Lab #3 Report: Wall Follower

Team #8

German Espinosa
Habeeb Salau
Izzie Torres
Angelina Zhang

6.141: RSS

March 5, 2022

1 Introduction [Habeeb]

Lab 3 was titled "Wall-Following on the Racecar". In the preceding lab, each team member individually developed a PID controller to allow a simulated racecar to follow along a wall. The team tested the simulated racecar varying speed, initial pose relative to the wall, and wall distance noise. The average error between the desired wall following distance and the distance of the robot in the simulation allowed the team to quantify how well each PID controller worked. The team took the best-performing code out of the simulation and applied it to a live racecar.

The transition from a virtual setting to a live racecar presented several challenges. A simulation is necessarily an approximation of the real world. The team had to determine how to tune the robot so that the simulated results were repeated with the unmodeled variance in the real world. Next, the team had to quickly learn to manage bugs and connectivity issues on hardware. The robot introduced hardware issues that were previously unseen in any of the previous labs that focused exclusively on simulations. Finally, the newly formed team had to quickly learn to work efficiently. The team presented lab 3 only two weeks after team formations which presented coordination challenges.

The goals for the lab follow closely with the presented challenges. The first goal was to establish team dynamics and working efficiency. This was important because the team will be working together for the remainder of the semester. A strong team dynamic allows each member to contribute and communicate effectively, write better code, and develop a better robot. The next goal was to

understand the robot and the hardware onboard. This involved understanding connection procedures, onboard hardware, electric safety, and data visualization. This was important because the team will be working with one robot for the semester. Understanding the robot well allows the team to better optimize for the robot. The next goal was to adapt the wall follower code to go from simulation to the actual racecar. This was important because it created a development process for testing in simulation and in real life. As the robotic tasks become more complex, this loop will become important to save engineering costs. The last goal was to implement a safety controlling module on the robot. The robot has several expensive sensors and computers and the team only gets one robot for the class. A safety controller allows the robot to safely stop when necessary to avoid damaging the car.

These challenges and goals informed the laboratory and our approach.

2 Technical Approach [Izzie, Angelina]

2.1 Initial Set-Up [Angelina]

Our team was presented with a car equipped with a lidar scanner, as well as a joystick controller. Given these tools, our goal was to enable our car to accurately detect its surroundings and safely drive a constant distance along a wall on a specified side (left or right).

In this section, we will describe the technical approach we devised in response to the problem, justify design choices, and explain how we integrated the different building blocks in our ROS implementation.

2.2 Technical Solution

2.2.1 Wall Follower: Lab 2 Code [Izzie]

At the beginning of this lab, the team decided to use the version of the Lab 2 code that received the highest score from the automatic grader as a starting point. This Lab 2 code is explained below.

The initial code importing all necessary libraries and message types is the same as in all other labs, and therefore trivial.

```
#!/usr/bin/env python2

import numpy as np
from scipy import stats

import rospy
from rospy.numpy_msg import numpy_msg
```

```
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped
from visualization_tools import *
```

Next, the WallFollower class is defined, and the necessary parameters are obtained and saved as variables.

```
class WallFollower:
    # Import ROS parameters from the "params.yaml" file.
    # Access these variables in class functions with self:
    # i.e. self.CONSTANT
    SCAN_TOPIC = rospy.get_param("wall_follower/scan_topic")
    DRIVE_TOPIC = rospy.get_param("wall_follower/drive_topic")
    SIDE = rospy.get_param("wall_follower/side")
    VELOCITY = rospy.get_param("wall_follower/velocity")
    DESIRED_DISTANCE = rospy.get_param("wall_follower/
        desired_distance")
```

Next, the instance is initialized and the publisher and subscriber are initialized.

```
    def __init__(self):
        self.pub = rospy.Publisher(self.DRIVE_TOPIC,
            AckermannDriveStamped,queue_size=10)
        rospy.Subscriber(self.SCAN_TOPIC,LaserScan,self.callback)
```

Next, in the callback function a list of all the angles corresponding to the ranges in the LaserScan data is created.

```
    def callback(self,LsrScn):
        # Create a list of what the angles are
        angles = np.arange(len(LsrScn.ranges))
        angles = angles * LsrScn.angle_increment
        angles = angles + LsrScn.angle_min
```

