Eddy Pan, Vivian Mak

Warmup Project

Introduction to Computational Robotics
Fall 2024

1 Overview

This warmup project serves as an introduction to ROS2 and working with the Neatos with various tasks. The 6 behaviors we implemented for this project were: robot teleoperation, driving in a square, wall following, person following, obstacle avoidance, and finite-state control (which combines multiple behaviors). Our approach to implementing these behaviors was to complete the MVP in a way that allowed for more complicated integrations.

2 Robot Teleop

The teleoperation (teleop) node controls the movement of the robot using keyboard input. We used a Twist publisher that sends out `cmd_vel` msgs to the Neato to control its velocity after keyboard input. For this node, we mapped the following keys to directions: “w” key maps to drive forwards, “a” key maps to rotate counterclockwise, “s” key maps to drive backwards, “d” key maps to rotate clockwise, and “Ctrl+C” maps to stop all movement and end the program. Any other key that the user inputs stops the movement of the robot. For the forward/backward movement, we published linear velocities in the x-direction to be positive/negative, respectively. For the counterclockwise/clockwise movement, we published angular velocities in the z-direction (which rotates around the xy-plane) to be positive/negative, respectively.

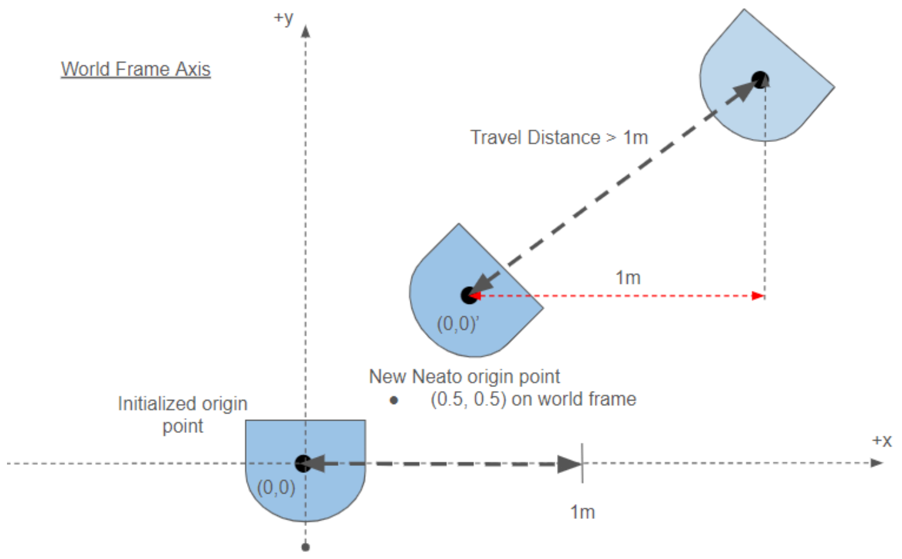
Teleoperation Keybinds

Key	**Action**
W	Drive forward
A	Rotate counterclockwise
S	Drive backwards
D	Rotate clockwise
Ctrl + C	Estop

