F1/10 Autonomous Racing

Assignment 4 - CS4501/SYS4582

Madhur Behl (madhur.behl@virginia.edu)

Autonomous F1/10 Car: Wall Following

Due Date: April 24, 2019

Overview

As described during the lectures, if you were able to successfully drive your car using the keyboard commands from the remote machine, you can now easily transition into fully autonomous operation. What you need is the following:

- An algorithm to generate the correct steering (angle) and drive signal using LIDAR data (perception), and
- A way for the car to autonomously follow the provided reference values of the steering, and the drive (control)

In this assignment you will implement the **Perception** and the **Control** ROS nodes for autonomous operation of the car.

The aim of this assignment is to implement a simple wall following algorithm which maintains the car parallel to a wall in a corridor. It involves using the sensor data from LIDAR and implementing a PID controller for *tracking* the wall.

Before jumping into the code, let us first understand the wall tracking algorithm.

Wall following algorithm aka Perception

Lets go through a simple procedure to calculate the distance of the wall. If we can calculate the distance from the wall, we can compare it with a desired distance and hence calculate the deviation from the desired trajectory. The LIDAR scans from right to left corresponding to 0 to 180 degree with 90 degree being the front of the car as depicted in Figure 1

We pick 2 rays: one at `0 degrees` *and the other at* `theta` *degrees;* `theta` *being some angle between* `0` *to* `70` *degree.* as depicted in Figure 2

A good value for `theta` is 45, but you should experiment.

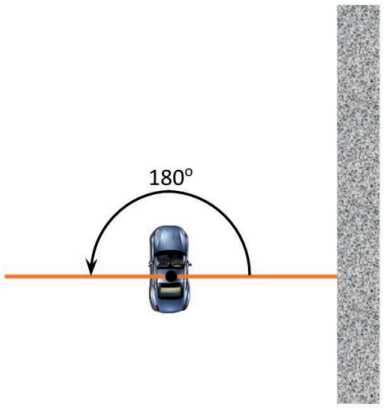


Figure 1: Lidar scan angles

Let `alpha` be the orientation of the car with respect to the wall. By solving the geometric problem in Figure 2, we can establish `alpha` as `tan` inverse of `[(a\*cos(theta)) - b/(a\*sin(theta))]`, and distance `AB` from the right wall as `[b\*cos(alpha)]`.

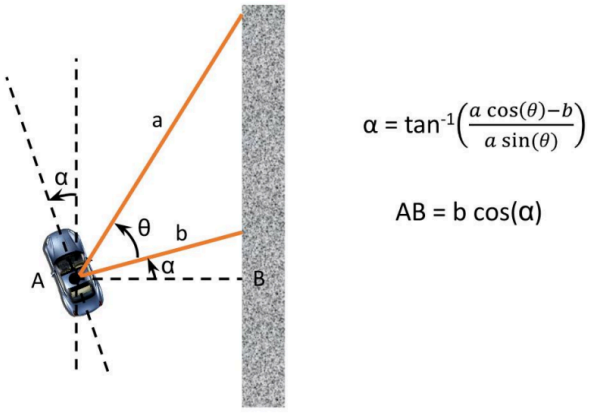


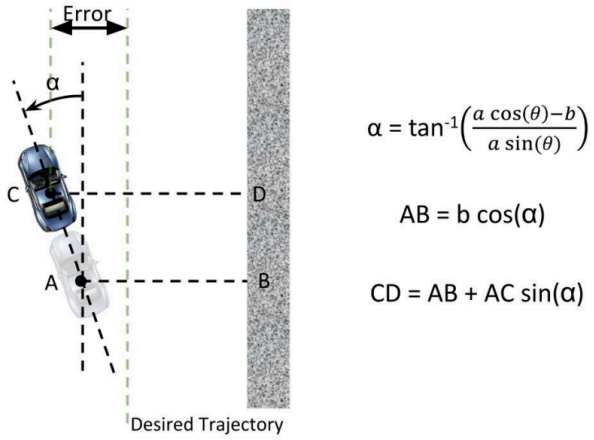
Figure 2: Calculating the orientation and distance from the wall

If the desired trajectory is say `0.5m` from the right wall then generally the error that has to controlled by the PID will be `[desired_distance - AB]` = `0.5- AB`

However, we cannot use this distance directly!

