6.4200/16.405 SP23 - Lab #3 Report: Implementing an Autonomous Wall-Following Vehicle with a Safety Controller

Team #20

Aaron Zhu
Jessica Rutledge
Chuyue Tang
Nihal Simha
Shara Bhuiyan
Zhutian Yang

6.4200/16.405 Robotics: Science and Systems (RSS)

March 10, 2023

1 Introduction

Wall-following presents a straightforward method in designing an elementary autonomous racecar system, where the vehicle moves accordingly to simply trace a wall-like object at a certain desired distance. Previously in RSS (MIT's "6.4200/16.405 Robotics: Science and Systems" course), we were given the task of writing a working version of a wall-following autonomous vehicle tested in simulation. In Lab 3, we are now presented with the task of modifying our simulated wall-following code to work on a real-life racecar that is able to execute certain safety precautions at appropriate instances where the vehicle is in danger of crashing.

There are many challenges associated with both transferring a working wall-following simulation to the physical real world and implementing a robust safety controller. In terms of transferring the simulation onto the vehicle, it first of all requires a substantial understanding of the hardware systems provided. Secondly, a tremendous amount of tuning is required to ensure the desired results that were constructed on the simulation are able to be replicated in the real world. Last but not least, constructing the wall-follower in the real world can result in more serious implications such as crashing into objects (rather than just simply passing through the wall as in simulation), requiring more rigorous tuning to ensure a more robust wall-follower. In terms of the safety controller, challenges are associated with balancing the necessity to guarantee the vehicle can avert any risks that may place its hardware in danger while ensuring the vehicle's desired functions also are not impeded at inappropriate instances.

In this lab, we build on our previous simulator lab code to present a working, real-life autonomous vehicle system that can move itself by accurately and safely executing a wall-follower. Given a wall-like object, the car moves parallel to it at a desired distance while autonomously executing safety precautions only at appropriate instances. We show that in the end, our system, through experimentation, robustly operates in a variety of scenarios. It does so by taking advantage of LIDAR laser scan data which is used to inform our system's wall-following and safety controller capabilities.

The contributions of this paper are:

- A comprehensive description of our system design, including our wall-follower and safety controller
- An experimental analysis of the measures used to determine the success of our system.

We detail our technical implementations of the lab in Section II and evaluate our experimental results in Section III. We conclude in Section IV and also add our personal lessons learned in Section V. Videos associated with this lab are available at: `https://drive.google.com/drive/folders/1IMhf-vWsCvO8wsje-Tpm68kaOgo3Tk51?usp=share_link`.

2 Technical Approach

This section details the design of the wall follower and safety controller. We begin by briefly describing our team's initial setup before detailing our wall-follower and safety controller in specifics.

2.1 Wall Follower

Initially, a code review was conducted where each team member's wall follower code was reviewed to understand different techniques that were used to obtain a solution. The following work is a combination of different codes after testing in simulation and on the actual car to obtain the most accurate wall-following controller. The goal of the wall follower is to obtain LIDAR measurements and produce a steering angle to drive the car and maintain a desired distance from the wall. This was achieved through the implementation of a PD controller, which is visualized in Figure 1. The desired distance d_r is specified to the controller after which the racecar produces a change in the distance to the wall L, which is processed by the sensor block to obtain a measurement of the distance from the wall y. This is then subtracted from d_r to obtain an error e which is processed by the PID controller block to produce a steering angle δ_s that the car must drive to follow the wall. Sometimes, this value can be larger than the achievable steering angle and the saturation block is used to limit this angle.

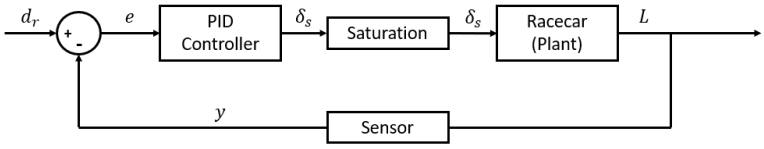


Figure 1: Wall follower block diagram.

To implement the aforementioned control system, the code must be laid out as in Figure 2. ROS uses a system of publishers and subscribers to send messages between the written code and the racecar itself. Therefore, to obtain LIDAR data from the car, a subscriber was used to obtain data from the `/scan` topic which provides `LaserScan` type messages from the Hokuyo LIDAR. It contains important data about the laser scan like the minimum scan angle value θ_{min}, the angle increment $\Delta\theta$, and the measured distance for each angle starting from minimum angle to maximum angle at the angle increment. For our car, the data had a minimum scan angle of $-135°$ and a maximum scan angle of $135°$ with the angle increment being chosen such that the `ranges` list of measured distances is of length 100.

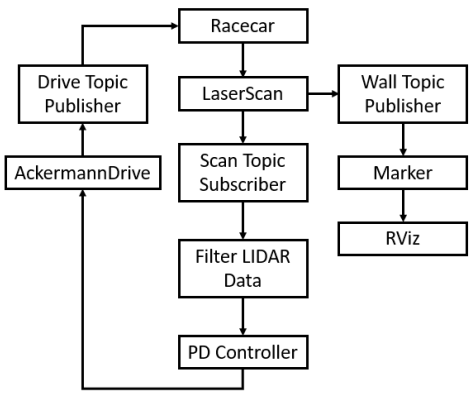


Figure 2: Code layout.