That list is used to determine which ranges are irrelevant for the wall follower code. Only the relevant ranges are saved in a new variable, and the angles list is updated to match.

```
    # Slice ranges to ...
    #...only look at one side ...
    relevant = np.where(self.SIDE*angles>0,1,0)
    new_len = np.sum(relevant)
    wall_obsv = np.array(LsrScn.ranges)[:new_len+1:self.SIDE*-1]
    #...and exclude behind the car
    in_front = np.where(self.SIDE*angles<np.pi*.6,1,0)
```

```
 final_slice  = -np.sum(in_front)
wall_obsv = wall_obsv[ final_slice :]
# Make angles match, reset both
angles = angles[:new_len+1:self.SIDE*-1]
angles = angles[ final_slice :]
angles = angles[:: self.SIDE*-1]
wall_obsv = wall_obsv[:: self.SIDE*-1]
```

The lists of ranges and angles are then converted to cartesian coordinates, where the origin is at the robot's location, and the positive y-axis is in the direction the robot is facing.

```
# Convert to cartesian coordinates
x = wall_obsv * np.cos(angles+np.pi)
y = wall_obsv * np.sin(angles+np.pi)
```

These x and y values are used to perform a linear regression, giving the parameters for a line that approximates where the wall is.

```
slope, intercept, r_value, p_value, std_err = stats.linregress(x,y)
```

For a moment, the code ignores the wall estimation line, and simply looks straight ahead, checking the distance to the wall ahead of the robot. This distance is compared to the distance of the closest wall to the robot, multiplied by the velocity. If the distance is smaller, that will be used as the distance input into the PD controller, causing the robot to start to turn the corner. Otherwise, the distance of the closest wall is used as the input to the PD controller. The multiplication by velocity allows the robot to start to make the turn sooner if it is going at a high speed, preventing crashes.

```
# If he approaches a corner, he thinks the wall is really close so he
    needs to turn a lot
if wall_obsv[ self.SIDE] <= min(wall_obsv)*self.VELOCITY:
  d = wall_obsv[self.SIDE]/self.VELOCITY
else:
  d = min(wall_obsv)
```

Once the distance d is set, the difference between d and the desired distance is calculated as the error, e.

```
# Calculate the error
e = d-self.DESIRED_DISTANCE
```

The error is then fed into the PD controller. The original "steer" variable attempts to turn the robot such that it is parallel to the wall, effectively acting as the derivative term. The $k_p * e$ term that is then added to the steer variable acts as the proportional term, trying to turn the robot such that it drives at the correct distance from the wall. Notably, the self.SIDE factor in the k_p variable reverses the sign, depending on what side the wall should be on.

```
# For just running parallel to the wall at any distance, steer
    according to the relative angle of the wall
steer = np.arctan(slope)
if abs(steer) > np.pi/3:
    steer = 0
else:
    k_p = self.SIDE*2.5
    steer = steer + k_p*e
    rospy.loginfo(d)
```

Now that the steering has been determined, an AckermannDriveStamped variable called levi is created, with the steering_angle set to the steer variable. This variable is then published.

```
# Publish the results!
levi  = AckermannDriveStamped()
levi.drive.steering_angle = steer #np.arctan(slope)
levi.drive. steering_angle_velocity = 0
levi.drive.speed = self.VELOCITY
levi.drive.acceleration = 0
levi.drive.jerk = 0
self.pub.publish(levi)
```

The final portion of the code initializes the node and creates the WallFollower instance.

```
if __name__ == "__main__":
    rospy.init_node('wall_follower')
    wall_follower = WallFollower()
    rospy.spin()
```

2.2.2 Wall Follower: Lab 3 Updates [Izzie]

The team made various changes to the code in order to account for errors with the LiDAR. The team first observed that the robot was constantly spinning in circles. After consulting the TAs, this error was determined to be due to dust on the LiDAR. However, wiping it did not solve the issue. Therefore, the team decided to filter out impossibly small range data in the following code:

```
# Filter out nonsense wall_obs values
 wall_obs_filtered  = wall_obsv[np.where(wall_obsv>0.2)]
```

Unfortunately, the team found that this did not fully solve the issue. The LiDAR was still detecting a point slightly behind the robot that was too far away to be filtered out, but closer than the distance to the wall. Since the code simply took the minimum of the selected range data as the distance to the wall, this single point of error significantly affected the robot's behavior. Thus, the code was changed to take an average of the three smallest range points, rather than just the smallest point.

```
# Calc distance from wall
if  wall_obs_filtered [ self.SIDE] <= 2: #close-ish to the wall
    d =  wall_obs_filtered [ self.SIDE]/2
else:
    d = np.nanmean(np.sort(wall_obs_filtered) [:3])  #avg of 3 smallest
        distances
```

Now, this was a quick fix implemented to get the code done in time. However, a more elegant solution would be to calculate the distance to the line generated from the earlier linear regression. This will be implemented for a future lab.

