Week 2 Lab Exercise 1: Driving Straight

The F1/10 Team

Introduction

This article contains the transcript of week 2 Lab exercise. This video walks you through the different structures of the various messages used and gives an outline on using the lidar data and implementing a simple control node.

Transcript of Video Tutorial

In this video we will walk through the skeleton code provided and the fields that have to be completed.

First lets see the implementation details for the node which determines the distance of the car from the wall. Download the week 2 workspace and open the template dist_finder.py in the src folder of the race package. Here you need to complete two functions., getRange and callback. This node subscribes to the Lidar scan data of type LaserScan[1] and publishes to pid error of custom data type called pid_input.

The LaserScan is a standard sensor_msgs datatype with various fields. The field ranges, which is an array consists of the distances in meters with first element being the distance at angle_min, the last element being the distance at angle_max and intermediate values at increments of angle_increment.

The pid_input message consist of 2 data elements. First the pid_error, or the error that needs to be compensated by the pid and pid_vel which the velocity the car should move at.

The callback is the function that is called when a new message arrives on the LaserScan topic. Lets fill this up. First step is to get 2 rays on the right side of the car to determine the distance of the car from the right wall and orientation with respect to it. We pick 2 rays at 0 degree and theta degree from the lidar scans.

Complete the getRange function that determines the distance of the wall at angle theta using the data. The various elements of the scan data can be accessed like a structure using the dot operator.

Using the equations provided in the tutorial implement the callback function in the space provided. Keep the speed of the car constant for now. Check the error by physically moving the car close to and away from the wall and at different orientations.

Next lets implement the PID node named as control.py in the same src folder. This node takes the error message of data type pid_input published by distance finder and publishes the drive parameters of custom message type drive_param. drive_param message type consists of 2 fields, angle specifying the steering angle between -100 to 100 and velocity specifying the throttle between -100 to 100.

The main function at start-up requests for kp, kd and velocity values. This makes it easier to tune the pid. control is the callback function that needs to be filled with the pid equations.

The variable servo_offset is used to trim the steering of the car to a center position if there is any mechanical misalignment.

Follow these 3 steps

1. First amplify the error by some suitable value.
2. Apply the PID equation to determine the correction value to the servo. Be sure to also consider the servo_offset.
3. Finally do a check to see if the steering angle is within bounds of -100 to +100.

Now you must be able to run your car by executing the following nodes.

```
$ roscore
$ rosrun hokuyo_node hokuyo_node
```

```
$ rosrun rosserial_python serial_node /dev/ttyACM0
$ rosrun race talker.py
$ rosrun race control.py
$ rosrun race dist_finder.py
```

I would encourage you to write a launch file[2] to ease this process.

References

[1] LaserScan message

[2] Launch file tutorial