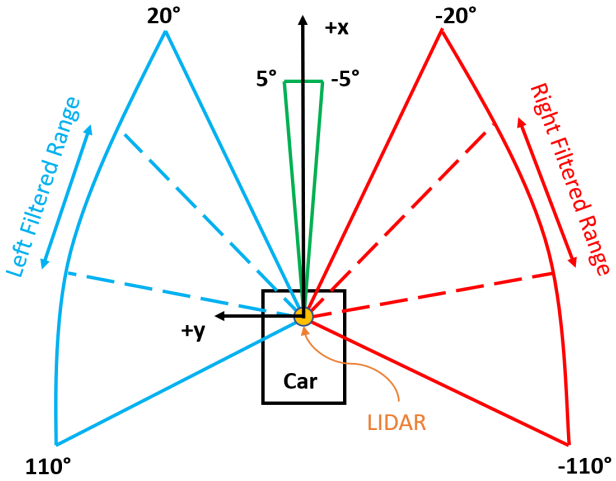


Figure 3: Laser scan lookout range selection. Since distances are measured from the LIDAR, the LIDAR is set to be the origin of the car's coordinate system. The positive x-axis points in the car's forwards direction, and the positive y-axis points to the left side of the car.

This data is then used to obtain a measurement of the distance from the side and front wall using Algorithm 1. Initially, it looks at whether it needs to follow the left or the right wall and selects the values of the measured distance from the LIDAR data in the ranges array depending on the lookout range. Figure 3 shows a representation of the lookout ranges being used in the code. According to the chosen reference frame, the x-axis points out of the front of the car and the y-axis to the left. Any angle taken anticlockwise with respect to the x-axis is positive. It can thus be seen that initial ranges of $20°$ to $100°$ and chosen for the left wall and $-20°$ to $-100°$ for the right wall. This can be converted into upper and lower bound indices in the ranges list using the equations in lines 8 and 9. However, this data is in a polar coordinates system and in order to obtain a measurement of the distance from the wall, these points must be converted into cartesian coordinates with respect to the LIDAR reference frame. This can be done by a simple transformation of coordinates given in lines 11 and 12.

Next, the standard deviation and mean of the x and y cartesian coordinate data are obtained after which these values are filtered if the difference between the x or y data point and its mean is greater than some constant times its standard deviation. This constant was found by tuning during experimentation. This allows for noisy data, that comes from the measured LIDAR distance being too large, to be removed providing better results. A linear regression is then run to obtain a line of best fit through rough data as the wall being measured might be uneven which might cause the commanded steering angle to change fast as the error would change very rapidly. Finally, the distance to the wall can be found by finding the length of the shortest line from the origin, since the LIDAR's location is assumed to be the origin, which intersects the line created by conducting linear regression on the wall's x and y coordinates. This line will be perpendicular to the line obtained by regression and using the general result that the product of the slopes of perpendicular lines is -1, the shortest distance to a line from the origin can be calculated using the equation in 23. Similarly, to obtain the front distance, the ranges list values from a lookout range of $-5°$ to $5°$ are averaged to obtain an accurate measurement.

The Wall Following node produces a δ_s with a PID controller whose parameters depend on the car's distance from the side and front wall. Algorithm 2 defines the procedure for the implementation of the PID controller which works on the basis of using a proportional, integral, and derivative gain in order to drive the error to 0. First, this error is calculated by subtracting the desired distance from the wall $D_desired$ from the sensed distance of the racecar to the wall d_{wall}. This error signal is then converted to a steering angle by multiplying the error by a proportional gain and summing the product of the integral gain with the integral term, which is the sum of the integral term from the past, along with the product of the derivative gain with the derivative term, which is the difference between the current and previous error. The time increment Δt was implicitly chosen by taking the difference between the time a drive command was published and the time a new sensor message was received. This way, the PID controller would always be working with new sensor data before computing the desired steering angle. Finally, a

check is applied to make sure that the maximum and minimum steering angle command is not exceeded in order to not damage the actuator.

Algorithm 1 Filtering Sensor Data

Require: DEG2RAD, ROUND, ARRANGE, COS, SIN, MEAN, ABS, POLYFIT, STD, ENUMERATE ▷ from `numpy`

procedure FILTER-SENSOR-DATA($scan$, $side$)
- $\Delta\theta \leftarrow scan.angle\_increment$
- $\theta_{min} \leftarrow scan.angle\_min$
- **if** $side = -1$ **then** ▷ When following the right wall
  - $b_{lower}, b_{upper} \leftarrow [-100, -20]$
- **else if** $side = 1$ **then** ▷ When following the left wall
  - $b_{lower}, b_{upper} \leftarrow [20, 100]$
- $i_{lower} \leftarrow \text{INT}((\text{DEG2RAD}(b_{lower}) - \theta_{min})/\Delta\theta)$
- $i_{upper} \leftarrow \text{INT}((\text{DEG2RAD}(b_{upper}) - \theta_{min}/\Delta\theta)$
- $values \leftarrow scan.ranges[i_{lower} : i_{upper}]$
- $index_{side} \leftarrow \text{ARANGE}(i_{lower}, i_{upper}, 1)$
- $x \leftarrow values * \text{COS}(\theta_{min} + index_{side} * \Delta\theta)$
- $y \leftarrow values * \text{SIN}(\theta_{min} + index_{side} * \Delta\theta)$
- $sdev_x \leftarrow \text{STD}(x)$
- $sdev_y \leftarrow \text{STD}(y)$
- $mean_x \leftarrow \text{MEAN}(x)$
- $mean_y \leftarrow \text{MEAN}(y)$
- **for** $index_y, val$ in ENUMERATE(y) **do**
  - **if** $\text{ABS}(val - mean_y) < 1.7 * sdev_y$ **and** $\text{ABS}(x[index_y] - mean_x) < 1.7 * sdev_x$ **then**
    - $filtered_x \leftarrow x[index_y]$
    - $filtered_y \leftarrow y[val]$
- $m, c \leftarrow \text{POLYFIT}(filtered_x, filtered_y, 1)$ ▷ fitting a line using filtered values
- $d_{wall} \leftarrow \text{ABS}(c/(m^2 + 1))$
- $i_{lower\text{-}front} \leftarrow \text{INT}((\text{DEG2RAD}(b_{lower}) - \theta_{min})/\Delta\theta)$
- $i_{upper\text{-}front} \leftarrow \text{INT}((\text{DEG2RAD}(b_{upper}) - \theta_{min}/\Delta\theta)$
- $d_{front} \leftarrow \text{MEAN}(scan.ranges[i_{lower\text{-}front} : i_{upper\text{-}front}])$
- **return** d_{wall}, d_{front}

