CompRobo Warmup Project

Alexis Wu, Philip Post

Introduction

The goal of this warmup project was to gain familiarity with ROS2 on our Neato (a vacuum robot with raspberry pi). The overall challenge was to communicate with the Neato using subscription and publisher. The subscription listens for the messages from the robot including sensor messages like a scan from its lidar, or the odometry on its wheel encoder. We need to collaborate with these input data to command the neato without the use of timers, for, or while loops. This is the most basic mindset of structures in ROS2, which is an operation platform for robots.

We implemented several behaviors on neato, including teleop (keyboard control), drive square, wall following (a visualization of neato is below), person following, and obstacle avoidance. Some challenges in development also include filtering noise from sensors, considering real-world physics (especially inertia), and proportional control of speed to reduce the margin of error. These challenges were compounded by the fact that python code for ros has to be designed to run as quickly as possible with no hanging within the code. Overall, this project provided a unique set of challenges that are explored within our report.

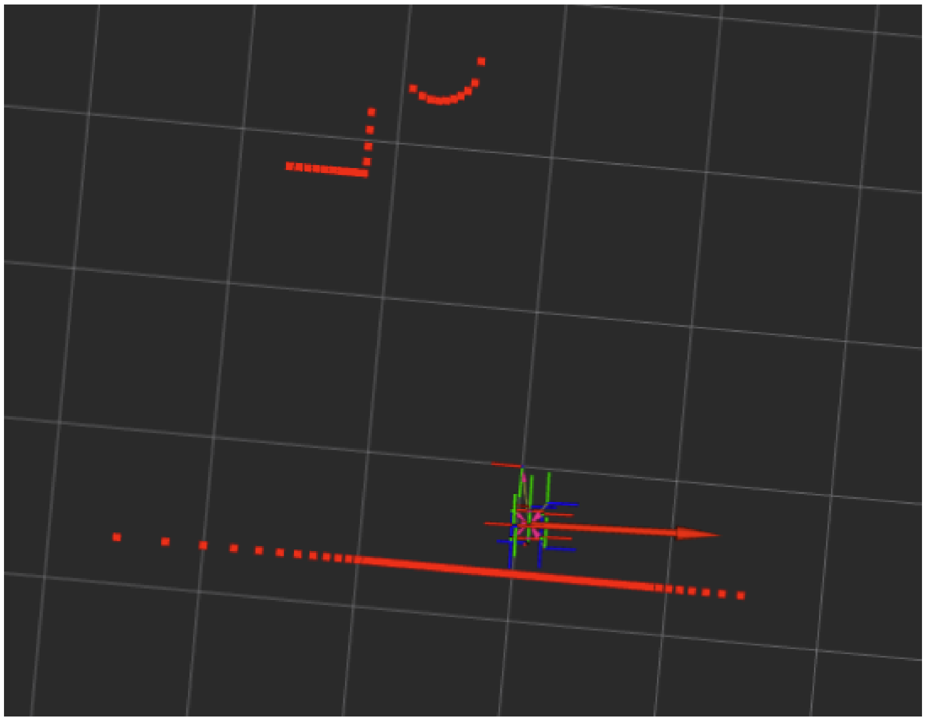

Figure 1. Visualization of wall_follower While Ignoring the Obstacles

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.

Drive Square

We need to let the neato drive 1m x 1m square. The moving pattern could be divided into a loop of 2 patterns: go straight for 1m and then turn 90 degrees. To approach this behavior, we tried 2 methods:

1. Hard code with given speed and time. The main challenge in this is to control timing using time elapse within the function that is repetitively called rather than a “while” or “sleep” function
    a. We chose the arbitrary linear speed of 0.1 and angular speed of 0.25, because a large speed would cause more inaccuracy. From there we calculated the required time.
    b. The problem with this approach is that the neato has to go really slow to avoid inaccuracy and is pretty much hard-coded.

