

ENAE450 Turtlebot Maze Final Project Report

Group members: Priyanka Nadella, Samantha Krakovsky, Jay Rana

Contents:

1. Introduction
2. Hardware: Methods, Results
3. Simulation: Methods, Results
4. Retrospective

1. Introduction

In our final project, we were tasked to make a Turtlebot 3 Waffle Pi robot exit a random maze within 3 attempts. To take on this challenge, we brought insights from our previous assignments, which were inside and outside wall following. This report will cover our methods, results, and any challenges we encountered along the way.

2. Hardware:

The problem for the hardware portion of this project was to program the Turtlebot 3 Waffle Pi to exit a random maze within three attempts. To accomplish this, we iteratively designed an algorithm by analyzing our current errors and created solutions to avoid those errors.

The first part of our code that stays consistent with all of our algorithm iterations is the way we capture distances from the Lidar sensor. We store the array of distances from the Lidar sensor in a variable named “self.ranges”. Then, we define four distances. “north_dist” refers to the minimum distance reported by the Lidar sensor in front of the robot, which spans the indices 270 to 450 in the ranges array. “east_dist” refers to the minimum distance reported by the Lidar sensor to the right of the robot, which spans the indices 90 to 270 in the ranges array.

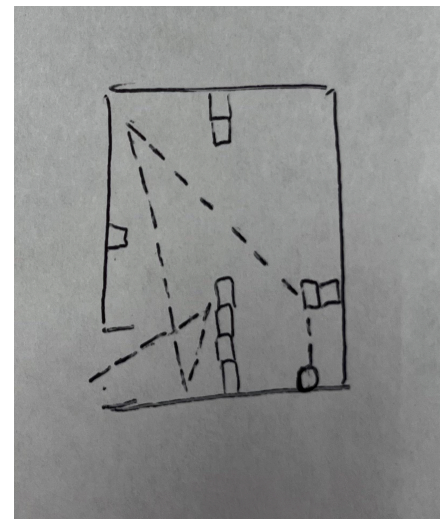
“south_dist” refers to the minimum distance reported by the Lidar sensor behind the robot, which spans the indices 630 to 720 and 0 to 90 in the ranges array. “west_dist” refers to the minimum distance reported by the Lidar sensor to the left of the robot, which spans the indices 450 to 630 in the ranges array.

```
north_dist = min(self.ranges[270:450])
east_dist = min(self.ranges[90:270])
south_dist = min(self.ranges[630:720] + self.ranges[0:90])
west_dist = min(self.ranges[450:630])
```

To get these values, we used our hand to sense where the front and back of the robot were and then calculated left and right accordingly. Once we had our coordinates set, we were ready to implement our first algorithm.

Our first solution was rather rudimentary. We initially set up an algorithm that moves away from the wall when it senses an obstacle. So the algorithm in pseudocode would be:

```
If left_dist < 0.15:
    Gonna crash on the left, go right
If right_dist < 0.15:
    Gonna crash on the right, go left
If front_dist > 0.5:
    Nothing in front, go straight
Else:
    If left_dist == right_dist:
        Rotate 360 degrees ( it is in a 3-wall corner)
    Else:
        Turn left by a certain amount
```



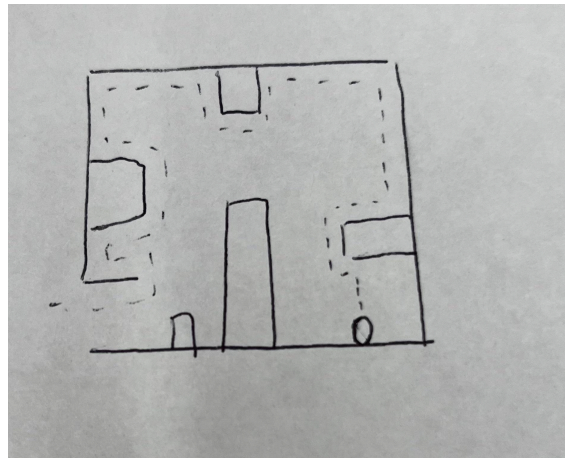
We fixed a certain direction to turn and only turned in that direction for the whole duration.

