CompRobo - Warmup Project

Benjamin Ziemann

September 21, 2018

The little robot that could

All code referenced in this document and more can be found in the project's git repo: https://github.com/zneb97/comprobo$_w armup_p roject$

1 Behaviours

1.1 Behaviour 1: Drive in a ~~Square~~ Regular N-gon

Originally presented as use "timing to drive the Neato robot in a 1m x 1m square", I ended up taking it a bit further, instead using the robot's odometry, it's distance from the point it was initialized, to drive in an n-sided polygon with any length sides starting from any offset in odometry.

In the background the class variables for the robot's x, y, and theta heading are constantly being updated by reading from the Neato's /odom topic. When the 'run' function is called this information is immediately used to per-calculate the corners for the regular n-gon based on the number and length of the sides which are passed as parameters to the class. This is then visualized in a matplotlib scatter plot. This was especially helpful when debugging the "calcCorners" function.

From there the robot drives forward until it is within a tolerated range of the corner. It knows this by comparing its current /odom data to the next corner location. Once there it turns in place until it reaches the heading necessary to drive straight to the next corner. Like driving straight, this also has a tolerance (linEp and angEp respectively) to compensate for network latency.

The most challenging part of this was figuring out the trigonometry for how much to add or subtract to the x and y for the next corner. Starting off it was difficult to think in terms of the robot and /odom world coordinates and heading but it became for intuitive as I went on. Plotting

it in matplotlib using my 'visualize' function also helped to make sure the shape I was making made sense regardless of its orientation.

One disadvantage to pre-calculating the corners is that if the robot tracks too far off it cannot recover. One way this could be improved is upon reaching a corner, calculate the next one. This way the errors in position do not accumulate.

1.2 Behaviour 2: Wall Following

True to its name, this behaviour attempts to have the Neato robot follow the wall given an initial position and orientation near the wall.

My solution to this was for the Neato to calculate the angle perpendicular to the "wall" and then try to bring that angle to either 90 or 270 (the sides of the robot so when travelling forward it is parallel to the wall) and maintain then maintain that angle. The way it calculates this is by comparing the lengths of laser scan of each angle scan and the scan X degrees (X is generally less than 90) ahead of it. Of all of these it then takes the halfway point between the two angles X degrees apart with the minimum difference in length (generally nearing 0) as the only way this occurs is if there is a plane (the wall) running perpendicular to that halfway angle.

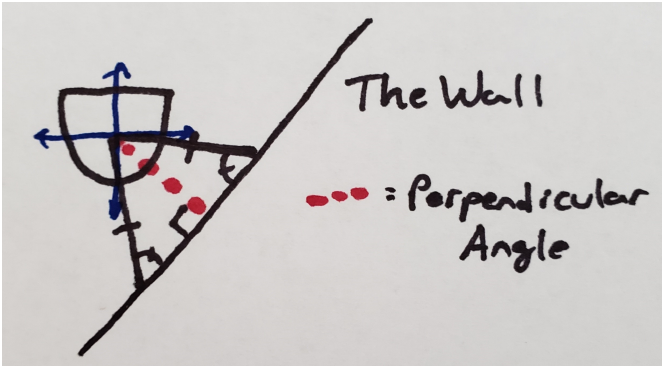

Figure 1: Finding the perpendicular

Once its perpendicular angle has been calculated, there are four cases that can occur during the initialization and run of the script and are represented in the code. These occur in the combinations of whether the wall is generally to the left or right Neato robot and whether it is turned towards or away from the wall. These can be thought of in terms of quadrants to the robot. Depending on which quadrant the perpendicular is, the robot turns to bring the perpendicular closer to 90 or 270 so the robot runs parallel to the wall.

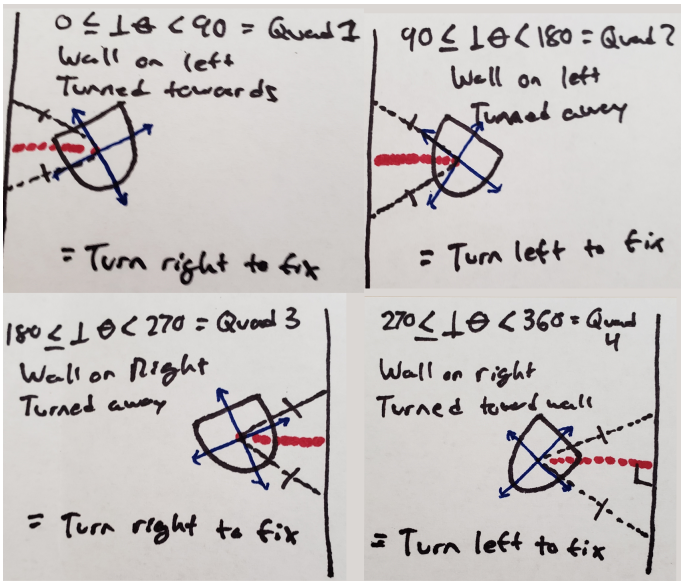

Figure 2: The possible cases for wall location and robot orientation

Visualization was performed by using the robot's odometry and the angles to the sides of the perpendicular angle to come up with a series points which I turned into markers to be displayed in rviz.

The subscriber to the /scan rostopic, after its data is processed to find the perpendicular angle, updates a class variable so that other processes, like updating the robot's driving, can be performed while it is running in the background. Unlike the wall follower, I do most of the data processing in the subscriber's callback function as I do not want to have to constantly store the entire laser scan list as a class variable.

Problems with this approach occur when the robot is in a crowded area as it is more likely to have scans of equal length and may calculate the angle perpendicular to an object rather than the wall. This could be solved by implementing line detection with an algorithm RANSAC but I did not put in the time to implement this.