2. Command the neato to go target points in the world frame using an odometry encoder. This method provides more flexible control of neato behavior (for example, we can have the neato draw any shape by providing vertex coordinates in odometry (world) frame.) We calibrate the world frame based on the position of the neato because the odometry encoder has errors and only initializes when a neato is first connected. We use read position (linear in x,y) and orientation (angular in z) information in the world frame and set the target angle and point both in the world frame. Then we would control neato speed and angle proportionally by setting the speed correlated to the displacement from current position to target position and current orientation to target orientation. There were a few challenges to overcome:
    a. figuring out how to get position and use odometry frame (there is no document on ros2 about this).
    b. How to deal with error induced by going past target position/ angle and sudden acceleration.

   We approached the second challenge by calibrating the odometry frame every time the neato approaches one target point to prevent incremental errors and using soft start and soft stop with proportional control.

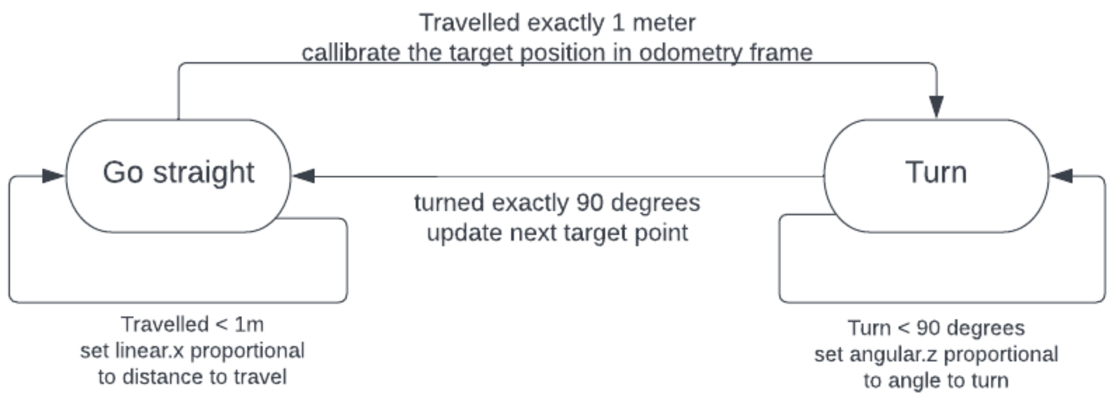
Travelled exactly 1 meter
callibrate the target position in odometry frame
Go straight
Turn
turned exactly 90 degrees
update next target point
Travelled < 1m
set linear.x proportional
to distance to travel
Turn < 90 degrees
set angular.z proportional
to angle to turn

Figure 2. Proportional Control System

There was still some margin of error even with proportional control, to improve that, we should use pid control to make the neato adjust its position all the way.

Wall Follower

We divided the problem into wall-finding, wall-approaching, and wall-following. For the wall finding, we optimized the function to distinguish a wall from random noise or a smaller obstacle.

To implement this, we first take a laser scan from 360 degrees and filter it by only considering a detected object with at least consecutive scans to be a wall. When the neato finds a wall, it takes only the shortest scan as the target angle as it should be vertical to wall.

