# Warmup Project

Yuzhong Huang, Shruti Iyer, David Zhu

## Driving in a square

When a robot is driving in a square, it traces four sides of a square of length 1m.

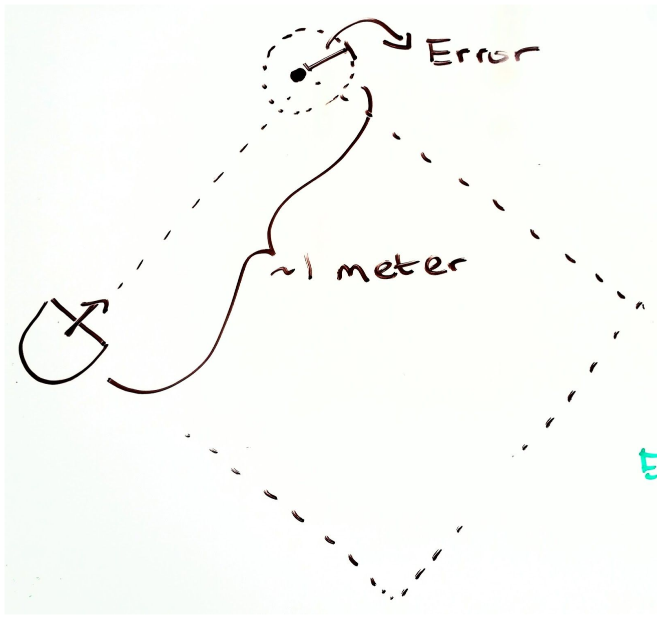

Figure 01: Location handling for error. After determine the goal, the robot moves straight until it is within reasonable range.

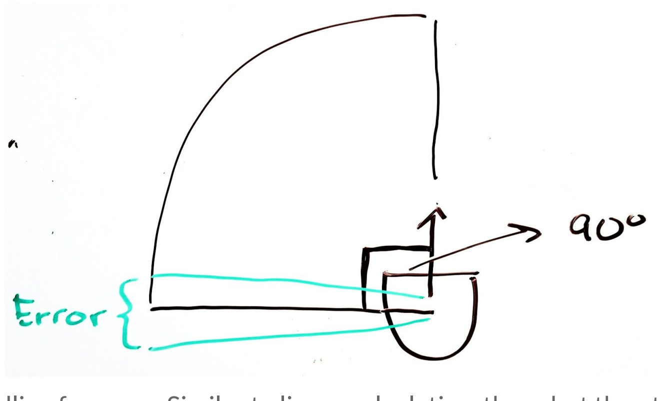

Figure 02: Angle handling for error. Similar to linear calculating, the robot then turns until it is within reasonable range.

To make the robot drive in a square, we used the robot's onboard odometry. Odometry provides us with the robot's position and heading in the 'world' coordinate frame. Here are the 5 states that the robot can be in. Here are their descriptions:

1. **Calculate Forward:** This is also the *start state* of the robot. Using the current (x,y) coordinates and the heading we get from odometry, we calculate the values of (x,y) that is 1m in the forward direction.
2. **Move Forward:** In this state, the robot moves forward until it reaches the calculated (x,y). The mean square margin of error is 0.2m. While covering the 1m distance, if the robot is within 0.2m of the calculated (x,y), it thinks that it has reached the destination. This error may seem high, but our robot tends to overshoot, so we calibrate for that.
3. **Calculate Turn:** After it has covered 1m distance in forward, it has to calculate the desired heading of the robot in order to make a 90° turn. We do that by adding 90° to the robot's current yaw.
4. **Move Turn:** In this state, the robot moves until the current yaw matches the calculated yaw. The margin of error is 8°. If the target angle is within the margin of error, the robot thinks that it has completed the 90° turn.
5. **Stop:** If the robot has finished tracing the four sides of the square, it changes its state to 'Stop'. While in this state, the robot stops moving completely.

