Lab #3 Report: Wall-Following

Team 2

Ritika Jeloka
Ian Perez
Jason Salmon
Esha Ranade
Sydney Chun

6.4200 Robotics: Science and Systems

March 11, 2023

1 Introduction

Ian Perez

The goal for this lab is to adapt our wall-following algorithm from the previous lab to run on our physical cars and to implement a safety controller that brings us closer to building a fully autonomous car. This lab is part of a series of labs aimed at developing autonomous systems and our work builds on the skills and knowledge gained from previous labs.

In this lab, we faced several challenges while adapting our wall-following algorithm to run on the physical cars. These challenges included obtaining accurate readings from our LIDAR sensors and tuning the PID controllers, which required significant effort from our team to overcome. We believe that the experience gained from solving these challenges will be valuable for future labs in the course.

In addition to the wall-following algorithm, we also implemented a safety controller that is design to protect the robot from performing any actions that could potentially harm it. We carefully considered different approaches and decided to implement the simplest method of sending a stop signal whenever an object is within a certain amount of centimeters of the front of the robot. This approach strikes a balance between safety and flexibility that we believe will be needed for future labs.

2 Technical Approach

2.1 Problem Statement

Jason Salmon

The overarching goal for the lab was not too different from that of our previous simulations. Overall, we still aimed to get the racecar to drive along the wall on the side of our choosing, with little to no error or crashes. However, in this case we had the added complexity of the safety controller, which was needed to prevent the physical racecar from damaging itself if our implementation(s) did not work. Keeping these in mind, we set out with the following goals:

1. **Implement the simulation code for wall-following on the racecar.**

   For this step, we felt that the transition would be fairly simple since, for the most part, the implementation was completed in the previous lab. Ideally, this step would have at most required changing a few lines of code to ensure the right topics were being subscribed/published to, and that we could properly interface with the robot via SSH and the Bluetooth controller.

2. **Tweak parameters within the wall-following code as necessary to improve performance on the plant.**

   This presents a more substantial opportunity for improvement, since it was extremely unlikely that the simulation exactly matched the real-world experience or accounted for small variations and imperfections on the robot and in the environment. The simulation code, at the very least, provided the foundation for our implementation.

3. **Implement a working safety controller.**

   As a fundamentally new block of code, this task allowed us to solve a new problem via a method of our choosing. We had hypothetical algorithms in mind before actually testing anything - which made the eventual coding of the controller easier - but the most uncertainty still revolved around this section because, unlike the others, we had no foundational code which we knew would work for certain.

2.2 Full Process

Esha Ranade

Our team subdivided the full logic into individual deliverables as follows to modularize our code structure:

- Reformat Velodyne input data to be consistent with the simulation

- Filter data according to the desired ranges
- Locate relative positions of walls and obstacles based on the closest distances measured at angles within the filtered data
- Calculate the error, the shortest distance to the perceived wall
- Determine robot steering angle based on required error adjustment
- Determine robot speed based on safety considerations

We will expand on the technical approaches to each below and discuss the challenges that our team encountered and needed to overcome in order to achieve our desired functionality.

2.2.1 Reformatting input data

Our unfamiliarity with the Velodyne LIDAR sensor and the additional preprocessing required to render its outputs useful led us to spend much of our time trying to understand how to parse the sensor data.

First of all, the Velodyne sensors produced 897 distance measurements, equally spaced around the full circle. The simulated sensor output from lab 2 only included 100 points total, spanning 270 degrees such that the 90 degrees in the back did not have any distances associated with them. To account for this increase in data points, we needed to scale our calculations converting from a desired angle to its corresponding index in the ranges vector.

The provided information suggested that the Velodyne sensors were placed on the robot with a 60-degree rotation, something we were able to verify for cars 77 and 88 by observing the direction of the logo on the robot's sensor. Car 85's sensor, however, had a rotation of close to 90 degrees. Figure 1 below depicts the angles measured by the Velodyne sensor and their correspondences to the robot's forward direction, which is what we used to control steering in our simulation from lab 2. We considered several strategies but ultimately decided to apply an algebraic offset to the measured angles in order to map them to their bearings in relation to the front of the robot.

