CS 489/689 Autonomous Racing: Lab 2 Wall Following

Section 1001, Fall 2023

1 Lab Description

In this lab, groups will learn to use a (1) PID controller to (2) autonomously drive the vehicle while maintaining a wall following.

2 Learning Outcomes

Outcome 1 A tuned PID controller

Outcome 2 A functioning left-wall following implementation in the f1tenth simulator

3 Rubric

Implemented PID	40
Tuned PID	40
Project Archive	20
Total	100

4 Submission Instructions

A completed project is first demonstrated to the teaching assistant in both the simulator and running on the vehicle.

Project demonstrations will be held during the TA office hours on September 16th. They include implementations demonstrated in simulator and on vehicle. In the simulator, the vehicle will be set along the track on the *right side* of the Levine Hall map (the default map) with the vehcile facing North. Please see image 1 below for reference. The `wall_follow` node will be started and the vehicle will drive a complete lap around the hallway using a *left-wall following implementation*.

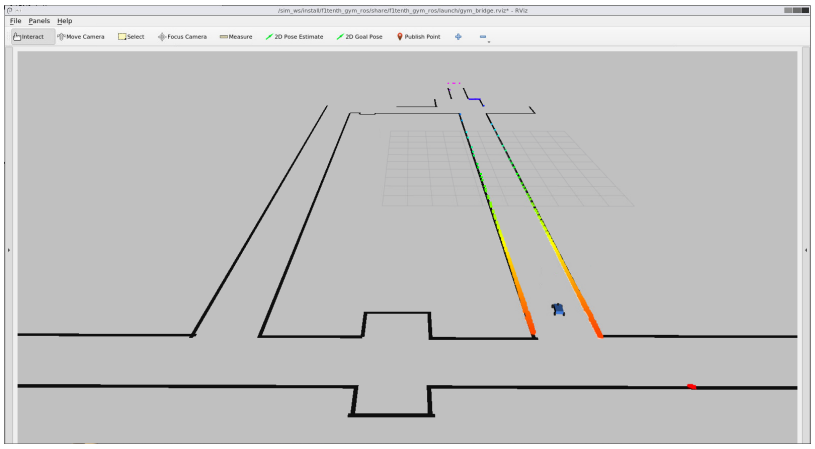

Figure 1: Starting Pose of the Vehicle for Simulation Enviornment

Be sure to implement the necessary changes to `wall_follow.py` to make it integrate with the vehicle before submission, these are listed in Section 5.6. It is expected for students to have tested their code on vehicle, before submission, to ensure it runs properly.

Submissions of the `wall_follow` package to canvas will be accepted after a successful demonstration. Afterwards, the docker image is exported for later inspection.

4.1 Demonstration

Before deploying to the vehicle, the wall following implementation must be successfully demonstrated within the simulation environment.

A satisfactory demonstration includes the following:

- The vehicle autonomously drives a lap, following the left wall, around the track without collision.
- The vehicle can drive with imperceptible oscillations after turning the first corner.
- The vehicle can successfully make it past the trap in the bottom of the simulator map.

In presentation of the simulator, start your `wall_follow` node in an adjacent terminal to the simulator preview. Have the node output a message to terminal indicating that the node has successfully been started. *Note: Continual terminal output delays node processing and may impact the correct operation of the vehicle. Provide terminal output for the successful launching of the node and no more.*

5 Lab Description

The examples provided in this lab concern a right-wall following algorithm. Translations to a left-wall following algorithm will be mandatory to recieve full credit for submissions.

5.1 PID Controller

The following PID controller equation will be used to calculate the vehicle's `steering_angle`:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\,dt + K_d \frac{d}{dt}(e(t))$$

Where:

$u(t)$ The output to the controller. $u(t)$ is the `steering_angle` in radians. *Note: A negative angle will turn the vehicle to the right, positive to the left.*

$e(t)$ The error at time t. This will be calculated as the difference between the desired distance from the wall and projected distance at time $t + 1$.

K_p The proportional control weight. Keeps the error proportional to the difference (error) of the distance.

K_d The derivative control weight. Helps identify a trend within the error and seeks to rectify it.

K_i The integral control weight. This term will help to account for the steady-state error accumulated over time.

