriculture, and even in everyday households. In the context of mobile robot navigation, many scenarios require the implementation of wall-following behaviors. These behaviors allow the robot to navigate along the contours of walls or obstacles while maintaining a safe and consistent distance from them. These autonomous robots exhibit the capability to operate in diverse environments, which can be characterized by non-linearity and partially real-time observations. To address challenges of this nature, a multitude of techniques and solutions have been explored and studied.

II. State of the Art

Robot navigation is a fundamental aspect of robotics that focuses on enabling robots to move autonomously and safely in their environment. It involves the development of algorithms, sensors, and control systems to guide robots in tasks such as exploration, mapping, path planning, and obstacle avoidance. Here are some key aspects of robot navigation: **Sensors -** Robots rely on a variety of sensors to perceive their surroundings. These sensors include cameras, LIDAR (Light Detection and Ranging), ultrasonic sensors, inertial measurement units (IMUs), encoders, and more. These sensors provide data on the robot's position, orientation, and the presence of obstacles. **Simultaneous Localization and Mapping (SLAM)**: SLAM is a critical technology in robot navigation. It enables a robot to build a map of its environment while simultaneously determining its own position within that environment. This is essential for autonomous navigation and exploration. **Path Planning**: Path planning algorithms help robots find a safe and efficient route from their current location to a target destination while avoiding obstacles. These algorithms take into account the environment's layout and the robot's physical constraints. **Local vs. Global Navigation**: Robots often employ a combination of local and global navigation strategies. Local navigation focuses on short-term decisions, like avoiding immediate obstacles, while global navigation involves planning routes to long-term goals. **Reactive vs. Deliberative Navigation**: Reactive navigation involves immediate reactions to sensory input, while deliberative navigation considers a broader context and plans actions in advance. Many robots use a combination of both.

Robot navigation is a complex field with applications in areas such as autonomous vehicles, industrial automation, search and rescue, space exploration, and agriculture. As technology continues to advance, robots are becoming increasingly capable of navigating a wider range of environments and scenarios.

III. Robot Implementation and Architecture

A. Robot Specification

For the implementation of the reactive robot, a simulated version of the Turtlebot3 [1] robot was used. Throughout the development, we mostly used the *burger* model of Turtlebot3 to test our simulation, however, one can change the model just by using a different environment variable. More on that in section III-D.

The robot is equipped with a LIDAR sensor, which is used to detect the distance to the walls and obstacles. The robot's movement is classified as differential movement, and as such, has two wheels, the wheels are controlled by the *wheel_left_joint* and *wheel_right_joint* joints.

The figure 1 shows the Turtlebot3 Burger in the real world and its simulated counterpart in Gazebo.

B. The Environment

The simulation environment is a question-marked shaped world. It was created using the Gazebo building editor tool.

[1]https://www.turtlebot.com/turtlebot3/

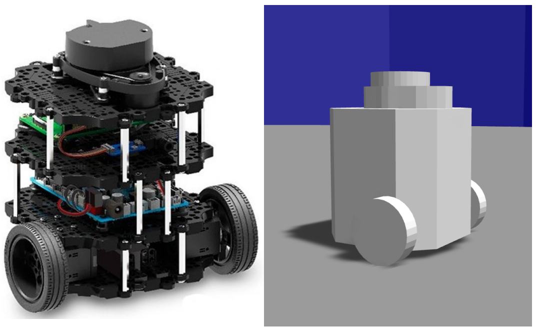

Fig. 1. Turtlebot3 Burger in the real world (left) and in Gazebo (right)

Project specifications specify that the bottom is straight with an edgy shape.

The figure 2 is a visualization of the environment, in Gazebo:

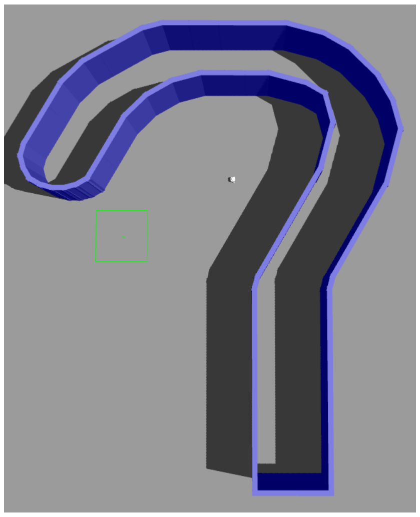

Fig. 2. The simulation world

C. Packages

Inside our workspace, two packages were created: *my_reactive_robot* and *my_controller*.