Algorithm 2 Wall Following

Require: FILTER-SENSOR-DATA

procedure WALL-FOLLOWING($scan$, $side$, $D_{desired}$, e_{t-1}, Δt)
- $\theta_{min}, \theta_{max} \leftarrow [-0.34, 0.34]$
- $D_{front} \leftarrow 1.75$ ▷ A heuristic threshold
- $d_{wall}, d_{front} \leftarrow$ FILTER-SENSOR-DATA(scan) ▷ Filter out outliers
- **if** $d_{front} < D_{front}$ **then** ▷ Too close to front wall
  - $K_p \leftarrow 0.8$
  - $K_i \leftarrow 0.0$
  - $K_d \leftarrow 0.0$
  - $e_t \leftarrow -side * (0.25 * (D_{desired} - d_{sensed}) + 0.75 * (D_{front} - d_{front}))$
- **else** ▷ Normal controller case
  - $K_p \leftarrow 0.6$
  - $K_i \leftarrow 0.0$
  - $K_d \leftarrow 0.05$
  - $e_t \leftarrow -side * (D_{desired} - d_{sensed})$
- $P \leftarrow e$ ▷ proportional term
- $I \leftarrow I + e_t * \Delta t$ ▷ integral term
- $D \leftarrow (e_t - e_{t-1})/\Delta t$ ▷ derivative term
- $\delta_s = max(min(K_p * P + K_i * I + K_d * D, \theta_{max}), \theta_{min})$
- $e_{t-1} \leftarrow e_t$
- **return** δ_s

The parameters were initially tuned in the simulation environment using the Ziegler-Nichols tuning method where K_p was set to allow the car to oscillate to find the critical gain K_C and the oscillation period T_C after which the gains can be calculated using Equation 1.

$$\begin{aligned} K_p &= 0.6 \cdot K_C \\ K_i &= 2 \cdot K_P / T_C \\ K_d &= K_p \cdot T_C / 8 \end{aligned} \tag{1}$$

However, while testing with the racecar in reality, the integral gain seemed to cause oscillations when the car was following its setpoint while in a steady state which meant there was no need for this integral term and instead, a PD controller was used. Furthermore, during the simulation and in-person testing, it became evident that a naive implementation of a follower that only looks at the distance to the side wall will result in a non-robust racecar which cannot turn around corners well due to the way the linear regression is being used to calculate the measured distance from the side wall. Therefore, the PID controller algorithm checks when the car is below a certain threshold distance to the front wall and computes a new error term by weighting the error to the side wall and front wall by 25 % and 75 % respectively. This creates the illusion that the racecar has a larger error than it does in reality while approaching the front wall in a corner, allowing the car to turn seamlessly around the corner. Additionally, a higher proportional gain was chosen after tuning with no derivative or integral gain to allow for a large steering angle to be commanded. Another case that was challenging in simulation and needed to be looked at was turning an open corner. This was achieved by having a large lookout range so that car could lookout behind itself in order to still try and identify a wall.

After the calculation of δ_s, the car publishes a message of type `AckermannDriveStamped` which specifies a velocity and steering angle that the car needs to follow by abstracting and encapsulating the low-level control taking place on the car to make that happen.
WRITE ABOUT GAIN VALUES CHOSEN BETTER

2.2 Safety Controller

Ensuring a vehicle system can automatically detect any impending crashes and administer the appropriate safety precautions is an essential part of any autonomous robotic system to protect the car, the user, and the environment from any damage. The objective is to create a safety control system that guarantees the vehicle will not crash into any object in any scenario. At the same time, however, the safety controller must be implemented in a way such that it does not hinder the vehicle from executing other actions that are in fact safe (i.e. will not crash into any objects). These objectives are achieved by creating a danger region via the vehicle's parameters, obtaining distances from the LIDAR laser scan, and running a basic algorithm to determine via the laser scan data if there are objects within the danger zone and thus the safety module should be triggered. The safety module simply commands the vehicle's velocity to be zero, effectively stopping the car before it collides with the detected object. Figure 4 illustrates the safety controller's process.

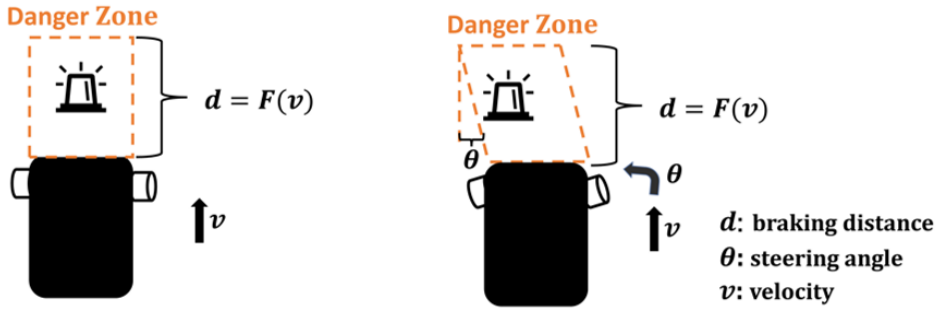


Figure 4: An illustration of the safety controller while moving forward (left) and turning (right).