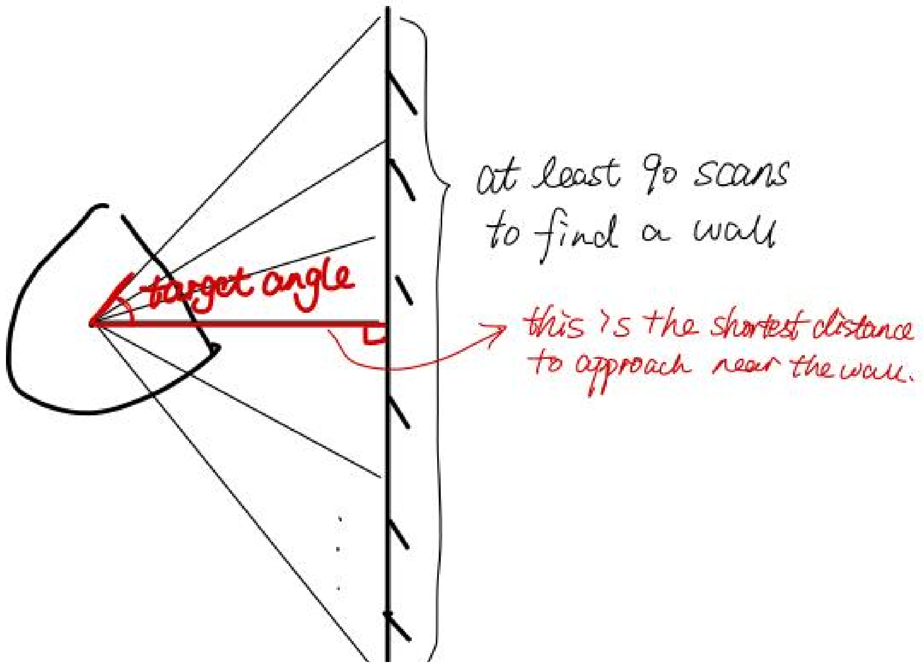

Figure 3. Wall Approach Trigonometry

Once we have the shortest distance and the angle from it, we command the neato to turn the target angle, having the x-axis facing vertically towards the wall. After the turn, the neato would go near the wall until it travels the shortest distance from measurement - 0.3m (arbitrarily set because the walls in gazebo have slopes). When the neato is facing towards and close enough to the wall, it should turn 90 degrees and just go straight to stay parallel to the wall. When a wall is no longer detected on the side or the neato bumps into another wall (like in a corner), it would just stops.

For the first iteration of the code, we had to make the neato spin to find a wall because we were only using the laser scan at the x and y axis. We improved the neato behavior and avoided some wrong wall findings by filtering the range of laser scans to be considered a wall. For further improvement, we could have the neato travel directly towards its forward direction if possible, then make only one turn to go parallel to the wall. Also if we could actively control the distance from the wall, the going parallel would be actually going parallel without going off.

Person Follower

For person follower, we assumed there is only one object in neato's detecting range for simplicity. Then we only take the index and distance from valid measurements. They are polar points with neato being the origin. We then transform them into the cartesian coordinate system.

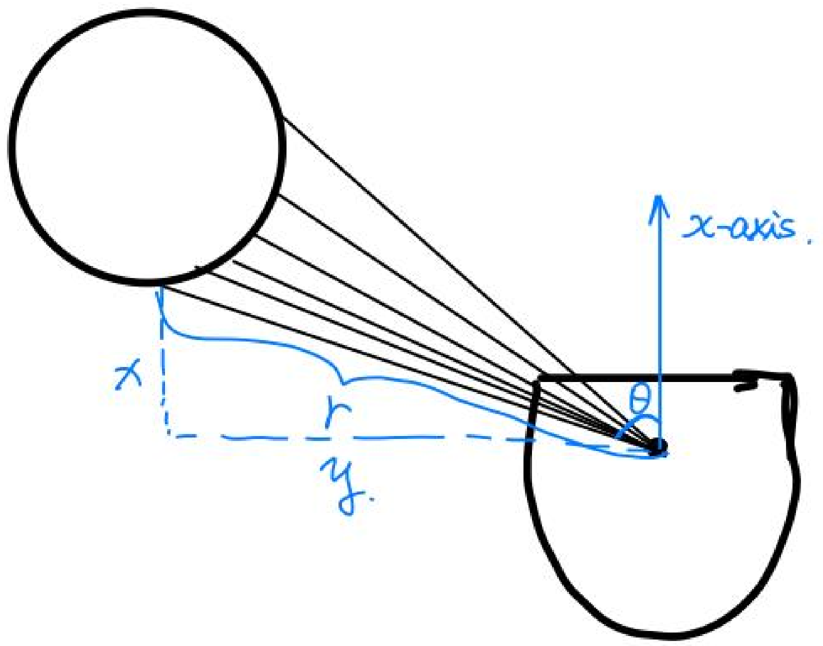

Figure 4. Object Approach Trigonometry

Then we find the x's and y's in neato's coordinate system. Once we have these points, we calculate the average of x and y as "center of mass" and then command the neato to go there. To approach the target point, we could have the neato rotate and go proportionally at the same time as it does not have to be extremely accurate. When the object is behind neato (target x < 0), the neato would only turn first and then go after it's actually facing the object to follow.

