Warm-Up Project Writeup

Victoria Coleman

Computational Robotics

9/19/14

Overview:

All of the code fulfilling the project can be found in the script “Basic_Behaviors.py” in the “src” folder of the project. The code was based of the code demoed in class.

Implemented Behaviors:

Wall Following

The wall following behavior successfully implemented follows walls; however, it does not only follow walls. The wall following behavior goes to the closes object in range and follows along its borders, keeping at a variable distance away. When another abject becomes closer, it then switches to following that border.

To accomplish that task, I used a method inspired by Diana’s approach by having an arbiter that combines the two wishes of the robot for this behavior: to be the correct distance from the wall and to be parallel to the wall. The controller consists of two basic P (position) controllers that calculate the angle between the closest point to the robot and 90 degrees to the robot and a desire turn angle proportion to the distance between the wall and the robot. The calculations for the P values handle angle wrapping and make it so the robot never wants to turn more than pi radians. The P value to keep the robot closer to the wall is calculated by finding the distance differential and multiplying it by pi/5, so the further the robot is from the desired distance, the more it wants to turn towards or away from the wall to get it to the desired positon. The arbiter is very simple and adds the two P values to determine how much the robot moves.

The code was made fully functional in the simulator, and handles going around continuous objects well and when surrounded by object. When two objects were close together but had a wide enough gap to fit through comfortably, the robot had a very waving behavior, but it still behaved correctly.

Obstacle Avoidance

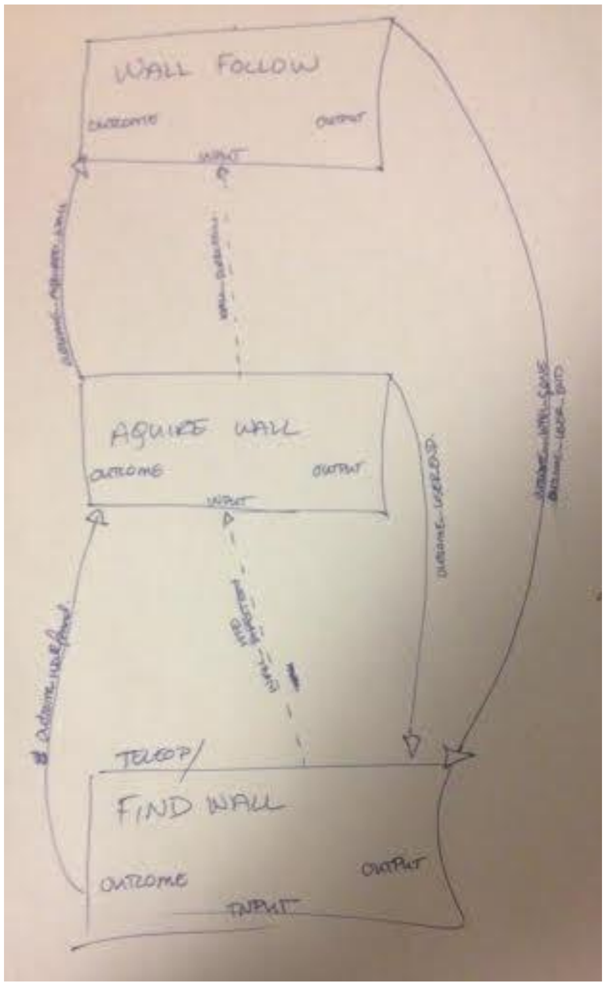
The implementation of obstacle avoidance had the robot move in a straight line, but deviates when it approaches and object and continues on the new strait trajectory when the object is no longer in its path. To determine the objects around it, it feeds through the sensor data and groups points together into objects if they are within .3 m of each other. It determines this by converting the current point and the next point into Cartesian coordinates and then compares the distance together. If they are within that buffer, the object is considered to be continuing. When the points get too far apart, it is assumed the object has ended, and an object Object with the following information is created: average distance of the object, the closest point, the width (in radians as it appears to the robot), and the angle to the center of the object from the robot’s perspective. (While this is more information than necessary, it would be useful for a more robust obstacle avoidance implementation.) When the loop gets to the last data point, it compares that one to the first one to determine if an object loops from the 359 degree to 0 degree. If that is the case, than the first object’s information is altered to account for the looping.

Once an array of the object detected is created, then the code sorts the list for objects in its path (ones with center angles within 15 degrees of 0 degrees). Of those, it chooses the closest one and veers away from it. Using a P controller, the closer to 0 the object is, the more the robot wants to turn away from it.

Finite State Machine:

My code uses a simple finite state controller and not the ROS smach package. While I played around with the package beforehand and got familiar with the implementation, I did not have time to debug integrating it with my code. I just used a simple state machine that switches between three behaviors: teleop, wall-following, and obstacle avoidance. To switch between the behaviors, the script queries the consul for user input to select what mode to be in any time while running. The robot only switched states when prompted by the user.

When I was initially planning my code, I thought to use a FSM for the wall-following behavior as mapped out in the picture below, but the actual implementation was used due to the simplicity and effectiveness of it. I include my initial though below because it was a nice exercise in thinking about FSM structure.



Note: teleop mode only works when the teleop package is run concurrently with the warmup_project .

Code Structure:

The code is in a simple script that includes many functions and one class that defines the data that describes objects the robot wants to avoid. When the code runs, a function initializes the appropriate subscribers and publishers then spins in a while loop to listen to user input to determine what state the robot should be in. The "scan_recieved()" function runs every time information is published to the /scan topic, and it passes the data to the appropriate function based on what state the robot is in. The functions return a Twist command, which the function sends to the /vel_cmd topic to control the robot. No command is sent when it teleop mode so the teleop package can control the robot if running.

Challenges:

The biggest challenge for me was working with the simulator and robots that both liked to crash. I got all the code running smoothly with the simulator, but due to waiting to test my fully developed code on the robots until the last day, meant that charged robots where difficult to find. I got parts of the behaviors mostly working the with robot, but not combined. Luckily I'm familiar to ROS already, so there was little difficulty with that facet of the project.

Improvements:

There are many improvements I would make to this project give more time. Here are a few:

1. I would make all my P controllers PID controllers for smoother behaviors
2. I would restructure my code and clean it up a lot. I'd probably break up the script into separate ones for the different behaviors or possibly make it all one class with sub methods so I would not be using terrible global variables
3. For wall following, I would analyze the image with openCV or try to analyze the Cartesian plot for linear patterns to distinguish between walls and other types of object
4. For obstacle avoidance, I would love to infule a little path planning so that it tries to return to it's original trajectory after avoiding an object (a simple point A to point B behavior)
5. I would use the smach package for my Finite State Controller
6. I would make the robot slow down when it got too close to objects
7. For obstacle avoidance, it currently only avoids the closest object in the emergency range. I would like it to consider other obstacles to determine which direction to turn and by what
8. Also in regards to obstacle avoidance, it should take into consideration the width of the object and nearness of the object to also influence how quickly it turns

Interesting Lessons:

I've never used a simulator before for programing robots, and I LOVED it. It made development of the code much quicker so just tuning and other minor details had to be smoothed out with the much more troublesome robot that takes a lot more time between runs to deal with (more overhead). It also makes planning time to work on the project much lest restricting. I wish I had taken more time to research other packages and libraries to use before starting the project, because after I made significant progress, I realized that I was re-inventing the wheel for many things, and I could have developed much faster and more robustly if I had taken advantage of what was already available in the world.