5.1.1 Calculating the Error Term

Figure 2 below is meant for a right-wall following implementation. It shows how to find the distance and orientation of the vehicle relative to the right wall. This image is to be used as reference, and will be modified to fit the purposes of this lab: to follow the *left* wall in the map provided in simulator.

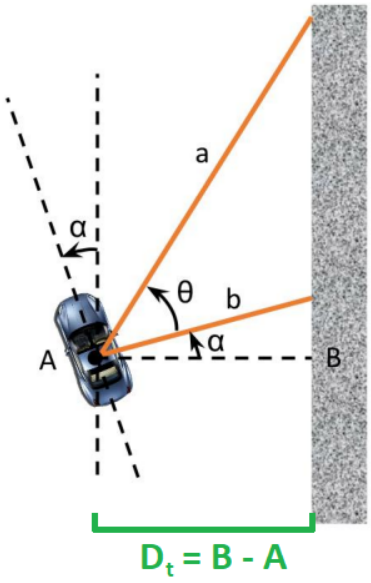

Figure 2: Distance and orientation of the vehicle, relative to the right wall

$$e(t) = D - D_t$$

The error term, $e(t)$, will report the instantaneous error. This is the difference between the desired wall distance and the vehicle's actual distance at time t. The desired distance from the wall, D, is a tunable value that each group may select from the range $[.5, 2]$ meters. The vehicle's "actual distance" is calculated from LiDAR scan data. It is the distance from the vehicle to the wall at a specific time t denoted D_t.

To calculate D_t, two distances measured at known angles from the vehicle are used. In the above figure, these are distances a and b. The angle for the perpendicular distance b is fixed, the angle of point in front of the vehicle, a, is a tunable value between [0, -70] degrees.

Using these two distances, and the angle θ between them, one can express an angle α that is the angle of offset from the desired angle, expressed by the following equation:

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

The actual distance, D_t, is then:

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

Calculating the error at time t is the instantaneous error (the current mismatch between the desired distance and actual distance). However, the vehicle is moving and the error cannot be corrected without advancing forward in time and distance. The goal is to reduce the error at the *next* timestep. To decrease the error at the next timestep the vehicle's current trajectory is projected by one timestep $t + 1$, shown in Figure 3 below.

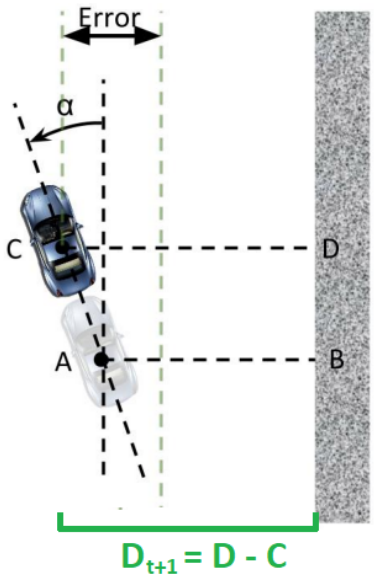

Figure 3: Predicting the future distance from the vehicle to the wall

With this projected distance, D_{t+1}, the `steering_angle` (control output) is adjusted to correct the vehicle's predicted distance from the wall. In the following equation, a look-ahead term L is a tunable value:

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

Now this term can be used to calculate the projected error by subtracting D_{t+1} from D. The projected error can be calculated by:

$$e(t) = D - D_{t+1}$$

Thus, the error function for use in this project is the projected error, not the instantaneous error.

5.1.2 Determining an acceptable speed based on the error

It is recommended to keep the velocity of the vehicle fixed while testing and in preliminary tuning, as the error of the system will increase with the speed of the vehicle.

The error of the system will also increase in a turn, due to the physical properties of the system (traction). This is best seen when a vehicle is rounding corners or needs to maneuver in precise detail to escape a trap (see Section 5.5 for more details).

Together, at higher speeds and maneuvering around corners, the error will be *amplified*. To account for this, a slower speed should be set. However, this will limit the speed even in the straights of the halls, and, in a racing enviornment, this is expected.

Therefore, the speed of the vehicle should be reduced only when necessary. To do so, adaptive speed control is recommended. The output from $u(t)$ will influence the speed of the vehicle. The following speeds are recommended for testing:

Note: These suggestions are ***one*** *possible method of adaptive speed control. There are many options, that may improve performance (speed).*

- If the `steering_angle` is between 0 degrees and 10 degrees, the vehicle 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.

