Lab #3: Wall Follower and Safety Controller

Team #5

Preston Tower
Waly Ndiaye
Cole Paulin
Erina Yamaguchi

6.4200/16.405: RSS

March 10, 2023

1 Introduction

1.1 Overview and Motivations

During this lab, we designed a low-level control system and safety controller to be embedded into our robot. The purpose of these two systems is to follow a wall at a user-defined desired distance, and to avoid collisions. While the control system is fairly rudimentary in the sense that we are only following a wall, it will serve as the building block for the rest of the features we will be implementing in the future (path planning, object identification, etc.). Eventually, we will use this controller to keep our robot on a desired path. In order to evaluate the success of these systems, we set our goals to minimize the positional error from our desired path, ensure no collisions between the racecar and objects, and limit dangerous controller behavior (such as overshoot). The deliverables for Lab 3 were split up into four sections: evaluating the controller in a simulated environment, implementing the algorithm onto the physical racecar, designing a safety controller, and testing our racecar with both controllers running.

1.2 Goals and Implementation

Our group decided that a Proportional-Derivative (PD) controller is the best fit for this situation, as the proportional action is designed to drive the car towards our desired point, and the derivative action reduced overshoot. During the testing of the racecar in the physical environment, the Ziegler Nichols tuning method to determine optimal gain values for both the proportional gain and the derivative gain. After obtaining experimental results, we began implementing the safety controller. Our method for the safety control works as follows: look at a certain range of data scans, determine if there is an object in that frame, then give the robot new commands to avoid crashes. If the robot detects an object, then the robot will either slow down or stop, depending on the distance from the robot. The PD controller and safety controller combine to create a more robust system that can accurately follow a wall at a desired distance, and will not collide with objects. In addition, we had to figure out where to place our nodes in ROS, and where they should be publishing and subscribing. In general, the robot subscribed to the LiDar scan data. This scan data drove many of our calculations, such as finding the wall, and determining if there are objects that might impede the robot's motion. The PD and Safety controllers publish their control decisions to the driving commands of the robot. The safety publish had precedent over the PD publisher so that we could override the robot's original path in order to stop, if necessary. After ensuring our robot could accurately follow a wall and avoid crashes, we collected some additional data that may lead us to helpful findings in future labs. Findings regarding the changing robot velocity and adjustable PD gains could eventually give the robot a more robust system.

2 Technical Approach

2.1 Wall Follower

The first goal of this lab is to design a node that subscribes to the LaserScan data, and publishes our PD controller to the navigation of the robot. Our PD controller acts on our error and adjusts the steering angle command str_cmd by the following formula:

$$str\_cmd = K_p * error + K_d * velocity * \theta$$

$$(error = desired\_distance - wall\_distance)$$

$$(K_p = \text{proportional gain})$$

$$(K_d = \text{derivative gain})$$

$$(velocity = \text{robot's velocity})$$

$$(\theta = \text{slope of linear regression of the distances the relevant LaserScan data traveled})$$

One of the parameters of the wall follower is to follow a left or right wall, defined by the user. Defining the positive x-axis as the direction of forward motion for the racecar, the PD controller must also take into account the change in direction of steering when following different sides of the wall. Therefore, a *side* parameter is introduced into the controller. The complete steering angle command is defined by:

$$str\_cmd = -side * K_p * error + K_d * velocity * theta$$

If we are following the left wall, and we are too close to the wall, *error* is a positive value, K_p is always a positive constant, and *side* equals one, resulting in a negative command. This makes sense because using the right-hand rule, turning right from the car's x-axis would mean we would be turning clockwise, towards the negative y-direction. The damping term's sign is only dependent on θ. θ, determined by the slope of the linear regression data is negative if we are moving towards the wall, which acts as a damper, making the car want to turn in the opposite direction of the direction it is turning now.
The rest of our Node acts to solve for these variables. The process for the controller can be split into three segments, determining our relevant scan data, finding an approximation for the wall distance, and solving for optimal gain values.

2.1.1 Determining Relevant Data

2.1.1.1 Slicing LaserScan Data

The LaserScan spans a range from $(-4 * \frac{pi}{3})$ to $(+4 * \frac{pi}{3})$. All of this data are not necessary when attempting to follow a wall in one direction. Our solution to this problem was to either take the $([angle\_min, 0])$ range, or the $([0, angle\_max])$ range, depending on the wall we are following. For example, if our car is following the wall on its right side, we will only use the positive LaserScan angles, because those scans determine the wall location.

