Warmup Project

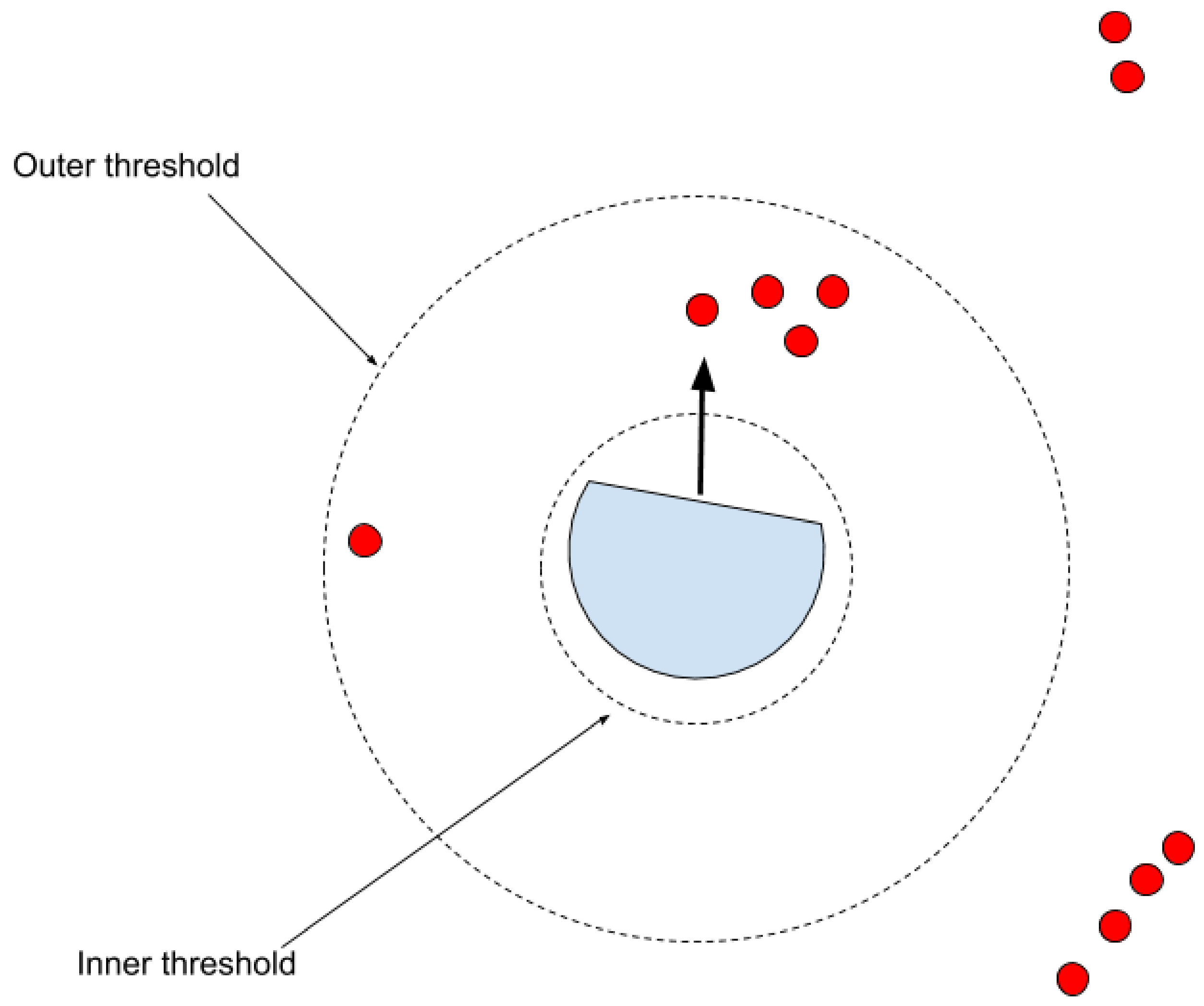
Anna Buchele and Ariana Olson

Wall Following

In the wall following state, the robot used the RANSAC algorithm with linear regression to detect a wall. If the wall was found to be too far away, the robot would approach the wall at a perpendicular angle. Once it was the appropriate distance from the wall, the robot would travel parallel to the found line angle of the wall.
The original implementation of the wall follower used proportional control. This worked well (actually better than our final implementation) for following a wall with the robot starting close to a wall and in an obstacle-free, low-noise environment. However, this implementation had a lot of trouble with any obstacles, and if there was anything close to the robot other than the wall. It also suffered a lot in noisy conditions. It also had difficulty detecting if a wall was actually a wall, or for finding and approaching a wall. The implementation with the RANSAC algorithm and linear regression attempted to fix some of these issues. The RANSAC algorithm was actually quite successful at finding the wall, once the parameters were properly tuned. This allowed our robot to successfully locate and approach a wall, even in a noisy, obstacle-ridden environment. However, the implementation with the RANSAC algorithm did less well with the parallel following of the wall. Its less-reactive manner had less allowance for fine-tuning the precise angle, so our robot ended up swerving slightly more than in the previous implementation. If we had more time on this project, we would implement a blended solution: using the RANSAC algorithm to find and approach a wall, and then once we were within tolerance distance, using the proportional control method to follow the wall. This would give our robot a much smoother method for following the wall, and we could switch between implementations based on the needs of the environment.