The `steering_angle` can be found as a message from the `AckermannDrive` topic. The angle is reported in radians.

5.1.3 Deciding on a desired distance

In the simulator, there is a measurement tool that will calculate the distance between two points. This distance can be found at the bottom left of the simulator screen once the tool has been placed.

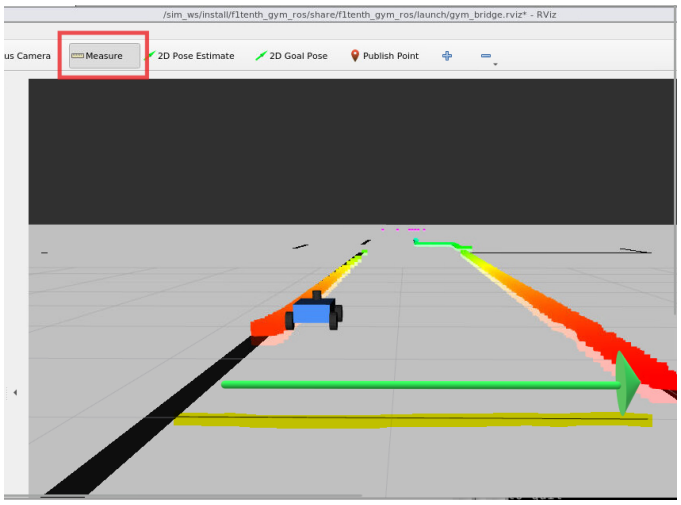

Figure 4: Measurement Tool highlighted in yellow

Figure 5: Example of a length measurement between track

5.2 Wall Following

This subsection is based upon the F1TENTH Lab on Wall Following: https://github.com/f1tenth/f1tenth_labs_openrepo/tree/main/f1tenth_lab3. Begin by cloning the F1TENTH Open Repository outside of the F1TENTH Gym container directory created earlier.

Within the open repository is a skeleton package that will be the basis of your submission. Copy the skeleton into the same directory containing the gym workspace.

```
gordon@f1sim:~$ cd ws/gym-one/
gordon@f1sim:~/ws/gym-one$ cp -r ~/git/f1tenth_labs_openrepo/f1tenth_lab3/wall_follow .
gordon@f1sim:~/ws/gym-one$ ls
f1tenth_gym_ros  safety_node  wall_follow
```

Don't forget to stop docker composition and edit the `docker-compose.yaml` file of the composition, include a volume for the newly copied `wall_follow` directory.

```
gordon@f1sim:~/ws/gym-one/f1tenth_gym_ros$ emacs -nw docker-compose.yml
>>> SNIP <<<
    volumes:
      - .:/sim_ws/src/f1tenth_gym_ros
      # Add the next entry
      - /home/gordon/ws/gym-one/wall_follow/:/sim_ws/src/wall_follow
>>> SNIP <<<
```

Restart the docker and re-enter the simulator container before trying to build the the `wall_follow` package.

5.2.1 Subscribers and Publishers

Note the following topic names for your publishers and subscribers:

`LaserScan: /scan`

`AckermannDriveStamped: /drive`, specifically, `drive.steering_angle` and `drive.speed`

