RSS Lab 3:
Wall Following and Safety Controller

RSS Team 9: SDWCH
William Chen, Henry Hu, Christina Jung, Silvia Knappe, Diego Mendieta

March 6, 2020

Contents

1 Introduction **2**

2 Technical approach **2**

2.1 Wall detection and following 2

2.1.1 Wall pose estimation via LIDAR scan linear regression . . 3

2.1.2 Wall following via pure pursuit controller 5

2.2 Safety controller design 8

2.2.1 Tuning parameters in the safety controller 9

2.2.2 ROS messaging implementation 10

3 Experimental evaluation **11**

3.1 Wall detection and following 11

3.2 Safety controller 13

4 Conclusion **14**

5 Lessons Learned **15**

1 Introduction

Author: Silvia Knappe

In Lab 3, we were tasked with getting to know our robot, as well as implementing 2 main deliverables: a wall following controller and a safety controller. The wall following controller should ensure that the robot constantly moves a set velocity forwards while staying a desired distance away from a wall. While wall following, the robot should be able to turn corners while staying at the desired distance from the wall, as well as follow uneven walls without much oscillation. We set out to create a wall follower that is robust to noise, minimizes oscillations, and is able to turn around both convex and concave corners.

For the safety controller part of this lab, we were tasked with creating a safety controller that overrides any autonomous command to the robot that would cause it to crash. For our implementation of the safety controller, we wanted our robot to be able to detect and stop in the presence of any obstacles, including small obstacles like chair legs. We also sought out to create a robust safety controller, so we also wanted the safety controller to consider the velocity at which the robot was moving.

Combining wall following and the safety controller would allow our robot to move around an unknown static environment while also avoiding crashes.

While getting to know the robot was not a deliverable goal for this lab, it was a prerequisite to being able to implement the wall follower and safety controller. Ensuring that each teammate was able to connect to the robot, as well as execute programs on the robot made the process of implementing the deliverables for this lab much smoother.

2 Technical approach

2.1 Wall detection and following

Author: William Chen

The wall following task can be broken into two distinct parts. First, the robot must detect the location of nearby walls. This detection method must be able to handle extraneous sensor data from both irrelevant environmental objects and inherent sensor noise. Second, the robot must generate and execute the control commands that allow it to maneuver next to said walls at a desired distance and velocity. The controller should ideally be able to get the robot to move to the correct trajectory from a range of different starting poses, even when these poses are far from the desired path and orientation of the robot.

2.1.1 Wall pose estimation via LIDAR scan linear regression

We decided to use our robot's Hokuyo UST-10LX LIDAR as the principal exteroceptive sensor for wall detection. After isolating the relevant sensor data, we then performed least-squares analysis in order to provide an estimate of wall pose with respect to our robot.

The LIDAR itself contains an internal rotating laser rangefinder that intermittently takes distance measurements as it spins. Each cycle, the measured data is stored and published as a LaserScan ROS message to the /scan topic. This message type includes several pieces of vital information:

angle_min and angle_max The start/end angles of a single LIDAR scan cycle, respectively. The front of the LIDAR is at 0 radians. Our LIDAR's scans begin at $-\frac{2\pi}{3}$ to the right and rotate counterclockwise to $\frac{2\pi}{3}$ to the left. Assume all angles are in radians, unless otherwise stated.

angle_increment The difference in angles between consecutive range measurements.

ranges A list of successive measured ranges in a scan cycle of length N. The angle θ_n corresponding to the range element at index n can be calculated as follows:

$$\theta_n = \text{angle\_min} + n * \text{angle\_increment} \tag{1}$$

where $n \in \{0, 1, 2, ...N-1\}$ where the first $(n = 0)$ range corresponds to angle_min and the last $(n = N-1)$ corresponds to angle_max.

Before performing any calculations on the LIDAR scan data, certain measurements are removed via two criteria. Firstly, if any ranges exceed a certain amount, they are excluded. Said points likely would come from background obstacles rather than the wall the robot is currently following, so they would only act as extraneous noise when estimating wall poses. Secondly, as we only wanted the robot to follow walls on one specified side at a time, some of the scan measurements from the other side are deleted. If wall following on the right, only points with angles in the range

$$\theta_n \in [\text{angle\_min},\ \text{angle\_min} + \Delta\theta] \tag{2}$$

are used. For the left, the angles are

$$\theta_n \in [\text{angle\_max} - \Delta\theta,\ \text{angle\_max}] \tag{3}$$

where $\Delta\theta$ is the difference between the maximum and minimum non-excluded ranges

$$\Delta\theta = F(\text{angle\_max} - \text{angle\_min}) \tag{4}$$

and F is the tunable fraction of total angles used. We chose to have $F > \frac{1}{2}$ so that some of the points in the front would be considered. That way, walls in front of the robot are detected and considered, meaning the car can turn to avoid collision and follow walls correctly.