2.1.1.2 Removing Noise

After determining the slice of data to use for wall-finding, we created a threshold distance to remove any of the LaserScan values that were either noisy or too far away. Outliers in the data set could affect our wall-finding algorithm. This is because an outlier data point can add incorrectly large distances to the linear regression algorithm used in the Wall Finding algorithm. Outliers were determined by a parameter set by the user as relevant distance. Depending on the environment, the walls could be double or triple the distance from the Stata basement simulation. Therefore, we set this to be a modifiable variable.

2.1.2 Wall Finding Algorithm

The relevant LaserScan data was used to determine an approximation for the location of the wall. The data showed us the relative distance and angle of where the LaserScan made contact with a solid object. Using the distance, angle, and some trigonometry, we broke each data point down to its relative x and y coordinates. Using these components, we ran a squared-loss linear regression algorithm on the coordinates. The output of the algorithm was a python list of $([slope, intercept])$, where the slope represents theta, and the intercept represents wall_distance (the robot's distance from the wall).

2.1.3 Determining PD Controller Gains

The optimal gain values were determined using the Ziegler-Nichols tuning method. First, the goal is to first find an ultimate gain. To find the ultimate gain, the K_d value is set to 0, and the K_p is adjusted such that the robot response shows steady oscillations.

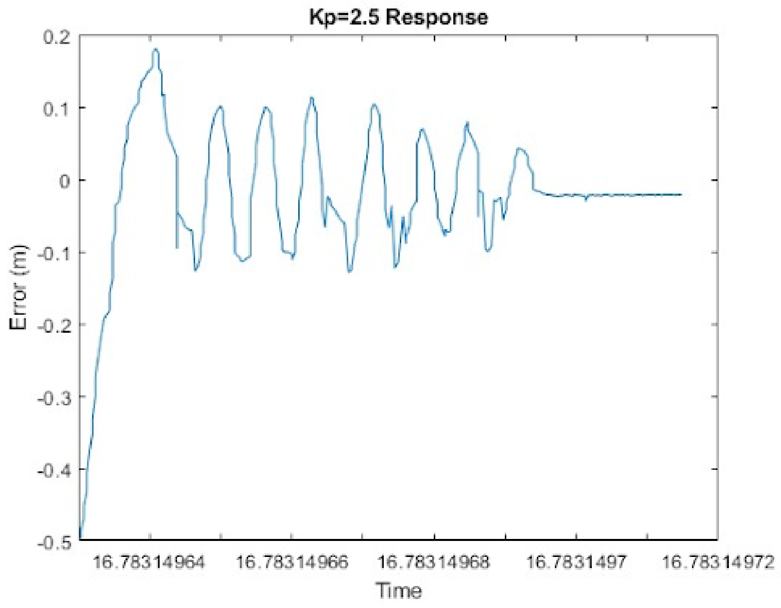

Figure 1: The y-axis represents the robot's deviation from its desired position. The PD gain values are set to $K_p = 2.5$ and $K_d = 0$

These oscillations show the robot's response to following a line. Due to the steady oscillations, we can determine that our Ultimate_gain 2.5. From Ultimate_gain, we approximate the gain values to be:

$$(K_p = Ultimate\_gain * 0.8 = 2.0)$$

$$(K_d = Ultimate_gain * .1 = 0.25)$$

We first verified the response behavior of $K_p = 2$ as shown below

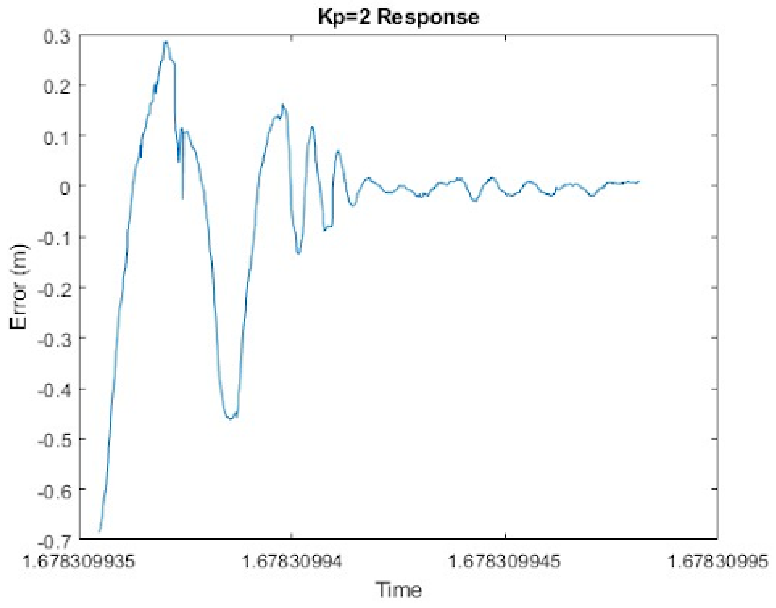

Figure 2: The y-axis represents the robot's deviation from its desired position. The PD gain values are set to $K_p = 2$ and $K_d = 0$

Using K_d = 0.25 seems fairly small, so we first attempted to use K_d = 0.4, and got the following response

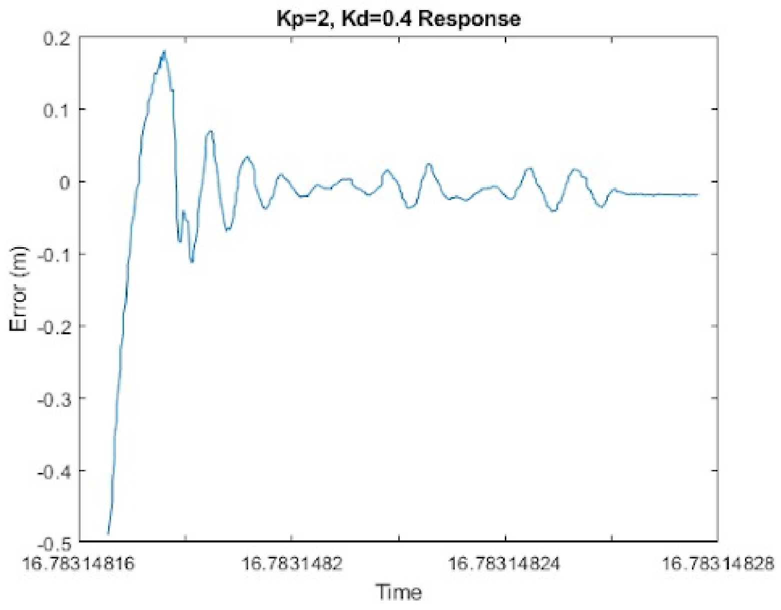

Figure 3: The y-axis represents the robot's deviation from its desired position. The PD gain values are set to $K_p = 2$ and $K_d = 0.4$

There is an initial 20 percent overshoot, and we also see some persistent oscillations. These are both properties of a system with too little derivative action. We then observed the response's distance error over time with a K_d gain of 0.8, and found that this corrected the issues of our previous gain.

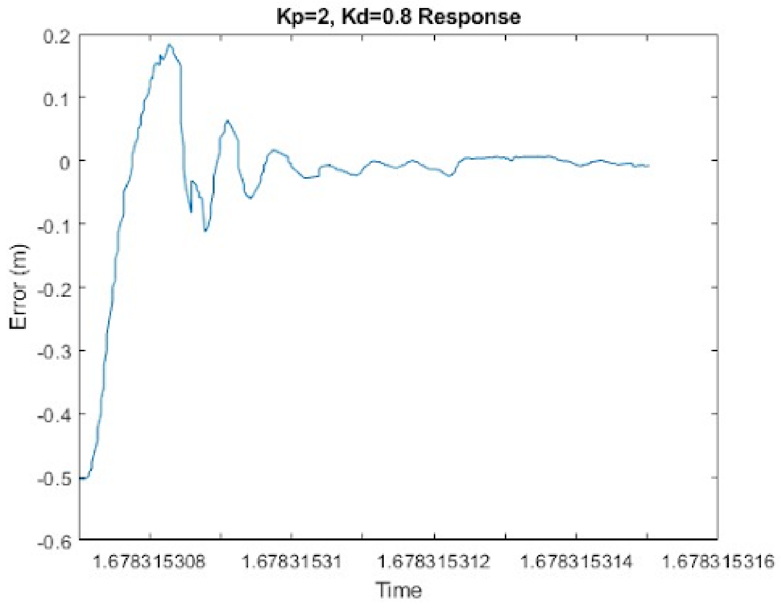

Figure 4: The y-axis represents the robot's deviation from its desired position. The PD gain values are set to $K_p = 2.0$ and $K_d = 0.8$

The response has a smoother overshoot, and has much steadier long-term behavior than the previous responses. Thus, we set our optimal K_p and K_d values as 2.0, and 0.8, respectively.