The final change from Lab 2 to Lab 3 was tuning the PD constants. This will be covered more extensively in the evaluation section, but where there was no k_d in the Lab 2 code (an implied k_d of 1), we added a variable for tuning in Lab 3.

2.2.3 Safety Controller [Angelina]

The primary purpose of the safety controller is to prevent the car from colliding with obstacles. While the car should stop if it is in danger of crashing, the safety controller should not be overprotective. In other words, the car should be able to drive fast, be close to walls, and turn corners without stopping. To accomplish this, our safety controller is only activated and overrides the default navigation when it detects that the car is at risk of crashing.

Our technical solution for the safety controller uses the pure pursuit model for a car-like robot. The goal is to calculate the minimum turning radius – the minimum distance our car needs, to complete a turn without hitting an obstacle. The pure-pursuit model takes in the following parameters: steering angle (δ) and "length" of the car (L). The maximum steering angle, δ_{max}, is 0.34 radians, as specified in the lab handout, and we measured the length of the car to be 0.33 meters.

After determining the parameters L, δ_{max}, we can calculate the minimum turning radius, R_{min}.

$$R_{min} = \frac{L}{tan(\delta)} = \frac{0.33}{tan(0.34)} = 0.9329 \tag{1}$$

The diagram on the following page also illustrates the calculations and trigonometric identities involved.

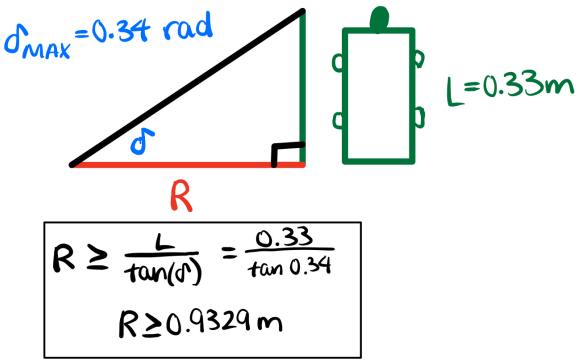

Figure 1: Pure-pursuit controller calculations to determine minimum turning radius

To summarize, the car detects that it is in danger of crashing when the look-ahead distance, the distance between the lidar scanner and the closest obstacle directly in front of it, is less than 0.9329 meters. We extracted the look-ahead distance as the middle element of the scan ranges (since they are sorted in increasing angle order), and checked if it was less than our minimum turning radius, 0.9329. The corresponding code snippet is attached below.

```
def lsr_callback ( self ,LsrScn):
  # crash distance is  directly  ahead (0 rad)
  num_scans=len(LsrScn.ranges)
  crash_distance  = LsrScn.ranges[num_scans//2]
  R_min = 0.33/np.tan(0.34) #0.9329, pure pursuit
  if  crash_distance  < R_min: # can't avoid
    self .speed = 0 # stop, override
   command = AckermannDriveStamped()
   command.drive.speed = self.speed
    self .pub.publish(command)
```

When the car is in such a position, it cannot drive forward, even while turning at its maximum steering angle, without colliding with the obstacle. Thus,

our safety controller overrides the wall follower's commands and sets the car's velocity to be 0 only when it detects this situation, preventing collisions when necessary. Otherwise, the safety controller is not activated and the car navigates as normal, under the wall follower's commands.

During the lab briefing, the teaching staff pointed out that the safety controller assumes the wall follower code is running. This will be an issue when the team runs different pieces of code in the future that behave differently. To address this, the team forced the robot to turn, at the maximum steering angle of 0.34 radians, if there is an obstacle within 1.5 meters. This code has not yet been tested, but it will be at the beginning of the next lab. The corresponding code block is attached below.

```
elif crash_distance < 1.5:
    self.steer = -.34*self.SIDE
    # Create and publish the variable
    command = AckermannDriveStamped()
    command.drive.steering_angle = -.34*self.SIDE
    self.pub.publish(command)
```

2.3 ROS Implementation [Angelina]

Our car utilizes ROS and drives autonomously through a series of Ackermann steering commands. We developed two main navigation modules: wall follower and safety controller. The wall follower subscribes to LaserScan data from the */scan* sensor and changes the steering angle by publishing Ackermann steering commands to the */navigation* topic. On the other hand, the safety controller subscribes to LaserScan data from the */scan* sensor and navigation commands from the */output* topic. It sets the car's velocity to 0 by publishing steering commands to the */safety* topic.