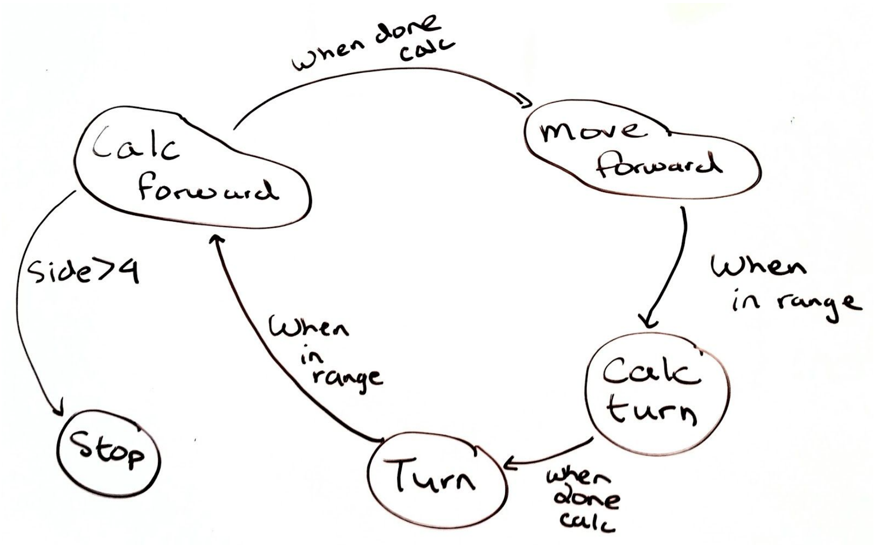

Figure 03: Drive Square state machine.

While moving between states, we also keep track of the number of times the robot was in 'Move Forward' state so that we know when the robot has finished tracing the four sides of the square.

## Wall following

This action requires the robot to identify a wall closest to the robot and follow the wall by moving parallel to the wall. Additionally, the robot should turn at the corner without bumping into the wall.

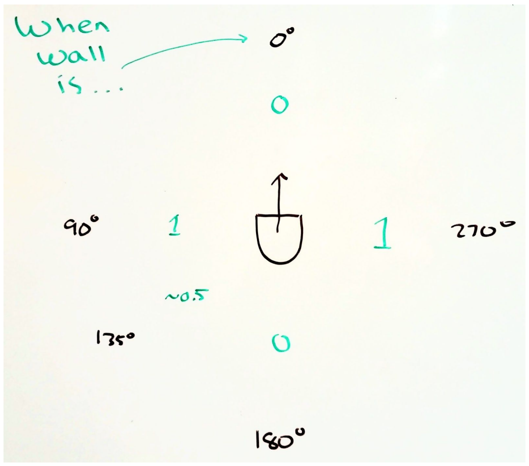

Figure 04: How the angle of the wall relative to the robot  affects its linear velocity.

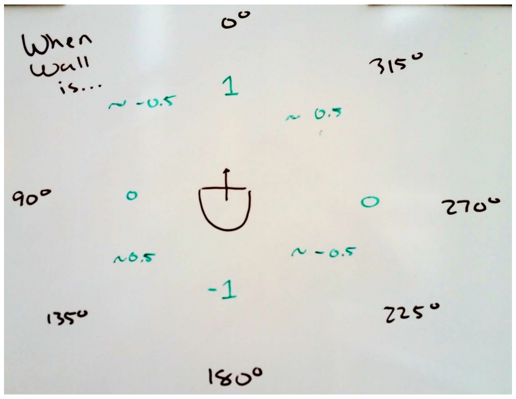

Figure 05: How the angle of the wall relative to the robot  affects its angular velocity.

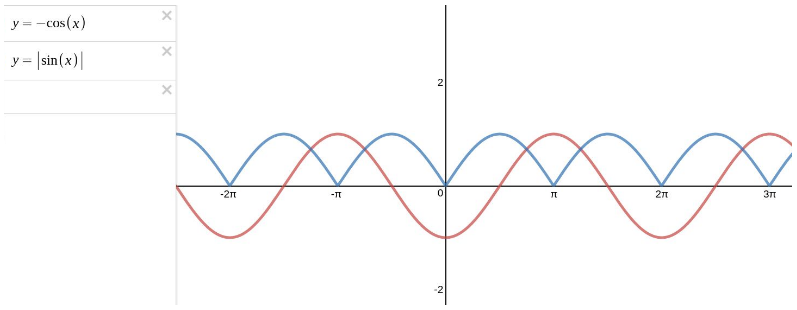

Figure 06: Change in linear velocity (y = |sin(x)|) and angular velocity (y = - cos(x)) with angle of the wall.

The strategy we used for this particular action is described as following: We find the minimum non-zero value point captured from robot's laser scanner. We then use its laser index to set the heading of the robot with respect to the wall, which is the angle from looking forward to that shortest point. Then we use this heading to make sure the direction of the robot is parallel to the wall by doing a proportional control on angular speed and linear speed respectively. The P control diagrams are shown above.

## Person following

Person following requires the robot to first identify the person closest to the robot and then follow the person regardless of the noise in the environment.