Our team noticed an unexpected fluctuation in the LIDAR sensor data: alternating high and low values in succession. Through discussion with the other teams in Pod 1, we realized that the Velodyne sensor appeared to be publishing two halves of the scan to the same topic, resulting in alternating data values. This split is shown in Figure 2. To address this, we created a custom intermediary node where the ranges of the two LaserScan messages data provided by the sensor were first concatenated into a single array, with all of the values and angles in order. This array was then sent to the /scan topic, allowing us to parse continuous data.

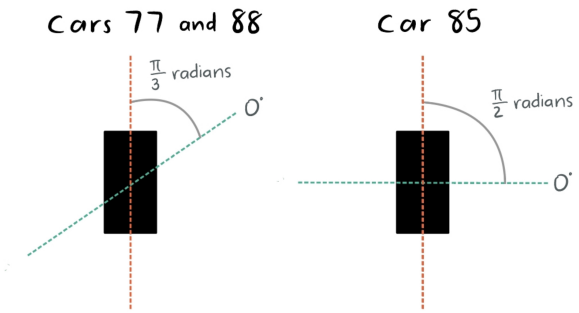

Figure 1: Velodyne sensor angle offsets, varying by car

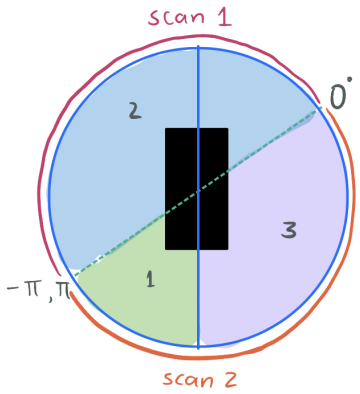

Figure 2: Angles seen by the two scans. The labels 1, 2, and 3 represent the order that distances at those indices would be added to the scan's ranges array to be consistent with the simulation.

We also noticed a phenomenon where several "inf" or infinite values were being measured by the sensor. We speculated that these occurred when the closest object being sensed was either too close or too far away from the sensor. Rather than filtering these out entirely, which would have decreased the total number of data points and changed our index calculations for a desired direction, we initially replaced the 'inf' values with the maximum number detected in the range. Using the maximum distance would prevent the robot from actually using it in a calculation to determine heading. However, we later realized that each of the two different ranges of scan data included the full 360 degrees worth

of distances, though only half of each was relevant. The 'inf' values represented the points not included in a certain half's intended range, as seen in Table 1 below. By combining every two consecutive scans in such a way that only the minimum (non-'inf') value corresponding to a desired direction was included in the distance calculations, we were able to get the correct values required for our PID control. For the angles representing the back of the robot, such as -$\frac{2\pi}{3}$, both scans reported a value of inf because other parts of the robot blocked the sensor in these places; these ranges were not included in any of our calculations.

Table 1: Scan Values by Angle

Sensor Angle	π or - π	$\frac{\pi}{2}$	0	-$\frac{\pi}{2}$	-$\frac{2\pi}{3}$
Scan 1 Value	valid	valid	valid	inf	inf
Scan 2 Value	valid	inf	valid	valid	inf
Value Kept	Minimum	Scan 1	Minimum	Scan 2	inf (not used)

2.2.2 Filtering data ranges

In order to ensure that only relevant data was included in our calculations to locate the robot's position relative to the wall, we focused on a specific slice of the range vector received from the LaserScan based on angles. If the robot is supposed to follow the right wall, it primarily looks at angles along its right side. On the other hand, if it's supposed to follow the left wall, its focus is on the sensor's left side. Through trial and error, we found the optimal angle range to be between 30 and 70 degrees relative to the forward direction of the robot. This is visualized in Figure 3. Additionally, the robot always considers measurements within 1 degree of the forward direction, as can be seen in Figure 4. This allows it to detect walls in front of it when it is approaching a corner.

