ENAE450 Final Report

Mohammad Durrani, Nitin Krishna, Pranav Shah, Karthik Taranath

Introduction

Our team's objective was to successfully navigate and get out of a maze using a TurtleBot robot platform. The challenge involved designing an effective algorithmic approach to traverse through a series of simulated mazes in the Gazebo simulation environment, as well as a physical maze using an actual TurtleBot3 robot equipped with LiDAR and running on the ROS2 framework. Additionally, there was a bonus task that required detecting Aruco markers while navigating the maze, utilizing the TurtleBot3's camera, and performing a specific action to indicate successful marker detection.

After exploring and iterating through various methods and techniques, our team ultimately developed a robust algorithm that combines distance-based parallelization and a state/action machine model. This was effective in enabling the TurtleBot to autonomously navigate through the maze environments. We were successful in completing both the simulation and hardware tasks by implementing our algorithm. However, we were unable to accomplish the bonus task of detecting and responding to Aruco markers during maze navigation within the given time constraints.

Background

Project Overview

The primary objective of this project was to develop an autonomous navigation system capable of guiding a TurtleBot3 robot through a series of mazes, both in simulation and in a physical environment. To achieve this goal, we adopted a modular approach, splitting the

maze-solving process into the two most basic components: detecting when to turn and when to go straight. The first component focused on accurately perceiving the maze's layout and identifying critical decision points. We leveraged the TurtleBot3's LiDAR sensor to continuously scan the surrounding environment, capturing precise distance measurements to nearby obstacles (walls). By analyzing this LiDAR data, we could determine whether the robot was approaching an intersection or a straight path and make a decision accordingly. We used these two foundational components and integrated them into a comprehensive maze-solving algorithm. This algorithm acted as a state machine, transitioning between different states (turn, straight, or halt) based on the input from the LiDAR sensor and the chosen maze-solving strategy. More detail is provided in the respective methodology section.

ROS: Robot Operating System

In this project, we utilized the Robot Operating System (ROS2 Humble) as the bridge between software, hardware, and simulation for controlling the Turtlebot. ROS's unique distributed and modular architecture with publishers, subscribers, and nodes allowed us to iterate quickly and interact with all of the relevant sensors. For reference in further sections of this report, a "topic" is a named bus over which nodes exchange messages, and a "node" is just a process that performs computation, communication, and more. You can publish information to a topic using a "publisher" and subscribe to topics (and get information) using a subscriber.

LiDAR Sensor

The Turtlebot is equipped with a LiDAR sensor (Light Detection and Ranging) which allows us to detect the walls of the maze and autonomously navigate as needed. LiDAR works by emitting pulsed laser beams and measuring the time of flight, providing an accurate estimate

of the distances in a 360 degree range around the turtlebot. By continuously measuring the surroundings, the LiDAR allowed us to build a comprehensive understanding of the layout and location of the obstacles, which then got fed to our control algorithm to navigate around them and avoid walls whenever possible.

Solving An Unknown Maze

In this project, we had no previous knowledge of the maze we would be presented with and knew that it could change at any time. The only guaranteed methods that allow you to find your way around the maze and get out are known as the right-hand and left-hand rule. These methods employ the idea of keeping one "hand" (right or left) on the wall at all times, guaranteeing that you will navigate all accessible areas of the maze and eventually reach the exit (provided there are no islands, which was specified in the project description). We chose the right hand rule as it would produce a shorter path on the ultimate maze.

Simulation

Overview

The simulation section will refer to all the code done in Gazebo / using a physics simulation of a TurtleBot rather than the physical hardware. Before dealing with any hardware, we designed, tested, and optimized our algorithms in simulation and then performed testing in the Robotics and Autonomy Lab. Due to challenges with simulation consistency, our methodologies slightly differed from real life due to the unique challenges we ran into, as detailed below.