There is a hierarchy of topics published that the racecar ultimately reads from. In total, there are four topics being published to: */output*, */teleop*, */safety*, */navigation*. The controller joystick publishes steering commands to */teleop* and has the highest priority; when activated, it will override all steering commands published to */safety* or */navigation*. The next highest priority comes from */safety*. When the car is in danger of crashing, it activates the safety controller and overrides commands published from */navigation*, as specified in the lab handout. The highest priority command is published to */output*, which the car follows. Figure 2 provides a visual depiction of the hierarchy of the topics as mentioned (we used a low level controller in this lab).

With this system of publishers and subscribers, our car is able to switch between different types of navigation: manual (joystick operation) and automatic (wall follower and safety controller). While driving autonomously, it also detects when it is in danger of colliding and intercepts commands as needed.

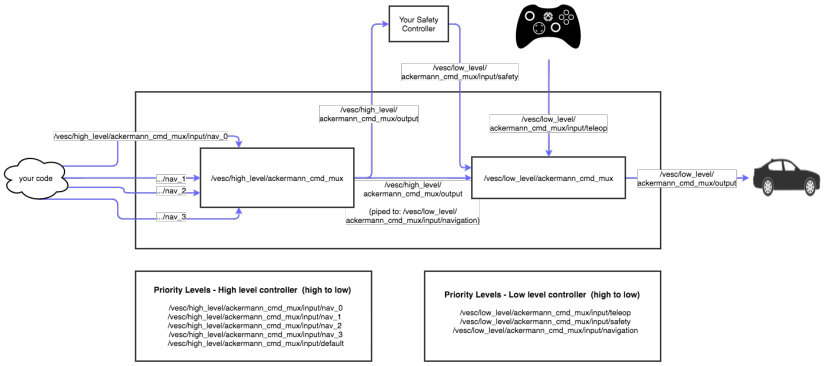

Figure 2: Command Mux with Different Levels of Priority (Lab 3 Handout)

3 Experimental Evaluation [German]

The transition from software to hardware is not always direct. The existence of real-world forces and frictions often push us away from the virtual conclusions we made. Therefore, it is important that we experiment with the robot itself to find the settings that make our car as efficient as possible. With regard to design, we had two problems to solve: writing a wall follower algorithm and developing a safety controller that minimized crashes. The following two sections summarize our experiments for each individual design objective:

3.1 Wall Follower

When discussing our wall follower implementation, there are primarily two parameters which we can adjust: proportional gain(K_p) and derivative gain(K_d). As mentioned in Section 2.2.1, we used $K_p = 2.5$ and $K_d = 1$ throughout our wall follower algorithm. However, this pair of gains was only optimal in simulation; therefore, we ran several tests while varying the gains in order to minimize error.

3.1.1 Defining Error

In order to accurately measure performance, we have to define a metric for our car. Given that our ultimate goal is to follow a wall at a specified distance(1 meter throughout our tests), we found that error should be defined as the difference between the car's current position and its desired position. Using ROS, we published the error to the /distance_error topic. Then we visualized the error as a function of time using rqt plot. In order to get an overall score for a test, we averaged the error over all samples. These calculations and plots can be seen in the following two figures:

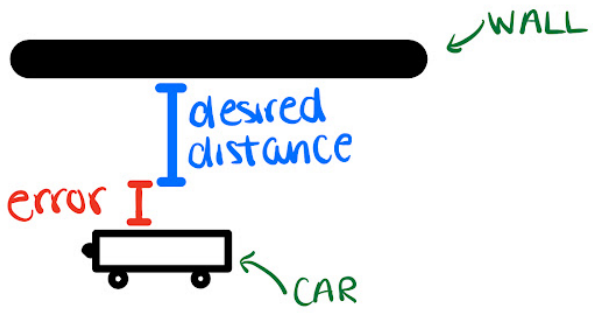

Figure 3: Visualization of error metric used throughout our experiments.