2.2.3 Locating relative positions of walls

Once the raw data has been filtered into the essential optimal set of distances, we use linear regression to find the line representing the wall relative to the robot. The filtered data from the previous step is still in the form of polar coordinates, because the distances are associated with their angle relative to the robot's heading. We translate these coordinates onto the Cartesian system by multiplying the distances by the cosine and sine of their corresponding angles to get the individual x and y components, respectively. In order to calculate the line of best fit, we utilize the Numpy library's polyfit function with a degree of 1 for a linear result. This allows us to extrapolate the $y = mx + b$ representation of a line fitting to the desired wall from the full scatterplot-like representation of the robot's full surroundings, as can be seen in figures 5 and 6.

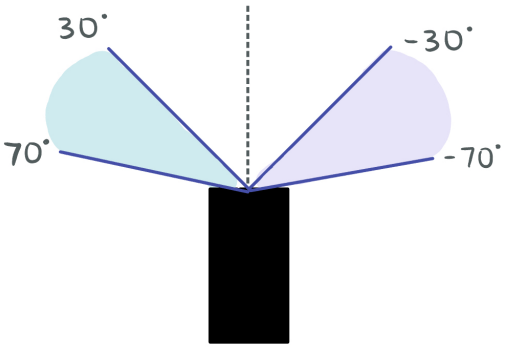

Figure 3: Angles considered for side wall detection: blue range for left-wall following and purple range for right-wall following.

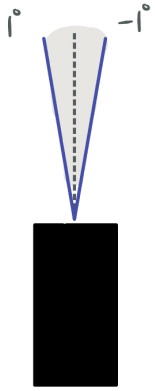

Figure 4: Angles considered for front wall detection.

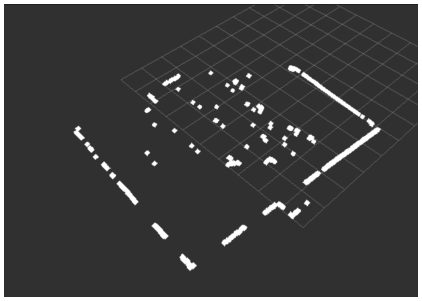

Figure 5: Full robot surroundings

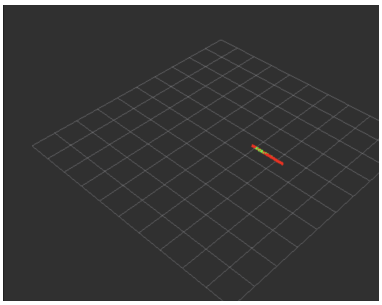

Figure 6: Detected wall

2.2.4 Computing error signal

Once the line of the wall is determined, we must compute the closest distance from the robot to the wall. This involves the following simple mathematical calculations, where m is the slope and b is the y-intercept of the line representing the wall.

$$perpendicular\_slope = \frac{-1}{m}$$

$$closest\_x = \frac{b}{perpendicular\_slope - m}$$

$$closest\_y = perpendicular\_slope * closest\_x$$

$$distance\_to\_wall = \sqrt{closest\_x^2 + closest\_y^2}$$

This series of calculations results in a numerical value *distance_to_wall*, representing how far the robot is from the wall it is supposed to follow. The error signal represents how far the robot is from its ideal path and can be computed as follows:

$$error = distance\_to\_wall - desired\_distance\_from\_wall$$

2.2.5 Determining robot steering angle

Given the error signal, we employ Proportional-Derivative control. This is a type of controller whose output varies in proportion to the error signal as well as its derivative. When tuned with proper coefficients, it determines the robot's steering angle in a way that minimizes error over time, allowing the robot to stay as close to the desired path as possible. The calculations follow the equation below, given the error signal and $\frac{d}{dt}$, the time between two consecutive measurements:

$$steering\_angle = K_P * error + K_D * \frac{d}{dt} * error$$

This steering angle is published to the ROS drive topic, sending the correct command to the robot to control the direction it drives in.

2.2.6 Determining robot speed

The final module of our code is the Safety Controller. When the robot gets too close to an unexpected obstacle, it is programmed to stop by overriding the other drive commands with a message that sets the speed of the robot to 0 until the obstacle is removed and the original functionality is resumed. The logic is

fairly intuitive:

if *min(distances)* $\leq$ *distance_threshold* **then**
  stop_robot();
end
else
  Continue as normal
end

Algorithm 1: Logic to stop robot when it gets too close to an obstacle

In order to make sure the robot only stops when someone is directly in its path, this logic is only applied for obstacles within 30 degrees of the front of the robot, as shown in Figure 7.