Due to the forward motion of the car and a finite delay in execution of the control and perception nodes; we instead, virtually project the car forward a certain distance from its current position. Hence now the distance of the car from the wall becomes `AB + AC*sin(alpha)` as shown in Figure 3.

Figure 3: Projecting the car future in time



Therefore the error to be compensated for is now the difference between the desired trajectory and CD. i.e. `[desired_distance - CD]`

We now have the error that we can use in the PID equation to determine the amount of correction to be applied to the steering angle.

PID steering controller

We use the standard PID equation, where `e(t)` denotes the error from the desired trajectory at time `t`. As explained in the lectures, it is sufficient to be using only `Kp (proportional)` and `Kd (derivative)` gains for the steering controller.

We will implement this as a tunable variable `Kp` times error plus another tunable variable `Kd` times the difference in error i.e.

$$V_\theta = K_p \times e(t) + K_d \frac{de(t)}{dt}$$
$$V_\theta = K_p \times error + K_d \times previous\ error\ -\ current\ error$$

We will use this value to increment or decrement the steering angle automatically based on the error. In the assignment you will be implementing the wall following algorithm and the PID controller in code.

Code description and download instructions

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

Your task in this assignment is to:

- Complete the `dist_finder.py` node.
- Complete the `control.py` node.

To begin the assignment, skeleton code is provided in the `race` package on the git repo. https://github.com/linklab-uva/f1tenth-course-labs

This is the same repository that you have used for previous assignments.

We will continue to use the same ROS package called `race` for this assignment, but with a few additional nodes.

By running `git pull` in your local `f1tenth-course-labs` folder, you can fetch the latest version of the repo which contains the same `race` package as used for Assignment 3, but a few new files are included in the `msg` and the `src` folder.

```
cd ~/github/f1tenth-course-labs
git pull
```

The updated contents of the race package folder are indicated below:

```
> race
    >> msg
        >>> drive_param.msg
        >>> drive_values.msg
        >>> pid_input.msg   <-- New!
    >> src
        >>> keyboard.py
        >>> talker.py
        >>> dist_finder.py  <-- New!
        >>> control.py      <-- New!
        >>> keyboard_new.py <-- New!
        >>> talker_new.py   <-- New!
    >> CMakeLists.txt
    >> package.xml
```

The updated code in this package uses three custom messages - `drive_param.msg` defines a pair of float values to define the desired throttle and steering (published by `keyboard.py`). And `drive_values.msg` custom message defines an int pair that is used to transmit the commanded PWM to the Teensy

After assignment 3, you are already familiar with these two custom messages.

The new custom message is called `pid_input.msg`

```
float32 pid_vel
float32 pid_error
```

It contains two floating value variables - `pid_vel`, and `pid_error`. You will see in a subsequent section how this plays into a new topic.

How to update the `race` package from your `VM` to the `Jetson`

Step 1: `ssh` into the `Jetson TK1`

As with assignment 3, make sure the Jetson and the Ubiquiti Wifi access point are powered on. The car need to be powered on.

```
ssh ubuntu@192.168.1.1
password: ubuntu
```

'ubuntu' is the default password. **Do NOT change this password**

Step 2: Ensure `catkin_ws` folder on the Jetson exists.

`You will likely skip this step, as the workspace already exists on the Jetson, following Assignment 3.`

In the same terminal as the ssh connector above. Follow the steps below to create a `catkin_ws` on the Jetson TK1.