After this preprocessing is completed on a scan cycle's data, all range measurements are converted to Cartesian coordinates. The conversion is defined by the following equations:

$$x_n = \text{ranges}[n] * \cos(\theta_n) \tag{5}$$

$$y_n = \text{ranges}[n] * \sin(\theta_n) \tag{6}$$

(Unless otherwise stated, assume all coordinates are in the robot's reference frame.)

Finally, least-squares line-fitting is performed on the scan points to estimate wall pose. Least-squares tries to find the line that minimizes the total square distance from it to all the data points. The problem can be represented as finding the vector $\mathbf{v}$ that minimizes the magnitude of the error vector ϵ, defined as follows:

$$\epsilon = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} - \begin{bmatrix} x_0 & 1 \\ x_1 & 1 \\ \vdots & \vdots \\ x_{N-1} & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \mathbf{y} - \mathbf{A}\mathbf{v} \tag{7}$$

Said optimal $\mathbf{v}$ (called $\hat{\mathbf{v}} = \begin{bmatrix} \hat{m} \\ \hat{b} \end{bmatrix}$) can be calculated as follows:

$$\hat{\mathbf{v}} = \arg\min_{\mathbf{v}} ||\mathbf{y} - \mathbf{A}\mathbf{v}||^2 = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y} \tag{8}$$

or

$$= \left(\begin{bmatrix} x_0 & x_1 & \dots & x_{N-1} \\ 1 & 1 & \dots & 1 \end{bmatrix} \begin{bmatrix} x_0 & 1 \\ x_1 & 1 \\ \vdots & \vdots \\ x_{N-1} & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} x_0 & x_1 & \dots & x_{N-1} \\ 1 & 1 & \dots & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} \tag{9}$$

In practice, we created numpy arrays for $\mathbf{A}$ and $\mathbf{y}$, then used numpy.linalg's lstsq() function to perform the above calculations efficiently. The resulting line of best fit, $y = \hat{m}x + \hat{b}$, acted as our estimation of wall pose in the robot's reference frame. On the actual robot, the above process is repeated at each time step (i.e. each time the LIDAR completed a scan cycle), which allows the robot to have near-real time estimates of nearby walls' poses.

This type of wall detection works well at both detecting corners and ignoring minor noise/wall roughness. When the robot approaches a corner, the resulting line of best fit goes through both the walls that form this intersection. The line acts as a ”virtual wall” that, when followed by the robot, allows it to turn appropriately. This least-squares method thus displays the robustness needed for wall following.

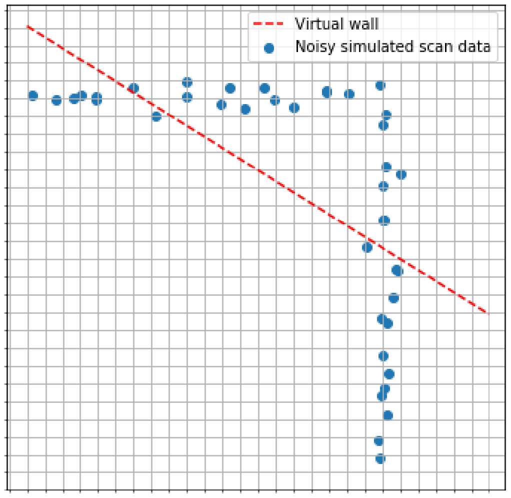

Figure 1: Least-square wall estimation performed on simulated laser scan points (with added Gaussian noise). This data is similar to what the robot would detect when near a corner. The detected ”virtual wall” does not correspond with a physical wall. However, if the robot detected and followed such a wall, the net result is the robot turning and avoiding collisions with the corner. This behavior further explains the motivation behind why laser measurements in the front of the robot are necessary.

2.1.2 Wall following via pure pursuit controller

We implemented a pure pursuit controller to generate drive commands needed for wall following. Said controller simplifies the robot’s dynamics to a bicycle model. From there, one can calculate steering angles that allow the robot to move along an arc connecting its present location to a reference point on some desired path at a set speed v.

Initially, the controller creates a desired path by computing a line that is both parallel to and some desired distance away from the previously-estimated wall line. If the robot were to follow said path, then it would consistently stay the desired distance away from the wall, making this method a suitable way of generating wall following trajectories. Mathematically, given a line of the form $y = \hat{m}x + \hat{b}$, a line that is parallel to and a desired distance d away from it would have the equation

$$y_{\text{desired}}(x) = \hat{m}x + \hat{b}' \tag{10}$$

$$\hat{b}' = \hat{b} \pm d\cos(\theta) \tag{11}$$

where $\theta = \arctan(\hat{m})$ is the angle the wall makes with the robot frame's positive x-axis. Moreover, the sign of the term in $\hat{b}'$ is determined by whether the robot is following a wall on the left or right. Left and right correspond to $-$ and $+$ respectively.

