# Wall Follower and Object Avoider

Brooks Willis

Computational Robotics, Fall 2014

## Introduction

For this project I created a wall following routine and an object avoidance routine. The wall follower was very successful, it is capable of locating the nearest wall (in the absence of other objects) and orienting itself to it, and then following it until the wall runs out. The object avoider can robustly locate all obstacles within range, and generate a force map that will cause it to arrive at the location furthest from all obstacles. It does not inherently want to move to a specific location, but is structured such that it could be easily added to another behavior that has other functionality. Both behaviors were structured as simple state machines using if statements and Boolean flags. Videos of the two scripts working can be found here - https://drive.google.com/file/d/0ByXNsDhR0k6qdEhJaTREMjNXb0k/edit?usp=sharing

## Wall Follower

The script for the wall follower is called wallfollow.py. At a high level, the wall follow script is structured somewhat like a finite state machine. The main loop contains a nested if statement that uses Boolean flags to force the code to stay in one state for an extended period of time. When initialized, the script takes in the desired follow distance from the user, and then looks for the nearest object of any kind. Once that is identified it verifies that the object is of a certain size, it assumes that the object is a wall. This assumption is not the best, but in this case we can assume that the wall will be the nearest object. At this stage the code is in the “approach wall” state. With the wall identified the robot moves directly towards the wall using a proportional controller until it is within +5% or -20% of the intended following distance. When that threshold is crosses a flag is flipped which put the code into the “orient” state. It will stay in orient until the wall is between $85^0$ and $95^0$ off of the front of the robot (effectively directly to the left). When this range is entered the code will enter “wall follow” mode, which flips the “orient” flag to false and the “wall follow” flag to true. Now the code is in the “wall follow” state, it uses a P controller to maintain its orientation to the wall while moving forward at a constant rate. To feed this P loop, the code checks the pair of point that is closest to 15 degrees forward and backward from the robot and compares the distances. If there is no pair of points in the dataset within the $\pm5^0$ to $\pm15^0$ range, it finds the points that are as far out as possible, to a maximum of $\pm20^0$. If these distances is separated by less than 0.5m, the robot will drive straight forward, if they are between .1 and 0.5m separated the P loop kicks in to correct the error. If ever the error exceeds 0.1m, the code will exit the “wall follow” state and return to the “approach wall” state. This will continue indefinitely.

Some logical extensions of this that I would like to implement are cleaner handling of internal and external corners, rather than continuous stretches of wall, since that will make the algorithm significantly more useful in the general case. To implement that I would add another state which specifically handles wall corners, and would probably be triggered when the robot can’t find a wall in front of it but can behind it, which is a check that is already happening to keep the robot aligned to the wall. Beyond that I would really like to refactor this code to be object oriented, there were a lot of workarounds that I used that would be solved by that change.

## Object Avoidance

The script for the object avoidance is called obj_avoid.py. This script is structured very similarly to the wall follower. There are three states: straight, curve, and turn. Each is fairly self-explanatory. Before reaching the state machine, the lidar data is grouped into clusters of points which are separated by less than the width of the robot. This way all gaps narrower than the robot get treated as solid walls. Once these clusters are formed they are transformed into force vectors which push the robot directly away from that each cluster, the magnitude of the vector is used to determine how hard to turn away from that object. Of course the robot can only move in one direction, so these vectors are then summed and the resultant vector is the input to the state machine. This code could easily be added to a larger arbiter by modifying it to output this vector rather than a command velocity, since the vector by itself represents the correct way to move in order to dodge all obstacles.

The initial state is determined by the first vector it is given. For desired angles $\pm 10^0$ it will enter the "straight" state, which simply drives forward since it has identified no objects in the robot's path. This state will be maintained until the desired heading deviates from the current path by greater than $20^0$. If the desired angle is within $\pm 80^0$ the robot will enter the "curve" state, which uses a P controller to turn towards the desired heading. The desired heading is scaled to a 0-1 scale and multiplied by a constant, and a ±1 (to determine direction), for both the translational and rotational motion. This state is maintained until the heading leaves the $\pm 20^0$ to $\pm 80^0$ range. Finally, if the desired heading is greater than $\pm 80^0$ the robot enters the "turn" state, which is a safe mode of sorts. It will turn towards the desired heading (in the closer direction) at a constant speed with no translational motion until it is within $\pm 20^0$ of the desired heading. The combination of these three states creates a reasonably robust object avoider that runs away from all objects.

Unfortunately I was unable to tune it properly to get it working on an actual neato, but it did work repeatedly in simulation with numerous object configurations. The biggest problem with obstacle avoidance is that the data we are getting off of the neato is fairly jumpy, it is hard to force the robot into a smooth path when the desired heading can jump upwards of $20^0$ per cycle. To get around that I created overlap between the exits and entrances to states, so once it enters a state a fairly significant change needs to occur for it to change again. That alone fixed most of the problems that I was experiencing, which also included very jittery motion (from rapid state switching) and inconsistent heading values.

For the scope of this project I do not think the "sum of forces" approach that I took is a very good one, it is reasonably complicated to get working even in a vacuum, however I think that this method would work excellently as part of a larger system if it was but one input to a larger arbiter. This is an area I would like to explore a lot. Some immediate improvements that this lends itself to include being converted to an object oriented behavior in a larger routine, or to adding functionality that lets it selectively ignore objects outside of a certain angle range until it reaches a shorter distance to them. That would significantly increase the versatility of this method. I would also like to give it the ability to find and navigate through gaps between objects more easily, since the current implementation does basically the opposite of that, to its detriment.

## Interesting Lessons Learned

First and foremost, I will make all of my scripts object oriented in the future. It is a cleaner way of structuring the code, and will generally improve the usability. Another major take away from this is to plan out the structure of the state machine before starting on data analysis and exact behavior. Even if it changes later on, having that structure to work in vastly accelerates the development process. Implementing more robust data filtering would have been a huge help on object avoidance, but I didn’t add any because of time constraints. Finally, I think that next time I will write out all of my logic inside the state machine before putting fingers to keyboard and run it by several other people to catch blaring logical errors before I implement them. On the whole, I learned a lot on this project and got much more comfortable with the neatos, python, and ROS. It was a very good refresher on methods of working out behaviors as well, since I hadn’t tackled a major programming problem in nearly a year.