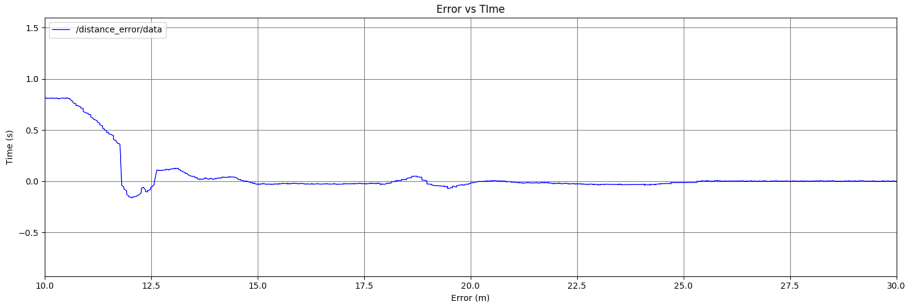

Figure 4: Example of rqt plot output demonstrating error versus time for Test 3.

3.1.2 Tests

Our tests covered all scenarios that our car would run into while driving. This includes variations in starting position, starting angles, and external objects such as a wall found in a corner. For each test, we varied the gains from 0 to 2.5 while keeping one constant at the value we found to be optimal in simulation. We kept velocity constant throughout the following tests.

- **Test 1:**
  Our first test explored the scenario in which our car was placed close to the wall (less than desired position) and angled towards the wall. This scenario is likely to happen when the car is driving quickly towards the desired position but overshoots and must adjust afterwards.

K_p	K_d	Average Error (m)
2.5	0	.236
2.5	.5	.225
2.5	1	.235
2.5	1.5	.195
2.5	2	.176
2.5	2.5	.198

K_p	K_d	Average Error (m)
0	1	.446
.5	1	.238
1	1	.202
1.5	1	.205
2	1	.195
2.5	1	.209

For test 1, the ideal pair of gains are: $K_p = 2$ and $K_d = 2$.

- **Test 2:**
  Our second test explored the scenario in which our car was placed close to the wall but facing away from the wall itself. This scenario would be expected as the car realizes it is too close to the wall and adjusts. Although this test is very similar to Test 1, the derivative component of our controller should smooth the transition to the desired whereas the proportional component dominated in the former.

K_p	K_d	Average Error (m)
2.5	0	.263
2.5	.5	.245
2.5	1	.157
2.5	1.5	.123
2.5	2	.120
2.5	2.5	.110

K_p	K_d	Average Error (m)
0	1	.302
.5	1	.133
1	1	.107
1.5	1	.100
2	1	.215
2.5	1	.183

As mentioned above, this test performs better when the derivative gain is weighted more heavily than the proportional gain: $K_p = 1.5$ and $K_d = 2.5$.

- **Test 3:**
  Our third test explores the scenario in which our car is placed far from the wall (greater than desired distance) and at an angle facing towards the wall. The expectation for this test is that the car should preform very

similar to how it did in Test 2. However, this test explores the car's ability to find walls at greater distances.

K_p	K_d	Average Error (m)
2.5	0	.221
2.5	.5	.134
2.5	1	.142
2.5	1.5	.154
2.5	2	.210
2.5	2.5	.269

K_p	K_d	Average Error (m)
0	1	.359
.5	1	.288
1	1	.193
1.5	1	.152
2	1	.205
2.5	1	.178

To our surprise, this test demonstrated trends that differed from Test 2. Instead of demanding a greater derivative gain, the car seemed content with a smaller value resulting in the proportional term dominating once again: $K_p = 1.5$ and $K_d = .5$.

- **Test 4:**
  Our fourth test explored the scenario where our car got placed far from the wall and angled facing away from the wall. Throughout our simulation, this particular test caused the most problems because the car might fail to find the nearby wall or confuse itself with another wall that is within reading range. Therefore, we expect this test to result in the largest error but converging, nonetheless.

K_p	K_d	Average Error (m)
2.5	0	.330
2.5	.5	.298
2.5	1	.307
2.5	1.5	.256
2.5	2	.336
2.5	2.5	.369

K_p	K_d	Average Error (m)
0	1	.609
.5	1	.400
1	1	.288
1.5	1	.256
2	1	.267
2.5	1	.241

As expected, we encountered the largest errors so far. However, these

values demonstrate a clear desire to have proportional gain dominate. If the derivative gain dominated, the car would be slow to change resulting in a false reading and forcing the car farther from the wall. Therefore, our optimal gains for this test are as follows: $K_p = 2.5$ and $K_d = 1.5$.