A significant improvement, if we have more time, would be distinguishing standing objects from the moving object that we want it to follow. To implement this, our plan would be to keep a record of each scan and look for the difference between scans to determine what's moving. But the whole thing would need to happen in the world frame rather than in the neato frame which would make the implementation more complicated.

Finite State Controller

The idea of finite state controller in fact has already been applied throughout previous behaviors drive square and person follower. The implementation is we create nodewise variables to represent different states. In the function that gets called repetitively in the publisher, we use if statements to determine what state is set True and implement the code for that state. Our finite state controller derives from person follower code, by adding an additional state to actively look for an object (drives around with random linear and angular velocity) and once an object appears in the detectable range, the neato would switch to the follower state to behave like in person_follower.

We made the finite state controller relatively simple so we did not really face a big challenge except for sometimes the gazebo simulator would mess with laserscan and odometry data. A potential improvement would be to distinguish wall and object to perform wall follower or person follower, which requires a combination of distinguishing standing and moving obstacles and recognizing a wall with the filter algorithm we performed in wall_follower.

Teleop

Teleop is the simplest of all of the nodes. All it does is print keyboard instructions for the user, wait for keyboard input, and then move the robot if the keyboard input is a certain character as highlighted by the figure below.

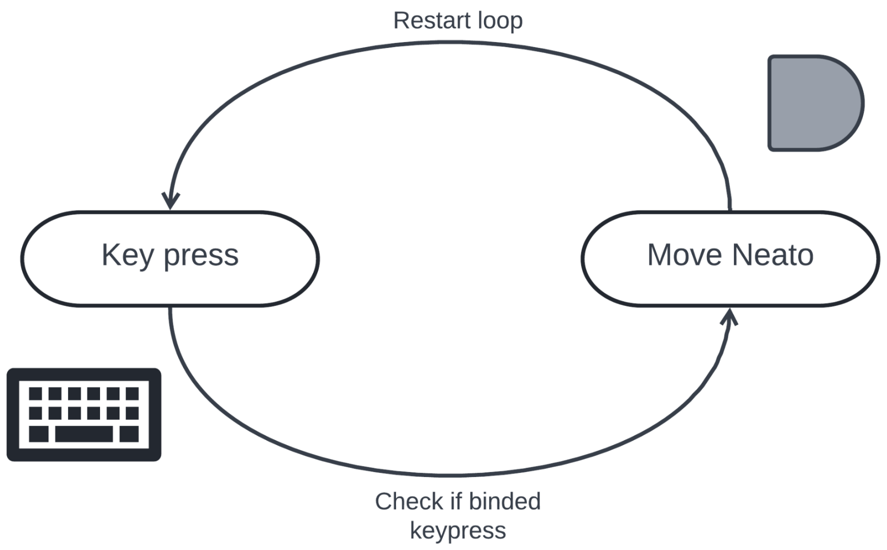

Figure 5. Teleop Loop

With our teleop, W moves forward, Q turns left, E turns right, A or D stops the neato, and S moves backward. It was attempted to include publisher capabilities of the key presses themselves to the other nodes, so they could also be activated upon key press. However, this was discontinued due to time limitations.

Obstacle Avoider

Obstacle avoider shifts between 5 simple states interlinked with a few logical checks. Like the finite state controller, this node used a helper function to keep track of the states as well as note any starting positions for turning. The hardest part of implementing this code was accounting for the fact that the global orientation of the neato goes from 0 to -0 and thus we had to write code to account not only for negative degree differences but also jumps from the start of one half and the end of another.

