Simple Reactive Robot to Follow *Cedilla-C* Shaped Map

1st Lisa Sonck
M.EIC, FEUP
up202202272@up.pt

2nd João Pires
M.EIC, FEUP
up201806079@up.pt

3rd Sergio Murino
M.EIC, FEUP
up202203049@up.pt

***Abstract*—This project is related to the development of a reactive robot under ROS. The robot is capable of following the walls that detects, within the environment in which is placed, maintaining a certain distance. The environment used is a *cedilha-C* shaped one. A stopping criteria was defined to make the robot stop when it is located under the *cedilla*.**

I. Introduction

This paper describes a simple reactive robot, which is placed on a 3D environment, but whose movement is limited to the 2D space. The robot position is identified by two two coordinates (x,y) and a direction (Theta). The robot has also a linear and an angular velocity, with respect to the z axis, hence with the usual counterclockwise positive orientation. In order to detect walls and measure distances from them, the robot is equipped with a sensor, which is able to scan the 360 environment that surrounds it. This will be discussed in more detail in the method section.

The model is based on the following assumptions and restrictions:

- It can't have memory, i.e. the robot has not the possibility of tracking the environment taking into account previous measurements.
- It can have a general idea of the map, but not a detailed one. This means that the robot cannot know in advance the coordinates of walls, corners, stopping point and so on, but it should be able to scan them and recognise them based on general information.

Since the coordinates of the end point cannot be known by the robot, this requires the robot to process information from its sensor.

Once the robot is placed, randomly, inside the *cedilla-C*, it should detect the nearest wall, point in that direction and approach it. Once it reaches a chosen distance, say 'd', it has to follow the wall keeping 'd' constant. Then, it will necessarily go outside the cedilla-C shape. The goal now is to keep going forward following the walls and then stop once it is located under the cedilla, which will be approximate by a straight line.

The paper is structured as follow:

- In section II, there's a briefly discuss about the state of the art.
- In section III, there's an explanation of how the problem was approached (description of ideas and algorithms used.
- In section IV, there's an analyse to the obtained results.
- In section V, the conclusions and future work.

II. State of the art

Models based on the "state" of the robot appears to be the most straightforward for the following wall problem [1], [4].

They are built on the idea of giving the robot a set of different states, all defined as having some particular attributes, that are supposed to be updated while the robot proceed with its path.

One of the simplest approach is the one described in [5]. It's the stop-turn algorithm and it is based on the alternating states of turning and going ahead, and states have either a angular velocity and null linear velocity, or the opposite.

Way more complex methods are available, for example based on fuzzy logical system [3], or on Kalman filtering and Hough transform [6].

III. Method

In order to solve the above mentioned problem, ROS (robot operating system) was used, equipped with the Gazebo tool and a burger turtlebot for simulation and Python as programming language.

A. General implementation

A TurtleBot class was defined, which is used to represent the turtlebot in the simulated world and which incorporates all necessary functionality. When an object of the class is created, it becomes a listener to LaserScan messages and it is a publisher of Twist messages. It also keeps track of some necessary variables, like a distance 'd' that should be constant when following the wall, if the stopping criterion has been reached, the state of the robot, a distance from the wall, a linear velocity and an angular velocity. etc.

The state determines what the robot has to do next. A state can change depending on how the robot is located relative to a wall.

Besides this, the class defines the function move, which determines the values for the Twist message depending on the state of the robot.

It also incorporates the callback message from the LaserScan, which will be called every time the scanner scans different values. These values will be used to change the state of the robot. By working with these states, the callback and the

move function, the robot can function autonomously without prior knowledge of the world.

At last there is a shutdown function which shuts down the simulated robot in a proper way.

B. Following the walls - States

An idea similar to the one proposed by Andrew Yong Zheng Dao was followed [1].

The main idea is to have thirteen states that define the robot movements in its environment. Every state is characterised by a number (from 0 to 12), a linear velocity (in the x direction, with respect to the frame of reference centred in the robot) and an angular velocity, which is the angular velocity with respect to the z axis, i.e. the axis perpendicular to the 2D plane where the robot is located.

Some states have a fixed linear and angular velocity, for instance the state 4 has a linear velocity of 0.1 m/s and and angular velocity of 0.1 rad/s. Other states have a constant linear velocity (0), but an angular velocity that is updated based on the what the laser detects.

The state of the robot is updated at each iteration thanks to the callback_laser function, which use the laser scan [2]. Now, let's briefly discuss how this method from the message_msg package works.

The laser scan, gives messages about all the 2D space that surrounds the robot, i.e. 360 degrees measurements of distances. The message type is the following: first of all, it has an header with the acquisition time of the first ray scan; then, it has the start angle of the scan, the end angle and the angle increment. If the laser is moving, there's also a time increment, but this is not the case of this project, since at each iteration it will scan the whole 2D before moving again; finally, it has the ranges which is a vector indexed with the angle of scanning, and its values are distances scanned at the respective angle. For instance, ranges[60] is the distance of the wall when the scan point is at 60 degrees (with respect to the robot direction).