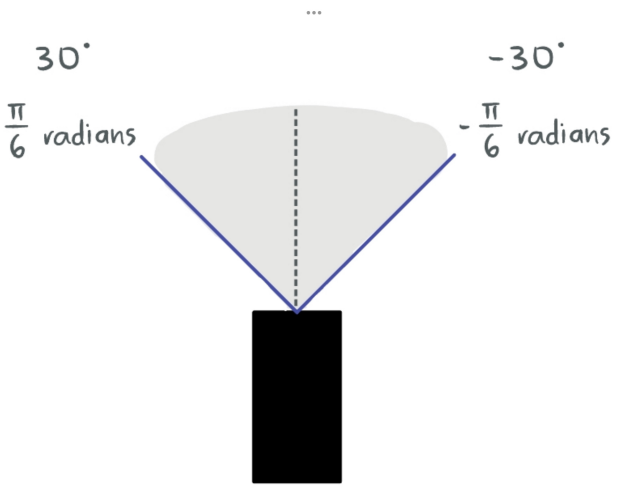

Figure 7: Angles seen by the Safety Controller.

2.3 Required Computations

Jason Salmon

There were many mathematical challenges involved with properly coding the robot. Having gone through our implementation above, this section explains how we arrived at the relationships and parameters we used.

2.3.1 Linear Regression

Linear regression is essentially a method for finding the best approximation of a linear relationship between two sets of data. Ordinarily, we understand this as having a set of points that seem to follow a trend, then (after some calculation)

drawing a line through those points. Imagine two vectors $\bar{x}$ and $\bar{y}$, where $\bar{y}$ versus $\bar{x}$ is assumed to be a linear relationship. The goal of linear regression is to find a vector of coefficients $\bar{\beta}$ which minimizes the magnitude of the error $||\bar{y} - \bar{\beta} \cdot \bar{x}||^2$. The eventual $\bar{\beta}$ vector represents the slope and intercept of the linear relationship.

Why use this to find our wall? For one, the LiDAR readings naturally involve some level of noise -both due to the natural imperfections of the surfaces we might wish to detect, and due to the noise introduced by the sensor itself. The linear regression allows us to extract a nice, smooth line, which helps to alleviate this issue. Secondly, the LiDAR records a finite number of distance measurements - thus, the possibility exists that the measurement corresponding to the true closest distance to the wall was simply not taken. Linear regression allows us to extrapolate from the points and pick the distance we desire.

An additional argument against the success of linear regression is that it does not handle the changing geometry of the walls. Rooms have corners, objects, turns - all profiles which are not necessarily smooth at all points and therefore cannot always be analogized to a perfectly straight line. Strictly, it is true that linear regression fails as an approximation of the wall. However, this behavior is still desirable. Imagine a set of points which follows a trend - perhaps representing a close wall. Now imagine that another portion of LiDAR data is taken beyond this set, fitting with an entirely new trend separate from the first - a far wall, in this example. The robot is at a corner, and so ideally we would like it to turn into the empty space. However, instead of creating two straight lines for the robot to attempt to follow (which it may or may not be able to do, depending on how far forward it can travel), the linear regression provides a line at each instance which poorly fits both sets of points. However, as the line evolves over time, it morphs continuously into the next wall the robot is to follow. The result is a relatively smooth turn, approximated as a polygonal surface with many, many sides.