3 Driving in a Square

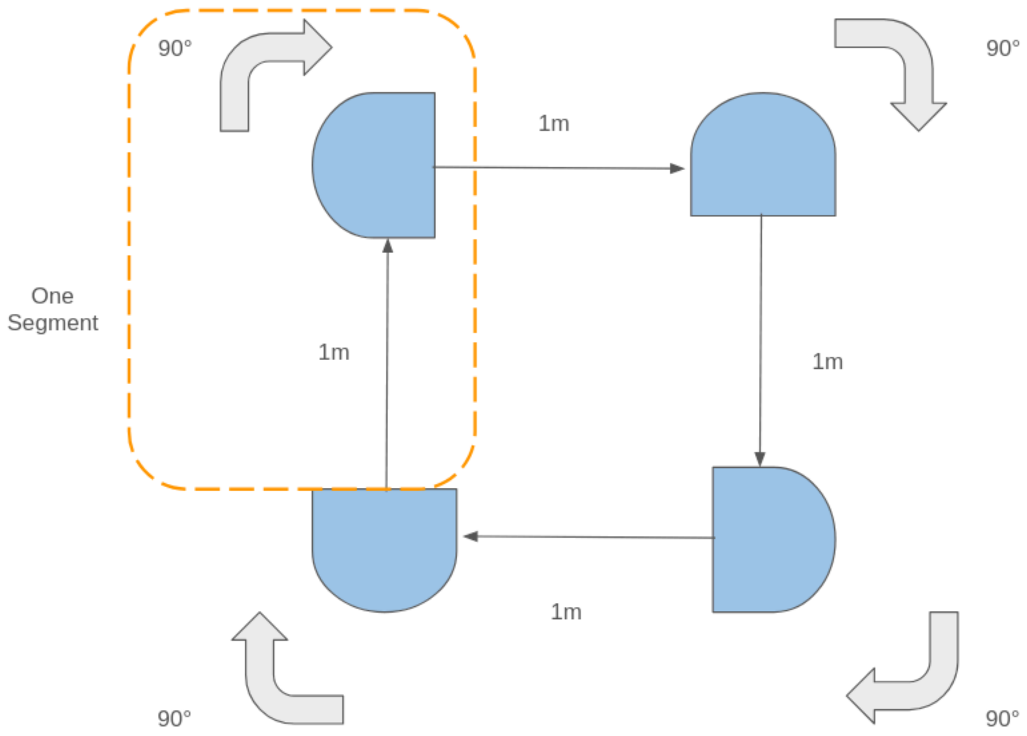
In order to drive in a square of 1m x 1m, we used a subscriber for the odometry of the vehicle and a publisher to send the `cmd_vel` msgs to the robot. With the odometry subscriber, we had access to the robot's position in relation to its starting position which served as the origin of the world frame. From there, we kept track of the number of turns, `num_turns`, the Neato had already completed, which served as our flag for whether the Neato had finished driving in a square (`num_turns == 4`). After some debugging, we realized that the origin point in the world frame was decided the moment we connected to the Neato, and not when the program started—which led to changing our approach in determining when the Neato reaches its target position using its relative position to its start points rather than constants.

This also brought up another issue where the travel distance of 1m is not accurate. The example below outlines a situation where the travel distance is actually greater than 1m. Because our software checks whether the x and y position has increased by 1, when the heading of the Neato is altered, as it traverses the travel distance line, both x and y are increasing. Simply with the Pythagorean Theorem, if we are just looking at the x axis, when x has increased by one, the travel distance is larger because it is the hypotenuse of the right triangle.



Shows that the travel distance is incorrect when Neato and World origin point and heading aren't the same.

Therefore, we set the body frame of the Neato, `body_pos`, to be whatever the first position the odometry sensor returned upon initialization of the node. Equipped with the `body_pos` and the position of the robot in current time (`self.pos`) from the odometry subscriber, we defined the relative position to be the difference of the two (`body_pos - self.pos`). Now, resetting the `body_pos` after each turn, we know that to travel on each segment of the square path, the vehicle needs to go straight 1 meter, then turn clockwise 90 degrees, and repeat until it has looped four times.



In order to drive 1m, we send cmd_vel Twist msgs with only a linear speed in the x-direction and check if the position of the robot has reached its destination yet. If it has, we send a linear speed of 0 and send angular Twist cmd_vel msgs until it has reached its desired rotation. We repeat this process of driving 1m and turning 90 degrees four times, thus completing our square path.

We also played around with a PID Controller to improve its accuracy of a square drive even further. We didn't fully implement it, though, only getting as far as a somewhat-tuned PI Controller. We were hoping that a modular PID Controller with odometry would help in implementing the other behaviors. Nevertheless, we decided to put a pause on it for the sake of time.

4 Wall Following

The goal of this behavior is for the Neato to follow and drive in parallel to the wall. Approaching this was simply a mathematical problem based on the lidar scan. The sensor on the Neato returned a list (ranges[]) of the distances objects are away for all 360 degrees. We constrained the problem by setting the Neato to always be to the right of the wall. This made calculations easier by pre-setting which index of the list to take values from.

To actually calculate the heading error of the Neato in relation to the wall, we have to find the slope of the wall. This can be done by picking any two arbitrary angles on the left side of the Neato. These two angles are converted to Cartesian Coordinates with the following equation:

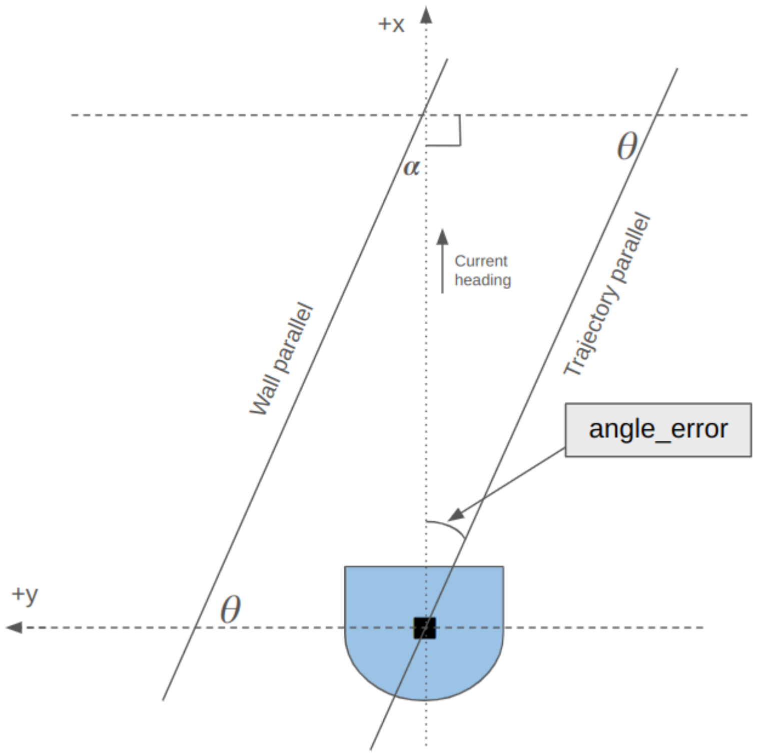
$$x = d * cos(\theta)$$
$$y = d * sin(\theta)$$

where d is the object distance away the sensor has recorded at angle θ. Once the two angles have been converted to points, the slope can be found with the following equation:

$$slope = \frac{y_2 - y_1}{x_2 - x_1}$$

Angular error then could be found with the following equation:

$$90 - tan^{-1}(slope),$$



Taking the arctan of the slope wall gives us theta. Subtracting theta from 90 gives us the remaining angle alpha, which is our angle error.

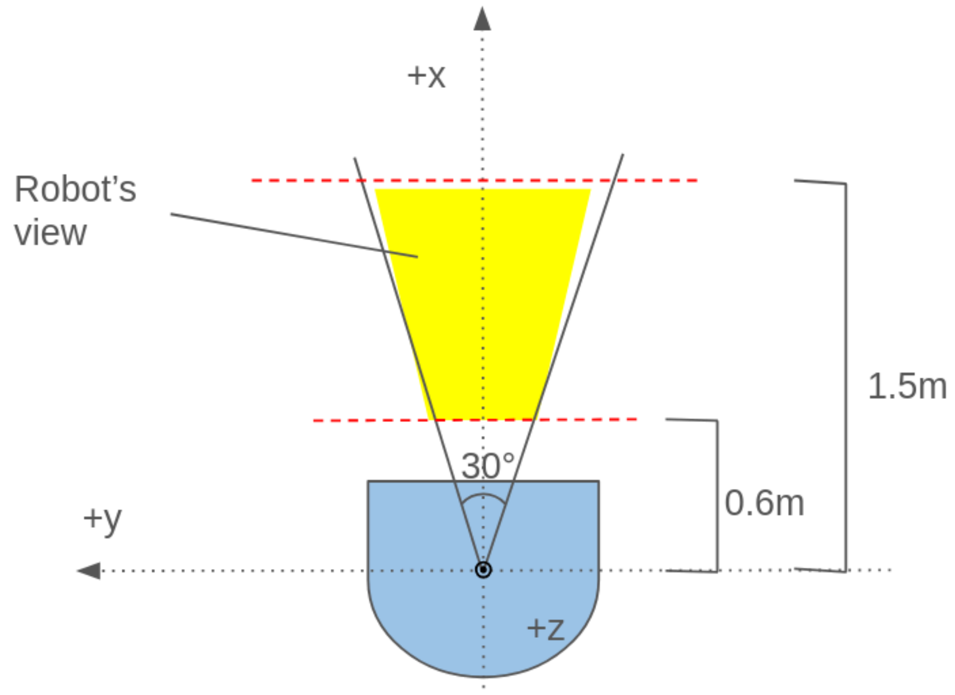
Moving the robot for this behavior involves a linear and angular component which we can control with the Twist topic. An *align_state* variable was created in the beginning to continuously check if the Neato trajectory is parallel with the wall. A tolerance of any angle error greater than 0.1 radians was chosen to set the Neato trajectory as not parallel to the wall.