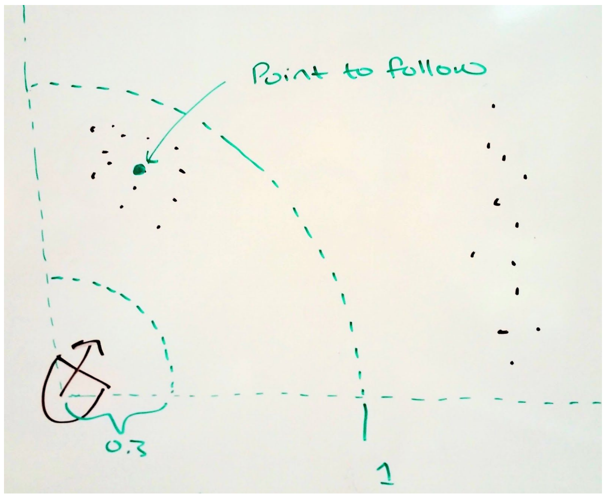

Figure 07: Range of the laser scan points that is used for further calculation

In order to filter out the noise in the environment, we filter the laser scan data so that we are only paying attention to the front area within certain range and angle. We use a naive algorithm to detect the center of the person by calculating the center of mass of the points in range. After that, we calculate the robot's distance and angle relative to the person. This distance and angle can be used to perform proportional control on the robot's linear and angular velocity to make the robot follow the person.

## Obstacle avoidance

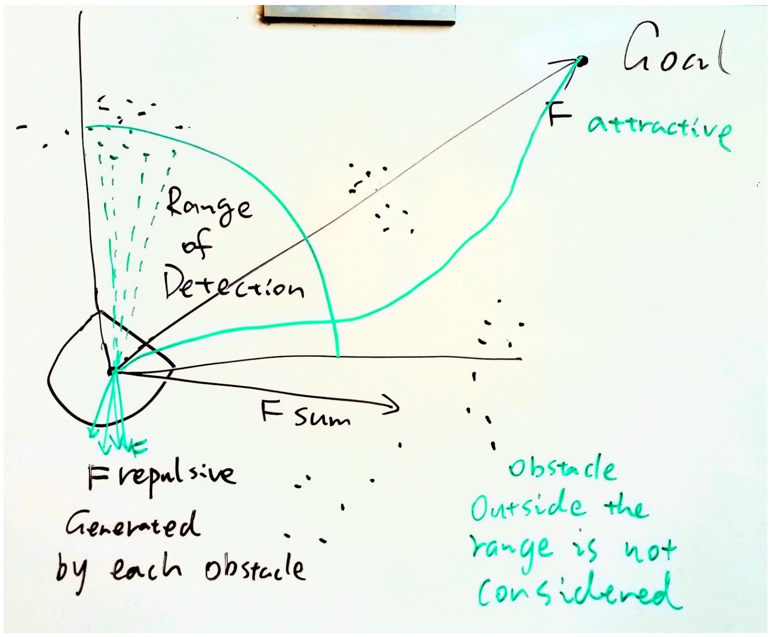
Goal
F attractive
Range of Detection
F sum
F repulsive Generated by each obstacle
obstacle Outside the range is not considered

Figure 08: Attractive and repulsive forces  of different points in the range

Obstacle avoidance requires the robot to avoid obstacles on its way to the goal. We categorize the forces acting on the robot to attractive forces and repulsive forces. Attractive force is produced by the goal we set in odom coordinate. The goal coordinate is then converted into robot's coordinate system to generate a relative goal to the robot. Since we want the attractive force to produce a stable force field when far away and reduce its force when robot is approaching the goal, we decided to normalize the attractive force by the *tanh* function over its magnitude, which is shown in the below diagram.

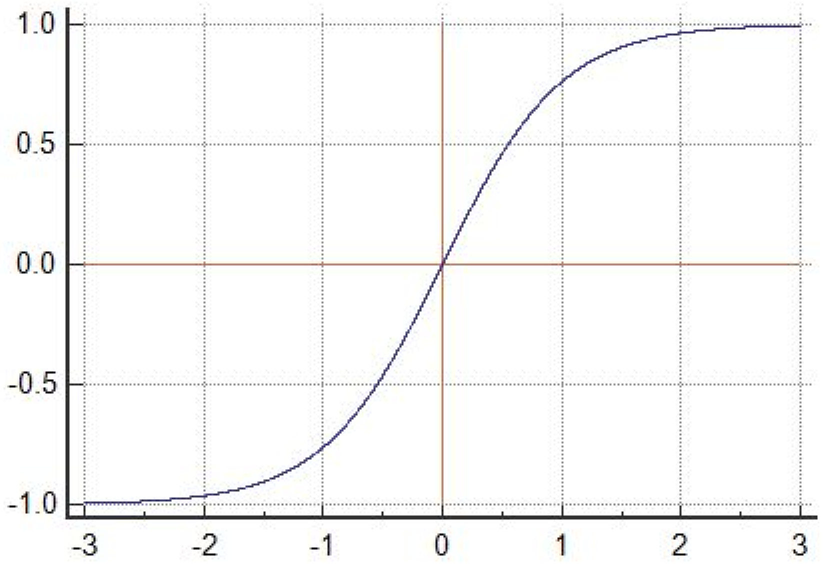

Figure 09: Change of attractive force with robot's distance away from the goal

After that, we calculate the repulsive forces. In our algorithm, each obstacle produces a small force vector. Then, we average all the repulsive forces to produce a sum force. Now we have both the attractive vector and repulsive vectors. We simply add them up to a sum force vector. That will then be converted to the robot's linear and angular speed.

