CompRobo: Spring 2017

Warmup Project

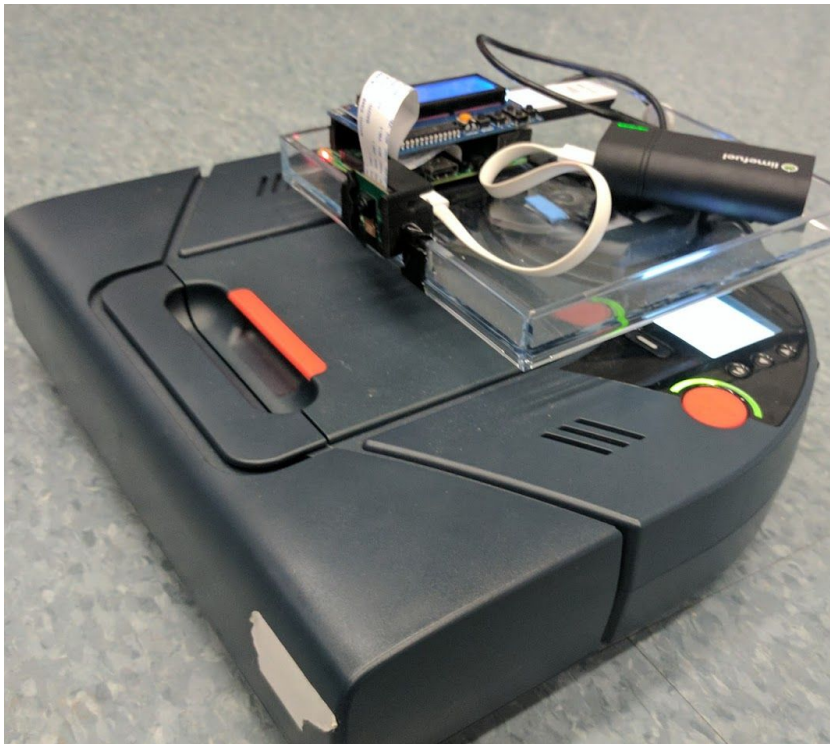
Eric and Carl

Introduction and Overview

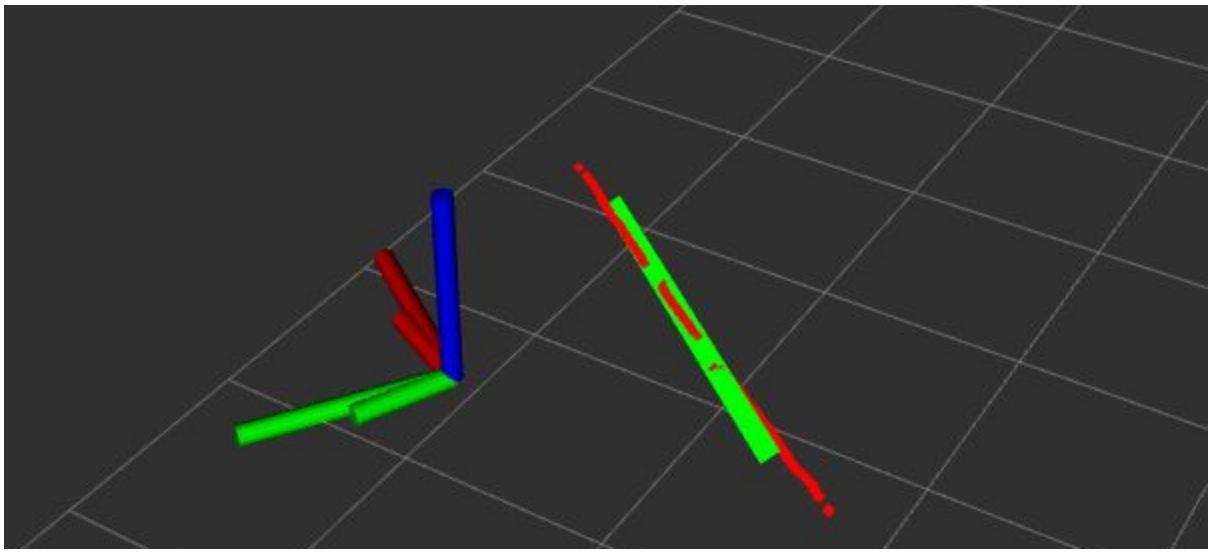
For each of the three main challenges, we used the same basic structure, composed of two nodes

- A *detection* node that finds the feature we are looking for and publishes it as a PointStamped or visualization_msgs/Marker
- An *actuation* node that outputs a cmd_vel message to move the robot towards or along the detected feature

This is important because it allows parallel development, creates a natural point for visualization, and enhances the reusability of the code, both within this project and for later tasks.