2.3.2 Distance Error

After we obtain the linear models at each time step, the next step that remains is to perform a calculation of the distance error to the wall. Fortunately, this is straight-forward, as the shortest vector to a line is always orthogonal. We simply calculate this vector by inverting the slope obtained from the linear regression, and calculating the x and y values accordingly:

$$x = \frac{b}{-1/m - m}$$

$$y = -1/m \cdot x$$

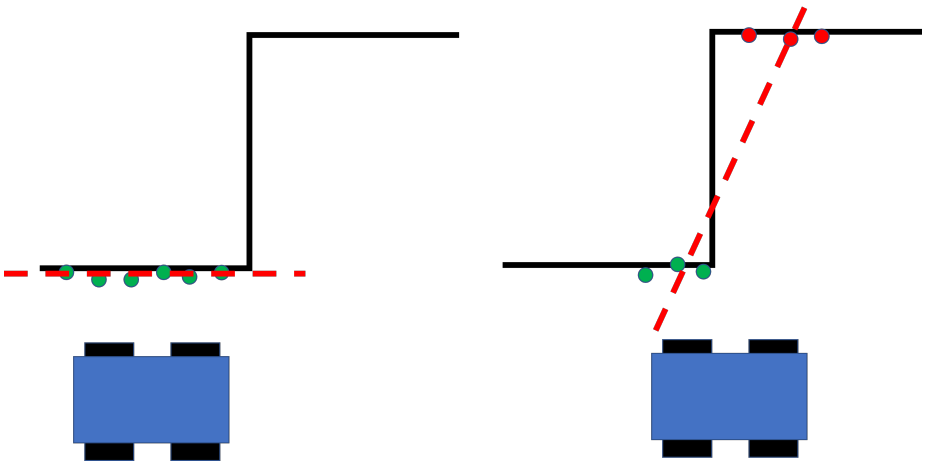

Figure 8: Visual explanation of how the fit from linear regression evolves across a corner. The gradual change in slope allows the car to parse a corner as multiple surfaces rather than a single edge.

2.3.3 Linearized Model

From lecture, we know that the control action we desire (lateral acceleration in the direction of our choice) has a trigonometric relationship with the steering angle:

$$a_y \propto \sin\theta$$

However, for small values of θ, we have the advantage of approximating this relationship as linear, where $\sin\theta \approx \theta$.

2.3.4 PD Controller

Given that we have a linear system, we seek to use the error to drive the car in the appropriate direction. A common solution to this problem is to use some subset of PID (Proportional + Integral + Derivative) control, where each of the terms is applied to a calculated variant of the error. With proportional control, it is possible to simply act (with some scaling value) in the opposite direction of your error, thus shrinking it. Generally, the higher the coefficient for proportional control, the faster the rate of change. A high coefficient has the advantage of fast convergence to a solution. However, it presents several issues:

- **It is typically impossible to implement control action beyond a certain value.** This is usually due to physical limitations (the maximum voltage that can be provided, the maximum turning angle, etc.)

- **A faster rate of action leads to more overshoot.** This is especially bad in our case, since a significant enough overshoot means crashing into a wall and possibly damaging the car.
- **Increasing the coefficient increases the amplification of noise.** This means that artifacts that have nothing to do with the control action begin to have a significant effect on the robot's movement.

We may alleviate some of these issues by combining proportional control with derivative control. By also managing the robot's movement by the rate of change of the error, we may dampen the system's action when the controller recognizes the situation is improving, or increase action in the opposite case. In ideal scenarios, critical damping occurs, leading to the minimum time to zero error, but miraculously zero overshoot.

We found that a PD with proportional gain of 1 and derivative gain of 0.55 achieved desirable convergence with minimal overshoot.