While we solved the maze with our first attempt, our second attempt just fell short. Our robot got stuck in a loop at the end of the maze, hence not being able to complete it. We quickly realized that bouncing off of walls would not be our solution. We also didn't want to be turning a fixed

amount but rather turn based on the environment around us. Rather than turning a fixed 15 degrees, turn based on how far away from the wall you are. This would allow for closer wall following and more precise results. We wanted to make a full-proof algorithm that would work for any maze and hence wanted to make it as comprehensive as possible. Hence, during our next lab, we went back to the drawing board to come up with another solution.

Our next solution was slightly better articulated. We decided to have a robot that follows the wall, rather than turning away from the wall. Our pseudocode for this algorithm :

```
If left_dist or right_dist < 0.15:  
    Dangerously close turn opposite direction  
If front_dist < 0.5:  
    Turn left  
Else:  
    If wall on right:  
        Go straight  
    If no wall on the right:  
        Turn right by a very small amt.
```



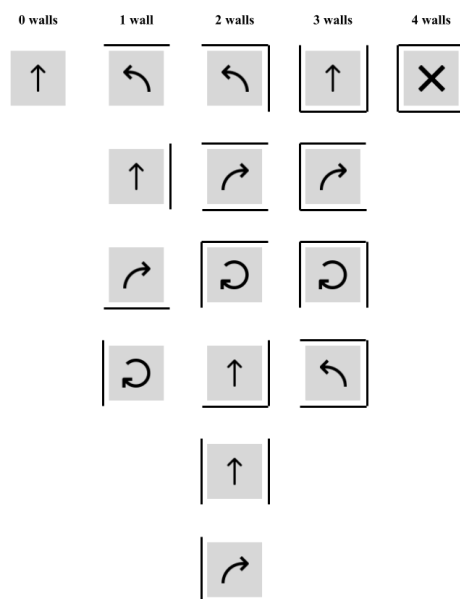
While this was a better solution that would work for any maze, we struggled to get the threshold correct, leading to very unpredictable results. Our robot would get very easily confused in large areas and get stuck figuring out whether there is a wall on the right or not. Although we could've fine-tuned this algorithm, we decided to go forward with an algorithm we felt more comfortable with.

Our final implementation was built on our previous algorithm. The primary goal of the algorithm is to find and follow the right wall in order to solve the maze. We, however, also managed to get the robot to handle every possible configuration of the maze by drawing each scenario out. This would allow for very robust maze-solving no matter the configuration.

















We used the same coordinate system as defined before but also added additional variables. We define a global array that holds the maximum distances that we would allow the Lidar sensor to report before we decide that there is a wall in each direction, and store it in the variable “self.wall_dists”. For our successful runs, all four values were set to 0.3 meters. Then, we define four boolean variables which will be True if the Lidar sensor reports a distance that is less than the defined maximum distances for each direction.

```
north_wall = north_dist < self.wall_dists[0]
east_wall = east_dist < self.wall_dists[1]
south_wall = south_dist < self.wall_dists[2]
west_wall = west_dist < self.wall_dists[3]
```

















Using this, we drew each possible combination of the values and determined how we wanted the robot to behave in each scenario. In the drawing below, the gray box represents the Turtlebot 3, the black lines represent what walls are present in that scenario, and the black arrow inside each box represents how we wanted the robot to move in that scenario. The X in the final scenario represents that the behavior of the robot does not matter, since this scenario is normally impossible.



We use a modified [Karnaugh map](#) (K-map) to group each of these scenarios to simplify the boolean logic. Instead of only using the values 0 and 1 for the K-map, we instead group our behaviors into five categories: “follow wall”, “turn clockwise 90 degrees”, “turn clockwise 180 degrees”, “turn counterclockwise 90 degrees”, and “go forward”. We then use those categories to fill out the K-map. For example, the first row denotes scenarios where there are no north or east walls ($N = 0, E = 0$).