2.1.4 Publish and Subscribe Information

The node we created was named "Wall_Follower", and we would publish our information to the "/Navigation" node, which determined the actuator commands to drive the robot. The only value that the Wall_Follower would adjust is the robot's steering angle. The velocity did not need to be changed because the steering alone had the ability to correct the vehicle's path. Wall_Follower subscribes to "/scan", which includes the data from our 2D LiDar scanner.

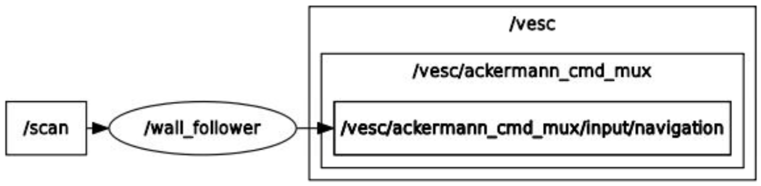

Figure 5: A ROS graph showing the interactions between the publishers and subscribers involved in our implementation of the wall follower algorithm

2.2 Safety Controller

Our second goal of Lab 2 was to create a safety controller that can be deployed on our car in order to avoid crashes. Since we will be utilizing it for the rest of the course, we put effort into making it safe from bugs through concise, clear code and ready for change through initializing variables mainly in our params.yaml file. We wanted to make sure the racecar's safety mechanism did not interfere with the wall follower's functionality, so we took care in testing that our car would not be "scared" of moving if it was touching or very close to a wall.

2.2.1 Determining Relevant Data

For our safety controller, we did not want the racecar to use all of the laser scan data; much of it is looking to the sides or slightly behind the car's LiDar scanner. We wanted to use just a set of LaserScan points that were facing the front of the car so that our car would be able to respond in the event of an object jumping or appearing in front of it. How the specific set of LaserScan points were acquired is detailed in 2.2.1.1.

2.2.1.1 Slicing LaserScan Data

In order to find the relevant data, we first sought out the middle values of our laser scan data. Since, as mentioned earlier, the LaserScan spans from angles $(-4*pi/3)$ to $(+4*pi/3)$, we first found the exact middle point of our scan. This was at 0 radians and the index of this point in the LaserScan.ranges array was the length of the ranges array divided by 2. We named this point front_facing. As an example, if there were 100 points in LaserScan.ranges, we set front_facing to 50.

We then created a variable called relevant_indices to help with slicing the ranges array for relevant data. relevant_indices was initialized in our parameters file in order to make our code ready for change and variability. Using this variable, we sliced our LaserScan.ranges data using a range of [front_facing - relevant_indices, front_facing + relevant_indices]. We set relevant_indices to be 8, so that we include 17 data points facing the front of our car in our safety controller's relevant sliced LaserScan data. This was a value we figured out experimentally by testing our car's ability to stop when considering a different number of data points aimed towards the front of the car. With too many datapoints, our car would inaccurately read the walls as an obstacle and with too few, it would not register an obstacle in time. Reading 8 points on each side of the front_facing point led to the best functionality.

2.2.2 Crash Detection

In order to detect crashes, the racecar used the sliced LaserScan data. We implemented two components to our safety controller. In the event that the car was within a concerning distance of an article, the car would slow down first. If it reached a point where it was in dire danger of crashing, the car's velocity was set to 0, causing it to come to a full stop.

Most of the steps for these two conditions were the same, with the differences being detailed in parts 2.2.2.1 and 2.2.2.2 below. We started our algorithm by creating a new AckermanDriveStamped() object that we would later pass to our publisher. In our params.yaml file, we initialized a few variables: a distance (in m) at which we should start slowing down our car (worry_distance), a speed our car should go at if it needs to slow down (worry_speed) and a distance at which we should stop our car (stop_distance). Another variable (point_threshold) was a threshold for our many points in our sliced LaserScan could be less than or equal to those distances before we told our car to slow down or stop. Similar to the relevant_indices variable from section 2.2.1.1, we tested this variable experimentally to find a value with the best performance. After this initial setup, we created two integer counter variables to track how many points in our laser scan indicated we need to 1) slow down or 2) stop the car.