2.2.1 Danger Zone

First, what is termed as the car's "danger zone" is constructed; that is, the region in front of the car such that if an object is detected inside this zone, the safety module should be executed. We choose to construct the danger zone to only be in front of the car because, for the purposes of this lab, the car will only be moving in the forward direction (not the reverse). The safety zone is constructed by calculating four coordinate points in the car's coordinate system (refer to Figure 3 which illustrates the car's coordinate plane). The general shape of the danger zone is a parallelogram whose area is a function of the car's width, current velocity, and current turning angle.
The first two base points of the danger zone are positioned at the front-left and front-right bumper of the car. We ensure the danger region is wide enough to cover the entire front width of the car, including its tires. We obtained the measurements using a ruler starting from the LIDAR which reads distances in meters. This is done because the distance measurements used to detect objects in the environment will come from the LIDAR; therefore, we set the LIDAR as the origin of the car's coordinate plane. To calculate the remaining two points of the danger zone corresponding to the depth and angle of the parallelogram, we utilize the car's currently commanded velocity and steering angle. As the car's velocity increases, the braking distance of the car increases because more time is needed to decelerate the car to a stop; thus, the depth of the parallelogram is written to be a function of the car's velocity. We rely on experimentation to determine this function due to the complexity associated with precise calculations. In short, we simply obtain the associated braking distance needed at various velocities to safely stop the car before colliding with an object, and run a regression to interpolate between the obtained data points. This data collection process is elaborated in detail under the Experimental Evaluation of the Safety Controller in Subsection 3.2 (see Graph **??**). //
As the car turns, the region it should detect objects in should change to better reflect the path the car will actually take in traveling. For instance, a car that is turning to the left should look in the area more towards the left. It should not stop for objects that are for example in the direction directly forwards of the car, as the car will be able to avoid hitting them because it is turning left. Thus, we also use the car's current steering angle to add a horizontal bias to the two depth points of our danger region, allowing the region to bend left or right by a magnitude related to the magnitude of the steering angle. This allows the car to still execute safe, collision-free turns that are tight to the wall yet still be able to stop the car appropriately in instances where the car recognizes it may hit the wall because it does not have enough space to safely conduct a turn. Figure **??** illustrates how the angling of the danger zone can accomplish such tasks. Thus, the locations of the danger region's vertices can be summarized by the following equations (illustrated previously in Figure 4):

$$leftbase = (lidar\_to\_front, car\_width/2)$$
$$rightbase = (lidar\_to\_front, -car\_width/2)$$
$$lefttop = (lidar\_to\_front + braking\_dist, car\_width/2 + braking\_dist * tan(current\_steer\_angle)$$
$$righttop = (lidar\_to\_front + braking\_dist, -car\_width/2 + braking\_dist * tan(current\_steer\_angle)$$

2.2.2 LIDAR Data

The safety controller then obtains data from the laser scan, utilizing the perceived distances of surrounding objects in the environment. Since our danger zone is set up to be in front of the car only, we obtain the distances corresponding to angles that are on the positive-x side of the car's coordinate plane (corresponding to $-\pi/2$ to $\pi/2$). These distances are then converted to cartesian coordinates via polar-to-cartesian conversions which will then be used to determine an object's presence in the danger zone:

$$x = distance \cdot cos(\theta)$$
$$y = distance \cdot sin(\theta)$$

2.2.3 Object Detection Algorithm

With the danger zone constructed and the distances obtained in cartesian form from the LIDAR, a simple algorithm essentially detects whether or not an object may be present in the danger zone and thus execute the safety module. We iterate through all the LIDAR points to see whether or not each point is located inside the danger zone, defined by the four calculated coordinate points. We utilize the ray casting algorithm **??** which essentially tests the number of times a ray emanating from a test point crosses the edges of a polygon to determine whether or not the test point is located inside the polygon.

When a test point has been located inside the danger region, the algorithm begins counting the number of successive points that are also in the danger region. If the algorithm has detected three successive points in the safety region, it immediately activates the safety module, commanding the vehicle a velocity of zero to stop the car before it collides with the object. A threshold of three successive points is used because it more accurately describes if a surface (spanning over $3 * angle\_increment$ radians from the LIDAR's perspective) is in the danger zone. This makes the object-detection algorithm more robust to potential noise that may be presented through the LIDAR (that it is more certainly an actual object surface) but also is quick to execute the safety module if needed (instead of needing to check every single point, it activates the safety module as soon as an eligible surface is detected). Figure _ illustrates this algorithm.

2.2.4 Publishing Safety Commands

To successfully obtain the necessary data and publish the safety module to the car, the safety controller intercepts the current driving commands via the topic `/vesc/high_level/ackermann_cmd_mux/output` and publishes any safety executions to `/vesc/low_level/ackermann_cmd_mux/input/safety`. The safety controller is able to work due to the racecar's command mux which has different levels of priority. The safety topic is a higher priority topic than the navigation topic (where the wall-follower commands are published to), so when the safety module is executed, the safety command to set the car's velocity to 0 and stop the car overrides any command the wall-follower attempts to publish, thus allowing the car to safely stop in time before colliding with the perceived object.