SW		00	01	11	10
NE	00	 follow wall	 turn CW 180°	 turn CW 90°	 turn CW 90°
	01	 follow wall	 go forward	 go forward	 go forward
	11	 turn CCW 90°	 turn CW 180°	 doesn't matter	 turn CCW 90°
	10	 turn CCW 90°	 turn CW 180°	 turn CW 90°	 turn CW 90°

Then, we group the terms with the same category, using the procedure for traditional K-maps. In the figure below, boxes with the same color denote groups that have been grouped together. The yellow and green groups wrap around the K-map.

SW		00	01	11	10
NE	00	 follow wall	 turn CW 180°	 turn CW 90°	 turn CW 90°
	01	 follow wall	 go forward	 go forward	 go forward
	11	 turn CCW 90°	 turn CW 180°	 doesn't matter	 turn CCW 90°
	10	 turn CCW 90°	 turn CW 180°	 turn CW 90°	 turn CW 90°

Using these groupings, we determine the most simplified boolean equations that represent our desired behavior. The chart below denotes what boolean equations relate to each behavior. A line over a variable denotes that this term will be true if the variable is false, and vice versa (e.g. \overline{N} will be true when there is no north wall). For example, the robot will only follow the wall if there are no walls to its north, south, or west.

Behavior	Boolean Equation
follow wall	$\overline{N} \overline{S} \overline{W}$
turn clockwise 90 degrees	$\overline{E} S$
turn clockwise 180 degrees	$N \overline{S} W + \overline{E} \overline{S} W$
turn counterclockwise 180 degrees	$N \overline{S} \overline{W} + N E S$
go forward	default behavior

We implement these terms in our code using the four boolean variables declared earlier.

```
if ((not north_wall) and (not south_wall) and (not west_wall)):
    # follow wall
    self.get_logger().info("Follow")
    if(abs(min(self.ranges[180:270]) - min(self.ranges[90:180])) < 0.06):
        self.publish_vel(x=0.08,y=0.0,az=-0.3)
    else:
        self.publish_vel(x=0.08,y=0.0,az=0.8)
elif ((not east_wall) and south_wall):
    # turn clockwise 90 degrees
    self.get_logger().info("CW 90")
    self.publish_vel(x=0.0,y=0.0,az=-0.5)
elif ((north_wall and (not south_wall) and west_wall) or ((not east_wall) and (not south_wall) and west_wall)):
    # turn clockwise 180 degrees
    self.get_logger().info("CW 180")
    self.publish_vel(x=0.0,y=0.0,az=-0.8)
elif ((north_wall and (not south_wall) and (not west_wall)) or (north_wall and east_wall and south_wall)):
    # turn counterclockwise 90 degrees
    self.get_logger().info("CCW 90")
    self.publish_vel(x=0.0,y=0.0,az=0.8)
else:
    # go forward
    self.get_logger().info("F")
    self.publish_vel(x=0.15,y=0.0,az=0.0)
```

For the “follow wall” behavior, we use additional logic to determine which direction to turn in order to avoid colliding with the wall. If the distance between the closest northeast wall and the closest southeast wall is smaller than 0.06 meters, then the robot will turn clockwise into the wall. Otherwise, the robot will turn counterclockwise out of the wall. The purpose of this additional logic is to avoid corners properly while turning.

Overall, this algorithm follows the right wall of the maze. This is similar to the previous algorithm; however, since we are accounting for all possible combinations of walls being present, the robot will always recover in the event that it loses the wall it was following. This algorithm also attempts to solve the problem we initially found where we sometimes collided with the convex corners of the maze.

During our successful run, we successfully followed the wall and exited the maze. However, we collided with three corners during the run and also encountered a problem where the robot was unable to exit the maze on the first attempt. We will talk about this issue during our retrospective.

