Introduction to Computational Robotics: Warmup Project

Daniel Park

September 24, 2024

Abstract

The first project in this course, rightfully named the "Warmup Project", is meant to be an exercise to warm up for the course by getting to know the ROS platform and working with the Neato robots. We developed a couple of behaviors for the robot: remotely user-driving the robot with keyboard commands (a TeleOp), driving the robot in a square, wall following, person following, obstacle avoidance, and finally combining two of the behaviors: driving in a square and obstacle avoidance.

1 Behaviors

For each behavior, describe the problem at a high-level. Include any relevant diagrams that help explain your approach. Discuss your strategy at a high-level and include any tricky decisions that had to be made to realize a successful implementation.

1.1 Tele-Op

The tele-operation behavior is responsible for the robot being able to be controlled through keyboard commands. The approach used to achieve this behavior was to create a series of if statements such that depending on the key pressed, the robot would perform a different action.

In standard video games, the WASD layout is used to control the robot. However, I decided to add an additional key to halt all actions on the robot which was incredibly useful when testing other capabilities in the robot when it was running.

For moving, the 'w' key would make the robot drive straight forward, the 'a' key would make it turn left in place, the 'x' key would make it drive straight backwards, the 'd' key would make it turn right in place. The 's' key would make the robot stop moving and pressing 'control-c' would end the program. The robot drives straight and can turn through publishing messages to the *cmd_vel* ROS topic to control the robot's linear and angular velocity at any given time, depending on what key is pressed.

1.2 Driving in a Square

As described by the project, this behavior drives in a meter-by-meter square as seen in Figure 1. It does so by continuously going forward for a couple of seconds then turning ninety degrees to go in a square shape pattern.

For this behavior, I decided to use timing to make robot go in a square. While it would have been a better implementation on our end to control the robot's movement with odometry measurements since I had to hardcode the times we wanted to move forward and turn to accurately

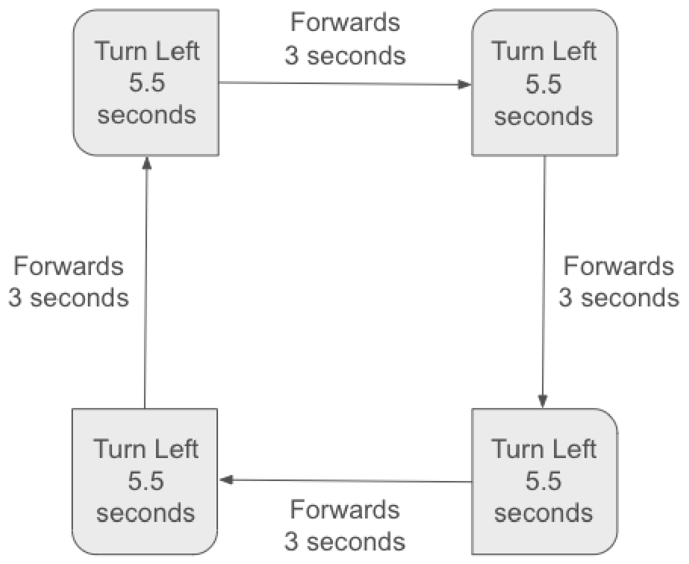

Figure 1: Drive Square Behavior Pattern along with their times of movement (as tested and calculated on the carpet outside of MAC113.)

make a square movement as shown in Figure 1. I decided to use timing to save time for the next few sequences in wall and person following, which would take more time than this sequence. In addition, I wanted to use this motion to test out how to control motors in an automated setting (without a driver control). So using the previous Tele-Op commands, I just needed to control the motors going up and turning left after specific moments of time to finish this sequence.

But to add simple complexity, the robot continuously moves in a square. This loop functionality would come later useful for the finite state control.

1.3 Wall Following

The wall-following behavior programs the robot drive parallel to walls when they are detected. I programmed the robot to drive along a wall along the Neato's left side. In this implementation, the $WallFollowerNode$ class subscribes to laser scan data from the Neato and process it.

While I have briefly experimented with proportional control, I decided to use a more rules-based approach to assign fixed angular and linear velocity values depending on how close or wall the robot is from the wall instead of continuously adjusting the control signals based on the magnitude of the error. This approach is not only intuitive and applies to the wall following problem but also to the other behaviors.