Algorithm 3 Safety Controller

```
1:  procedure SAFETY-CONTROL(scan, ack, v)
2:      δ ← ack.drive.steering_angle
3:      D_lidar ← 0.1524          ▷ The distance between lidar and the front edge of the car (m)
4:      W_car ← 0.33                                              ▷ The width of the car (m)
5:      θ_view ← π                                  ▷ The whole front view taken into consideration.
6:      N_point ← 3               ▷ Stopping threshold when counting obstacle points in a streak.
7:      n_point ← 0
8:      p_0, p_1, p_2, p_3 ← CONSTRUCT-DANGER-ZONE(D_lidar, W_car, δ)
9:      x, y ← CARTESIAN-CONVERTER(scan, θ_view)
10:     for x, y do
11:         if POINT-INSIDE-DANGER-ZONE(x, y, p_0, p_1, p_2, p_3) then
12:             n_point ← n_point + 1
13:             if n_point ≥ N_point then
14:                 break
15:         else
16:             n_point ← 0
17:     if n_point ≥ N_point then
18:         new_ack ← CREATE-ACKERMANNDRIVE-MESSAGE(δ)
```

2.3 ROS Implementation

An overview of our system can be shown in Figure

3 Experimental Evaluation

To ensure an effective Wall Follower and Safety Controller, a series of tests were designed to verify the functionality of the design. The tests were centered around creating controlled, reliable, and safe mobility for frequent and necessary autonomous vehicle maneuvers. Each test was repeated three times to guarantee robustness and error was recorded to evaluate success.

3.1 Wall Follower

To guarantee the safety of the car, create uniformity among the tests, and due to the limited space in our testing environment, we chose a constant velocity of $0.5ms^{-1}$ for the majority of the Wall Follower

test cases.

Table 1: Wall Follower Test Cases and Results

Priority Score	Test Type	Test Description	Measurement of Success
1	Straight Wall	Align the car parallel to wall, let it run for 5 seconds	The abs(error) should be <0.05m.
1		Align the car parallel to wall, offset by -50% and 50% from desired distance.	Time it takes for robot to converge to 0 error (+/- 0.05m) should not exceed 0.5 seconds. Maximum error should not exceed original offset.
2		Align the car head direction at -45° and 45° compared with the wall, and from twice the desired distance.	
1	Corners	Align the robot parallel to the wall facing a 90° closed corner. Run until the robot has turned the corner and no longer oscillates.	Time it takes for robot to converge to 0 error (+/- 0.05m) after peak error should not exceed 1 second.
1		Align the robot parallel to the wall 3m away from a doorway. Stop after the robot has travelled through the doorway.	
3	Cluttered Environment	Run the robot along an unsmooth wall in the lab classroom. Run until failure.	Robot should successfully navigate most obstacles in cluttered environment. Record limitations.

3.1.1 Straight Wall Test

Throughout this section, we set our set-point/goal value to be 0 for our error. We have an epsilon of $\epsilon = +/-5\%$ defined to be an acceptable range of our error values.

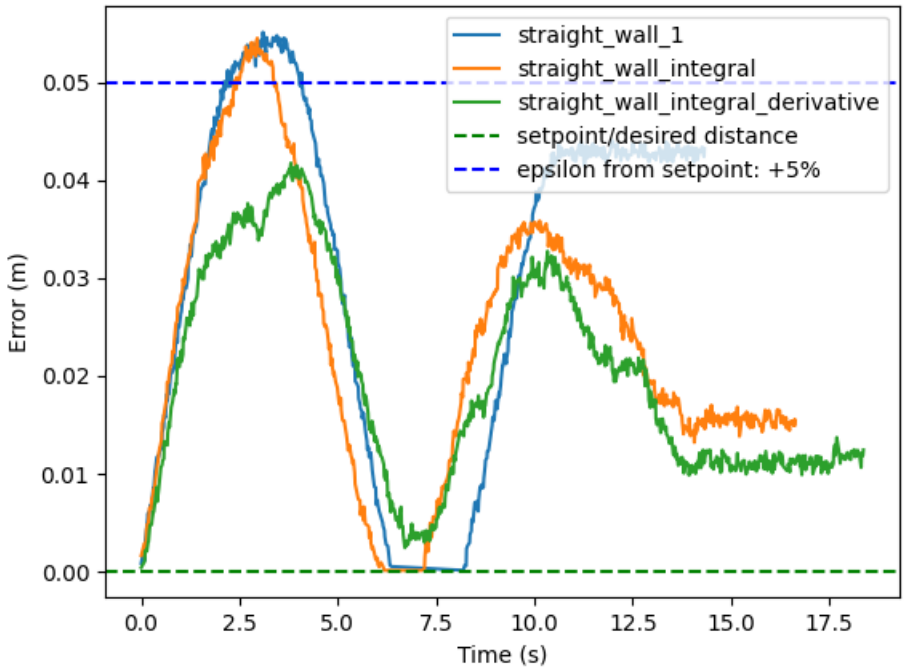


Figure 5: Straight wall test 1 - starting at the desired distance with different PID parameters.