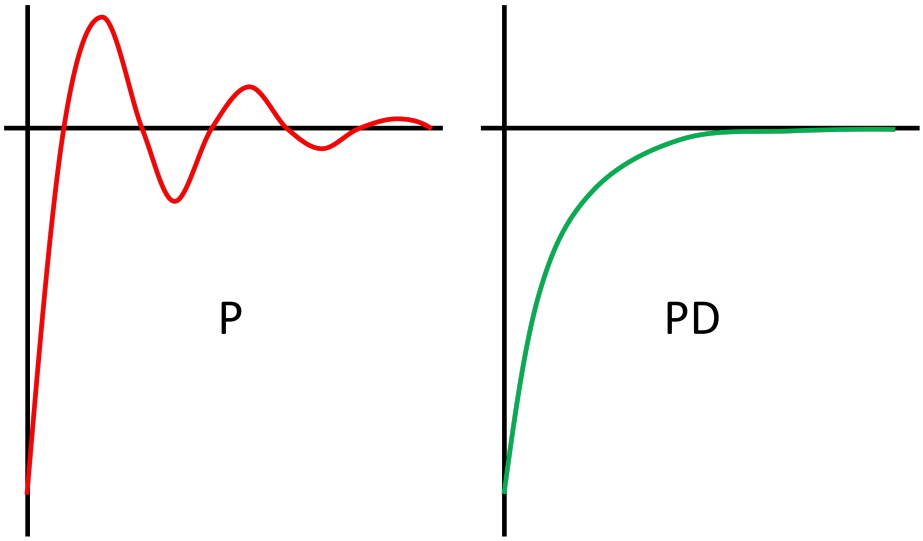

Figure 9: The ideal damping effect of derivative control.

2.4 ROS Implementation

Ian Perez

The ROS architecture we used for this project is depicted in Figure 10. The nodes colored in green are the ones we created and the rest were given to us. At

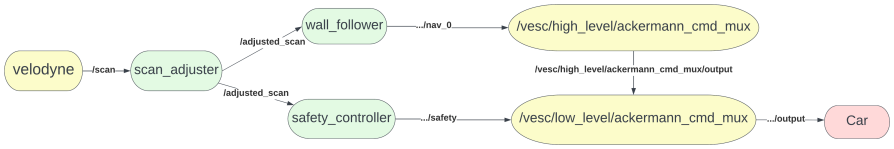

Figure 10: Breakdown of our ROS architecture for this project. Only the nodes in green were created by our team.

a high level we have a node called "scan_adjuster" that subscribes to the /scan topic. It is responsible of merging every two scans by taking the minimum of both and publishing it to the "adjusted_scan" topic. The reasoning for this was explained in Section 2.2.1. We also have a node called "wall_follower" which subscribes to the "adjusted_scan" topic. It is responsible for listening to the scans, detecting the wall, and publishing the appropriate steering commands to the "/vesc/ackermann_cmd_mux/input/navigation" topic which effectively allows the robot to move. Next, we have the "safety_controller" node that also subscribes to the "adjusted_scan" topic. The role of this node is to override any steering commands produced by our other nodes with one that stops the car whenever an obstacle is less than a given distance threshold away. It is able to do that by publishing the stop messages to the "/vesc/low_level/ackermann_cmd_mux/input/safety" topic.

3 Experimental Evaluation

Ritika Jeloka

After establishing our wall_following code with the modification of the LaserScans output as explain the section above, we moved on to optimization.

Our team tested various PID values, adjusting Kp, Kd, and Ki, to reach an optimal state of wall-following. We started by only setting a Kp value, and then increased Kd slowly to decrease oscillation time. In the end, we chose the PID values that led to the least average distance error (actual distance-desired_distance to the wall).

We then plotted this average distance error for all cases as shown below.
Figure 11 plots the distance of the car from the right wall in meters (y axis) by time (x axis). Kp was set to 1 and Kd was 0.44. Ki was not used.

In this graph, the desired distance was set to 0.5m. The car was set to an initial distance of 1.2m away from the right wall and the wall_following.py code was run at approximately 23 seconds. The car nears the desired distance to the wall (0.5m) in about 1 second and then oscialltes around the desired distance for

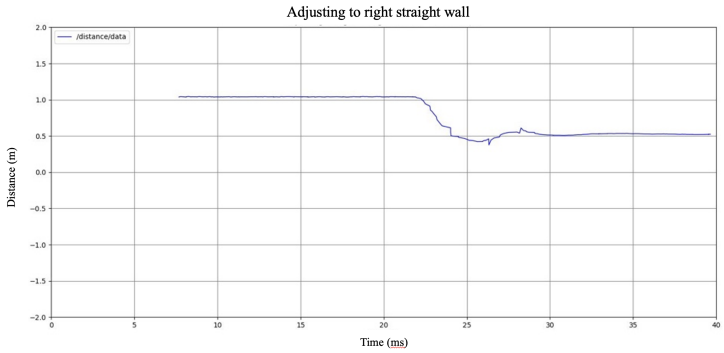

Figure 11: Average distance error following the right straight wall

about 6 seconds until it reaches a steady state, which is at the desired distance from the wall.