Upon initialization, the node sets up two critical functions: a publisher and subscriber. The publisher sends velocity commands to the robot's motion control system via the */cmd_vel* topic, while the subscriber listens to laser scan data on the */_scan* topic. The laser scan data provides distance measurements around the robot, shown in Figure 2 which are essential for wall detection and navigation.

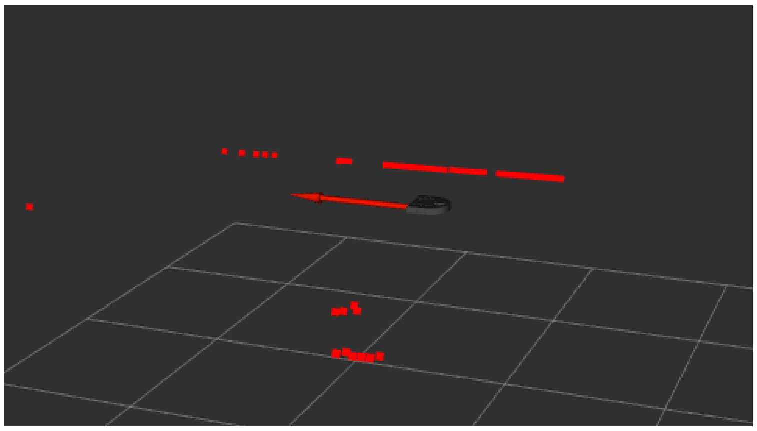

Figure 2: LiDAR Scan of the Wall from the Neato shown in rviz2.)

In order to use the scan data for adjusting the robot, we need to process every time a new set of laser scan data is received. Within the *process_scan*() function, the first step is to extract the range values specifically from the left side of the robot. These range values represent the distance to objects on the left, which is where the robot aims to follow a wall. The selection focuses on angles around 90 degrees relative to the robot's forward-facing direction, which corresponds to the left side.

Once the left range values are gathered, the code calculates the slopes between consecutive range values using the *calculate_slopes*() function. These slopes represent changes in distance between successive measurements, giving an indication of the wall's orientation relative to the robot. With the *is_wall_present*() function, if the slope values are consistent and relatively flat, it suggests that a wall is present and running parallel to the robot. If the slopes are within the hardcoded threshold, a value set to filter out variations in slopes that would indicate the absense of a wall, the function confirms the presence of a wall, allowing the robot to proceed with its wall-following behavior.

Finally, the robot's movement is controlled by the *control_robot_movement*() function, which generates velocity commands based on the presence of a wall and the distance to it. The robot attempts to maintain a specified distance from the wall, defined by a wall distance variable. If the robot is too far from the wall, it will turn slightly left to correct its course. On the other hand if it is too close, it will turn right. If no wall is detected, the robot will rotate in place, searching for a wall to follow.

1.4 Person Following

Based on my approach for wall following, I decided to follow a similar approach for person following to keep it as simple as possible. Leveraging laser scan data, the Neato detects the person's presence and dynamically adjust its movement to maintain a consistent distance.

Similar to wall follower, the robot utilizes a laser scanner, subscribing to the *_scan* topic to receive distance data in the form of LaserScan messages. To process the incoming range values and angular data, the *process_scan*() function is called to represent the field of view of the laser scanner.

Instead of a standard clustering algorithm, in this implementation, the robot is set to rely on basic laser scan data and follow any object detected within a range of 1 meter. For each distance, it checks if the value is less than *person_distance_max*. If a distance is less than the threshold, it assumes that a person is detected and the Neato will try to approach it.

For calculating the angle of the detected person from the laser scan data is derived from the way laser scanners report their measurements in ROS. I have found the formula below and how to use it in this formula found here.

The laser scan message provides an array of distance measurements (ranges) and corresponding angular data, including the starting angle of the scan (*angle_min*) and the angular spacing between consecutive measurements (*angle_increment*). The laser scanner sweeps across a field of view, starting at *angle_min* and increasing by *angle_increment* for each subsequent measurement in the ranges array.

$$\text{angle\_to\_person} = \text{angle\_min} + (i \times \text{angle\_increment}) \tag{1}$$

In the above formula, i is the index of the detected object in the ranges array. This formula allows us to compute the angular position of any detected object relative to the robot's forward-facing direction. The starting angle, *angle_min*, sets the baseline, and the term ($i \times$ *angle_increment*) calculates the offset from that baseline based on the index of the detected object. This approach ensures that the robot can accurately determine the person's location in its field of view, enabling it to adjust its movements accordingly to follow the detected person.