Challenges

The biggest challenge that we faced when using simulation was the inconsistent timing of simulation which led to a wide variation in runs. Our original code, which used timing based turns, would work perfectly on one computer and not work at all on another computer. This led us to abandon the turn based timing code because of its consistency. This inconsistency in timing was primarily attributed to the varying computational resources and performance characteristics of different machines. Factors such as CPU speed, memory availability, and background processes could influence the execution speed of the simulation, leading to discrepancies in the timing of our turn commands. To address this issue, we had to abandon our initial time-based turn approach and devise a more robust solution that was less reliant on precise timing. This led us to explore alternative techniques, such as utilizing the robot's odometry data and feedback from the LiDAR sensor to determine when a turn was complete, rather than relying solely on predetermined time intervals. Another simulation-related challenge we encountered was the discrepancy between the simulated and real-world environments. While the Gazebo simulation provided a valuable testbed for our algorithms, it could not fully replicate the complexities and uncertainties of the physical world. Factors such as sensor noise, environmental conditions, and uneven terrain could affect the robot's performance in ways that were difficult to predict or simulate accurately. We had to incorporate additional checks and error handling mechanisms into our code, ensuring that our algorithms could adapt and recover from unexpected situations. Despite these challenges, working with the Gazebo simulation environment proved invaluable in the development and testing phases of our project. It allowed us to iterate rapidly, experiment with different approaches, and identify potential issues before deploying our solutions on the physical TurtleBot3 robot.

Methods

Our robot's code was designed using the right-hand rule in mind. Due to this, while designing our code, we made some key assumptions in order to allow for simplicity. First and foremost, we had our robot operate under the assumption that for the majority of the run, there would be a wall within a small distance of the robot's right side. When the run begins, we cannot actually guarantee this, so we have our robot drive forwards until it detects a wall in front of it. From there we can turn left, so that the wall moves from the front to the right of the robot. This allows us to begin the main looping body of the code. As long as there is a wall to the right of us, there are 2 types of corners we have to watch out for: inside corners and outside corners.

Inside corners are defined as corners such that the right side wall extends forwards, and intersects with a wall in front of the robot. At such a corner, the only path to be taken is a left turn. So, we have our robot turn left until its right side is roughly parallel to the wall which used to be in front of it. We find parallel-ness by comparing the length of 2 lines from the LIDAR to the walls. If we let 90° represent the right side of the robot, we had one of the 2 lines at a small angle "θ" larger than 90°, and the other line at an angle of "θ" less than 90°. The robot will be parallel to the right-side wall if and only if the length of these lines are equal. So, we kept monitoring the length of those 2 lines, and stopped the robot's turning once those 2 lines became a similar length. This allowed us to turn left at an inside corner, and guarantee that the robot's right side is next to a wall again.

The next case was that of the outside corner. More commonly known to us as the elusive "right turn", the outside corner case was the fall of many of our attempts to create robust code to conquer the maze. The outside corner case occurs when the right side wall stops going forwards, and the wall makes a 90 degree right turn. When our robot encounters one such wall, we lose

sight of the right side wall. Since our philosophy was that we want to just follow the right wall, the right wall stopping short leaves the robot with a completely open right side, and we have no guarantees that there are any other walls around the robot which we can use to make the robot travel parallel to a wall. To counter this, we decided to have our robot operate on some approximations for a short time. In our final code, upon finding an open right side, the robot would assume it has reached an outside corner, and so it would execute its "right turn action". This action consists of traveling forwards for a small timer-based distance, until there is a high likelihood that the robot has passed the wall. Then, the robot would start turning right, while monitoring a distance at an approximate angle of 45 degrees right of front. When this distance becomes short enough, we assume that the wall is now diagonally forward right of the robot. So, we have the robot drive forwards until the distance from the right of the robot to a wall is short enough. This would mean that the robot has moved forwards until the wall that was diagonally in the front and right of the robot is now only at the right side of the robot. Now that we have re-established our key assumption of the right wall's existence, we can continue moving straight with the right wall that we have found.

Last but not least, we had to add some complexity to the action of moving "straight". If the robot goes completely straight while it is not perfectly parallel to a wall, it will slowly start moving away from the wall, until at some point we lose our key assumption. So, the robot continuously monitors its distance from the right side wall. If that distance ever got smaller than or larger than 2 specific threshold values, the robot would add some angular velocity in the direction to counteract this distance, and move closer or farther from the right side wall, as needed. This worked very well in Gazebo, as the walls were perfectly straight, and so once the

robot established a mostly parallel trajectory, only minor changes in angular velocity were required to keep the robot following the wall.