The average error (measured by the absolute value of the car's actual distance-desired distance from the right wall) was 0.029651138995160069 for this run.

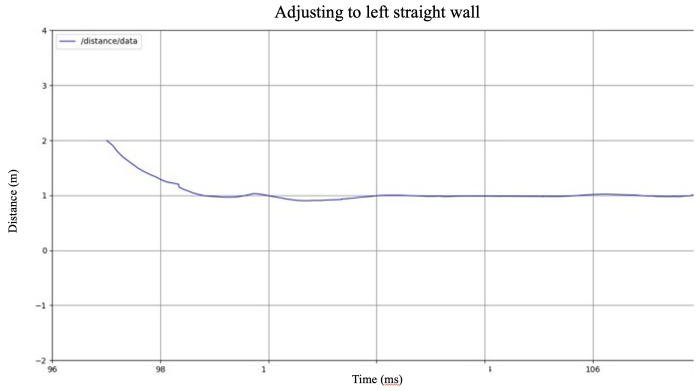

Figure 12: Average distance error following the left straight wall

Figure 12 plots the distance of the car from the left wall in meters (y axis) by time (x axis). Kp was set to 1. Kd and Ki were not used.

In this graph, the desired distance was set to 1m. The car was set to an initial distance of 2m away from the left wall and the wall_following.py code was run at approximately 97 seconds. The car nears the desired distance to the wall (0.5m) in about 2 seconds and then oscillates around the desired distance for about 3 seconds until it reaches a steady state, which is at the desired distance

from the wall.

The average error (measured by the absolute value of the car's actual distance-desired distance from the left wall) was 0.032651138995160069 for this run.

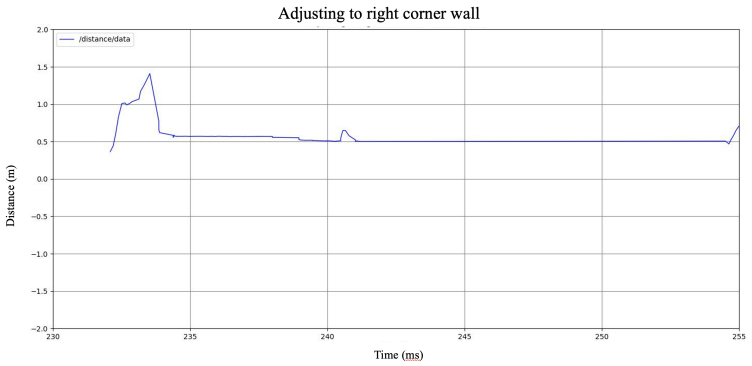

Figure 13: Average distance error following a right corner

Figure 13 plots the distance of the car from the right corner in meters (y axis) by time (x axis). Kp was set to 1. Kd was set to 0.55, and Ki was not used.

In this graph, the desired distance was set to 0.5m. The car was set to an initial distance of 0.5m away from the right wall at the start of the corner and the wall_following.py code was run at approximately 233 seconds. As the car approaches the corner, it increases distance from the wall to around 1.4 meters away in 1 second. It then turns and returns closer to the wall in another second. The car then uses the 5 seconds to approach the steady state of the desired 0.5m away from the right wall.

The average error (measured by the absolute value of the car's actual distance-desired distance from the left wall) was 0.0064828373833442374 for this run.

When our racecar was following a straight line, the time taken to oscillate before reaching a straight line ranged from 3 seconds to 6 seconds.

Additionally, our racecar reliably did not crash into the wall. In Fig 1, 2, and 3, it is seen that once the car reaches the steady state of the desired distance away from the wall, it does not move significantly further away from the desired distance. Because of this, the robot will never crash into the wall.

The closest distance reached to the wall was 0.45m away, when the desired distance was 0.5m away. This distance occurred when the car was following the right straight wall. The closest distances to the wall when the car followed the

left wall and both corners were significantly larger, emphasizing that the car will reliably not crash into the wall. The furthest distance from the corner reached was 1.4m.

