PID and Wall Following Lab

Niraj Basnet
basnetn@oregonstate.edu

April 15, 2020

1 Laplace Transform

Submit written answers for this part. Your submission can be PDF, Word, or scanned handwritten solutions (as long as they're legible). Submit these on Canvas (not through the repo).
1) Prove the following transform theorem: Let $f : \Re \to \Re$ be a twice differentiable function s.t. $f(t) = 0 \, \forall \, t < 0$ and whose derivatives admit Laplace transforms, and let $F(s) = \mathscr{L}(f)$ be the Laplace transform of f. Then

$$\mathscr{L}(d^2f/dt^2) = s^2F(s) - sf(0) - f'(0)$$

2) Complete the missing steps, in the lecture, in the solution of the ODE

$$x'' + 3x' + 2x = 0, x(0) = a, x'(0) = b$$

3) In the lecture, we saw a plant with inertia load ($(Js^2)^{-1}$) controlled by a PD controller. Give the diagram of a system with inertia load with a PID controller, and derive its transfer function $H(s) = X(s)/R(s)$.

2 Wall-Following: An Application of PID

In this lab, you will implement a PID (**p**roportional **i**ntegral **d**erivative) controller to make the car drive parallel to the walls of a corridor at a fixed distance. At a high level, you will accomplish this by taking laser scan distances from the Hokuyo LiDAR(simulated LiDAR sensor), computing the required steering angle and speed (drive parameters), and publishing these to the VESC(mux node in simulator) to drive the car.
Note: Review the simulator by going through the Readme file in the racecar_simulator package

in the skeletons repo. You need to press Key 'n' on your keyboard to enable navigation mode or if you want manual keyboard control, press key 'k'.
Before starting this lab, review the lectures on PID and LiDAR to ensure you are familiar with the material.

2.1 Review of PID in the time domain

In the lecture we saw PID in the frequency domain, since that brings out most clearly the effects of the various gains and why we might want to add, say, a derivative term. Here we look at PID in the time domain, in relation to the task of wall-following.

A PID controller is a way to maintain certain parameters of a system around a specified set point. PID controllers are used in a variety of applications requiring closed-loop control, such as in the VESC speed controller on your car.

The general equation for a PID controller in the time domain, as discussed in lecture, is as follows:

$$u(t) = K_p\, e(t) + K_i \int_0^t e(t')\, dt' + K_d\, \frac{d}{dt}(e(t))$$

Here, K_p, K_i, and K_d are constants that determine how much weight each of the three components (proportional, integral, derivative) contribute to the control output $u(t)$. $u(t)$ in our case is the steering angle we want the car to drive at. The error term $e(t)$ is the difference between the set point and the parameter we want to maintain around that set point.

In the context of our car, the desired distance to the wall should be our set point for our controller, which means our error is the difference between the desired and actual distance to the wall. This raises an important question: how do we measure the distance to the wall, and at what point in time? One option would simply be to consider the distance to the right wall at the current time t (let's call it D_t). Let's consider a generic orientation of the car with respect to the right wall and suppose the angle between the car's x-axis and the wall is denoted by α. We will obtain two laser scans (distances) to the wall: one at an angle θ ($0 < \theta \leq 70$ degrees), and another at an angle of 0 degrees relative to the car's x-axis. Suppose these two laser scans return distances a and b, respectively.

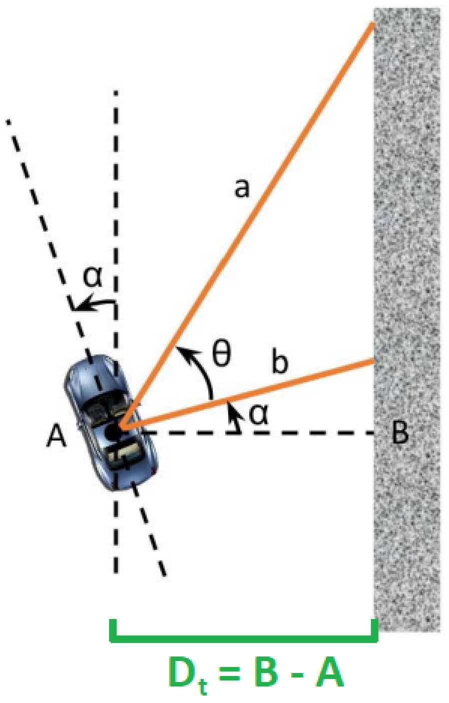

Figure 1: Distance and orientation of the car relative to the wall

Using the two distances a and b from the laser scan, the angle θ between the laser scans, and some trigonometry, we can express α as

$$\alpha = \tan^{-1}\left(\frac{a\cos(\theta) - b}{a\sin(\theta)}\right) \tag{1}$$

We can then express D_t as

$$D_t = b\cos(\alpha)$$

to get the current distance between the car and the right wall. What's our error term $e(t)$, then? It's simply the difference between the desired distance and actual distance! For example, if our desired distance is 1 meter from the wall, then $e(t)$ becomes $1 - D_t$.