Results

Ultimately, our simulation code worked well, getting around 3 minutes and 15 seconds of total run time. In order to speed up our time we could increase the linear speed and modify some of the tuning parameters to get a faster time, but by making it through the maze while hitting no walls was well within parameters for success for our group.

Hardware

Overview

Hardware is defined as any task done with the physical TurtleBot (ie. not simulation). Before dealing with any hardware, we designed, tested, and optimized our algorithms in simulation and then performed testing in the Robotics and Autonomy Lab. Due to the differences between simulation and the real world, our methodologies slightly differed due to the unique challenges we ran into, as detailed below.

Challenges

For the specific hardware, we used a waffle model (rather than the alternative burger model), which posed unique issues while dealing with its turning behavior.

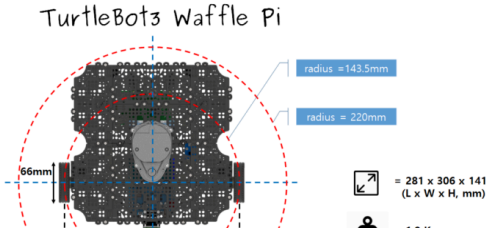

Figure 01

As we can see from Figure 01, the turtlebot waffle real life model does not actually pivot around its center. When we try to rotate the waffle model around its z axis, on the hardware, the axis of rotation for the robot is actually between the 2 wheels. Given that the 2 wheels are not centered on the robot, the robot then does not in fact rotate around its center point. Given that the LiDAR is approximately in the center of the robot, the LiDAR and the point of rotation do not match up. This made static turning difficult, because we could not compare LiDAR ranges between turns to determine the degree of the rotation.

One thing we noticed about the LiDAR in gazebo is that it returns a range of 360 different distances. In Gazebo, the LiDAR senses 1 distance for every degree around the robot. However, in the real hardware, the LiDAR actually provides a range of seven hundred and twenty distances, with 1 distance given for every half degree around the robot. Another major difference between the LiDAR data in hardware and gazebo was that the real life LiDAR data ranges started from the back of the robot, and rotated counter-clockwise. In contrast, while the Gazebo LiDAR still rotated counter-clockwise, its ranges actually started at the front of the robot. This meant that we could not use the same angles in Gazebo and real life to determine the position of the robot and its distance from the walls. Therefore, we decided to use a flag in our code, which we would alternate between True or False, depending on whether the code was to be run on real hardware, or on Gazebo simulation.

ROS 2 seems to update topics at a non-uniform frequency. It publishes at different times depending on the power of the machine on which it runs. Due to this peculiarity, we had to work

hard to find a reliable way to turn the robot to be parallel to a wall. This leads us to our discussion of our methodology.

Methods

For our code on the hardware, we started out by trying to use the same code as the simulation, only changing a few threshold values and LiDAR range values, to reflect the difference in size between the real life and simulation robot, and the difference between the return values of the Gazebo LiDAR and real life LiDAR sensors. However, we quickly found out that executing our code on hardware posed unique challenges which forced us to make major changes to our code in order to make the robot successfully exit the maze.

The largest issue we faced, which was something we definitely did not think about during the Simulation phase, was that in real life the maze was made of many blocks, and so the walls were not perfectly straight. While our main key assumption was that our robot would have a wall to its right at all times, we also unconsciously made another assumption, that the walls would be perfectly straight. The mild imperfections in the walls made the parallelization code on left and right turns work far less efficiently, as sometimes the robot would have its 2 parallelization lines between 2 blocks which had different distances from the robot, and so the robot would think that it was not parallel with the wall, when in reality it was just that some of the blocks were sticking in or out of the wall.

One other issue we faced was that our approximations for right turns did not always give us proper right turns. We realized that the timer for the straight motion in the right turn would have to be increased, so that the robot could have enough time to clear the right wall and then turn to the proper angle. Also due to some inconsistencies in LiDAR readings, we would sometimes get readings of infinite from very few LiDAR ranges even when there was a wall in

that direction. To rectify this, instead of reading singular LiDAR ranges, we would take a small range of LiDAR outputs within 3 to 5 degrees on either side of the actual value we desired, then we would average those values, ignoring any readings of infinite.