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
cd ..
catkin_make
source ~/catkin_ws/devel/setup.bash
```

Step 3: Selectively move the updated `race` package to the Jetson

We will again use scp, to transfer the updated files in the race package to the Jetson

The reason why we synchronized our local git repository first is so that we can be selective about which files to `scp` to the board. Using scp, you can dictate which files you want to move, instead of complete folders. For instance, you may not want the template `talker.py` and template `keyboard.py` in the git folder to override your nodes (from Assignment 3) in the race package on the Jetson.

Use `scp` without the recursive `-r` option individually for new files:

- scp ~/github/f1tenth-course-labs/race/msg/pid*input.msg ubuntu@192.168.X.1:home/ubuntu/catkin*ws/src/race/msg/
- scp ~/github/f1tenth-course-labs/race/src/dist*finder.py ubuntu@192.168.X.1:home/ubuntu/catkin*ws/src/race/src/
- scp ~/github/f1tenth-course-labs/race/src/control.py ubuntu@192.168.X.1:home/ubuntu/catkin_ws/src/race/src/
- scp ~/github/f1tenth-course-labs/race/src/keyboard*new.py ubuntu@192.168.X.1:home/ubuntu/catkin*ws/src/race/src/
- scp ~/github/f1tenth-course-labs/race/src/talker*new.py ubuntu@192.168.X.1:home/ubuntu/catkin*ws/src/race/src/

If you try to build the packages on the Jetson right away, you may get errors since the `dist_finder.py` and `control.py` files are incomplete and will not compile.

The `keyboard_new.py` and `talker_new.py` are implementations of the solutions for Assignment 3. We encourage you to continue using the `keyboard.py` and `talker.py` nodes that you submitted for Assignment 3 but have provided the Assignment 3 solutions as `keyboard_new.py` and `talker_new.py` and you can use those if you wish.

Just, FYI, to transfer the entire contents of the `~/github/f1tenth-course-labs` folder to the `home/ubuntu/catkin_ws/src` folder on the Jetson, use the command below. However, exercise caution, as you may override your previously written files in the race package in the Jetson.

```
scp -r ~/github/f1tenth-course-labs ubuntu@192.168.X.1:home/ubuntu/catkin_ws/src
```

Code walkthrough and assignment description

Step 1: Perception: `dist_finder.py`

Lets see the implementation details for the node which determines the distance of the car from the wall. 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` topic which has a message type `LaserScan` and publishes to the topic `error` of custom message 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 two data elements. First the `pid_error` , or the error that needs to be compensated by the pid and `pid_vel` is the velocity the car should maintain.

The `callback` is the function that is called when a new message arrives on the `scan` topic. In order to complete this function - The first step is to pick two 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 (See the wall following explanation above - Figure 1)

Complete the `getRange` function that determines the distance of the wall at angle theta using the data received on the topic `scan` . The various elements of the `scan` data can be accessed like a structure using the dot operator. For example, in the `getRange(data, angle)` function. The distance reported by the LIDAR for the i'th ray will be given by:

```
dist = data.ranges[int(index)]
```

Where index = i;

Using the equations provided above in this assignment, 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. Remember that the error reported is between the LIDAR and the wall, and not necessarily the car and the wall.

```
Pro Tip: One way to debug your distance finder code and error is to only put the steering channel of the
in autonomous mode and keep the ESC channel in manual mode - to prevent your car from going rogue at the
speed.
```

Step 2: Control: `control.py`

Next lets implement the PID node named as `control.py` in the same src folder.

This node subscribes to the `error` topic, listening to messages of data type `pid_input` published by `dist_finder.py` . This node publishes to the `drive_parameters` topic using the custom message type `drive_param`

[Recall that these were getting published by the `keyboard.py` node from Assignment 3. We are replacing manual control with PID

control here].

`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. This should be very familiar to you based not he keyboard control exercise. In `control.py` we ought to ensure again that we pass a correct steering angle value (between -100,100) to `talker.py` which will then convert it to the correct PWM values.

You can use the saturation condition below:

```
if angle<-100:
        angle = -100
if angle>100:
        angle = 100
```

The main function at start-up requests for `kp` , `kd` and `vel_input` values. This makes it easier to tune the pid directly from the command line.

`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.

The following step may help:

- First, amplify the error by some suitable value. [say be 4, or 5 - similar to proportional gain]
- Apply the PID equation to determine the correction value to the angle. Be sure to consider any non-zero `servo_offset` .
- Perform a saturation/sanity check to see if the steering angle is within bounds of -100 to +100
- Construct the `drive_param` message, with the two fields `velocity` , and `angle` . Publish this message.

What you need to demo.

During the demo, you will run your car by executing the following nodes:

Start roscore: `roscore`

Start the LIDAR: `rosrun urg_node urg_node`

Start rosserial: `rosrun rosserial_python serial_node /dev/ttyACM0`

Start `talker.py` from the previous assignment: `rosrun race talker.py`

Start the PID controller: `rosrun race control.py` Provide appropriate kp, kd, ki, and velocity values.

Start the perception: `rosrun race dist_finder.py`

The car should run autonomously and follow the right hand wall.

Extra Credit

1) Launch file

Create a launch file `autonomous.launch` in the race package which launches **all** the above demo nodes in the correct order and accepts user inputs for kp, kd, and velocity.

2) Collision avoidance

Modify the exiting code, or write your own nodes to implement basic collision avoidance, i.e. if the LIDAR detects an obstacle right in front of it, it immediately stops the car.

3) Velocity PID

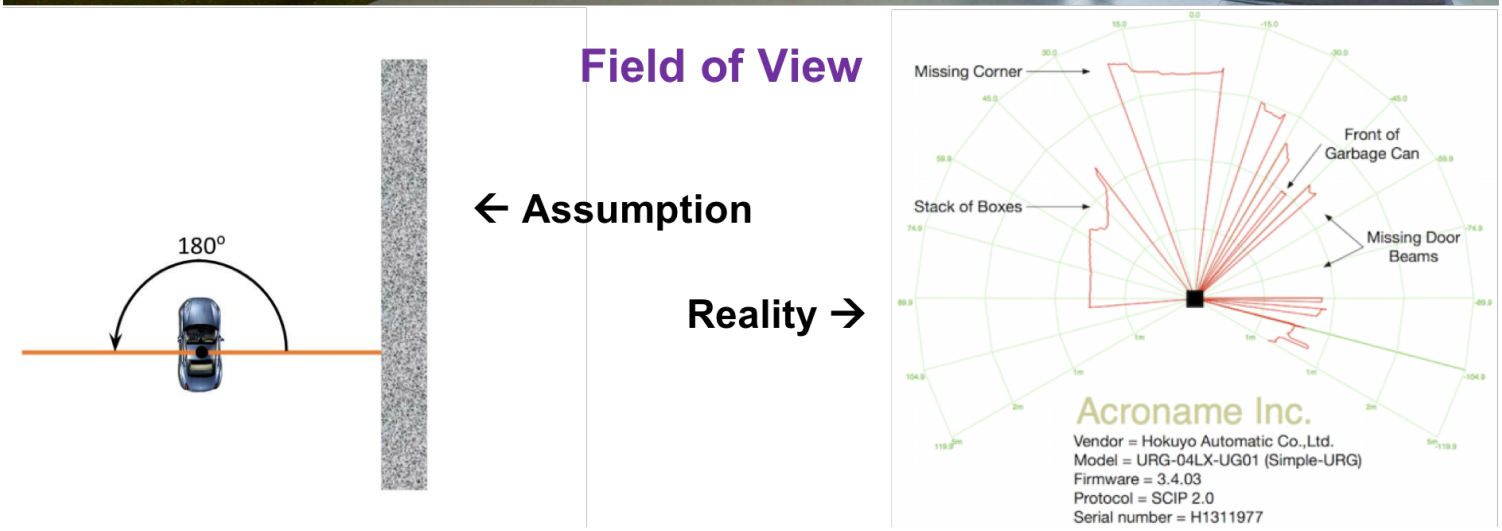
You will notice that in the assignment we are correcting the steering based on the error, but what about velocity ? It is bing held constant. Modify the `control.py` file so that velocity of the car also changes with error, i.e. on the straight parts of the track when the car is parallel to the wall, and error is low (or zero), the car drives at a higher velocity, but during the turns when the error is high, the velocity of the car reduces appropriately.

Debugging: Useful tips for Assignment 4.

[1] Picking LIDAR rays correctly:

One of the things required to determine the error is the measurement of the distance from the wall, or to be precise, the future distance from the wall (when the car is virtually projected forward in code).

While we assume in Figure 1 that the LIDAR scans from 0 degrees to 180 degrees, in reality the field of view of the LIDAR is larger than 180 degrees. The figure below shows a comparison of what we assume, vs what the LIDAR sees.



In reality the field of view (or the maximum angular range) of the 4LX is 240 degrees

The linear range of the LIDAR is from 20mm to 560mm, just for reference.

Which means that if we pick the ray, corresponding to zero degrees in our assumed FOV, it will correspond to the 30 degree ray in the LIDAR's FOV

There is a way to fix this directly by specifying the `angle_min` , and `angle_max` , parameter values for the Hokuyo (refer to documentation at http://wiki.ros.org/urg_node)

However, there is a easier way to account for the FOV of the LIDAR in your code directly.

The `Laserscan` message http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html has a field called `ranges` which stores the distance measurements.

We know that the angular range is 240 degrees, therefore for any angle `theta` , you can convert it to the correct ray index in the FOV of the LIDAR using:

```
index = theta * (len(data.ranges)/240)
```

Therefore, for the desired horizontal ray, which is 0 degrees in the assumed case, and 30 degrees in reality, the index of the distance measurement will be:

```
zero_ray_index = 30 * (len(data.ranges)/240) = 0.125 * (len(data.ranges)/240)
```

Similarly, the ray at 45 degrees from horizontal will be (45+30 =75) degrees in the fov of the LIDAR.

[2] Visualizing LIDAR data in `rviz`

The Jetson is running roscore.

But `rviz` will run on your remote machine/VM.

Here we are assuming that you have successfully `ssh` into the Jetson using the static IP configuration assigned to your team.

You need to now setup `ROS OVER NETWORK` so that the remote machine can connect to the `ROSMASTER` running on the Jetson.

To do so:

Step 1: Remote machine config

Open your `.bashrc` file Your `.bashrc` file is located in your user directory.

Add the following lines to the end of the `.bashrc` file on the remote machine/VM

```
# ROS network options for the remote machine or VM
export ROS_MASTER_URI=http://192.168.1.1:11311
```

Step 2: Jetson config

Open your `.bashrc` file on the Jetson. (After `ssh` connection)

Add the following lines to the end of the `.bashrc` file on the Jetson and save and exit. Now everytime you open a ssh terminal instance on the Jetson, the ROS OVER NETWORK will be correctly configured.

```
# ROS network options for Jetson
export ROS_MASTER_URI=http://tegra-ubuntu:11311/
export ROS_IP=192.168.1.1
```

Reverting back to normal behavior

If you just want to revert back to running tests on your remote machine/VM (like to run the ROS Tutorials etc), set these environment variables in the `.bashrc`:

```
export ROS_HOSTNAME=localhost
export ROS_MASTER_URI=http://localhost:11311
# export ROS_MASTER_URI=http://192.168.1.1:11311
```

Notice that we can comment out the `ROS_MASTER_URI` which points to the Jetson.

Upon successfully setting up the ROS over Network, you can ssh into the Jetson to start the `urg_node` to start publishing on the `scan` topic.

Then launch `rviz`

```
rosrun rviz rviz
```

In `rviz` , on the left panel, set /laser as Fixed Frame in Global Options. Then click on `Add` . In the dialog box set /scan as topic in LaserScan. You should now be able to see the LIDAR scan on your remote machine.