Using the sliced LaserScan data, the algorithm iterated over all the values, checking if a given LaserScan value was less than worry_distance and/or stop_distance, and if it was, 1 was added to the respective counter. If either of the counter variables eventually reached a value equal to or greater to point threshold, a command was published to the car to slow down/stop, depending on what was needed.

2.2.2.1 Stopping the car

For stopping, the algorithm specifically checked if enough of the points in our sliced LaserScan data were less than stop_distance. If this condition was reached, the code stopped iterating through the data. Stopping the car always took precedent over slowing down the car. That is, if both conditions to slow down/stop were reached, only the command to stop (setting the velocity to 0) was published.

2.2.2.2 Slowing down the car

For slowing the car down, the algorithm specifically checked if enough of the points in our sliced LaserScan data were less than worry_distance. Velocity was then set to worry_speed and published.

2.2.2.3 Visualization of Safety Controller

Below is a figure that sums up the steps taken to make our safety controller functional. It shows how our car responds to potential crashes. In this example, to simply illustrate functionality, our point_threshold was 5.

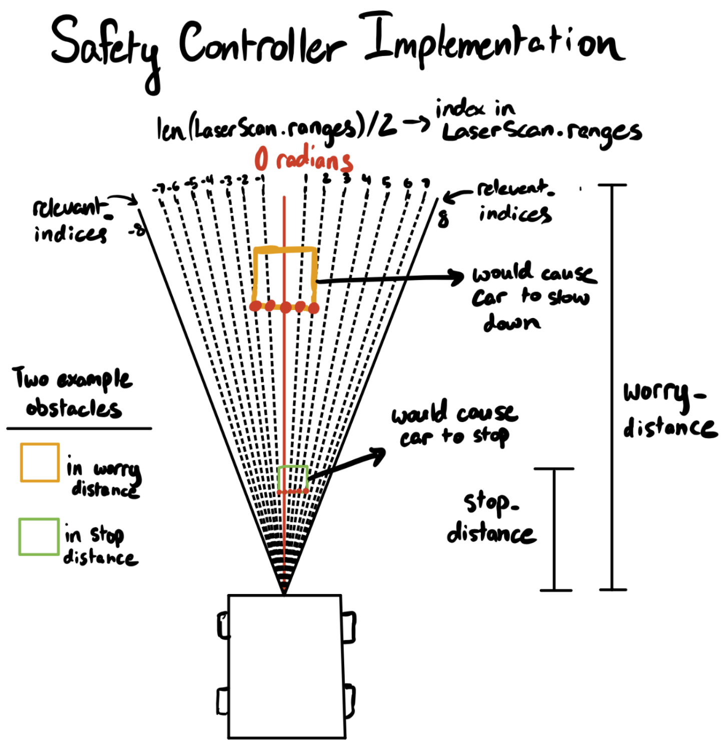

Figure 6: A depiction of how our safety controller works at causing our car to stop or slow down depending whether obstacles are in the range of stop distance or worry distance respectively

2.2.3 Publishers and Subscribers

In order to fully implement our safety controller, our code had to utilize two subscribers. The first of these subscribes to the "/scan" topic. This is where our racecar got our laser scan data from, in order to slice it and determine the likelihood of a crash. The other subscriber that was listened to was "/vesc/high_level/ ackermann_cmd_mux/output." From this subscriber, we were able to get information regarding the car's driving. The resulting decisions made by the algorithm in section 2.2.2) Crash Detection was published to our safety top_cmd_mux/ input/safety." Since this topic is 2nd on the priority order, after teleop but before any navigation topics, our team always tested our safety controller through running our wall follower. Below is a picture of our ROS implementation of the safety controller.