The first package:

- Launches Gazebo, by running both *gzserver.launch.py* and *gzclient.launch.py*
- Publishes the robot state with the *robot_state_publisher* node
- Spawns our Turtlebot3 robot in the simulation environment, with the *spawn_entity.py* service, from the package *gazebo_ros*

The second package, as the name suggests, is responsible for controlling the robot. It contains the following nodes:

- *scan_to_velocity_node*: This node subscribes to the *scan* topic, which contains the LIDAR data, and publishes the velocity commands to the *cmd_vel* topic, which controls the robot's wheels.
- *spin _randomly _node*: This node is a workaround that we created to allow for a random initial robot orientation. It spins the robot randomly for a few seconds and then stops it.

The robot's initial position and pose are random but are constrained to the round area of the question mark.

D. How to Run the Simulation

To run the simulation, you will need the following ROS packages/dependencies:

- *turtlebot3_gazebo*: This package contains the Turtlebot3 models and the Gazebo plugins.
- *gazebo_ros*: This package contains the Gazebo ROS interface.

Assure that you have the necessary dependencies and that you have sourced your ROS distribution's setup file. Please refer to the README file in the project's repository, if you're having any issues with this process.

Build the package and source the setup file:

1) `cd FEUP-RI-PROJ` - Go to the project's root directory
2) `colcon build --symlink-install` - Build the packages
3) `source install/setup.bash` - Source the setup file
4) `export TURTLEBOT3_MODEL=burger` - Set the Turtlebot3 model. You can change this to *waffle* or *waffle_pi*.

Run the simulation:

1) Open two terminal windows.
2) Run `ros2 launch my_reactive_robot launch.py` in one of them.

This will launch Gazebo and spawn the robot in the simulation environment. Please wait for Gazebo to load before running the next command.

In the other terminal window, run `ros2 run my_controller controller_node`. This will launch the nodes that control the robot.

Please note that we set up Gazebo so that it starts with a paused state. This means that the simulation will not start until you press the play button in Gazebo's GUI.

E. Implementation

In addition to the tangible physical limitations of the robot, the robot controller must adhere to a primary constraint: it must operate without any recollection of past actions and/or states. Keeping this constraint in focus, the controller script is structured into three primary conditional evaluations:

1) The controller checks if the robot has reached the final position.
2) Is the robot lost.
3) The robot confirms it is within follow distance from the closest wall.

a) If it is too far away, then the robot will turn and face the wall, moving closer to it.
b) If it is close enough, then it will enter the follow wall algorithm.

1) Final Position: The robot has reached the final position when it is directly at the center bottom of the question mark.

In order for the robot to recognize that it reached the center of the wall both of the first and last LIDARs that detect a wall need to detect the wall at the same distance.

Additionally, for the robot to recognize that the wall it has reached is the bottom wall of the question mark, the calculated length of the detected wall has to match the length of the actual bottom wall. In order to achieve this, we decided to use the law of sines, this law was chosen because of the data we had: a) we have the distance from the robot to the two edges of the wall; b) we have the angle between the robot and both edges of the wall and c) we had the guarantee that the other two angles of the triangle were equal because the distance to both edges of the wall is the same.

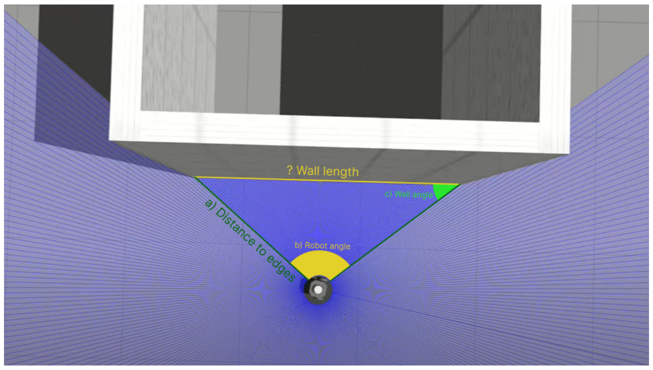

Fig. 3. Data needed to compute robot's end position

2) Lost Robot: The robot is considered lost when no walls are detected in its LIDAR scans.

In this scenario, given that the robot lacks a concrete sense of direction, our approach was to maintain continuous forward movement. This particular state doesn't carry significant contextual significance, and it is not anticipated to happen during a standard program iteration.

3) Move to Wall: In order to better follow a wall the robot should move closer to it before entering the follow wall algorithm.

To enhance the robot's ability to approach the wall, it's imperative that the robot aligns itself perpendicular to the specific wall it intends to move closer to. To achieve this, the robot should rotate so that any of its front LIDARs detect the wall in the closest proximity.