Based on whether the Neato heading is facing towards or away from the wall, we will turn clockwise or counterclockwise, respectively. We will continuously set the turn angle speed to 0.1 until the angular error of the Neato is within ±0.2 degrees to be exactly parallel with the wall.

While implementing this behavior, the main issue was figuring out what information we needed to successfully carry out wall_follower. There were many ways to reuse variables, but we decided to make a new heading variable to determine whether to turn clockwise or counter-clockwise. Having this extra variable makes it easier for more complicated systems in the future. For example, if the wall were to be on the other side of the Neato, deciding the turn direction with the slope could get complicated. This approach was relatively smooth with occasional cases of the Neato jittering back and forth. Implementing a PID controller to this will improve that aspect, and a next step would be to check for walls on either side.

5 Person Following

In order to follow a person, the Neato needs to (1) identify if there is a person in its vicinity and (2) know where it is in relation to the person. Since the Neato is equipped with a LiDAR sensor, it can see objects all around it. However, since the range for the LiDAR sensor is quite sizable, we'll use only a 30° cone as its view to reduce noise. We'll also limit this cone to have a max distance of 1.5m, and minimum distance of 0.6m to ensure it doesn't crash into the person it's following.



We went with a simple approach to detect the center of mass of the person: using every point in the 30° cone, we check if there are any in our distance constraints. If there are, we'll take the average θ and average distance r from the set. The polar coordinate

$$(\theta_{average}, r_{average})$$

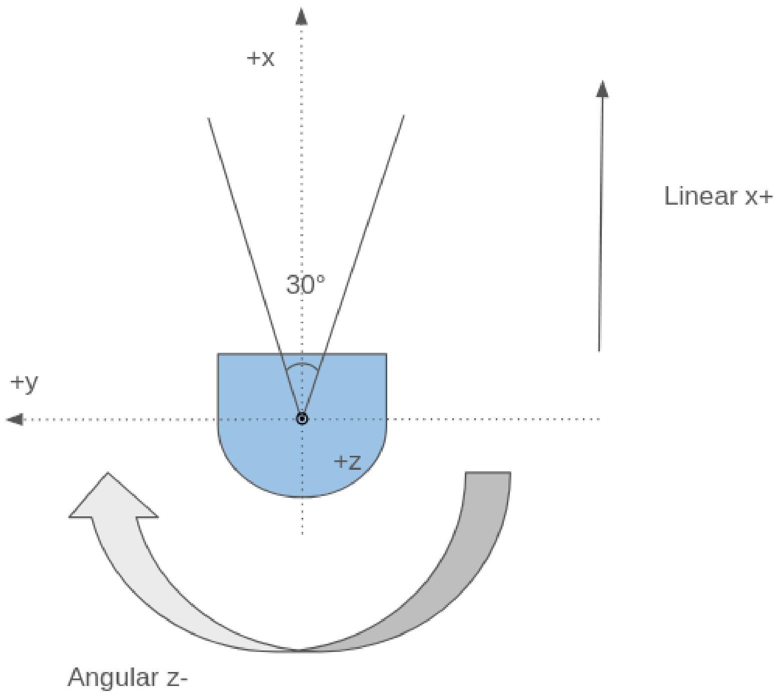
Represents the center of mass of the person. Now, knowing that the Neato is always at $(0, 0)$ relative to the center of mass of the person, we can rotate the Neato until the θ of the person is close enough to 0—thereby meaning that the Neato's heading is aligned with the person—and until the r is close enough to the 0.6m minimum distance we specified earlier. For this behavior, we implemented a simple movement controller. We struggled to implement a PI controller again for this behavior, so we opted to use the following approach instead:

- ➢ If the person's $\theta_{average} > 0.05$, where 0.05 just represents a positive tolerance, then we send positive angular velocities to the robot, turning it counterclockwise.
- ➢ Otherwise, if the person's $\theta_{average} < (-0.05)$, then we send negative angular velocities to the robot, turning it clockwise.

- ➢ Else, the person is within a tolerance of 0.1 from the Neato's heading, which is sufficient to track and follow them. Set the angular velocity to 0.