Person following

For person following, we has to determine a target for the robot to follow. We did this by assuming that in a reasonably non-noisy environment and for a small radius around the robot, a person could be identified as the “center of mass of the points”. We took all of the range points from the LIDAR scan in a ring shape around the robot, each on represented by a polar vector. We then took the average of these vectors to calculate a center of mass that represented the person being followed. We used proportional control on both the angular and linear velocity of the robot. One of the most challenging aspects of controlling the robot in this way was that noise from the surrounding environment could greatly interfere with the robot’s target. For this reason, we had to add bounding regions on the acceptable LIDAR points, which meant that tuning the proportional controller and tuning the threshold radius were difficult to tease apart.

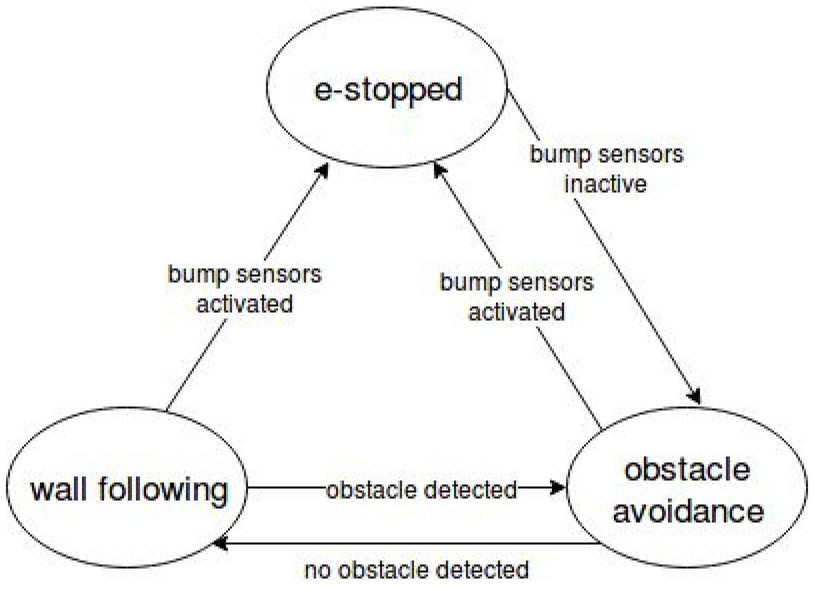


Obstacle avoidance

For the obstacle avoidance, we used the concept of potential fields. In this implementation, each object found in our lidar produced a “pushing force”, attempting to move the robot in the opposite direction, with a stronger force the closer the object was (at a strength of 1/(d^3), where d is the distance from the robot). We then added up every “force” and then sent the robot in the combined direction. This had the effect of our robot aimlessly wandering around in search of the area furthest away from every object. The obstacle avoidance became more useful when we also added in a “target”, which would “pull” the robot in the direction of the target, with a force strong enough to be the prevailing direction of motion unless an obstacle was very close. Once the obstacle was avoided, the robot would return to its primary direction of movement until it encountered another obstacle. If we had more time for this project, we would likely fine-tune the parameters of this system more, as our system currently runs “good enough”, but likely not as well as it could.

Finite State Controller

The finite state controller allowed our robot to switch between three states: e-stopped (when the bump sensors were pressed), obstacle avoidance (when an obstacle was detected very close to the robot), and wall following (finding a wall and traveling parallel to it). In the e-stop state, the robot turned off its motors and did not move. In the obstacle avoidance state, the robot turned away from the closest and largest obstacle. In the wall following state, the robot would seek out and travel parallel to the nearest wall. The finite state controller switched between the states in the following manner: the emergency stop overrides all other states, and the obstacle avoidance overrides the wall following. The obstacle avoidance behavior is chosen if the obstacle avoidance node attempts a turn above a tolerance, which indicates that an obstacle is quite close to the robot’s path. If the obstacle avoidance node is not publishing an attempted turn above this tolerance, then the wall following state is chosen. This hierarchy was designed to minimize the robot’s likelihood of hitting obstacles or ending up stuck.



Code Structure

Our code was built in an entirely object-oriented structure. Each of our nodes (emergency stop, wall following, person following, obstacle detection, finite state controller) is a class with its own rosnode, with each function being a function of that class. Each node subscribes to a few topics, and publishes its own recommendation for how the robot should move (/node_name/cmd_vel). The finite state controller, also its own node, then selects the correct course of action based on the available information.

Challenges and Lessons Learned

We had a lot of trouble with the reliability of the lidar, especially since the lighting conditions and materials reflected from had a large effect on our data. As Olin students, we work at all hours of the day and night, and in many different locations, and so we found it very difficult to control for the constant variations. We also spent a lot of time debugging, which is to be expected.

We learned a lot from this project, other than the basics of ROS and rospy. First was that visualizations could be extremely helpful: we used markers in rviz to denote the desired line, and the desired point to pursue. This helped us immensely in debugging exactly why our robot kept ramming itself into the wall (the RANSAC parameters weren't properly configured, allowing noise to to pass as a wall). Another lesson was the importance of finding methods to control for changes in the environment. We had to re-calibrate a lot of parameters nearly every time we switched rooms, or worked on the robot in different lighting. If we had found one room we could reliably work in early on, we wouldn't have needed to spend nearly so much time recalibrating. Another big takeaway was that if a “dumb” (read: reactive) system was able to perform well on a task, adding in smarter programming might not actually improve its performance. In our case, the inclusion of the RANSAC algorithm and linear regression for smarter wall finding, while helping us find the wall, actually introduced a ton of new problems and performed worse on the initial task. The most important lesson, however, was that when our robot is behaving oddly, the best thing to do is to visualize what it's seeing, visualize what it's trying to do, and then sit down and piece through the code why it might be trying to do that. It's very easy to think “changing this one random parameter will fix everything!” but that usually isn't the case. Using the tools available to us and debugging slower led to much better results throughout this project.