Wall Following



Detecting the Wall

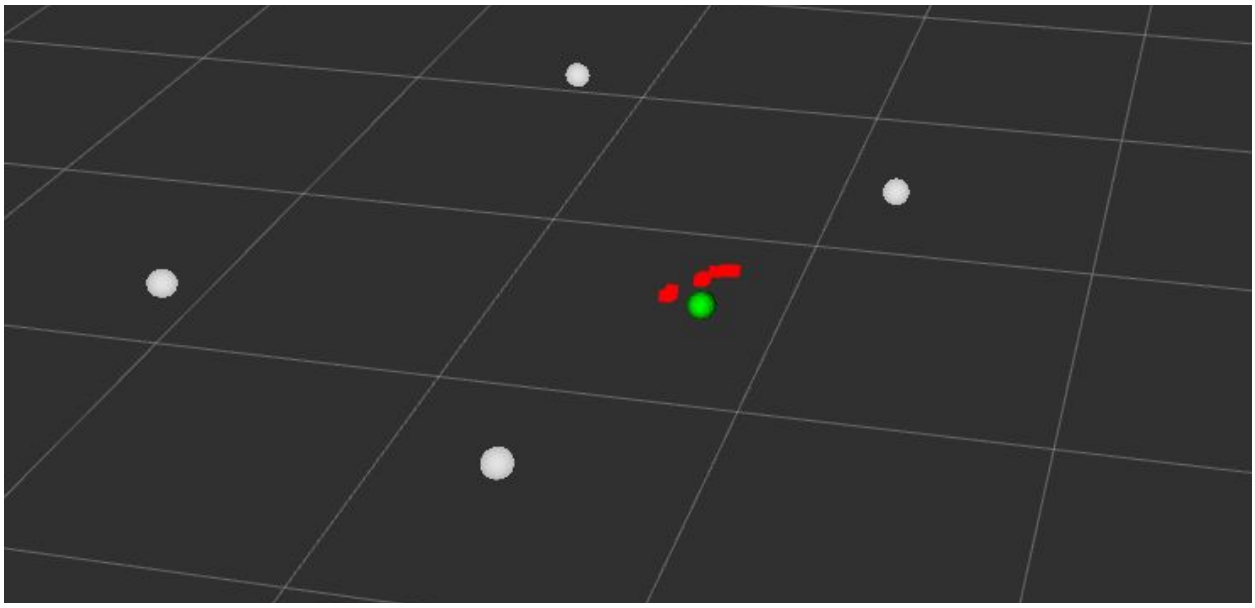
To detect the wall, we decided to convert the laser scan into cartesian coordinates and then filter out points that we did not want. We filtered out points that were to the left of the robot as well as points that were in front of the robot. From there, we chose to pick 2 points, one in the upper and one in the lower n^{th} percentile to construct a line. To transmit and visualize the data, we published it as a LINE_STRIP marker.

Following a Line

To follow the detected wall, which is treated as an infinite line defined by two arbitrary points, we use a two-stage proportional controller. First, we convert the received line into the robot's base_link coordinate frame, allowing publishers to operate in any frame of their choice. In the base_link frame, we calculate the distance from the origin to the line, and the angle of the line relative to the x-axis of the robot.

The first controller looks at the current distance of the robot from the wall and outputs what absolute heading relative to the wall that the robot should use to reach the desired distance offset. This goal angle is published to the transform tree so that it can be easily seen and debugged. The second controller compares this desired heading to the current heading, and drives the motors to attempt to bring the two into agreement.

Person Following



Detecting a Person

The location of the person is approximated as the center of mass of the points contained inside a “following region”, with both the following region and the detected centroid being published to the visualization topic. Additionally, the detected person location is published as a PointStamped in the frame of the robot with the timestamp of the original laser scan.

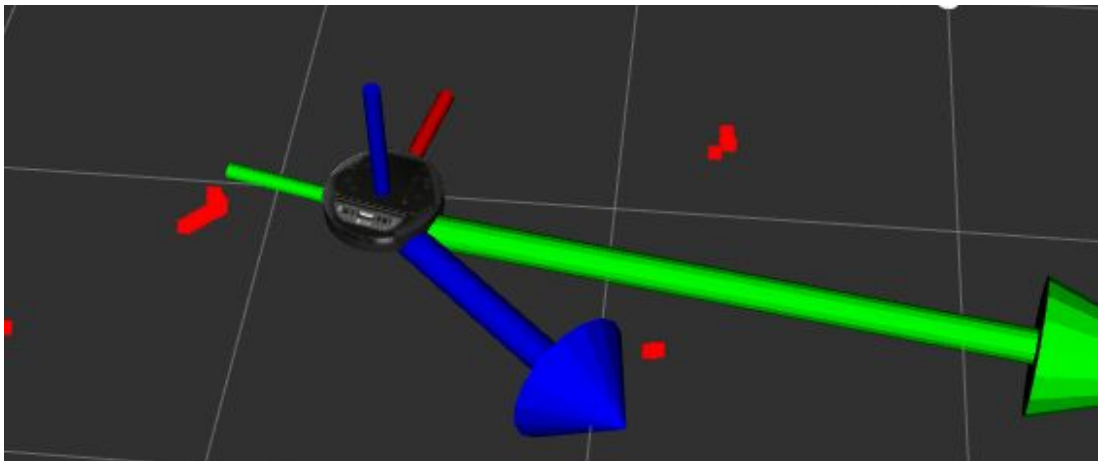
Following a point

To follow the published point, a simpler algorithm is used than the wall follower. Because an offset is not needed, a simple single-stage proportional controller attempts to keep the robot pointing at the target, with the linear velocity determined based on how closely the robot is pointing at the target.

