6.141/16.405 Robotics: Science and Systems

Laboratory Exercise 3: Obstacle Detection and Wall Following

In this laboratory exercise, you will be implementing a safety and corridor controller using the laser scan data from your RACECAR. This exercise can be divided into the following tasks:

1. Implement a simple obstacle detection using the laser scan data. This obstacle detector should be able to detect if there is an obstacle in front of the robot.
2. Implement a simple wall detector using the laser scan data. This wall detector should be able to detect the relative orientation and the distance of the robot to a wall to the right of the robot.
3. Implement a safety controller that stops the RACECAR when there is an obstacle right in front of the car.
4. Implement a PID or PD steering controller that adjusts the steering angle to follow a wall.

The figure below shows a simulation snapshot of the laser scan in one of the tunnel corridors. As shown in the figure, you should divide the laser scan into segments for wall detection and obstacle detection. For example, ± 30 degrees of the laser scan is used for detecting obstacles which are less than 2 meters away from the RACECAR. For wall detection, you should fit one straight line using laser scan data in the range 30 degrees to 135 degrees, following which you should find the relative distance of the car from the wall as well as the error in orientation.

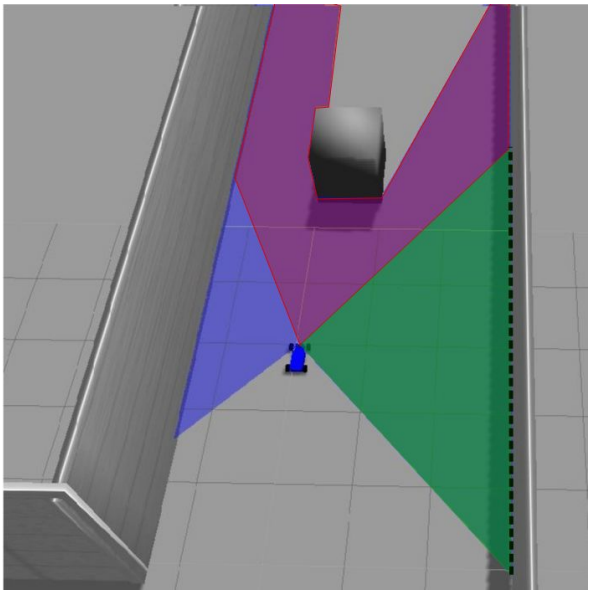

Figure 1: Laser scan simulation in a corridor

Update your workspace to pull in upstream TA changes

```
cd ~/racecar-ws/
# Pull in upstream changes
wstool update
# Install all dependencies for this workspace
rosdep install -r --from-paths src --rosdistro kinetic -y
catkin_make
```

ROS Node Architecture

As you may have noticed in previous labs, the RACECAR platform has a few topics that output commands to the VESC through a priority mux node. We recently changed it to make it more flexible - a node diagram of the latest current architecture can be found in figure 2.