After calculating the detected person, the method calls $follow\_person()$ with the calculated $person_angle$ and directs the robot to move forward and adjust its heading to follow the person. I also added a marker to track the person's movement in rviz visualized in Figure 3.

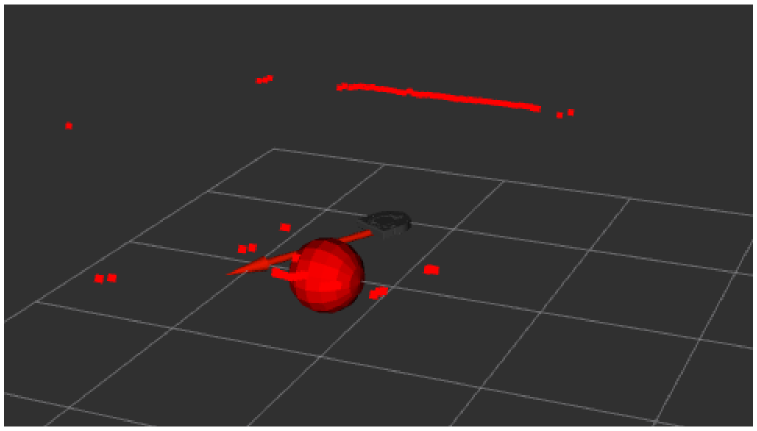

Figure 3: LiDAR Scan of the Neato following a person.)

1.5 Obstacle Avoidance

As for a robot to avoid obstacles, I derived the system by adapting the core logic of the person follower code and to steer away from nearby obstacles instead of following them.

In the original system, the robot detects a person within a specified distance and adjusts its velocity to follow the person. In contrast, the obstacle avoider detects objects within a defined threshold distance and steers the robot away from them, prioritizing obstacle avoidance over forward movement.

Similar to person follower, the LaserScan data is processed to detect any obstacles within a threshold distance, which is set to 1.0 meter in the code. The robot checks if any objects fall within this distance, storing both the closest obstacle's distance and its angular position relative to the robot using the same calculation from Equation 1. If an obstacle is detected within the threshold distance, the robot stops moving forward and turns in the opposite direction of the obstacle, adjusting its angular velocity to steer clear.

1.6 Finite State Control

The Finite State Control allows the robot to manage itself in two different states based on the conditions met. I decided to combine driving in a square and obstacle avoidance. As shown in Figure 4.

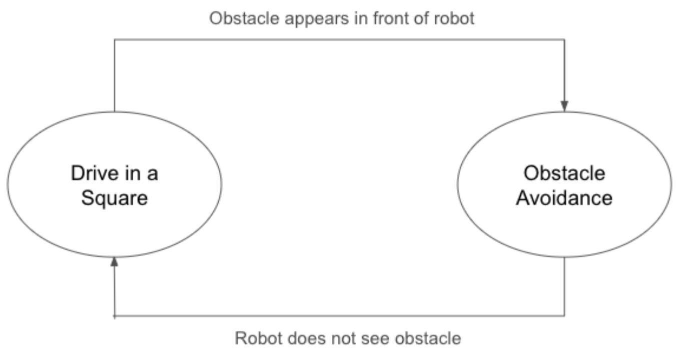
Obstacle appears in front of robot
Drive in a Square
Obstacle Avoidance
Robot does not see obstacle

Figure 4: Finite State Control where Robot drives in a square unless it sees an obstacle, which the robot will avoid.)

The state transition diagram demonstrates one flow moving to another. When the robot is moving in a square, it is actively looking for an object to follow. If it sees an object in front of it, the robot will avoid it as in obstacle avoidance. If the robot loses sight of the object in front of it, it will return to its default behavior of driving in a square.

2 Challenges, Improvements, and Final Remarks

The main challenge was doing the entire warmup project by myself. Given more time and possibly a partner, I would have liked to attempt the parts in the Go Beyond sections.

In addition, it would have been a better idea to take advantage and experiment with the robot's odometry readings that was written in the odometry ROS topic. The readings on headings, angles, speeds, and distance that would have been extremely useful, especially for Drive Square where instead of having to experiment with the times of the driving and turning (with the additional variable nature of the MAC carpet floor), the Neato would perfectly drive in a square in any floor environment.

Going into the next project, I would personally like to start tackling things early, and hopefully with a partner next time!