The callback_laser function takes as input the messages from laser scan, and updates the state of the robot, and, if necessary, it change some class attributes. First of all, it creates an np.array from the ranges variable of the laser message, and save the minimum detected distance in the attribute self.distance. Its activity depends on the state of the robot, which are translated in different if statements, that perform different operations based on the identifying value of the state.

Let's clarify this better with an example: imagine the robot is in the state identified with 2, i.e. it has an angular velocity equal to 0 and a linear velocity that is the maximum between 0.1 m/s and one fifth of the difference (in absolute value) between self.d and self.distance, which is the latest update of the distance from the wall (a brief parenthesis is needed here in order to explain where this criterion for the linear velocity came from: indeed the max between these two quantities is being taken, because 0.1 is the default one, but, if the robot is really far from the wall, it is better to increase the velocity in order to reach the wall within a smaller number of steps). Since the robot is in the state 2, it will enter in the third if statement of the function; then, a check is made in order to see if the minimum distance detected from the wall is less then 'd' (the desired distance - to have when following the wall), if this happens then a change of direction becomes necessary, and, in order to do this, the angular velocity is updated, which was 0, proportionally to the difference of angle between the robot direction and the wall. The state is changed to state 3, which is defined as having 0 linear velocity and angular velocity equal to the updated one. If the nested if statement is false, the robot simply keeps going forward.

C. The map

As mentioned before, our map is a *cedilha-C* shaped one, built by the authors of this article. It was built in Gazebo, using non-curved walls (the only ones that were available), which means the 'C' part of the map is not completely round. However, each wall has an average width of 1 meter and a couple of walls were used so that the curve of the 'C' could be achieved in the best possible way.

As it's shown in figure 1, the *cedilha* part of the map is completely straightforward, while the 'C' part is made of consecutive walls that form a curved shape, with the limitation of not being completely curved.

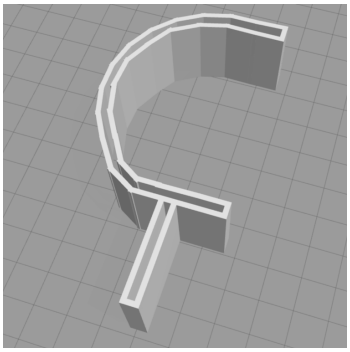

Fig. 1. *Cedilha-C* shaped map used in this project.

Alongside with the map, the standard sun and the standard ground plane from Gazebo, which appears also in the figure 1 bellow the map. The sun is responsible for the illumination of the map, otherwise it would be impossible to see it.

D. The stopping criterion

Since the robot could not have the possibility to know the map with details, i.e. it cannot know coordinates, defining the stopping criterion is one of the most challenging aspects of this work. As previously stated, the goal is to make the robot end its path below the *cedilla* part of the 'C' shape, with whatever orientation.

The idea is to define two flag variables, self.criterion and self.stop, initialised as False, that combined with some states, can make the robot understand when to stop. Thus, a description in detail of how these two variables work in needed.

The intuition is based on the fact that the robot is approaching a ninety degrees corner only once in its path, and that

point is the corner between the *cedilla* and the 'C' in the right side of the *cedilla*. It's the right side because the algorithm is built to make the robot always going out the 'C' from the bottom. Since this happen only once, it can be described with a mathematical condition, and have a bijection between that point and the truthiness of an equation. From basic geometry, it's known that a point nearby a corner is nearer to both the front line and the up line, than it is to the corner. Hence these distances are measured and the evaluation build on top of that.

All the procedure is executed within the callback_laser function. Once the code execution enters in the while loop, the first if statement always checks two conditions: 1) if the minimum distance measured between five degrees angle and minus five degrees angle is less than the minimum distance measured between minus forty and minus fifty degrees angles (with the usual counterclockwise positive orientation this is indeed the diagonal distance when the robot is heading left, measured in a ten degrees range with middle point in minus forty five degrees angle); 2) if the minimum distance measured between minus eighty-five degrees angle and minus ninety-five degrees angle is less than the minimum distance measured between minus forty and minus fifty degrees angles. If both these two condition are true (together with the condition self.criterion = False, i.e. it will enter this first if statement only once), the if statement is verified and the next state is set as state number 12, and the flag variable self.criterion equal to true.

The state 12 has a linear velocity of 0.1 m/s and a null angular velocity. Now the rest of the path for the robot is going down, turn and then stop below the *cedilla*. Hence, once there's the detection of the wall at a distance less than 'd', the state is changed to state number 8 and the robot starts to rotate down. From now on, the robot keeps switching between state 8 and state 9, in order to following the wall of the *cedilla* going down. Once the robot detects infinity in the minus ninety degrees angle, with respect to its direction, this means that the *cedilla* is no more present on the right side of the robot, and so the state is changed to 10, since it needs to rotate and locate itself below the *cedilla*. Indeed the state 10 is defined as having linear velocity of 0.1 m/s and angular velocity of -0.2 rad/s, since the robot must rotate clockwise.