## FSM

For the FSM activity, we need to incorporate two different robot tasks into a new task connected through a finite state machine. To accomplish this, we need to repurpose our existing tasks and build some transitions between the states.

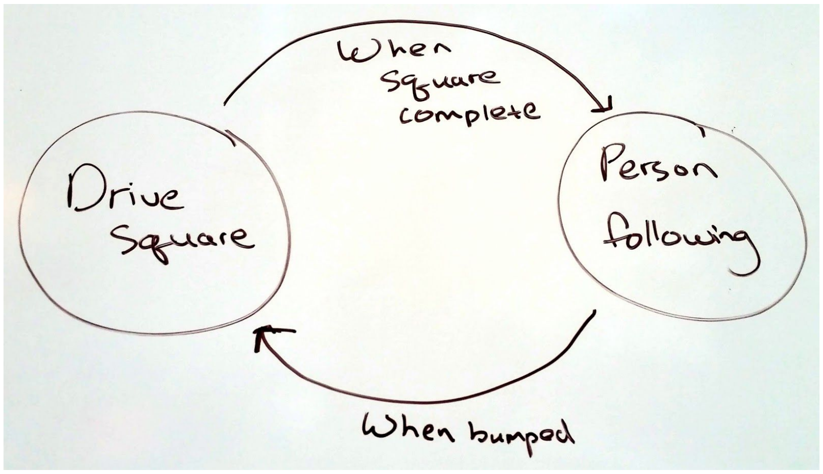

Figure 10: FSM of the robot behavior. This robot can person-follow or drive in a square. It goes into square driving mode after a bump is detected.

For our FSM behavior, the robot first follows you when you are in front. When the robot bumps into something, it shifts its state and starts to drive in a square. After completing the square, the robot is back in person-following mode, and it can start following your movements. While drawing a square, the robot will not follow you.

To tackle this problem, our strategy is to select two tasks and connect them through transitions. We decided to choose person following and drive square because they were our most stable implementations. To connect them, we decided to use another sensor - bump sensors. We retrofit the existing person-following code to detect bumping, and we exit this mode when a bump is detected. After the robot exits, it transitions into the Drive Square state. The robot then reverts back to person-following when the square is complete.

We implemented our state machine using Smach. Using two custom classes, we encompass each state and connect them through ‘outcomes’. Because this is a new file, we had to remove `rospy.init()` from our existing code to prevent conflict. When the other files could be imported, we encoded them into the classes to build the new robot behavior.

## Code structure

Throughout all of our robot behavior code, our code structure is fairly consistent. We implement a single class that contains our methods. The `run` method contains our run loop that normally publishes our events to the robot. There are also `on_…` methods that are listeners to events we subscribe on the robot. Inside of these event listeners, we set instance variables that we reference elsewhere in our code.

We also have several helper methods that we use to process the data from our event listeners. Usually, these are also called by the event listeners. The while loop runs independently of the event listener updates.

At the end of the file, we keep an `if __name__ == ‘__main__’`, where we `rospy.init()`, instantiate our controller, and execute our `run()`. In this format, we can also use these controllers in other, larger classes (such as our FSM implementation.)

## Challenges

One of our biggest challenges was to solve the problem of converting from robot’s coordinate system to odom. We had problems testing it and also had to redo our math calculations a few times.

Another recurring bug was the units in which the robot’s angle was being stored. We prefered degrees for visualizing and doing mental math. But, the python trigonometry functions expect the angle in radians. Although such a small bug, we kept running into it.

## Improvements

The current obstacle avoidance is not that accurate when the obstacle is right in front of the robot. Extending the laser scan range to include the points closer to robot doesn't improve the behavior. We could improve the existing code by incorporating the bump sensors. We could also add wiggles to make the robot move away from the obstacle. We could also try a completely different algorithm in order to make the obstacle detection better.

For person detection in the person-following behavior, we found the coordinates of the person by finding the center of mass of laser scan points within  a certain range. We could have detected the person differently by doing cluster point finding.

Overall, we only used proportional control. An improvement could be to add PID control instead of the simple P control.

## Key Takeaways

The key takeaways for us in this assignment is that we got ourselves comfortable with libraries in ROS. Specifically, we can create and run packages; take advantages of the marker object for debugging; get familiar with the data returned from odom, twist and scanner and etc. Additionally, we are more familiar with a general code structure for ROS. Also, during our teamworking, we figured that drawing things on board and listing helper function is pretty useful.