However, we have a problem on our hands. Remember that this is a race: your car will be traveling at a high speed and therefore will have a non-instantaneous response to whatever speed and servo control you give to it. If we simply use the current distance to the wall, we might end up turning too late, and the car may crash. Therefore, we must look to the future and project the car ahead by a certain lookahead distance (let's call it L). Our new distance D_{t+1} will then be

$$D_{t+1} = D_t + L\sin(\alpha)$$

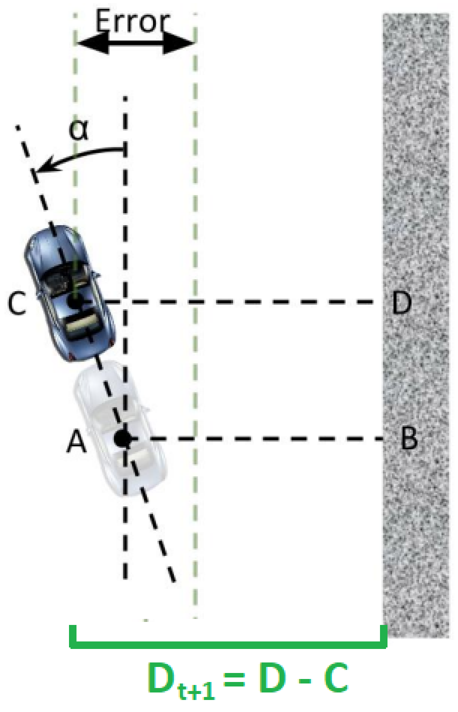

Figure 2: Finding the future distance from the car to the wall

We're almost there. Our control algorithm gives us a steering angle for the VESC, but we would also like to slow the car down around corners for safety. We can compute the speed in a step-like fashion based on the steering angle so that as the angle exceeds progressively larger amounts, the speed is cut in discrete increments. For this lab, we would like you to implement the following speed control algorithm:

- If the steering angle is between 0 degrees and 10 degrees, the car should drive at 1.5 meters per second.
- If the steering angle is between 10 degrees and 20 degrees, the speed should be 1.0 meters per second.
- Otherwise, the speed should be 0.5 meters per second.

So, in summary, here's what we need to do:

1. Obtain two laser scans (distances) a and b, with b taken at 0 degrees and a taken at an angle θ $(0 < \theta \leq 70)$
2. Use the distances a and b to calculate the angle α between the car's x-axis and the right wall.
3. Use α to find the current distance D_t to the car, and then α and D_t to find the estimated future distance D_{t+1} to the wall.

4. Run D_{t+1} through the PID algorithm described above to get a steering angle.
5. Use the steering angle you computed in the previous step to compute a safe driving speed.
6. Publish the steering angle and driving speed to the VESC(mux node in case of simulator).

3 Code

Open a terminal, navigate to the directory containing your course Git repository, and run

```
$ git pull origin master
```

to pull the starter code for the wall following lab. Copy the directory `labs/wall_following` into the `algorithms` folder of your ROS workspace and run

```
$ catkin_make
```

in the root of your workspace to build the packages. You can run the code using

```
$ roslaunch wall_following wall_following_sim.launch
```

There are two files you will be editing for this lab (both under `src/algorithms/wall_following/scripts`): `pid_error.py` and `control.py`.

- `pid_error.py` is where you will compute the error between the desired distance and the distance from the car to the wall.
- `control.py` is where you will take the error you computed and run it through the PID algorithm to get a steering angle and speed, which you will then publish to the VESC(mux node in case of simulator).

Make sure you do not change any code outside of these two files!

3.1 Using the Hokuyo LIDAR to find distances to the wall

In `pid_error.py`, complete the function `getRange(data, angle)`, which, when given a scan of the LIDAR (i.e., a `LaserScan` message) and an angle, returns the distance measured at that angle.
Some tips and important things to note:

- "0 degrees" for the LIDAR is defined as 45 degrees into the sweep. The sweeps are done in a counterclockwise fashion.
- Remember that some values in the data will be above the LIDAR's maximum range.What should your function do in such cases? See how the return value of the function is being used.