While inside state 10, the angular velocity keeps being updated proportionally to the difference of angle between the bottom of the *cedilla* and the direction of the robot, and once the minimum distance is measured in the range minus ninety-three degrees angle and minus eighty-seven degrees angle, this means that the robot is at the end point and hence the variable self.stop is set to true. Finally, the robot enter the state 9, with linear velocity equal to 0.1, goes straight until it detects infinity in a seventy-five angle range (from zero to minus seventy-five), and finally it stops, setting the final state to 11 with both an angular and a linear velocity equal to 0.

There is a key point in this procedure. Indeed when self.stop is set equal to 0, the reader may argues that the condition "minimum distance measured in the range minus ninety-three degrees angle and minus eighty-seven degrees angle", it does not necessarily happen only once in the robot path. However the function callback_laser is implemented in a way that state 10 is reached by the robot only once that the robot has scanned the corner and then switched the variable self.criterion to true, and only once that it has scanned infinity on its right side. This makes a one to one correspondence between the combination of flags and states and the end point.

E. Choice of parameters

For the minimal distance to the wall, it's used the maximum between 0.7 and the initial minimum distance between the robot and the *cedilla-C* shape. This is done so the robot can be placed in worlds with changing size (a bigger world will give a bigger distance, which also allows more flexibility for the robot). At the same time, there is a lower limit, so the robot is not too close to the wall and does not hit it.

For the linear and angular velocity, sometimes hard-coded values are used. For the linear speed this is 0.1 for states 4, 6, 9, 10 and 12. 0.1 is chosen based on the size of the robot and also relative to the minimum distance to the wall (0.7). This value ensures that the robot is not driving to fast to be able to make enough and accurate measurements, so changes are detected early on and the right actions can be taken on time. For example is the robot drives too fast, it is possible it does not detect a wall on time and hits it. The angular velocities in states 6 and 7 are -0.15 and 0.15 respectively and these are used to keep the robot near the wall, so it doesn't drift too far. These angular velocities are chosen, so the robot can keep driven, but still a good curve can be made. When the robot is again on track the states change, so it drives straight again. For state 10 the angular velocity -0.2 is used, so the robot can make a big enough curve to make a turn around the *cedilla* and can still drive.

When these values are not hard coded they depend on the relative place of the robot to the wall. Angular velocities are determined depending on the number of degrees between the wanted direction and the robot's current direction. The bigger this difference, the faster the robot will turn. This is also the case for the linear velocity, where the value is determined based on the distance between the robot and the wall. If the robot is far away, it will drive faster.

IV. Results and discussions

In order to test the developed robot, a set of executions was made to assess the robot behaviour. Since all the executions were made inside a Virtual Machine with Ubuntu as the Operating System, and considering all the limitations in terms of performance of the VM (an average FPS of less than 2), it was not possible to run a set of executions with the same conditions to check for the average time of completion of the task (start at a random position and stop at the *cedilha* part of the map).

However, all the executions, no matter the starting position (just needed to be inside the 'C' part), resulted in the robot following a similar path in order to achieve a state where the

stopping criterion could be applied, as explained in the section 'The stopping criterion'.

V. Conclusions and future work

The main goal of this project was to create a simple architecture robot able to following walls in a know-shaped world, and with a known end point. The main goal was successfully achieved, in what can be considered as a really simplified approximation of a real environment. Some nice upgrades of this model could be reached by implementing a more sophisticated robot, able to move in a totally unknown environment and with some obstacles in it; moreover the robot could be built in such a way that some obstacles could be hit if this resulted in a big gain of time (or some other quantity), but some other must strictly need to be avoided; and even more there is the possibility to make the robot learn by itself based on how much it gets hurt by them.

It's a known limitation of this project to not have data from execution to analyse the robot performance and, thus, it's recognised as being one of the tasks to be made in the future.

References

[1] Andrew Yong Zheng Dao. Maze escape with wall-following algorithm, 2021. Last accessed 31 October 2022.

[2] ROS Official Documentation. Laserscan message, 2022. Last accessed 31 October 2022.

[3] S. M. Bhagya P. Rayguru Madan Mohan Ramalingam Balakrishnan Elara Mohan Rajesh Muthugala, M. A. Viraj J. Samarakoon. Wall-following behavior for a disinfection robot using type 1 and type 2 fuzzy logic systems. 2020.

[4] Github @nimbekarnd. Wall-follower-in-ros-using-python, 2020. Last accessed 31 October 2022.

[5] RoboWiki. Wall following, 2017. Last accessed 31 October 2022.

[6] Xueming Wang. Wall following algorithm for a mobile robot using extended kalman filter. 2011.