1.3 Behaviour 3: Person Following

Like the name implies, this behaviour attempts to follow a human around. This behaviour works best in a large open space with the human remaining within a given angle to either side of the robot and within a given distance.

My way about this behaviour was to define a region of interest (some degrees out from the robot on the left and right and some distance. It would then calculate the center of mass of all points by just averaging the distance and angle. Using proportional control with a goal of the person being directly in front of the robot and some distance away and by normalizing the data between -1 and 1 for turning and 0 and 1 for driving the robot the robot would then turn correctly and follow at a reasonable speed depending on how far off the human position was from the goal position.

The trickiest part of this program was working with angle wrapping. Because moving from left to right in front of the robot corresponds to the laser scans X to 0 then wrapping around to 359 to Y, I ended up writing several cases for when it does wrap and processing the laser scan data into a list in specific ways (index 0 being the left most from the front of the robot, n-1 being the right most) that made it easier for me to debug and conceptualize.

One problem with my solution is that it can get "distracted" by any large objects it happens to pass by as the center of mass is calculated using all points within the region of interest. Two ways I could counteract this is to have it remember where the human last was and move based on that or have it recognize the two legs of a human.

1.4 Behaviour 4: Obstacle Avoidance

Obstacle avoidance attempts to have the robot move forward while avoiding obstacles and returning to the intended direction after passing them. I tackled this by treating each point around the robot as a force acting on the robot. This approach was implemented after reading about potential fields on the papers linked on the project page[1,2].

My first approach involved only taking in the objects in front of the robot and turning based on it. This did not work as the robot just continually veered off to one side. This was I didn't take into account that the robot needed a way to reset it's trajectory to its original path. Instead by treating object behind it as something to turn towards (this is making the assumption that the robot turned to move past the object in the first place), the robot will curve back to its original trajectory.

The "spread" or area around the object which it would have influence on the robot path is used to in comparison with the robot's distance to the object to determine how fast the robot should be turning away from the object. The closer the object, the more slow the robot's linear velocity (to allow for more time to turn) while having a greater angular velocity to turn more quickly from close objects.

The trickiest part of this behaviour was identifying values for the spread and radius of objects along with the proportional constant. Tuning these took up a lot of time and the end result still isn't great.

1.5 Behaviour 5: Finite State Controller: Approximating Tangential Circles

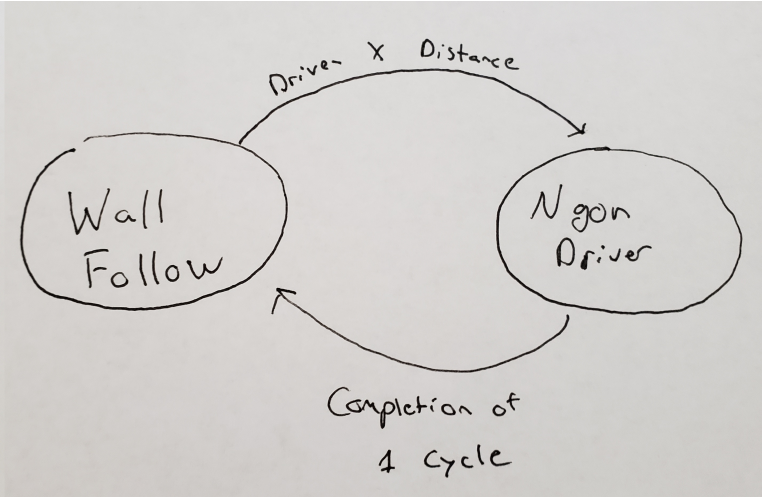

Figure 3: State and transition diagram

For state control I decided to combine my shape driving and wall following. The program runs wall follower for a given distance than creates a triangle before returning to following the wall. As it goes along at every distance interval it will add an additional side to the shape while decreasing the side lengths, creating a closer and closer approximation of a circle that is tangential to the line of the wall. The states were the behaviours themselves, either following a wall or driving the n-gon.

2 Code Structure

For all the behaviours I used a class structure which allowed me to take advantage of class variables rather than making everything global. This was extremely helpful as I could use the callback functions of my subscribers to update class variables that could immediately be used in the classes' run functions. The one time I broke from this structure was for obstacle avoidance where instead everything is run out of the laser scanning callback function. This was due to me running into problems with my of original approach of a list of points updating and changing lengths partway through the run function, causing indexing errors.

3 Challenges

Besides the small challenges of each behaviour (see the write ups for each), the largest challenge came with debugging. Often the robot would act in two very different ways despite seemingly being initialized and encountering similar conditions each time. This was and still is especially apparent in my obstacle avoidance behaviour. Between managing the (for a human) large amount of scan data, running multiple functions at once (callbacks, runs, state controller), and the actual math, there is a lot to take in and a big part of this project for me was learning how to isolate my debugging and using process of elimination to find what went wrong. It was also quite a challenge of patience. In the future I'm going to try to mitigate this by really trying to build my code up function by function (see next section, takeaways).

4 Key Takeaways

Some of the key takeways from the warmup projects were:

- Visualize. As much as it can be a pain to do out the tiny bit of extra math to put in markers in rviz, it makes debugging so much easier rather than just staring at a list of terminal numbers.
- Draw pictures! Lots of pictures! There's a lot of math going on in any one of these behaviours. Trying to do it all mentally is difficult and taxing. It also lets you more easily spot cases you may have not accounted for and provides easy documentation for when you come back later!
- Build it step by step, function by function. Get one smaller aspect of the robot working first then move on to the next. This will usually end up being the workflow of your robot anyways!

5 References

1. Potential Fields Tutorial - Michael A. Goodrichtics
2. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots - Oussama Khatib