5.2.2 Wall Follow Node Template

```python
import rclpy
from rclpy.node import Node

import numpy as np
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped

class WallFollow(Node):
    """
    Implement Wall Following on the car
    """
    def __init__(self):
        super().__init__('wall_follow_node')

        lidarscan_topic = '/scan'
        drive_topic = '/drive'

        # TODO: create subscribers and publishers

        # TODO: set PID gains
        # self.kp =
        # self.kd =
        # self.ki =

        # TODO: store history
        # self.integral =
        # self.prev_error =
        # self.error =

        # TODO: store any necessary values you think you'll need

    def get_range(self, range_data, angle):
        """
        Simple helper to return the corresponding range measurement at a given angle. Make sure you take care of NaNs and infs.

        Args:
            range_data: single range array from the LiDAR
            angle: between angle_min and angle_max of the LiDAR

        Returns:
            range: range measurement in meters at the given angle

        """

        #TODO: implement
        return 0.0

    def get_error(self, range_data, dist):
        """
        Calculates the error to the wall. Follow the wall to the left (going counter clockwise in the Levine loop). You potentially will need to use get_range()

        Args:
            range_data: single range array from the LiDAR
            dist: desired distance to the wall

        Returns:
```

```
            error: calculated error
        """

        #TODO:implement
        return 0.0

    def pid_control(self, error, velocity):
        """
        Based on the calculated error, publish vehicle control

        Args:
            error: calculated error
            velocity: desired velocity

        Returns:
            None
        """
        angle = 0.0
        # TODO: Use kp, ki & kd to implement a PID controller
        drive_msg = AckermannDriveStamped()
        # TODO: fill in drive message and publish

    def scan_callback(self, msg):
        """
        Callback function for LaserScan messages. Calculate the error and publish the drive message in this function.

        Args:
            msg: Incoming LaserScan message

        Returns:
            None
        """
        error = 0.0 # TODO: replace with error calculated by get_error()
        velocity = 0.0 # TODO: calculate desired car velocity based on error
        self.pid_control(error, velocity) # TODO: actuate the car with PID


def main(args=None):
    rclpy.init(args=args)
    print("WallFollow Initialized")
    wall_follow_node = WallFollow()
    rclpy.spin(wall_follow_node)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    wall_follow_node.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

scripts/wall_follow_node.py

5.3 Tuning your PID Weights

When tuning the PID weights for the controller, it's helpful to know what each term provides to the overall calculations. Please reference the document How to Tune a PID Controller below for a helpful guide on tuning each weight. This will give an idea of what each term does, and how it looks on vehicle.

5.3.1 K_p

$$K_p e(t)$$

The first weight that will need to be tuned is K_p, proportional weight. The equation presented above will push the output controller towards the desired distance from the wall proportionally to the error calculated.

This means if the error term, $e(t)$, has a greater magnitude, it will proportionally affect the output controller. When tuning this term, notice the tendancy of the vehicle to drift back and forth across the center-line (the line the desired distance away from the wall). Remember, if the proportional term calculated is smaller, then $u(t)$ will also be small. This means a small `steering_angle`, and thus a wider oscilation width around the center-line. Try to tune this so the vehicle has a very small breadth of oscillation.

5.3.2 K_d

$$K_d \frac{d}{dt}(e(t))$$

Next, the differential, or derivative, term will slow the approach from the proportional term by counter-inlfuencing its calculation. K_d holds the rate of change expected in the future projection of the vehicle. Therefore, if the error term is likely to change faster, the projected error should (upon a correct implementation) be expected to shrink significantly. This will create a larger control force on the proportional term.

The student's task will be to estimate the derivative using the following equation:

$$\frac{d}{dt} = \frac{e(t-1)-e(t)}{t-(t-1)}$$

Conceptually, the derivative is the difference in error separated by the amount of time between error calculations. Note, t is treated as in index not an absolute time value, your implementation may benefit from incorporating wall time.

5.3.3 K_i

$$K_i \int_0^t e(t)\ dt$$

The integral weight, K_i, is the last term to be tuned. The integral accounts for the steady-state error caused by K_p and K_d, the accumulated error over time. With a correctly tuned weight, the vehicle's oscillation will become imperceptible over time. This term will need to be tuned, and some time term will need to be stored.

5.4 Clamping

Clamping is used to control the calculated control output, $u(t)$, which is to be considered the `steering_angle`. While testing, it is possible this number will quickly get out of hand due to integral windup. Using a clamp will ensure $u(t)$ is always within an acceptable range.

The most the `steering_angle` can turn is by 20 degrees, in both directions. A negative `steering_angle` will turn your vehicle right, and a positive angle to the left. The `steering_angle` is stored under the `AckermannDrive` in radians. Be sure to keep this in mind while progressing through the lab.

5.5 The Trap

On the South facing side of the simulator map lays a rectangular irregularity, illustrated below by Figure 6. This portion of the map is a trap. It is used to test implementations and catch the vehicle within its confines. Due to the tight corners, it can be nearly impossible for the vehicle to maneuver out once it has entered.

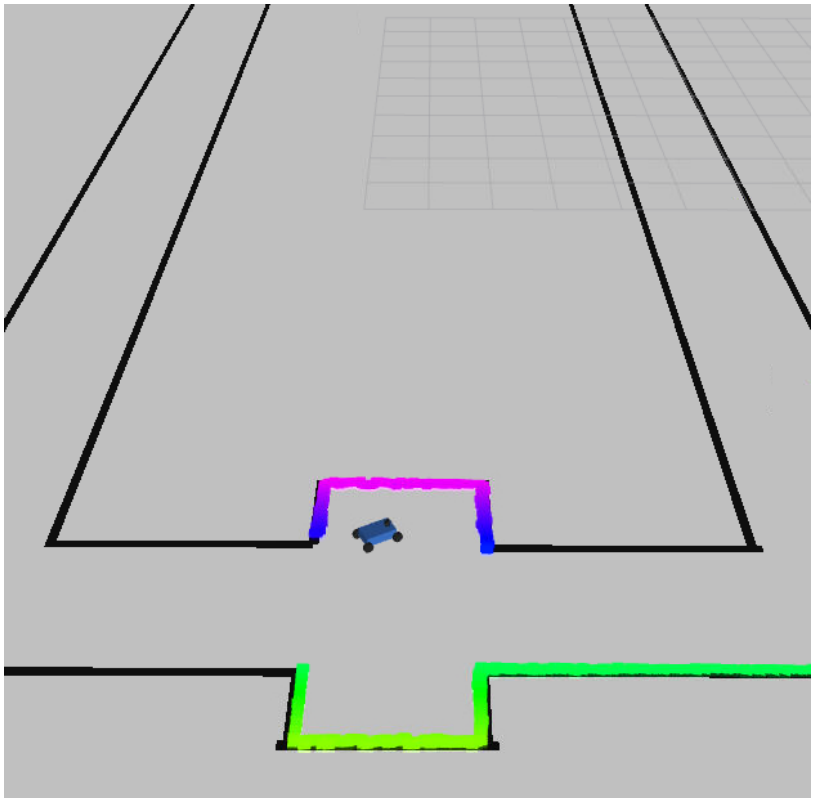

Figure 6: The Vehicle Caught in the Trap

5.6 On Vehicle Demonstration

The presentation on vehicle will be held in person on a track set up in the classroom. The vehicle will be expected to complete the following tasks:

- The vehicle autonomously drives a lap, following the left wall, around the track without collision.
- The vehicle can drive with imperceptible oscillations after turning the first corner.
- The vehicle can successfully make it past traps located on the track.

As the physical vehicle is a different model from simulator, the tuned PID weights will not carry over. This means the weights will need to be tuned in person, on vehicle, for a proper demonstration.

5.6.1 `mux.yaml` Prioritization

Previously, the vehicle's control priorities were modified to prioritize autonomous control over manual control. As witnessed in simulator, without the correct weights for the PID controller implemented, the vehicle may produce unwanted results, and unsafe conditions that require manual intervention. Thus, the `mux.yaml` file will once again need to be updated to allow manual override.