- **Test 5:**
  Our final wall follower validation test explored the car's ability to handle corners. This test is also expected to have a relatively large error since our car is never expected to follow the straight line path along the wall; instead, we would expect it to curve along the corner. Given that our car would ideally prevent a crash at all costs, we expect large proportional gains to result in high errors.

K_p	K_d	Average Error (m)
2.5	0	.203
2.5	.5	.211
2.5	1	.154
2.5	1.5	.142
2.5	2	.143
2.5	2.5	.160

K_p	K_d	Average Error (m)
0	1	.090
.5	1	.100
1	1	.141
1.5	1	.156
2	1	.161
2.5	1	.091

  As mentioned in our hypothesis, this test favored a larger derivative gain. This should not be surprising because an opposition to change would force the car to follow the wall closely and waiting till the very last moments to change direction. Nonetheless, we are more likely to crash at higher speeds with the low proportional gains found throughout the experiment: $K_p = 0$ and $K_d = 1.5$.

- **Test Conclusions:**
  For the sake of finding an overall pair of gains, we have decided to average the gains found throughout all tests to find our final optimal set:

$$K_p = .2 * 2 + .2 * 1.5 + .2 * 1.5 + .2 * 2.5 + .2 * 0 = 1.5 \tag{2}$$

$$K_d = .2 * 2 + .2 * 2.5 + .2 * .5 + .2 * 1.5 + .2 * 1.5 = 1.6 \tag{3}$$

3.2 Safety Controller Validation

In order to accurately measure the performance of our safety controller, we had to once again define an error metric. However, this metric would differ from the one used throughout our wall follower experimentation. Our code currently makes it so that our car looks ahead 2 meters and executes turning early on; therefore, increasing speeds along the corner would rarely result in a crash. We would only encounter a problem if our starting position was within 2 meters of the wall, but our safety controller would terminate all operations the moment it reads a wall .9329 meters ahead of us. Therefore, we chose to vary the starting distance from the wall and the car's velocity. The error metric used in the previous example would be uninformative for this test which is why we decided to use a simple binary (pass/fail) metric to gauge the effectiveness of our controller. After executing 25 different tests with this metric, our safety controller was successful for every test resulting in a 100% effectiveness when riding along the wall. This result is unsurprising since our controller uses the minimum turning radius as the stopping distance. In addition, the code itself is built to think ahead when encountering obstacles; therefore, we thought it would also be beneficial to see the cars response to dynamic obstacles placed within .9329 meters of the car.

Throughout our short distance obstacle test, we varied the amount of space our car was able to react to. In addition, we increased the speed of our vehicle in order to test the car's reaction time. A value of 1 means the car did not run into the obstacle. The results are as follows:

- **Speed = 1**

Obstacle Distance (m)	Pass/Fail
.9	1
.8	1
.7	1
.6	1
.5	1
.4	1
.3	1

- **Speed = 2**

Obstacle Distance (m)	Pass/Fail
.9	1
.8	1
.7	1
.6	1
.5	1
.4	1
.3	1

- **Speed = 3**

Obstacle Distance (m)	Pass/Fail
.9	1
.8	1
.7	1
.6	1
.5	1
.4	1
.3	0

- **Speed = 4**

Obstacle Distance (m)	Pass/Fail
.9	1
.8	1
.7	1
.6	1
.5	1
.4	1
.3	0

- **Results:**
  Overall, our safety controller was successful at stopping within a short amount of time. Looking at our results, our car is 100% responsive at distances greater than .3 meters. When within this range, the car has a problem stopping at higher speeds due to delay and other factors. These tests are an extra measure to gauge the performance of our safety controller but these scenarios are very unlikely since any dynamic obstacle (such as a person) would only be placed in the car's path with at least 2 meters of anticipation. Nonetheless, our overall effectiveness and error are as follows:

$$Effectiveness = successes/tests = 26/28 \approx .929 \tag{4}$$

$$Error = failures/tests = 2/28 \approx .071 \tag{5}$$

4 Conclusion [Habeeb]

The team found that the optimal gain values for the PID controller to follow the wall were when K_d was 1.6 and K_p was 1.5. This was determined by bringing the code from simulation to the robot and quantifying the mean distance error between the path of the robot and the desired distance.