The last issue we faced was that of 1 block long walls not being picked up by the original code. Our code would try to look straight in front of the robot when looking for an inside corner to turn at. However, 1 block long walls would mess up this algorithm, because the robot's LiDAR would be centered at a point such that there is no wall straight in front of the LiDAR center point, but there is a wall in front and slightly to the right of that point. This issue did not occur in Gazebo since all walls in the Gazebo mazes were much larger than the robot, so the distance of the robot from the right wall did not affect its ability to pick up a wall directly in front of it. To fix this issue, we added another distance that the robot would monitor, that being the distance 45 degrees right of the front of the robot. If this distance became shorter than a certain threshold, we could safely conclude that the robot had reached an inside corner, so it could begin turning.

Results

Despite our difficulties between simulation and real life, we ended up with two successful challenge runs with the 3rd fastest time within our group. By using a robust algorithm, we were able to complete the unknown maze in 3 minutes 36 seconds with only one minor brush against a wall. Overall, the hardware portion of our project was a success.

Retrospective

This project taught us a lot about how ROS works, and it taught us about how important it is to have robust code. Since our inside wall turn code used parallelization and constant monitoring of nearby walls, it was able to be quite robust, and inside walls ended up being the specialty of our robot. Also, after some redesigns, our robot was able to travel straight and hug the right side wall, while constantly monitoring its distance from the right-side walls. Since ROS topics don't always update at constant intervals, timer-based code would not always give the same outputs, and so we learned about how useful sensor-output based movement was.

As stated above, our sensor-output based movements ended up becoming quite robust. However, an astute observer might realize that our outside corner used some timer-based actions, and one would be right to conclude that our outside corner turning was less consistent. In both hardware and Gazebo, our right turns were never completely perfect, and at each right turn our robot would either travel too close or too far from the perpendicular wall, so when we ran our straight-travel code after a right turn, our robot would have to make many adjustments to properly adjust its angular momentum in order to accurately follow the wall. If I were to redo this part of the code, I would change the timer-based action to a sensor-output based action, where the robot would move forwards until enough sensor readings output that there is no wall to the right of the robot. Only after the sensors have verified that we have traveled far enough would we actually begin our right turn. This would allow our robot to more consistently and accurately travel to a certain distance away from the right side wall, which would lead to less error, and less adjustments that the robot would have to make later on during the run.

Once we had finished the design of our algorithm, while it was very robust, it took a long time to complete the maze. The fastest maze time was 50 seconds but we were at around 3

minutes and 36 seconds. We could have increased our time by using a less strict algorithm and using a faster linear and angular speed. We also could have looked into a RANSAC / linear regression based method where we take the polar coordinate points from the lidar and approximate the wall at all points using regression, allowing us to not need any special state machine logic besides trying to keep parallel to the wall at all times.

Work Division

At the start of the project, the group got together and we collectively worked through the logic of the code. Together, we each contributed ideas to form a general strategy of how to traverse the maze. During the lab, we hosted a Visual Studio Code Live Share session so everyone could edit the code in real time. This way, we could all code and contribute to the project simultaneously. We connected to the turtlebot to test our code on it directly.

Outside of class, however, we did not have the turtlebot to run tests on, so we had to resort to using Gazebo. Nitin and Mohammad were the only ones able to get Gazebo running on their laptop, so any changes that Karthik and Pranav made would be uploaded to a shared GitHub so that Nitin or Mohammad could test these results. To avoid the repeated back and forth, Karthik and Pranav mostly worked on the overarching logic, such as how the robot would know to turn at an inside or outside corner and how to keep the robot parallel with the walls when going straight. This way, they would not have to check in with Nitin and Mohammad when they had any updates to their code. Nitin and Mohammad would also work on the general logic, but they would then implement this code and work out the more intricate details. This included implementing model specific features, like the and choosing angles and distances to detect walls

from, and correcting any flaws or bugs in the code. This way, by the time we returned to the lab, we reached a working solution in Gazebo.

There were differences in the simulation and the hardware that caused the turtlebot to not complete the maze. We repeated what we did during the last lab and used a Visual Studio Code Live Share session, and we all discussed ideas about solutions to the issues we were having and tested these together. We continued this process of creating an algorithm that worked in Gazebo and then testing it on the physical turtlebot until we were able to converge at a solution that worked both in Gazebo and on the turtlebot.