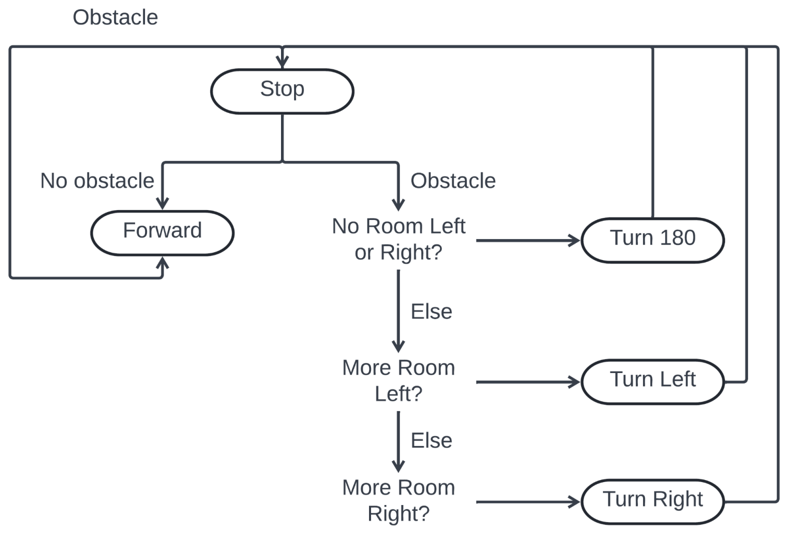

Figure 6. Obstacle Avoider States and Logic

The fundamental logic of the function was quite simple with the stop state serving as the in-between state of any transition in a bid to reduce unwanted movement. The general premise is that the neato will move forward as long as the lidar data indicates there is no object blocking its path. When an object is detected, the neato will check if it can turn either left or right. If neither of these is an ideal option, the neato will turn 180 degrees with the state shifter function noting its station orientation is that it can keep track of its relative change. If there is room to either side, the neato will weigh which side has more space and then turn 90 degrees in that direction. All of these states feedback into the same stop forward pattern that will continue until the code is terminated.

For the finite state controller, what was the overall behavior. What were the states? What did the robot do in each state? How did you combine and how did you detect when to transition between behaviors? Consider including a state transition diagram in your writeup.

Two approaches to state controllers were used within our code. One approach was an individual node represented by the finite_state_controller that instructed the neato to shift between two states: look for object or follow person. In the look for object state, the neato would move around at random angular and linear velocities until the environment presented a perceived object. Once this condition is met, the neato will switch to the object following state where it will head towards wherever the object is located within the neato's frame of reference. If there is no object within range, the neato will change back to the object following state.

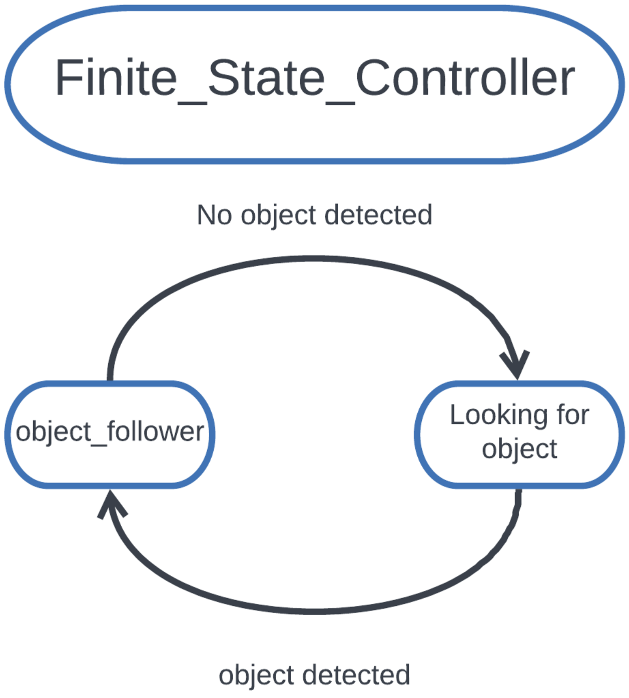

Figure 7. Node Changing Between States

All of these transitions were detected and handled by interpreting the lidar data to identify whether an object was present or not. We used self.states to track the state of neato and what code to implement within the node spin.

Our second approach was using a helper function within the obstacle_avoider code to handle both the changing of states as well as recording an initial value when a state was changed. For this approach, the neato used the stop state as a mediator between all other states to ensure it did not move more than desired.