Another measure in place to ensure that the robot did not crash into the wall, or anything else, was the safety controller. Our safety controller worked to stop the car if a minimum distance of 0.2 was recorded in the front 60 degrees of the car. The car then immediately sends out a drive message with a velocity of 0 to stop the car. Once the obstacle is removed, the car proceeds following the wall path.

4 Conclusion

Ian Perez

The goal for this lab was to adapt our wall-following algorithm from the previous lab to run on our physical cars and to implement a safety controller that brings us closer to building a fully autonomous car. This lab was part of a series of labs aimed at developing autonomous systems and our work builds on the skills and knowledge gained from previous labs.

We were able to adapt our wall-following algorithm so that it was able to run on our physical cars and implemented a safety controller to protect the robot from harm. We faced several challenges along the way such as obtaining accurate LIDAR readings and tuning the PID controllers, but we overcame them as a team.

We encountered some difficulty with our wall-following algorithm. The robot had trouble following walls only on the right side. This issue however was resolved by changing how we were measuring distances to the estimated wall. Additionally, we recognize that our safety controller may need more work if it limits us in future labs.

Looking forward to our next design phase, we will focus on integrating more sensors such as the camera into our autonomous vehicle and improving the overall performance and functionality. We look forward to continuing towards our goal of building a fully autonomous vehicle by the end of the semester.

5 Lessons Learned

5.1 Ritika

Completing this lab has taught me many things. Most importantly, that if something can go wrong- it will. Our team spent countess hours debugging issues that were unrelated to the lab, but over time got better at handling them. In terms of communication, I learnt that it is best to be proactive when

working with a team. I have had teams in the past where people stop replying or showing up, but I was very happily surprised when all of group2 was so involved and always held up their side of the project. We also all communicate often and effectively. Technically, by watching other members on the team code, I have learnt various little tricks to improve my coding. For example, one member knew how to create a graph to start collecting data as we ran our experiments. This was an extremely helpful skill to learn. I hope to learn more, and grow as a coder and collaborator over the course of this year!

5.2 Ian

While completing this lab, I gained a better understanding of useful ways to debugging common issues that might arise again in the future. In particular, I found that power-cycling the racecar was an effective solution to many of the problems we encountered. Additionally, I learned the importance of clear communication when working in a team. By openly discussing our progress and challenges with both our team members and TAs, we were able to avoid misunderstandings and ensure that everyone was on the same page. Finally, I also improved my collaboration skills by learning to effectively share ideas and work together to find the best solutions to problems. This involved being receptive to the input of other team members and combining different ideas in order to come up with the most effective solution.

5.3 Esha

This lab taught me about the many new challenges that come along with the introduction of hardware. Our team faced many moments of confusion and delays related to unexpected behavior from the robot, from sensors timing out to power supplies being interrupted. The most consistently successful solution was to turn the robot off and back on, which frustrated me because it didn't necessarily reveal the root cause of an error. Surprisingly, though, I found that having one single robot and one single computer that code could feasibly be written on at a time was much less of a bottleneck than I had expected; my team members' involvement and constant willingness to help in proactive ways allowed us to keep moving forward. Having multiple pairs of eyes to spot various bugs in our code and perform sanity checks with allowed us to catch things that would've taken me much longer to find alone. I am excited to continue working with this team throughout the semester and learn more from them in future labs!

5.4 Jason

Throughout this lab, the most important thing that I learned to appreciate was the value of determination. We experienced many hardships throughout the duration of this lab - some of which was due to our general inexperience with the platform and techniques required, some due to hardware failures and general

uncertainty around how the robot should function. Nonetheless, we managed to achieve a working robot by putting in the work and persevering. Alongside this point, an important takeaway was that sharing information, understanding and tricks between our fellow groups is paramount. Having more than one group operating in parallel experiencing the same hardships and difficulties was extremely valuable, since other groups may take other approaches and try other better implementations that do not match our own. Overall, I am glad that I have had this experience thus far, and I look forward to other fruitful (and hopefully less Velodyne-intensive) labs in the future!