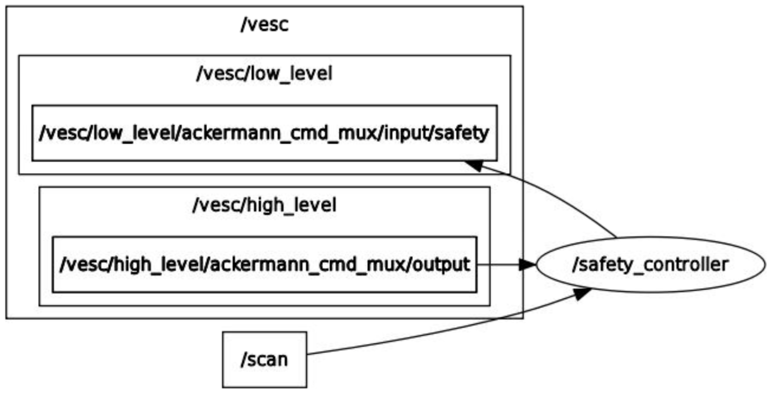
/vesc
/vesc/low_level
/vesc/low_level/ackermann_cmd_mux/input/safety
/vesc/high_level
/vesc/high_level/ackermann_cmd_mux/output
/safety_controller
/scan

Figure 7: A ROS graph showing the interactions between the publishers and subscribers involved in our implementation of the safety controller algorithm

3 Experimental Evaluation

Going from a simulated RVIZ environment to a physical racecar can lead to some unexpected results. There are physical limitations that are inherent to the robot. For example, a simulated model may show a large steering angle, but our front axle will only rotate the wheels to a finite degree. Another reason we are gathering data on our robot's performance is to see what facets are robust, and which are prone to failure. We ran tests on both our Wall Follower, and our Safety Controller. These tests evaluate robustness to different positional and velocity values.

3.1 Wall Follower

3.1.1 Effects of Velocity on PD Gain Values

3.1.1.1 Introduction

While testing different parameters of our PD controller, we realized the response varied depending on the velocity. Doubling and tripling the initial velocity, set to one, with the same gains resulting in the racecar oscillating from its desired position. At a velocity set to three, these oscillations grew such that the car would have crashed into the wall without stopping the controller. All of our initial optimal values for testing the functionality of the wall follower were calculated for a velocity of one. Deviating from this velocity, we found responses with far more overshoot and longer settling times. This implies that there is a relationship between PD Gain values and velocity. In order to improve the robustness of the wall follower performance to varying velocities, we experimented to find the optimal values and extract their relationships.

3.1.1.2 Analysis of Changing Velocities on Optimal PD Gains

First, we doubled the velocity to 2. As an initial guess, we decided to halve both the proportional and derivative gains. This decision was made through the assumption that the relationship of the response rate to velocity is approximately linear. We expected the period of oscillations would halve since the doubled velocity would cover the same oscillation distance in half the time.

The original optimal values for a velocity of 1.0 were ($K_p = 2.0$), ($K_d = 0.8$).

Halving these values, the gain values for a velocity of 2 were set at ($K_p = 1.0$), ($K_d = 0.4$)

The response of this experiment is shown in Figure (8).

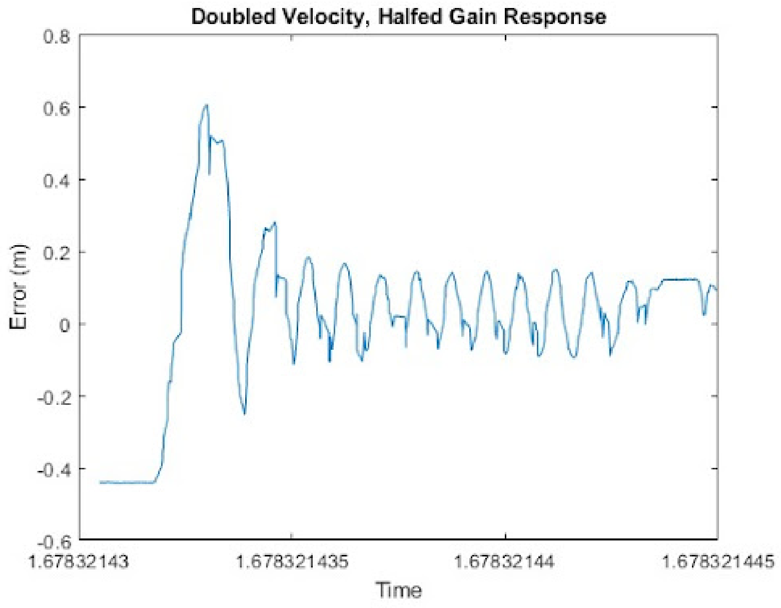

Figure 8: With both K_p and K_d values halved, there are consistent oscillations starting at the third overshoot.