Major decisions

All of the parameters, including the size of the tracking box, are controlled by ROS Parameters. This allows us to test simple changes without rebuilding the code.

Obstacle Avoidance



Algorithm overview

The obstacle avoidance operates directly on the list of points retrieved from the LIDAR, and uses a vector field navigation model to determine which direction the robot should be driving. It accepts navigation destinations from the standard /move_base_simple/goal topic published by rviz, and creates a constant-magnitude force vector pointing at that goal location. It then generates a repulsion vector from each point in its laser scan of magnitude $k_{avoid} * (d - d_{thresh})^{-exp}$ where k_{avoid}, d_{thresh}, and exp are tunable parameters. These vectors are all added together, and the direction of the resulting sum represents the desired direction of robot travel.

To better utilize code from the previous part, this desired direction is represented with a Point3D placed 1km away in the direction, allowing the same “move to point” node to be reused.

Major decisions and learnings

Operating directly on the raw points in cartesian 2D means our robot is able to see small objects, but responds less dramatically to a small object than to a large object.

This type of algorithm would work much better on a holonomic vehicle, our robot spent most of its time turning.

State Machine

We decided to combine all three of the previous behaviors together into the finite state machine, creating a system that will follow a person until it collides with the person or a wall, and then transition into wall-following mode. If it encounters an obstacle during its wall-following, it will attempt to use obstacle avoidance while navigating to a point 1m on the far side of the obstacle, at which point it will transition back to wall following.

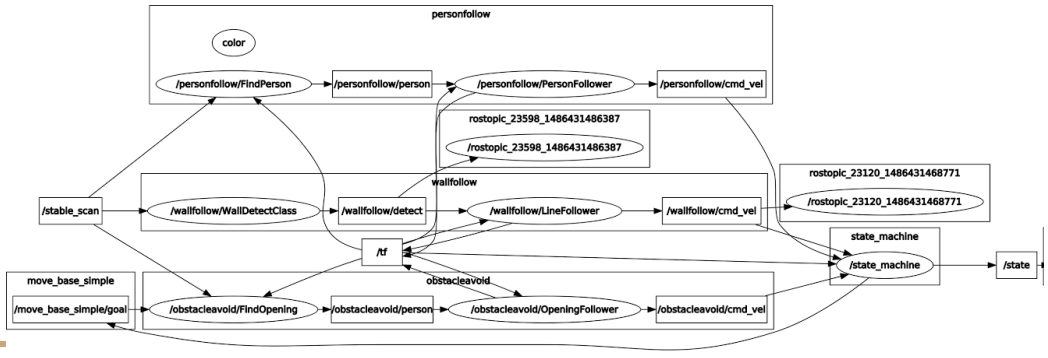
Code structure

Because our ROS nodes for other tasks already existed and worked, we wanted to make minimal alterations to their behavior and internal details. As a result, we decided to create one new node, state_machine, and use a roslaunch file to execute the other nodes inside contained namespaces. With this technique, shown below, we could have confidence in the ability of each node to behave exactly as it had in earlier tests and keep the code segregated into separated units.

When a state was active, state_machine simply forwarded /**wallfollow**/cmd_vel to /cmd_vel to give that behavior control over the robot.

Handling transitions

Because each of our state transitions was triggered by the bumper being activated, we chose to add a cooldown time of 2 seconds before another state change could be triggered. Additionally, starting right next to an object often prevented the successful execution of the next state, so we added an additional “backing up” state handled as a special case during state transitions.



Reflections

Challenges

- Working with a physical robot makes the write-test-debug cycle significantly slower that in a pure-software project. One technique we used to try to overcome this was rosbag, and if we were to do this again, we would use it earlier and more often.
- Simply finding times to meet proved surprisingly challenging, and one person's progress would sometimes be held back because the other person was busy. More intentionality can help to resolve this, as can a better division of tasks.

Improvements

- The person detection code is depressingly mundane in its current state, and does a really bad job of rejecting large disturbances. Making a "laser scan processor" that extracts lines and blobs from laser scan data would allow us to use essentially the same code for scan handling in all three major behaviors, and produce better, more smooth results.
- Additionally, many of our tuned control loops exhibited oscillatory behavior at about 1hz, which is approximately the round-trip lag of the laser scan data. This makes us think that the robot is hindered by having antiquated data about the position of objects in the world, a situation that could be improved by using the history data in the transform tree.

Key takeaways

- Object-oriented code is nice. Each node internally used one object to store local state, and the state_machine node used a further Enum class to keep track of the current state, all of which made our code more portable and readable.
- Creating intermediary visualizations is really important, but don't use them to transmit data to other nodes. Custom messages are better for that task.
- tf is amazing, and being careful to put the correct header on each message you publish is worth the effort.
- Pair programming allows both people to contribute evenly and better understand the code