Desired Distance Test For our first experiment, we decided to test whether our car could follow the straight wall starting at the desired distance from the wall. Moreover, we studied how different values for our control impacted our computed error. The values used for kp, ki, and kd are shown in the table

below. In the figure, we can observe that the error converges to a large value and this is due to the lack of control from our integral term. We introduced the integral value into our testing after observing this behavior, so that we could have better control over our steering angle and how much we overshoot. Increasing the integral term and our derivative term helped us control how much we converged.
(TODO: figure out a better legend, maybe just use the param value triple?) (Params used for test 1.1) kp=0.6 ki=0.0 kd=0.05 kp=0.6 ki=0.001 kd=0.05 kp=0.6 ki=0.005 kd=0.1

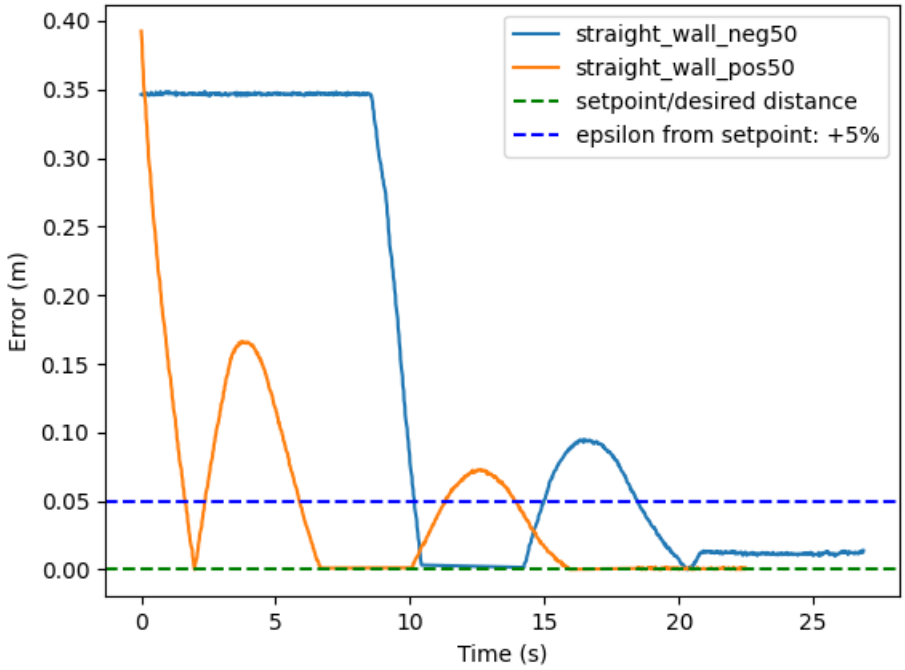
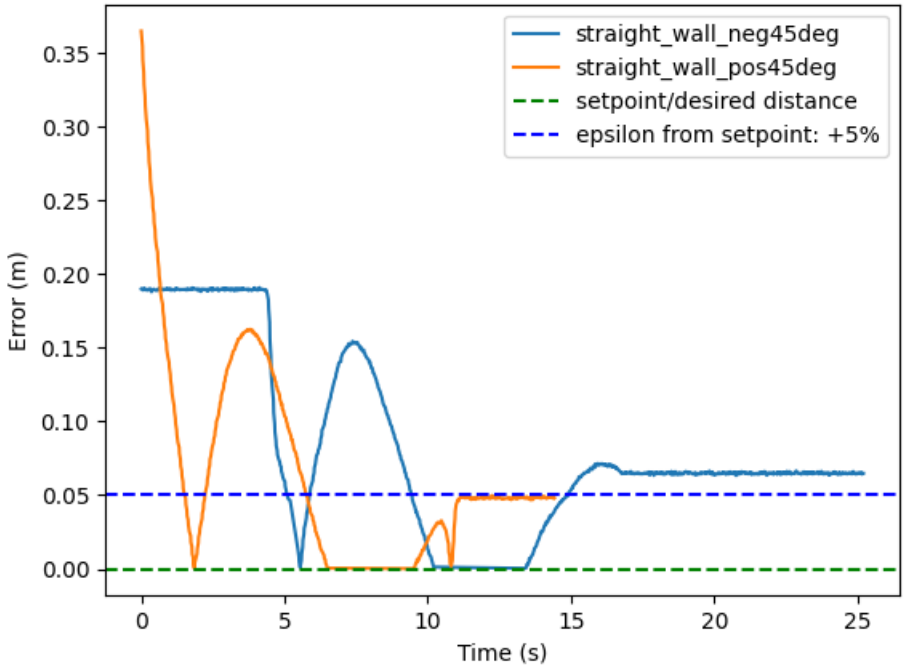


Figure 6: Straight wall test 2 - starting at different distances from the wall.

kp=0.6 ki=0.0 kd=0.05 Let starting distance from the side wall: $s_0 = 25\text{cm}(-50\%), 75\text{cm}(50\%)$ Given that s_0 from our wall being greater than our desired distance, the behavior in the graph exhibits readjustments to our robot's position and a convergence towards a smaller error term once we reposition ourselves to be closer to our setpoint.



$s_0 = 100cm$ kp=0.6 ki=0.0 kd=0.05 The angle of our robot is +/- 45 degrees in relation to our wall. We can observe that this allows us to minimize our error and stay our close to our desired distance.
This concludes our first section of tests, we have observed that we are able to drive straight along a wall without crashing.