As seen in the response graph constructed from the error data of the racecar's position from the desired distance from the wall, we see consistent oscillations after the initial overshoot. The inability of the controller to converge to a steady state informed us that the proportional term was too aggressive. In the range starting at the third peak, around 0.5-.8 seconds, the robot demonstrates consistent oscillations with an overshoot of approximately 20%.

Therefore, in the next test, we reduced the gain terms by $\frac{1}{4}$ from the original optimal values. The results of this experiment can be seen in figure (9). In an attempt to have a better system response, we ran the same test, but with one-fourth of the gain values. The gains for velocity of 2 for this test were $(K_p = 0.5)$, $(K_d = 0.2)$.

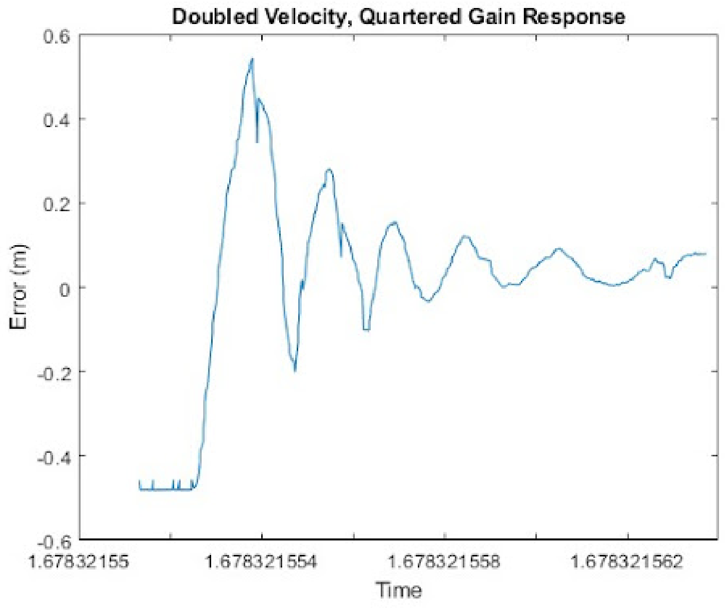

Figure 9: Both K_p and K_d values cut to a quarter, initial overshoot remains the same, but oscillation amplitude decrease to convergence.

There is still slightly more oscillatory behavior than we would like, but it converges to a steady state in under around 6 oscillations. Also, given the car's speed, we can be more forgiving with the response. Generally, the faster the vehicle is moving, the harder it is to find PD Gains.

3.2 Safety Controller

3.2.1 Crash Avoidance Testing

In order to evaluate the safety and robustness of the safety controller, the car was made to approach various obstacles to verify the car will stop before a collision. A white plastic bin lid, small cardboard box, or human person was placed on the floor next to the wall. The car was then made to follow the wall at a distance which would result in a collision with the obstacle.

The white plastic bin lid was representative of a 0.5m wide or larger obstacle, such as a column or wall. The small cardboard box was placed such that its car-facing side was detected as a 0.15m wide obstacle. The human person stood with both legs in the path of the car. These tests were meant to explore the full design requirement that the car not collide with any objects or people.

3.2.2 Crash Avoidance Testing Success

The car successfully stopped before hitting all obstacles. It consistently came to rest with its front bumper 0.35m from the obstacles. The above graphs show the car's velocity over time as it approached each obstacle and slowed before coming to a rest.

Once the car was stopped, we removed the obstacle and the car accelerated forward to resume its wall-following operation. This behavior demonstrates that the safety controller's interruption to operation only lasts as long as the obstacle is present in the path of the car. In future labs, this will allow the car to experience minimal interruption from inadvertent passerby interruptions.

3.2.3 Minimizing Interference with Wall Follower

As mentioned in 2.2.2, we worked to ensure there was minimal interference between our safety controller and wall follower. In order to test this, after we had some confidence in our safety controller, we ran our wall follower through a series of trials. We operated it following right and left walls first. Then we tested convex and concave right and left corners to make sure the safety controller's forward point detection was not causing the car to stop midway through its turn. After we did these initial tests, we tried potential edge cases, like starting the car both very close to a wall and rather far away from a wall. This was to ensure that the wall following behavior of correcting the gap between the current and the desired distance was still functioning as intended.