```
ackermann_mux:
  ros__parameters:
    topics:
      navigation:
        topic   : drive
        timeout : 5.0
        priority: 150
      joystick:
        topic   : teleop
```

```
        timeout : 0.2
        priority: 100
```

Listing 1: Previously Modified `/home/f1/racecar_ws/src/f1tenth_system/f1tenth_stack/config/mux.yaml`

```
ackermann_mux:
  ros__parameters:
    topics:
      navigation:
        topic   : drive
        timeout : 0.2
        priority: 10
      joystick:
        topic   : teleop
        timeout : 5.0
        priority: 100
```

Listing 2: Newly Modified `/home/f1/racecar_ws/src/f1tenth_system/f1tenth_stack/config/mux.yaml`

Do not forget to rebuild the racecar workspace to implement the changes to file.

Lab 2 Resources

F1TENTH Lab 3: Wall Following The F1TENTH group's description of Lab 3: Wall Following. Instruction URL: https://github.com/f1tenth/f1tenth_labs_openrepo/tree/main/f1tenth_lab3

`LaserScan` Message Quick link to the ROS `LaserScan` topic reference data. Instruction URL: http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/LaserScan.html

`AckermannDriveStamped` Message Quick link to the ROS AckermannDriveStamped topic reference data. Instruction URL: http://docs.ros.org/en/melodic/api/ackermann_msgs/html/msg/AckermannDrive.html

How to Tune a PID Controller An informational guide to help tune the PID weights. Instruction URL: https://pidexplained.com/how-to-tune-a-pid-controller/