3.2 Computing the PID error for right, left, and center wall following

Now that we can find distances to the wall using the LIDAR, we can use these and the equations we derived in the background section to compute the angle α between the car and wall and the projected future distance D_{t+1} from the car to the wall.

Complete the functions `followRight`, `followLeft`, and `followCenter` in `pid_error.py` using the `getRange` function you wrote earlier and the equations derived above. The names are self-explanatory: `followRight` should make the car follow the right wall, `followLeft` should cause it follow the left wall, and `followCenter` should make the car drive between two walls at an equal distance.
For this part, determine appropriate values for the lookahead distance L and the desired distance, and use those values along with the formulas for α and D_{t+1} to compute the PID error $e(t)$. (What should the error be for `followCenter`?) For all three `followX` functions, a positive error should cause the car to steer right, and a negative error should cause it to steer left.

Once you've computed the error, assign it to `msg.data` in the callback function at the bottom of `pid_error.py` to publish it over the topic `pid_error`. This will allow your controller (which you'll implement next) to calculate the steering angle and drive speed.

Hints and tips:

- The `numpy` package (imported as `np` in `pid_error.py`) has trigonometric functions available to help you calculate the car's distance to the wall. Be mindful, however, that many of these functions operate in radians and not degrees.
- Adding `print` statements or using `pdb` (the Python debugger) is useful for verifying the correctness of your distance-measuring functions, as well as the angle α between the car and wall.
- You can see the PID errors you're computing in real-time (use `rostopic echo` on the appropriate topic).
- Use variables that are `const` for the lookahead distance, desired distance to the wall, K_p and K_d gains, and for the speeds corresponding to different steering angle ranges - avoid magical numbers! (Python doesn't really support `const` so this is more of a convention: an all-uppercase variable name above your `followXXX` functions that you can then reference in all of those functions; e.g. `LOOKAHEAD_DISTANCE = 2.5`. C/C++ does support `const`s of course.)

3.3 Implementing the PID controller and tuning the gains

In `control.py`, you'll implement the PID algorithm described above, taking in the error from the `pid_error` topic and running that through your algorithm to get a steering angle. As part of this, you'll also tune the gains of your PID controller so the car can achieve the set distance to the wall as quickly as possible and with minimal oscillation. (This is known as a "critically damped" response in control theory.)

Note that for this lab, we'll be implementing a PD controller instead of a full PID controller for simplicity, since implementing the integral term would require us to keep track of time. In practice, this means that you can leave the integral term out of the PID equation in your code.

Depending on your PID gains, you might find that the steering angle goes beyond 25 degrees, which is the limit to the steering angle on the VESC. You should cap the steering angle between -25 and +25 degrees.
Once you compute a steering angle, use it to compute a safe speed for the car as described in the background section above. With both the steering angle and speed in hand, you should publish them to the VESC over the `drive_parameters` topic using the skeleton code at the bottom of `control.py`.

When tuning your PD gains, tune the proportional term first until the car quickly approaches its set point, followed by the derivative term to reduce oscillation and smoothen out the response.

4 Submission and evaluation

Run your wall follower in the simulator on the Levine and Porto tracks and a third track of your choice. Choose the third track from the map files located inside the maps folder of the `racecar_simulator` package in the repository. Things to consider:

- Check the wall-following behavior in the levine track or other tracks. Make the car go around the track just by wall-following (so depending on whether you're going clockwise or ccw around the hallway, you'll be doing left or right following).Check right, left wall-following, and with center lane-following, running each of the three `followXXX` functions separately to ensure they work. Submit video for each of the three modes.
- Instead of always doing wall-following at constant speed, adjust the speed depending on what the car is doing (straight, about to go left, going left, etc).Isolate this logic in a separate function for modularity.
- **Bonus:** instead of following at a constant distance from the wall, adjust that distance depending on what the car is doing (straight, about to go left, going left, etc). E.g. if the car is about to do a left turn, it's good, from a racing perspective, to hug the right wall on approach, turn left quick and accelerate. When driving straight, stay in the middle. Whatever algorithm you use for this bonus part, submit a writeup explaining it as well.
- All code should be in one package, submitted through the individual's repo (in a new branch for this lab, as usual. Please review submission instructions in the syllabus) **Note: This is an individual assignment.**