3.1.2 Corner Tests

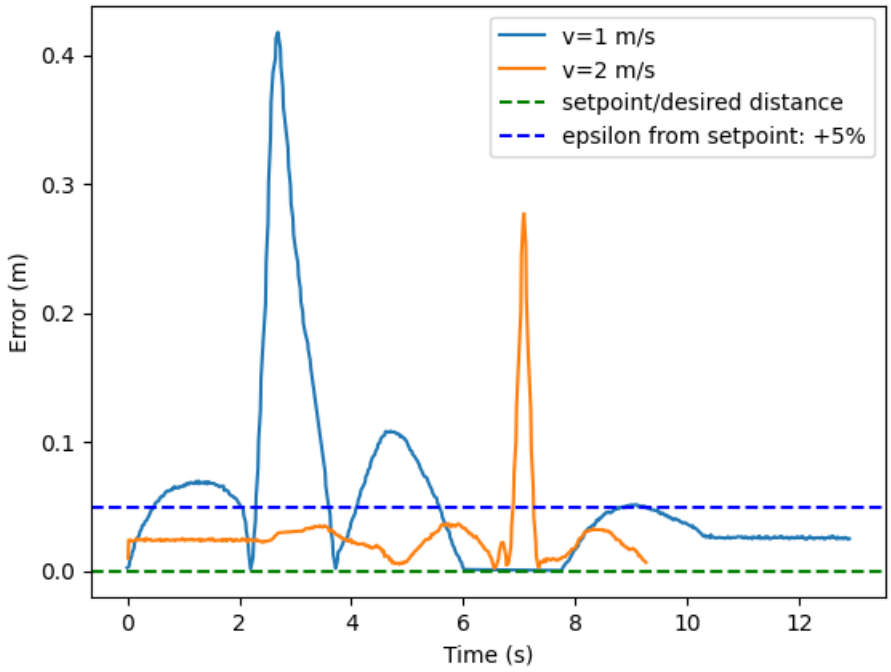


Figure 7

(TO-DO: Dfront = reacting distance) (In our code, if we found our front distance less than reacting distance, we would start to take the front distance error term into account, which hopefully drives the car to turn.) Currently, our reacting distance is a constant (1.75 m) and as we increased the speed of our robot we found it was more difficult to turn accurately at a higher velocity. Our reacting distance should be a function of velocity because higher speeds result in wider turns and larger turning radiuses, so we must account for this with a higher reacting distance so that we can handle turning at higher speeds.

closed 90deg corner kp=0.6 ki=0.0 kd=0.05, velocity = 1.0m/s kp=0.6 ki=0.0 kd=0.05, velocity = 2.0m/s kp=0.6 ki=0.0 kd=0.05, velocity = 3.0m/s kp=0.6 ki=0.0 kd=0.05, velocity = 3.0m/s

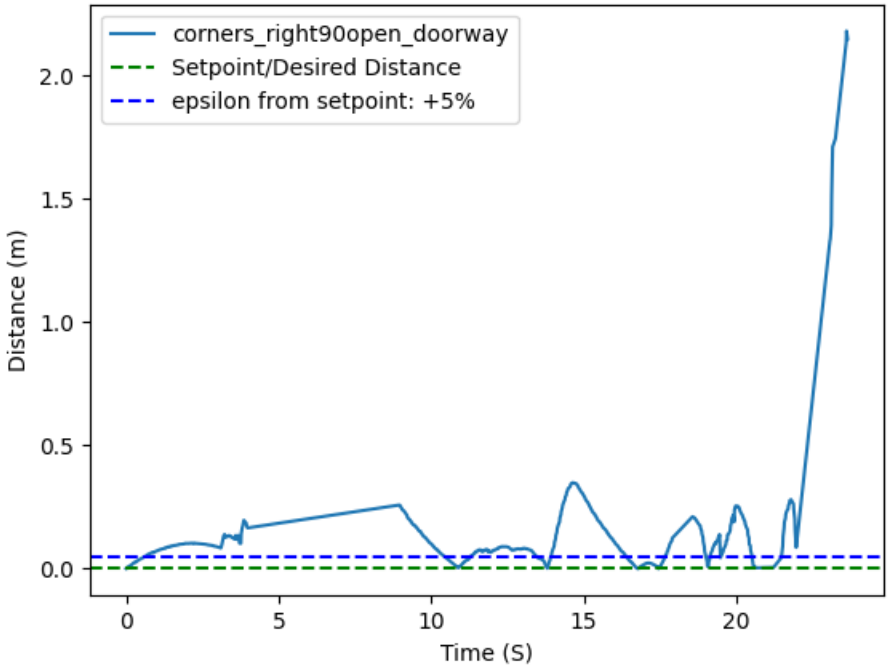


Figure 8: Caption

3.1.3 Cluttered Environment Tests

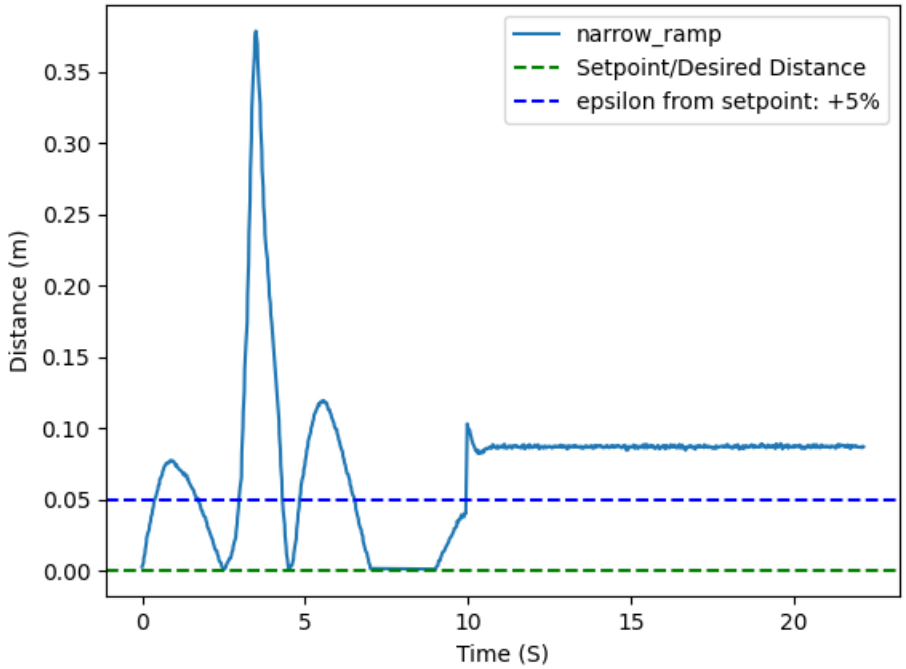


Figure 9: Caption

3.2 Safety Controller

Our initial testing for the safety controller focused on creating a function dependent on the car's velocity to calculate the breaking distance. This is vital to implement a robust safety controller that is effective at different velocities. Assuming there is no lag time from data processing and autonomous control, the minimum breaking distance (D_{min}) is given by Equation 1 where V is the car's velocity and d is the car's deceleration.