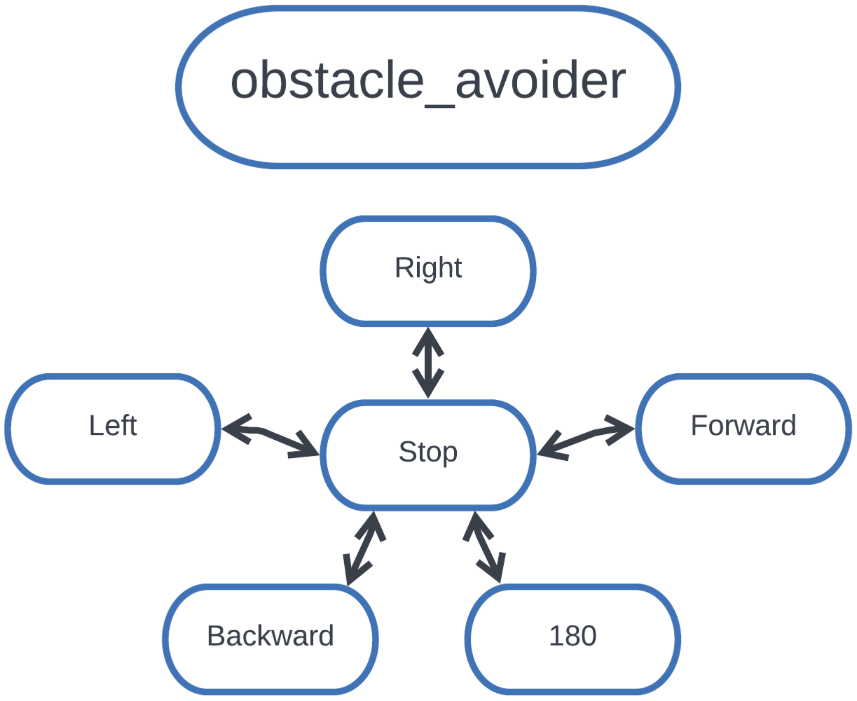

Figure 8. State Situation for Node

The stop state stopped all movement, the right state moved the neato clockwise, the left state moved the neato counterclockwise, the 180 state moved the neato counterclockwise, the forward state moved the neato forward, and the backward state moved the neato backwards. You may notice that 180 and left state seem to do the same thing but the difference is that our state controller also keeps track of how much a state needed to be active. For 180, this meant it would keep the state operating until the neato had turned pi and pi/2 or either left or right. The neato would always move forward until the lidar data indicated it had hit a wall at which point it would turn either left, right, or 180 depending on whichever one had the most room to move then continue to move forward.

How was your code structured? Make sure to include a sufficient detail about the object-oriented structure you used for your project.

Our code was structured into six separate nodes: drive_square, obstacle_avoider, teleop, person_follower, wall_follower, and finite_state_controller. In addition, we had the angle_helpers function that aided all of the nodes in converting cartesian coordinates into orientation angles.