The controller then tries to find a reference point on the path (located at the point $r_{\text{ref}} = (x_{\text{ref}}, y_{\text{ref}})$) that is a certain "look-ahead distance" (L_1) in front of the car's back wheel axle. The origin of the robot reference frame is defined to be at the LIDAR, meaning the coordinates of the back wheel axle are

$$r_{\text{back axle}} = (0, -L_{\text{LB}}) \tag{12}$$

where L_{LB} is the positive distance between the center of the LIDAR and the center of the back axle. To find the reference point, then, one has to find the intersection coordinates of the desired path and a circle of radius L_1 centered around $r_{\text{back axle}}$. That is, $r_{\text{ref}} = (x_{\text{ref}}, y_{\text{ref}})$ must satisfy:

$$L_1^2 = (x_{\text{ref}})^2 + (y_{\text{ref}} + L_{\text{LB}})^2 \tag{13}$$

and

$$y_{\text{ref}} = \hat{m}x_{\text{ref}} + \hat{b}' \tag{14}$$

Solving the system of equations, r_{ref} has the following closed form solution:

$$x_{\text{ref}} = \frac{-(\hat{m} * \hat{b}') + \sqrt{(\hat{m} * \hat{b}')^2 + 4(\hat{m}^2 + 1)(L_{\text{LB}}^2 - L^2 - \hat{b}'^2)}}{2(\hat{m}^2 + 1)} \tag{15}$$

$$y_{\text{ref}} = y_{\text{desired}}(x_{\text{ref}}) = \hat{m}x_{\text{ref}} + \hat{b}' \tag{16}$$

where $L \approx .325$ m is the distance from the front wheel axis to the back. Note that, usually, the look-ahead circle and the desired path line should intersect at two locations. However, we only consider the intersection point with the higher x-value, as that corresponds to an intersection in front of the car. As the car only drives forward, that is the only reference point utilized. We chose L_1 to be a linear function of the robot's velocity. This relation was motivated by the fact that, at higher velocities, it would be harder for the robot to make sharp turns (as there were limitations to the steering angles the robot could

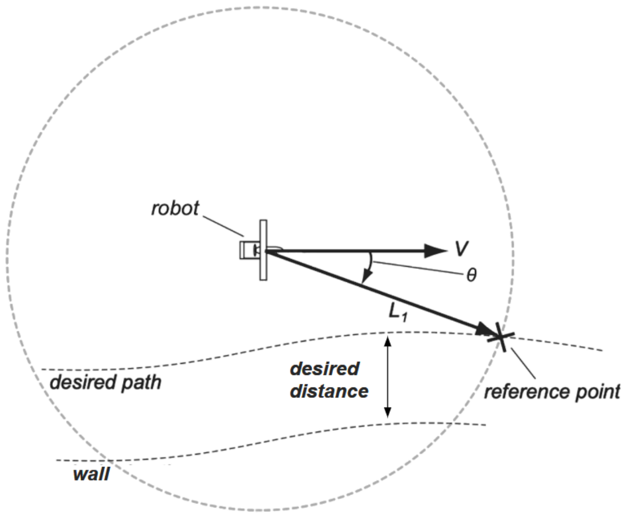

Figure 2: Illustration of pure pursuit controller. The desired path is parallel to the wall and the desired distance away. The outer circle shows the look-ahead radius L_1. Shown also are the reference point and the heading angle θ. Diagram adapted from RSS 2020 lecture 6.

achieve). Therefore, the aforementioned arcs the robot would travel upon at high speeds would require larger radii, and thus the look ahead distance should also be greater. We defined a proportional gain term K to determine L_1 based on robot speed:

$$L_1 = Kv \tag{17}$$

Lower K values result in the robot getting to the desired path faster, but also result in more oscillations (as the shorter L_1 values cause the robot to make sharper turns). Higher values do lower oscillatory behavior, but also, since the robot looks ahead further, it takes longer to converge to the desired path. Rough path shapes for these two cases are shown in figure 3. In the case that L_1 is too small and no reference point can be found, L_1 is instead marginally incremented until it reaches the desired path.