- You must also provide a ROS node which receives, from your controller, the computed distance from the car to the wall. This node then computes the running average absolute error, and the running maximum absolute error, over a given simulation. (Running average means the average of all values so far. Same for max.) The node publishes these two numbers to topic `wall_following_analysis`. Call the node `studentname_analysis.py` (or .cpp if you're using C++). Make that node part of the `studentname_wall_following` package.
- Create a launch file (which you can base on the `wall_following_sim.launch`) such that, when we run it, it calls everything that `wall_following_sim.launch` calls, in addition to your `studentname_analysis.py/cpp` node. The launch file also causes the third track to be loaded in the simulator.
- Create your own parameter files (like config.yaml) for storing all the necessary parameters like PID gains, left/right/center choice and all other values so that you can change them without editing your program.You can search for parameter server in `wiki.ros.org` to know about it more. We will also check your parameter files.
- Submit a write-up in which you justify:
  - how you compute the steering angle based on D_{t+1}.
  - your choice of look-ahead distance L
  - your tuning procedure for PID gains: what did you look for when choosing K_p, show us a graph/image demonstrating that you achieved desired goal. Similarly for K_d.
  - equation (1) above for α (i.e., prove it).

Note: The third lab is only upto this point. The following section is a continuation of this and is your fourth lab.So, submit only up-to this segment.

5 Wall Following With Explicit Instructions

Note:This is an team assignment for the lab in the fourth week and will be due in the fifth week.Check your syllabus for the info.
You will now run wall following on the car, but with explicit instructions like whether to follow left wall, right wall or center of the track.
The entire discussion like in the earlier wall following part still applies.
1) First order of business is to tune your wall-following PID gains on the car.For this, check that the commanded speed is reasonable so that you can experiment safely (you need to find where the speed is set).Follow the same procedure to set these PID gains when tuning the gains in simulation. Start with K_p, then K_i, then K_d.

2) Check the wall-following behavior in all three modes(left,right and center).

3) We now add something in the mix: as you noticed, the track(like levine map) has T-junctions or crossroads at which the car could go left or right or keep going straight. We want the ability to tell the car which direction to take. So the `wall_following` package in the repository contains file `instructions.csv`. It is formatted as follows:

```
direction1 velocity1
direction2 velocity2
etc
```

For e.g, two instructions('center 2' and 'left 1') means go straight at 2 m/s and then take the next left turn at 1m/s. And so on.
Here is the expected behavior of your car: start by driving down a straight stretch. When the car comes to a turn (at which it could go left, right, straight, or any combination of those), it fetches the next instruction from the instructions file, and executes it if applicable. E.g., if the car comes upon a left turn and the next instruction says 'Left 1.5', it executes that and discards it. On the other hand, if the instruction says 'Right 1.5' it doesn't execute it and keeps it intact in the list, since there is only left turn in the track. If the car comes to a T-junction where it could go either way, it decides what to do based on the instruction.

Note that you also need to detect whether a turn has been completed. A turn is not done instantaneously. So if you're executing an instruction to turn left, say, the car should finish that turn before detecting for the next opening and checking it against the next instruction. You have to create your own instructions based on the different cases you might encounter in T-junction or crossroads(4-way intersection). We will test your submission with our own instructions and see if it passes all the cases.
Note: You can also attempt and submit the bonus part of the earlier lab. Submit it with a writeup if you did something.
For this part, you can leverage your gap finder to detect the openings, as it will help you to discard or execute your instruction depending upon the turns encountered.
Use RViz to visualize the car's operation: plot the lidar scan, gap center (if doing gap-finding), indicate somehow that a turn has been detected (and whether it's left or right), and indicate that an instruction is going to be executed.

5.1 Submission and Evaluation

You will submit

- your `teamname_wall_following` package, and other packages you compiled from source. We will copy those into our clone of the skeleton repo, compile, and run the launch file. Please do not submit the whole workspace.
- A Word or PDF document explaining your algorithm for implementing wall-following with explicit instructions. It will highlight critical parameters and how you set them. You should present a cogent, coherent procedure for setting parameter values. I.e., you were looking to achieve a desired effect and tuned until you achieved it.

- the video

You will demonstrate, in this order:

1. Pure wall-following. we will request left or right and tell you where to start from, and you will run it. The Left/Right choice should be in a configuration file and take no more than 5 seconds to change. No re-compilation should be needed.
2. Pure wall-following at higher speed: we will request the max speed and left/right-wall follow. Max speed should be settable in a config file.
3. Wall-following with instructions. we will request the default mode (left or right-follow), the sequence of turns with corresponding speeds, and where to start from. Similarly here, 10 seconds to change the config and start the car. The choice of top speed, default mode (left/right follow), whether using an instructions file or not, should all be in the same config file that we will look at.
4. Throughout the whole thing, we will want to see things on Rviz!! No guessing what the car is seeing or trying to do. This includes: Lidar returns, gap and gap center if using, current driving mode (left/right-wall follow), output of `wall_following_analysis` node that you created, `turn_initiated` and `turn_completed` info, instruction currently executing if applicable.