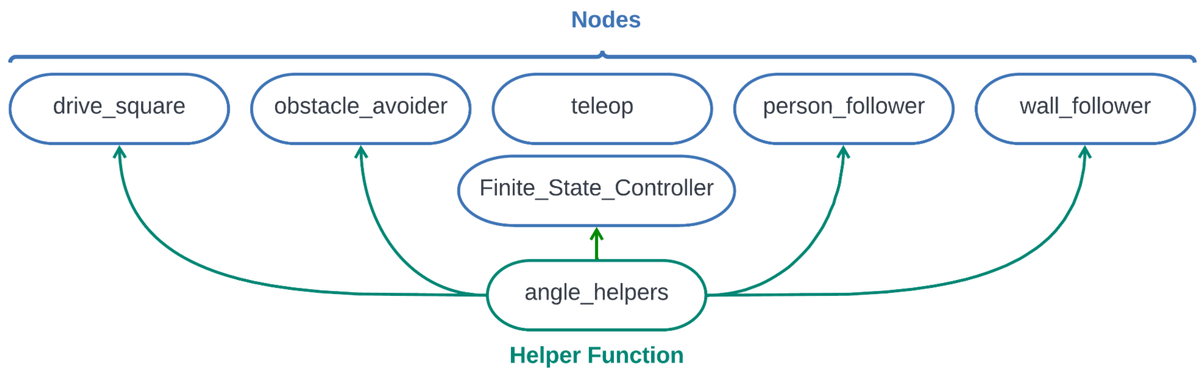

Figure 9. Abstracted Code Structure

Each node was its own class with multiple internal functions to handle subscribing to all of the incoming lidar, orientation, and other sensor data while publishing relevant motor commands to the neato. The internal functions also helped the neato determine what state it should be in depending on the goals of the node in a given situation. Not all code used the same structure and approach. Obstacle_avoider and finite_state_controller used a state-changing approach to keep track of and facilitate distinct stages for the neato to be in. The drive_square_function, wall_follower, and person_follower used proportional control to accomplish their goals while teleop simply translated keyboard input into velocity messages to send over ros. Except for the angle_helpers function, all of our nodes were self-contained and only communicated between themselves and the neato. Communication between nodes was attempted with teleop but this was shelved due to time constraints.

What if any challenges did you face along the way?

The largest problem we ran into was not using time within our code. Until this project, we did not appreciate how reliant we were on using timers, for, and while loops in order to do repetitive tasks or keep track of whether something had been completed. The first challenge was realizing that we shouldn’t use these techniques as Ros relies on being able to quickly loop through the code and any holdups would cause everything to break.

This really became a burden when we had to implement any code that relied on the robot turning a specific amount, which was most of our functions. Since turning is just angular velocity, of course, our first instinct is to turn for a certain amount of time to yield the desired distance. However, as time is imperfect when you are looping through your code constantly, and avoiding hanging at all costs, we sought an alternative. What we settled on was using the odometry, position of the neato within the global frame, to keep track of the relative amount it has turned. The largest challenge associated with this is that our robot is not necessarily only going to move counterclockwise or clockwise; thus we need to handle the trig necessary to yield the constant degrees moved no matter which direction the robot is turning. This is compounded by the fact, within the global frame, the neato does not have a traditional 0 to 360-degree orientation system with it instead being -0 to 0.

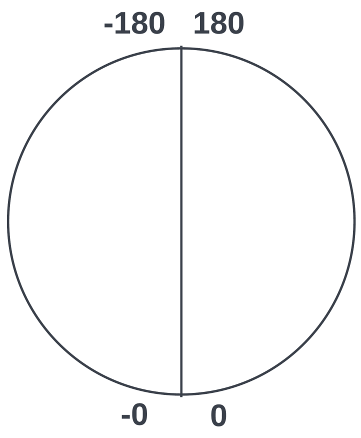

Figure 10. Orientation System of Neato in Global Frame

Therefore, we had to implement even more code to keep track of what frame the neato is shifting from as well as make sure the relative turn amount was always positive. Overall, implementing this system was the largest hurdle we had to overcome.

What would you do to improve your project if you had more time?

If we had more time, we would focus on improving how our neato identifies and categorizes objects. As most of our time was focused on moving the neato and keeping track of its position, we spent far less time on how we would also intelligently include obstacles within our movement algorithms. Our implementation, only dealt with objects when they were directly near or in front of the neato. However, our ideal code would take into account these objects far in advance so the neato could choose the most efficient path rather than figuring things out when it has already potentially taken a wrong turn. Overall, with more time, a better system of identifying, categorizing, and storing objects for the neato to avoid or move towards would be the aspect we would have improved.

What are the key takeaways from this assignment for future robotic programming projects? For each takeaway, provide a sentence or two of elaboration.

For loops, while loops, and timers should be avoided in code that works with Ros as they cause the program to hang, throwing subscribers and publishers out of wack.

Before starting a program, you should consider what repetitive actions your robot will take and implement those within states that you can repeatedly call.

Writing a state is half of the challenge as you also need to consider under what circumstances you want a state to trigger, avoiding any scenario where your robot can become stuck in a state for infinity.

Ros should not just be thought of as useful for communicating between robots but also for communicating within the robot itself. A robot may have multiple different systems that have their own parameters, but they can be united by using Ros as the interlinking chain.

Sometimes there is a lag before data from a subscriber starts arriving so before you run the main code you need to make sure that the data exists to avoid any None type errors in your logic.