3. Simulation

One of the first things we noticed when trying to run the simulation was that while the ranges array for the lidar scan for the actual robot had 720 entries, the one for the simulation only had 360. In order to figure out which side corresponded with different degrees, we had a script that outputs the index of the minimum distance of the lidar scan. We then used Teleop to align each side of the robot, one at a time, with a corner. This way we knew the minimum distance was at the desired side.

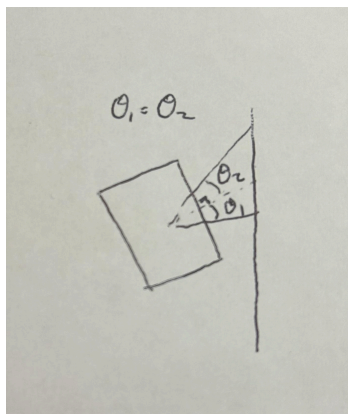
Even though we had a successful run with the hardware code, when we tried to implement the same logic in the simulation, we ran into some difficulties. In-person, even if the robot grazed the wall, the bot could continue on its original path. If the robot hits a wall in simulation it starts rotating about the x and y axes and cannot recover. This meant we needed a much more robust turning and obstacle avoidance algorithm. In the hardware code, when the turn

condition is satisfied the robot will turn in small increments until the condition isn't satisfied anymore. To keep the robot moving parallel to the walls, we decided that it would be better for the robot to turn in 90-degree increments. In the simulation code, when the turn condition is satisfied, the robot checks the distance on the side that we want it to turn towards, saves that distance, and continues rotating until the front distance is equal to that saved distance. For example, if we wanted to perform a 90-degree counterclockwise turn, the robot would check the left distance before the turn, and then keep rotating until the front distance was equal to that original left distance.

Even though this new algorithm resulted in fairly clean turns, the robot still was able to drift toward the walls and bump into them. To rectify this, we implemented another algorithm to keep the robot aligned with the wall.

This algorithm chooses to follow either the right or left wall based on whichever is closer and then checks the distance at two opposite angles on the side of the robot. If the difference between the distances is greater than some tolerance value, the robot rotates so that the angles are equal.

In addition to the new turning and wall alignment algorithms, the robot checks the minimum distance on the side following the wall and rotates and moves away if the distance is too small.



We tried many different algorithms to navigate the maze in the simulation, but our final solution is pretty simple. When the front distance is below a certain threshold, we check the distances on the left and the right and turn whichever way has the greater distance.

Once we detect that the minimum distance around the bot is greater than 1.5 meters, we turn 180 degrees and stop.

4. Retrospective

For the hardware portion of the project, we were on the right track with our algorithm that uses the combination of walls present near the robot in order to determine its behavior. Also, our additional code for following walls minimized the number of collisions with the convex corners of the maze. Finally, our iterative algorithm design process helped us determine what aspects we needed to improve on and eventually create a reliable solution.

We were able to successfully complete the maze within three attempts. However, our first attempt failed since our turning angle around corners was too large, which resulted in our robot incorrectly detecting another wall and looping around. For our second attempt, we adjusted the angular velocities while following the wall, which fixed this issue. However, we also collided with three corners during our attempt. Additionally, our robot was slow and took nearly three minutes to complete the attempt.

There are several parts of the algorithm which we can improve. For example, instead of only relying on the current inputs (i.e. which walls are present) to determine the output (i.e. desired behavior), we could also add a current state (e.g. if the robot is currently trying to follow a wall). Our current algorithm is prone to detecting certain walls too early, which leads to looping certain sections, as seen at 2:00 in our video. If we also kept track of a current state, we could make sure to ignore certain walls while turning around corners.

Another improvement would be to make following the wall smoother. Our current logic for following the wall is slow and jittery since it will either turn clockwise or counterclockwise depending on whether the northeast or southeast wall is closer. A smoother approach would be to vary the direction and magnitude of our turning angle based on the difference between the two direction values.

When working with the simulations, we implemented a better algorithm that corrects itself after turning to return parallel to the wall . With more time, this is something we could've implemented with our hardware code as well. This would lead to smoother turns and better results overall.