4 Conclusion/Lessons Learned

4.1 Conclusion

When the robot velocity was doubled, our optimal K_p and K_d values were quartered. More data is needed to make a correct statement, but it is possible the change in K_p and K_d values have a relationship with the change in velocity. This is an inverse-square relationship, where doubling the velocity leads to $\frac{1}{4}$ of the gains. Similarly, three times the initial velocity might lead to optimal gains of $\frac{1}{9}$ of their initial value. In future labs, when we will be operating the robot in a faster environment, it will be helpful to know how to adjust our controller to still achieve desired behavior.

Similarly, tests were performed to confirm the robot will be able to detect and avoid crashes for objects in its path, including non-stationary objects. Additionally, the racecar is able to continue on its path, undisturbed, if the obstacle is removed, even after stopping. Testing for the safety controller and stopping rates were determined so that the racecar operating at faster speeds in future labs would have a greater look-ahead range such that it has enough time to identify and react to obstacles in its path.

4.2 Lessons Learned

4.2.1 Negative Cartesian Coordinates

During our work on the wall follower, we encountered a particularly pervasive issue in our code. Our car would follow the left wall with ease, but couldn't follow the right wall. Eventually, we thought to rospy.loginfo() the (x,y) coordinates of the points. It then became clear to us that what we failed to relate was that depending on which direction the robot was facing, y values might be negative. Since we failed to use absolute value when checking if laser points were to be neglected, we were not properly parsing through our data. This served as a helpful reminder to us to make sure our calculations in future labs account for the possibility of x/y values often being negative values.

4.2.2 Debugging Time

At one point we wrote a more developed safety controller which has a more precisely-defined area in which it checks for obstacles. On trying to run this controller on the car, we encountered a series of bugs. After a few hours attempting to debug it, and with the deadline to complete this report fast approaching, we decided to opt for our more rudimentary but functioning safety controller. The lesson from this experience is to expect at least a few hours of time debugging a completely new system, so this time must be accounted for when deciding if the redesign is feasible.

4.2.3 Publishing Data and Bag Files

In order to analyze experimental data to test the performance of the wall follower and safety controller, bag files were recorded by publishing the desired data as nodes. There were issues with the need to go back and rerecord data because some of the data turned out empty or unusable. These trials however allowed us to create a flow of always starting the launch files first before starting to record, verifying the rosbag has started to record before starting the test, and waiting after the experiment has completed to first close rosbag, then move the robot, so it is clear to tell when the data stops.

4.2.4 Simulated to Physical Environment

In order to quickly debug code, the Proportional-Derivative (PD) controller was implemented in a simulated environment. The simulation had walls already embedded in them from previous LaserScan efforts. The simulated environment does not take the robot's physical limitations into account, such as turning radius and drag. In the simulated environment, the optimal gain values were ($K_p = 5.2$) and ($K_d = 5.0$). This is a drastic difference from the physical robot's optimal gains, which were calculated earlier to be ($K_p = 2.0$) and($K_d = 0.8$). The overall lesson learned is that the simulation can be great for implementing general ideas and strategies, but there must be additional fine-tuning to ensure success in the physical world.

5 Addendum

In order to not clutter the report, we listed out by team member which sections they primarily authored. The listing is recorded below.

5.1 Waly

- 2.2) Safety Controller
- 2.2.1) Determining Relevant Data
- 2.2.1.1) Slicing LaserScan data
- 2.2.2) Crash Detection
- 2.2.2.1) Stopping the car
- 2.2.2.2) Slowing down the car
- 2.2.2.3) Visualization of Safety Controller
- 2.2.3) Publishers and Subscribers
- 3.2.3) Minimizing Interference with Wall Follower
- 4.2.1) Negative Cartesian Coordinates

5.2 Cole

- 1.1) Overview and Motivations
- 1.2) Goals and Implementation
- 2.1.1.1) Slicing LaserScan Data
- 2.1.1.2) Removing Noise
- 2.1.2) Wall Finding Algorithm
- 2.1.4) Publish and Subscribe Information
- 3) Experimental Evaluation
- 4.1) Conclusion
- 4.2.4) Simulated to Physical Environment

5.3 Erina

- 2.1 Wall Follower
- 2.1.3 Determining PD Controller Gain
- 3.1.1.1) Introduction
- 3.1.1.2) Analysis of Changing Velocities on Optimal PD Gains
- 4.2.3) Publishing Data and Bag Files
- Graphs and Visualizations

5.4 Preston

- 3.2.1 Crash Avoidance Testing
- 3.2.2 Crash Avoidance Testing Success
- 4.2.2 Debugging Time
- Graphs and Visualizations