$$D_{min} = \frac{V^2}{2d} \tag{2}$$

While we determined it was not realistic to measure the deceleration, we can extrapolate the quadratic relationship between the breaking distance and the velocity. We measured $\frac{1}{2d}$ experimentally by determining the ideal D_{min} at different velocities. This experiment was conducted by placing the autonomous car 0.5m from a wall. The autonomous code is then run at a given velocity, with the braking distance decreased by 0.01m each test, until a tolerance of 0.05m was achieved. This experiment was repeated for velocities in range[0.5, 3] at increments of 0.5 as seen in Graph 10. After determining a quadratic regression with low variance ($R^2 = 0.9985$), we implemented the equation into our code to ensure the safety controller operated effectively at multiple velocities.

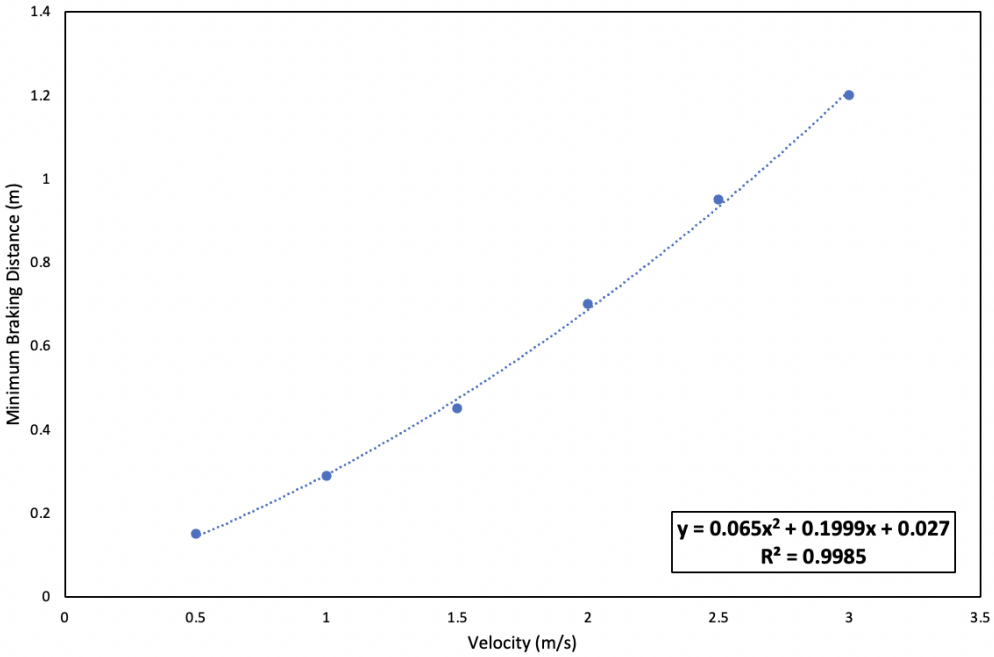


Figure 10: Braking distance at different velocities.

Table 2: Safety Controller Test Cases and Results

Priority Score	Test Type	Test Description	Measurement of Success	Result
1	Non-Impedance of Wall Following	Run all Wall Follower test with the safety controller implemented	None of the tests below $2ms^{-1}$ should fail	Pass
2	Dynamic	For all brick and chair tests run, the brick or chair should be moved out of the way of the robot after it successfully stops. (all at velocity=1.0)	After the brick is removed the robot should resume its path.	Pass
1	Different Object Types	The car should safely stop on a variety of different object types of different sizes, such as a char leg, a brick, a human, a wall, and a cone.	100% stop rate	Pass
1	Different Car Velocities	The Safety Controller should be properly executed when the car is running at different velocities.		Pass
3	Different Desired Distance	The car should safely go and stop with a large and small desired distance.		Fail
2	Turning	The safety controller is robust to objects that may be found while turning.	100% stop rate if object will impede (at higher velocities won't turn)	Pass

You can find ideas and suggestions in the "Good Experimental Evaluation" Recitation on Canvas (Modules section).

(no more than 1250 words)

3.3 Suggestions for evaluation

1. qualitative properties wall follower: speed, reliability, ...; safety controller
2. quantitative
3. understand when the system works and when it doesn't.
4. ablation study
5. different problem contexts
6. different algorithm modules

4 Conclusion

Summarizes what you have achieved in this design phase, and notes any work that has yet to be done to complete this phase successfully, before moving on to the next. May make a nod to the next design phase.

(no more than 500 words)

5 Lessons Learned

Aaron:
Jess:
Chuyue:
Nihal: The most important lesson I have learned completing this lab on the technical side of things is to have a plan of action before starting the experimentation for the lab. There is a lot of data that we could have collected including videos of working and failed tests which we couldn't because we didn't think about our testing procedure thoroughly. Also, having made a meeting notes document with all the important information there was a good decision. In terms of communication, we can split the times that

we work on the car better by understanding when people are free a little better. Otherwise, I personally feel like the team has put in all their effort to make meetings and to get the lab completed.
Shara:

6 Individual Contributions

Although the project was largely a collaborative effort, team members took the lead for the following components of the project:

7 References