The team also found that using a pure pursuit turning model for the safety controller resulted in a robust safety module. This was determined by testing the safety module with a variety of speeds, obstacles, and driving angles. These initial tests indicate that the robot can respond to obstacles without being too

timid and stopping unnecessarily.

While the initial goals of the lab were met, the team will be looking into tangential areas and problem spaces to improve the robot's performance. The current implementation calculates the robot's position to the wall by averaging the closest three points picked up in a filtered slice of the LIDAR scan. While the filtering helped to protect the system against outliers, the team will look into increasing the resilience of the position code. Potential distance metrics for investigation include using the y-intercept of the wall and point-to-line distance.

The current implementation of the safety module with pure pursuit assumes that the robot is traveling at speeds where the tires are not skidding. For the competition, the robot may face conditions where this may occur (eg. speed increases due to declines and varying coefficients of friction). In this case, the safety controller would need to factor in skidding and speed effects and stop earlier than currently programmed.

In addition, the current implementation for turning the car around a corner depends on a binary condition based on distance. The car performs a maximal turn once this binary condition is met. The team doesn't yet have a conceptual understanding of how the distances measured during the turn are impacting the turn radius. The next step here would be to plot out the distance measurements during sharp turns and develop a system that works well and is simple to model.

These results and future areas of investigation will be explored in future labs and briefings.

5 Lessons Learned

5.1 German

I've learned that the transition between software and hardware is not always smooth. In our first test run, our code worked well and was able to turn around corners and follow the wall; however, the robot started malfunctioning an hour later virtually out of nowhere. The team spent hours searching through the recent commits to find the bug, but nothing came up. Fortunately, this was a common problem and one of the assistants was able to quickly identify the issue: the lidar was dirty. This issue caused it to identify walls that were centimeters away from the car which forced it to always drive at the max steering angle. When working with hardware, the problem can literally come from anything and the software component is not always at fault.

With regard to interpersonal skills, I found that clearly communicating goals and expectations are crucial within a team. Since this lab had very little modularity, many of us found ourselves with nothing to do at times. This was

incredibly inefficient and I share the blame for not facilitating with this problem. For our next task, it is important that we communicate individual goals early on so that everyone feels like they have an equal share in the project.

5.2 Habeeb

Over the past week, we started learning how to work well together as a team. In our earlier two-hour lab sessions although we had effectively 8 man/woman-hours dedicated to the lab, we were making relatively slow progress. As a result, we had to increasingly cram in time closer to the deadline to finish in time. Towards our later meetings, I started to list out specific objectives and success criteria for the working sessions. With a specific objective we were able to work together more efficiently and in parallel toward the task. In the future, this learning can be coupled with time boxing – estimating how long a task will take and having each team member work deeply on a small subsection of the objective.

5.3 Izzie

During this lab, I noticed how difficult it is to collaborate on hardware. Not only does our code work differently in real life than in simulations, but also meeting in person in an area appropriate for testing makes coordination difficult. It is clear that we will need to be very organized in the future in order to use our time efficiently. One case where this was a problem was when we were working on fixing the LiDAR-based errors. The problem only required one or two people to work on it, and having already written the next section of code to implement, I had nothing to do. This might have been more efficient if I had coordinated my coding work with the blocks of time when I wasn't needed for help with implementation. Looking back, I could have spent that time debugging our issues with calculating the distance to the line generated by linear regression, a problem that wasn't essential to finishing the lab but would ultimately make our code more elegant and more resistant to errors. In the future, I will attempt to plan out what tasks I will work on during group meetings when I have nothing else to do.

5.4 Angelina

I've learned that complications in hardware arise often. Our car would sometimes start spinning in circles, even while parallel to the wall. After analyzing log statements and scan data, we discovered that our lidar scanner was misinterpreting nearby scans, and erroneously believed that a wall was nearby. We resolved this by cleaning the lidar and filtering out illogical scan values. Through this process, I discovered that it is difficult to reproduce and isolate bugs, making it hard to debug hardware. I also became more familiar with problem-solving processes as my team worked together to find the error.

On the communication side, I learned about the importance of planning ahead. Since our team did not anticipate having difficulties with hardware, such as the faulty lidar readings, our progress was significantly impeded and we experienced time pressures before our initial briefing. For future labs, we will budget additional buffer time to account for debugging and additional issues that arise.