All the while, we check for the distance $r_{average}$ the person is from the Neato:

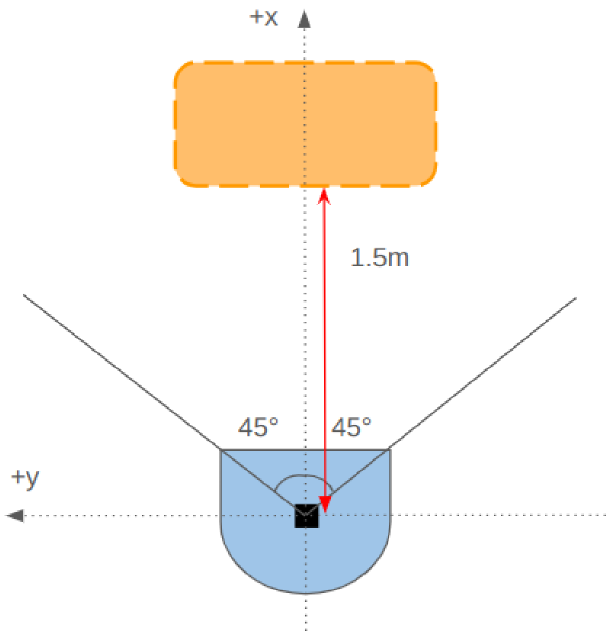
- ➢ If the person's $r_{average} > 0.6$, then we send positive linear velocities to the robot, bringing it closer to the person
- ➢ Else, the person's $r_{average}$ must be less than 0.6, our defined distance we want the Neato to keep from the person. We stop the robot by sending linear velocities of 0.0.



While this behavior worked as intended for most cases, we found that the distance we wanted the Neato to keep from the person was inaccurate. Even though we define the distance to be 0.6m, we found that the Neato stops within approx. 0.2m of the person instead. We also found that the Neato struggled to track the person when there were other objects it detected with its LiDAR within the 1.5m max distance constraint. Therefore, to reduce the risk, we decreased the degree of view to a 30° cone. This method of only being able to recognize objects in a specified cone region meant that the robot would be clueless on how to follow the person if they were in the remaining 330°. An interesting extension to this behavior would be to distinguish the person from other objects the Neato might pick up on with its LiDAR, check all 360 degrees of its LiDAR, and use a better movement controller like PID to follow the person.

6 Obstacle Avoidance

The goal of this behavior is for the Neato to traverse an environment without bumping into any objects. Approaching this was a recursive process to keep the Neato moving forward unless an obstacle was detected. We constrained the scanned data we looked at to a 90 degree angle (from 45 to -45 degrees.



0 degrees is positive x, and going in a counter-clockwise motion, positive y is 90 degrees.

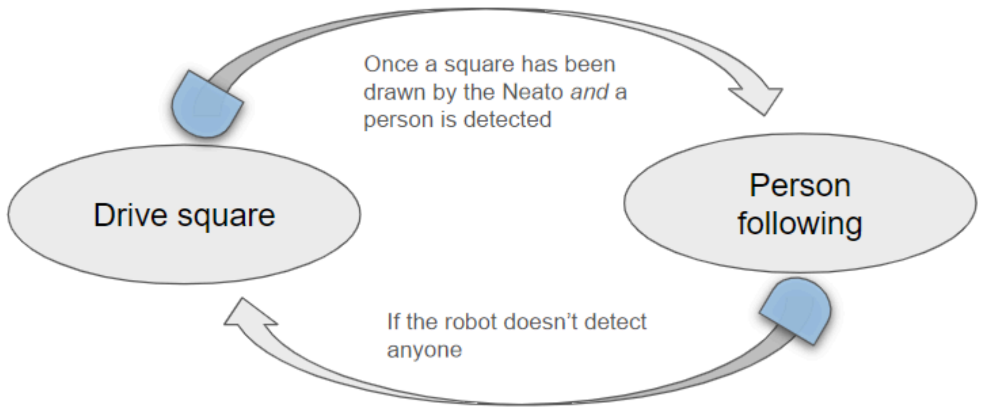
An object less than 1.2m away is considered close, and while turning, the clearance distance to check is 0.5m.

We checked the values in ranges[ ] for any values less than 1.2m which would turn our variable *detect_obstacle_state* to be true. To know which side of the Neato the obstacle is, we had another variable *turn_direction* which set whether we needed to turn clockwise or counter-clockwise. If an object is detected on the left side, the Neato will turn right until all its constrained points are greater than the clearance distance, and vice versa for if an object is detected on the right side.

With this, the Neato can smoothly traverse any environment without bumping into any objects. A next step could be to implement planning into the software. This would give the Neato a start and end point, and with the initial lidar scan, an ideal path is calculated to follow. The tricky part of this behavior was calibrating the away distance and clearance to make sure that the turn speed is fast enough to not bump into any obstacles.

