F1/10 Autonomous Racing

Assignment 2 - CS4501

Madhur Behl (madhur.behl@virginia.edu)

Autonomous F1/10 Car: Wall Following

Due Date: May 3, 2021 : Live Demo + Code Submission

Overview

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.

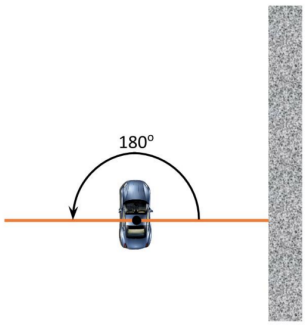


Figure 1: Lidar scan angles

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)]`.

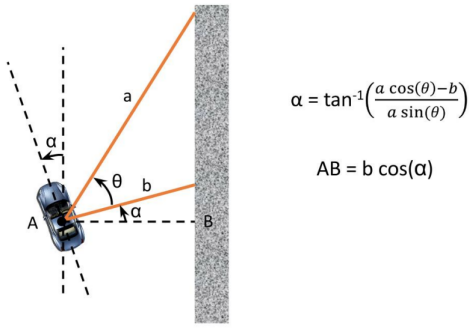


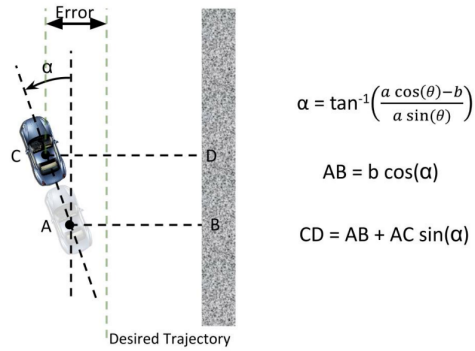
Figure 2: Calculating the orientation and distance from the wall

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 template 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

You will use a new ROS package called `race` for this assignment.

Clone the repo in your user accounts catkin_ws/src folder on the simulation server. (You need to ssh into the server with your team account first)

The contents of the race package folder are indicated below:

```
> race
    >> msg
        >>> drive_param.msg
        >>> drive_values.msg
        >>> pid_input.msg
    >> src
        >>> dist_finder.py
        >>> control.py
        >>> scan_test.py
    >> CMakeLists.txt
    >> package.xml
```

The template code in this package uses custom messages -
The custom message you will use for this assignment 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.

Remember that you need to ensure that the race package is present in your team's catkin_ws.

Step 1: `ssh` into the simulation server as your team.

Assuming that you were able to complete the simulator lab and able to run teleop on the sim, proceed with the following:

```
ssh team_name@rosmaster
password: <has been sent to each team>
```

Do NOT change your team password

Step 2: Clone the `race` package into the team's catkin_ws

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

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 `/team_name/scan` topic which has a message type `LaserScan` and publishes to the topic `/team_name/error` of custom message type called `pid_input`.

You must replace 'team_name' with your team's name; e.g. `/team_alpha/scan`

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 `/team_name/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 `/team_name/scan`.
The various elements of the `/team_name/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.

To help get started with this, a separate testing node called `scan_test.py` is provided in the package as well.

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 `/team_name/error` topic, listening to messages of data type `pid_input` published by `dist_finder.py`.
This node should publish to the `/team_name/multiplexer/command` topic using the message type `AckermannDrive` from the ackermann_msgs package.

You need to publish the steering angle between -100 to 100 and velocity specifying the throttle between -100 to 100.
In `control.py` we ought to ensure that we pass a correct steering angle value (between -100,100).

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. This does not matter for the simulated car.

The following step may help:

- First, amplify the error by some suitable value. [say be 4, or 5 - similar to proportional gain]
- 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 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 either the right or the left hand wall and complete 2 laps of the track.

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.

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.