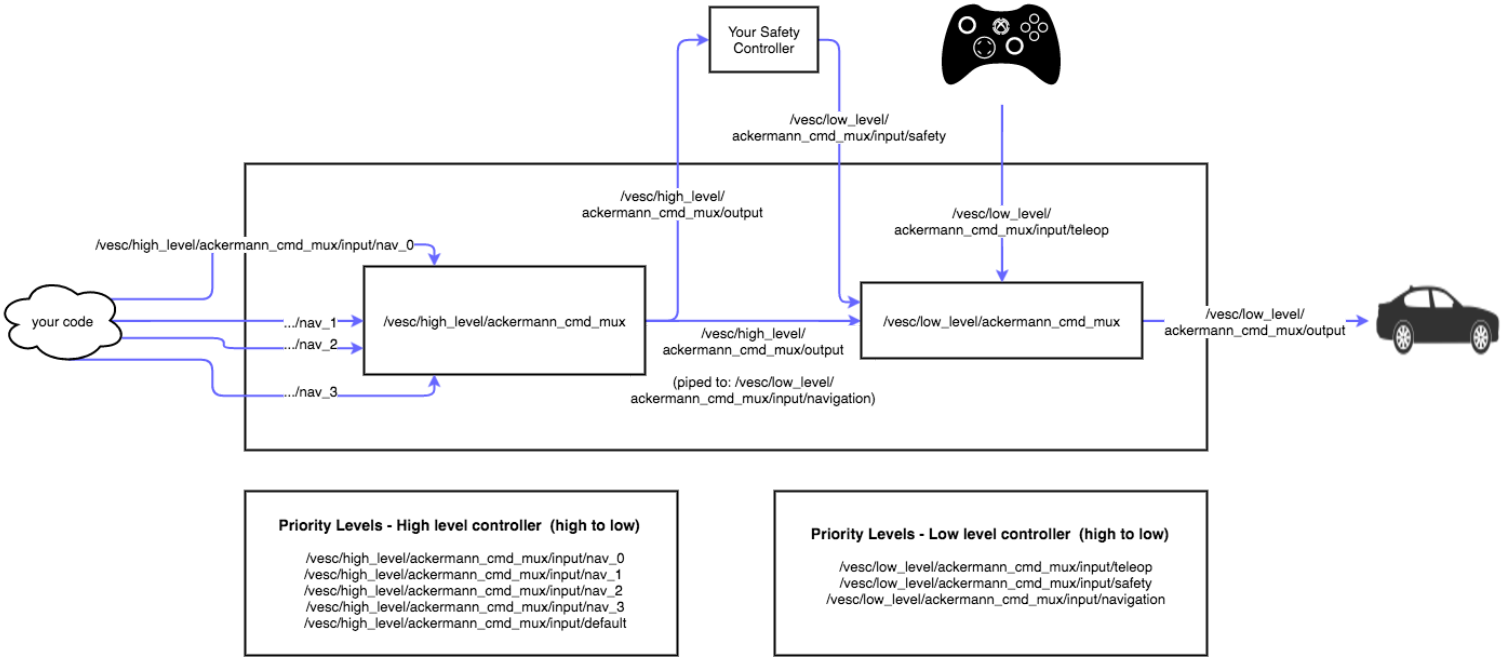

Figure 2: The mux architecture of the RACECAR platform.

Your autonomous *wall-following* code should use the `high_level` inputs, seen on the left side of the diagram. These will be switched in order of the priority listed and piped into the `/vesc/high_level/ackermann_cmd_mux/ output` of the high level mux node which goes into the navigation topic of the low_level input.

The low level mux is responsible for issuing commands to the car. It listens on the `/vesc/low_level/ackermann_cmd_mux/input/{teleop/safety/navigation}` topics and in order of priority issues those commands to the car. This allows the joystick

controller to override your safety controller, and your safety controller to override your autonomous navigation code.

There are two muxes (high and low level) that allow your safety controller to listen in on what messages your autonomous controller is sending and decide whether those controls would lead to disaster. While the easiest thing to do would be to stop the car if it's directly in front of a wall, doing so without eavesdropping on the higher level navigation messages would also prevent the car from (e.g.) backing away from the wall. Take a look at the diagram above to see how this could be implemented.

Backwards Compatibility

To remain compatible with the older topic names which you may be more familiar with, we have remapped the topics into the new system. The remapping is as follows:

```
/vesc/ackermann_cmd_mux/input/safety
      remapped to -> /vesc/low_level/ackermann_cmd_mux/input/safety
/vesc/ackermann_cmd_mux/input/teleop
      remapped to -> /vesc/low_level/ackermann_cmd_mux/input/teleop
/vesc/ackermann_cmd_mux/input/navigation
      remapped to -> /vesc/high_level/ackermann_cmd_mux/input/nav_0
```

Your code should not send commands directly to teleop as that would conflict with the joypad controller. Later in the class you may choose to do your own low level architecture, but don't worry about that until you have more experience with the car.

How muxing works in the RACECAR platform

The priority mux works by first figuring out which topics are actively being transmitted on, then picking the highest priority active topic to pipe directly through to the node's output. If no topics are currently active, an empty Ackermann command that stops the car is piped through to the VESC from the "default" channel.

For the mux to consider a topic to be active, it must receive messages at a frequency of 5Hz or faster on that topic. Thus, if a particular controller would like to hand off control to a lower-priority controller, the higher priority controller should simply stop publishing messages to the mux. **Piping messages any slower than 5Hz to the mux while a topic is supposed to be active will cause erratic behavior as the mux switches back and forth between input channels.**

Hints & Suggestions

A robust wall follower algorithm that can handle wavy wall contours and corners should probably use some sort of PID or PD control. Simple P (proportional) control is often not enough to create a responsive and stable system.

For PID control, some sort of line fitting is required to extract the angular and perpendicular error. A common method that you could try is least squares regression on a windowed angular "slice" of the laserscan data.

If you find that the presence of outliers in your laserscan data is causing your robot to have trouble tracking, line fitting using RANSAC can be pursued as a more robust alternative to least squares regression. However, using RANSAC is optional for this lab since we will not learn about RANSAC until later in the semester.

Another potential way of dealing with outliers before applying linear regression is the median filter - a filter which replaces each element in a list with the median element from a windowed region around that element. Scipy includes a good implementation of a median filter if you are interested.