Once the robot is properly oriented towards the wall, it can proceed by moving directly forward.

To ensure the robot effectively reduces the distance between itself and the wall, we employ a spinning action in the direction that will require the least amount of rotation while, simultaneously, we apply a modest amount of forward movement in order to accelerate the overall process.

4) Follow the Wall: For the robot to follow the wall, it needs two conditions met, 1) it needs to be close to the actual wall, and 2) its left or right sensor needs to be perpendicular to the wall it wants to follow.

Given the simplicity of visually identifying the shortest trajectory, we have opted to designate the left sensors as the primary sensors for analysis.

In theory, for a robot to effectively track a trajectory parallel to a wall, it is essential that its left-front sensors consistently maintain the same distance from the wall as the corresponding left-back sensors.

Our follow-wall algorithm is divided into four main conditions:

1) The left sensor is not detecting any wall. In this situation, the robot will spin in the direction that requires the least amount of rotation in order so that the left sensor detects a wall.
2) The left sensor detects a wall to close to the robot. We prioritize the robot's safety by instructing it to execute a rightward spin while maintaining a slow forward pace to prevent collisions.
3) The left-front sensor and the left-back sensor return the same distance to the wall. This is our main goal and the state we want to maintain the robot in, when this occurs the robot is travelling parallel to the wall and will move forward.
4) The left-front sensor and the left-back sensor return different distances from the wall. In this situation, the robot should continue its forward motion while applying a slight spin to rectify this distance gap.

IV. Experiments

As previously stated, the robot's initial pose is random: the robot spawns randomly inside a circle area with 1 meter of radius at X = 2 and Y = 2.5.

To randomize the robot's orientation, specifically its yaw, we created a workaround node that spins the robot with a random angular velocity (z axis) for a few seconds and then stops it. This node is called *spin_randomly_node*.

We have ran two different experiments:

- In the first one, the robot's position is fixed at X=2 and Y=2.5, and its orientation is variable.
- In a second experiment, it's orientation is fixed, and we randomize the spawn coordinates

A. Experiment 1

In order to test how the robot's starting orientation affects the loop-time, we have ran the simulation 11 times, only changing the yaw (orientation at z) values, that range from -1 to 1, with a step of 0.2.

The table below shows the loop time in regards to the yaw value.

Another way to visualize this data is in an radar chart, with this chart the user can better understand in what orientation the robot performed worse and what the change between each orientation is.

B. *Experiment 2*

The second experiment tests the robot's performance, at different initial spawn positions.

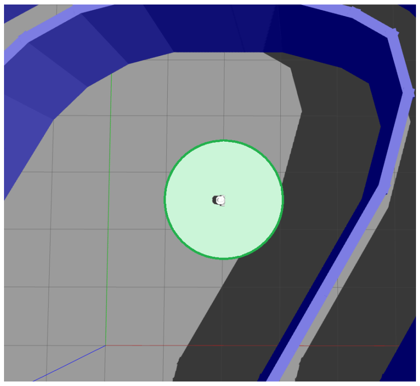

Fig. 4. The robot's initial position

The green area in figure 4 shows the possible initial position of the robot. We've conducted this experiment 10 times, and the robot's orientation was fixed at 0.0 (facing right on the image).

V. Results and Discussion

A. *Experiment 1*

The goal of this experiment was to analyse how much of an effect the orientation of the robot relatively to the closest wall affects it's travel time.

As previously mentioned, the robot's position was fixed at X=2.0 and Y=2.5. When viewed from a bird's-eye perspective, the closest walls to this point are situated either to the right or at the top. Consequently, it was anticipated that the robot would require more time when facing the bottom-left direction, as evidenced by the maximum time taken when the robot's yawn rotation was set to 0.6.

This result aligns with our expectations, based on the design of our robot's controller. However, it may not immediately appear intuitive to a first-time observer. When examining the robot's trajectory, the orientation often ends up being close to 0.6 while it follows the initial curved wall inside the question mark. This behavior arises from the specific distance we agreed our robot needs to maintain from the wall in order to follow it effectively. If we were to increase this threshold, along with expanding the LIDAR's detection range, it is expected that the value of 0.6 should shift from being the slowest to the fastest time, as the robot wouldn't need to make significant adjustments to its initial rotation, unlike other orientations.

B. *Experiment 2*

The goal of this experiment was to analyse how much of an effect the initial position of the robot relatively to the closest wall affects it's overall travel time.