7 Finite-State Control

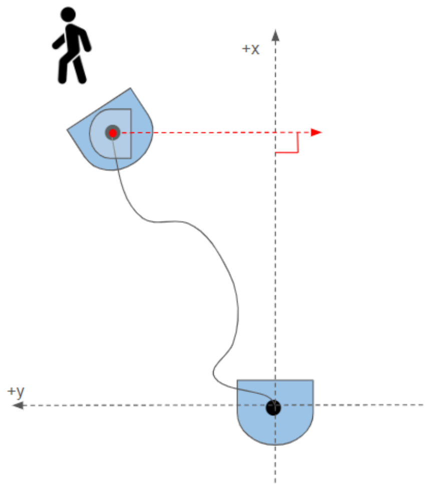
The goal of this behavior is to combine multiple previous behaviors with a smooth transition. We decided to combine the drive-square and person-following behaviors. This means the Neato will drive in a square until a person is detected. Once a person is detected, the Neato will start the person-following behavior until the person is no longer detected. It will then start driving in a square again.


Once a square has been drawn by the Neato *and* a person is detected
Drive square
Person following
If the robot doesn't detect anyone

Visualization of logic of Neato switching between behaviors.

In addition to the code of the original behaviors, we also defined state-control variables, and a separate function that would check if there were any people in the robot's view. The variables were meant to run continuously until stopped; however, with finite-state control, we wanted to switch between the two. The detect_person_state and node_state were used for the logic to switch between states.

Using odometry to drive a square allowed the Neato to remember its position and heading, so even after switching to person_following, if the heading is 45 degrees from its 0 degree person following, the Neato will rotate 135 degrees to the heading it was supposed to be at since initialization.



The heading of the Neato will always move back to its supposed relative heading.

An issue we ran into was figuring out how to implement these behaviors. Originally, we tried initializing a copy of the node in this file to call it, but it did not end up working. As a last resort, we approached this by copying and pasting code from the original class. However, this did not seem to be the most ideal solution in terms of efficiency, so the next step would be to find ways around this.

8 Conclusion

This project provided a good scaffold for future projects and introduction to the class with the practice of using ROS2 commands, writing good code, logic thinking, integration, and documentation.

Takeaways

Our key takeaway was that breaking down these complicated behaviors into smaller steps makes implementing the code much easier. A big challenge when starting this project was figuring out where to start, having no previous experience in ROS2. For each behavior, thinking through what the publisher and subscriber is narrowed down what exactly we needed to implement. From there, we used variables assigned to the topics, calculated how to find what we were looking for, wrote pseudocode, and calibrated as needed. This learned step-by-step process made approaching more complex tasks like person following more manageable.

Another takeaway was how to actually manage a project without knowing too many details about it. This is a skill that comes up a lot when approaching group projects at Olin because our motto is learn by doing. Before starting the project, our first meeting sorted out all the details of timeline, task list, scheduling work blocks, and setting up for parallelizing documentation. We used a Notion workspace to document all assigned tasks, timeline, creating a sample write up, and saving often used command line calls. This process became very helpful also for breaking down the projects into more manageable sections. With this, we were able to see clear steps and milestones, and delegate tasks evenly among us.

Code Structure

A class was created for each behavior we implemented inherited from Node. Each class had a publisher and subscriber topic depending on what the behavior needed. For example, for wall_follower, the publisher is Laserscan and the subscriber is Twist (commands velocity). The publisher will send data to the subscriber, and with our written logic, the robot is able to move with constraints. Within the class, there is a minimum of one function to process data from the publisher and a run_loop function which actually commands the Neato. This function will run indefinitely and is where the logic happens.

Challenges

There were quite a few minor challenges we ran into along the project. In terms of the setup, we each were only able to connect to either the Neato or Gazebo. This made it difficult to test asynchronously. For technical challenges, figuring out the math for wall_following took a while, but we eventually figured it out. There were more individual behavior related challenges, but overall, we learned a lot through each obstacle and the limitations of what each topic could provide.

Next Steps

This project was very open ended in the sense that there are many ways to improve the software. PID control is something we wanted to apply to all behaviors. Currently the robot jitters as it makes a decision, so having that implemented with a calibrated tolerance will solve that issue. Fixing up the software to work around the limitations of each topic is something we want to do. For example, for drive_square, if the heading or position changed from initiation, the square's length won't exactly be 1m. Know that we know the small details like that for each behavior, developing software for more complex behaviors will be easier.