As expected, the robot reaches the end position faster when it spawns closer to the wall. This is due to the fact that the robot doesn't need to move as much to reach the wall, and therefore it doesn't need to rotate as much at the beginning.

As observable in table II, the worst runs are the 6th and 7th runs, to no surprise, as these runs are the ones where the robot spawns in a way that puts the top wall as the closest wall to the robot, either because the x value is too low or the y value is too high.

The fastest runs, on the other hand, are the 4th and 9th runs, where the robot spawns in a way that puts the right wall as the closest wall to the robot.

VI. Conclusions and Future Work

To sum up, this project has proven to be a great way to practice using the Robot Operating System (ROS) and the Gazebo simulation environment. It offered insights into autonomous navigation, sensor-actuator integration, and real-time robotic behavior, making it a useful chance to dig into the complexities of ROS robotics.

Nevertheless, it's important to recognize that more time than expected was spent correctly setting up and launching the project. This demonstrates how crucial it is to comprehend the nuances of ROS and Gazebo in order to execute projects without any problems.

A. *Future Work*

Future work might include improving the robot's ability to spawn inside the environment's walls, which would add a new element of difficulty to the navigation challenge. Consider a situation in which the robot must find its way from within the labyrinthine structure to the bottom of the '?' form, opening up new avenues for study and development.

Moreover, there are a few improvements that our robot would welcome, namely:

1) Presently, the robot adheres to a fixed rotation value. To improve efficiency, it could be advantageous to adjust the rotation based on the specific situation it encounters. For instance, when the robot needs to perform a 180-degree spin, a higher rotation value could expedite this process.
2) Similarly, the robot's movement speed could be made adaptive. It should vary depending on the circumstances. However, it's essential to establish an upper limit to prevent potential collisions.
3) The robot's current approach to stopping when it reaches the middle of the bottom wall lacks a specific endpoint. To enhance its path-following capabilities, it might be valuable to designate a predetermined endpoint for the robot to navigate towards.

VII. Acknowledgments

First and foremost, we would like to express our gratitude for the collaboration and cooperation we have maintained throughout this relatively brief yet intricate task.

Secondly, we thank the course's professors, for their guidance and support throughout the development of this project.

Finally, We would also like to extend our gratitude to the authors of the ROS packages that we used in this project.

Appendix A
Tables

TABLE I
Experiment 1 : Results

Yaw (Z-Orientation)	Time to Finish (s)
-1.0	95
-0.8	90
-0.6	92
-0.4	91
-0.2	91
0.0	93
0.2	87
0.4	91
0.6	103
0.8	94

TABLE II
Experiment 2 : Results

X	Y	Time to Finish (s)
1.81	2.44	94
1.92	2.34	90
2.43	2.88	94
1.87	2.39	94
2.52	2.39	82
2.21	2.90	91
1.59	3.19	130
1.99	3.49	126
2.46	1.67	83
1.62	2.67	102

Appendix B
Images

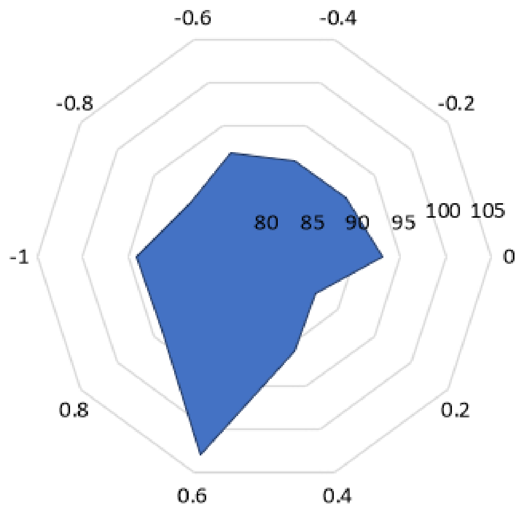

Fig. 5. How the initial rotation affects time taken in seconds

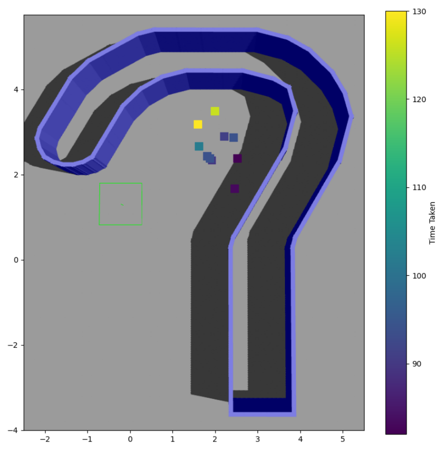

Fig. 6. How the initial position affects time taken in seconds