Once the reference point is found, the necessary steering angles (δ) can be calculated using the following equation:

$$\delta = \arctan\left(\frac{2L\sin(\theta)}{L_1}\right) \tag{18}$$

where θ is the heading angle from the robot's x-axis to the reference point:

$$\theta = \arctan\left(\frac{y_{\text{ref}}}{x_{\text{ref}}}\right) \tag{19}$$

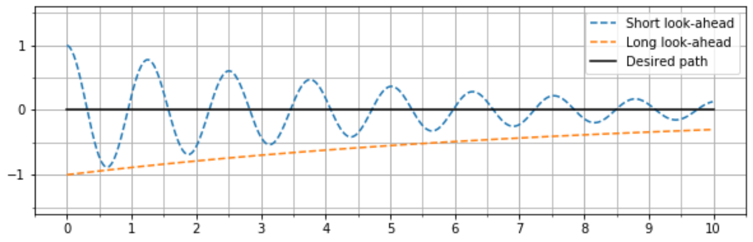

Figure 3: Rough path shapes for long and short look-ahead distances. Shorter distances result in oscillations, effectively being underdamped. Longer distances result in slow path convergence, essentially being overdamped.

Each time the LIDAR completes a scan, the wall detecting algorithm (2.1.1) runs and the corresponding steering angle is computed. The commanded angles are then stored in an AckermannDriveStamped ROS message and published to the topic /vesc/ackermann_cmd_mux/input/navigation. There, it is received by the robot's pre-built computational nodes and converted to low-level motor commands, which the robot executes. Repeating this process iteratively allows the robot to constantly traverse along paths a distance d away from any perceived walls, resulting in wall following behavior.

2.2 Safety controller design

Author: Henry Hu

The purpose of our safety controller is to prevent the robot from autonomously navigating into environmental obstacles such as walls, chairs, and people's legs. Because the controller needs to prevent crashes in a variety of situations, with both anticipable and sudden obstacles, we decided to implement the simplest controller that still performed well.

Our overall approach in the safety controller is to react to laser scan messages by issuing a braking command whenever an obstacle is detected within a critical distance. The critical distance is determined formulaically based on the velocity of the robot and the angular position of the obstacle.

A detailed description of the safety controller algorithm follows. First, we reduce the field of view of the robot to only consider its front. We then split up the field of view into a large number of angle sections, each of which is evaluated separately when finding obstacles. For each angular section, we average the laser scan ranges. Because of the small angle approximation, this is a good option for eliminating noise and obtaining consensus from the scans in a given angle sector. We also calculate the critical distance for that section using the formula:

$$\texttt{critical\_distance} = c + K \cdot v \cdot \cos\theta \tag{20}$$

$$\begin{aligned} c &= \text{baseline critical distance} \\ K &= \text{braking gain} \\ v &= \text{velocity (obtained from most recent navigation message)} \\ \theta &= \text{angle offset of the section from the robot's front} \end{aligned}$$

If the average range of the laser scan in any region is closer (less) than the critical distance for that region, a braking message is sent to the appropriate ROS topic (discussed in ROS messaging implementation). A braking message is an ordinary drive message containing the instruction to drive at zero velocity and at the last-detected steering angle.

As noted in equation 20, we use the velocity obtained from the most recent navigation message to calculate critical distance. Our robot does not contain an IMU, so we were not able to use sensor data to determine the actual velocity of the robot. This leads to suboptimal braking, as the robot can only react to the commanded velocity rather than its actual velocity. In future iterations, we look forward to incorporating IMU data.

One final detail of the implementation is the multithreaded aspect of the software. To keep track of the most recent navigation velocity, as well as the most recent steering angle, the safety controller exposes two global variables to a ROS subscriber model. When the subscriber receives a navigational message, it updates the shared data values via a callback function. When the safety controller requires these values, it reads the most up-to-date version. The ROS architecture guarantees that we do not experience issues with the interleaving of these two processes.

2.2.1 Tuning parameters in the safety controller

Our safety controller as described above contains four tunable parameters: the angle range to monitor, the number of angular sections used, the baseline critical distance, and the braking gain on critical distance.

We estimated the angle range to monitor by observing the angle of the lidar to the front left and right corners of the robot. We determined that the angle range to monitor would be from the angles $-\frac{\pi}{3}$ to $\frac{\pi}{3}$, where an angle of 0 denotes directly forward.

We measured the baseline critical distance by holding a posterboard around the robot and observing the laser scan data. We determined that the robot would be considered "crashed" if it was within 20 centimeters of the LIDAR. We used this value for the baseline critical distance.

The remaining two parameters were tuned empirically, by observing the robot's behavior at different levels of the parameters. To minimize risk to the robot, we tuned for correctness at lower velocities first ($\approx 0.5\frac{m}{s}$), then at higher velocities ($\leq 3\frac{m}{s}$).

2.2.2 ROS messaging implementation

To ensure that the safety controller could be used together with another autonomous driving controller such as the wall follower, we took advantage of existing infrastructure on the robot, shown in the figure.

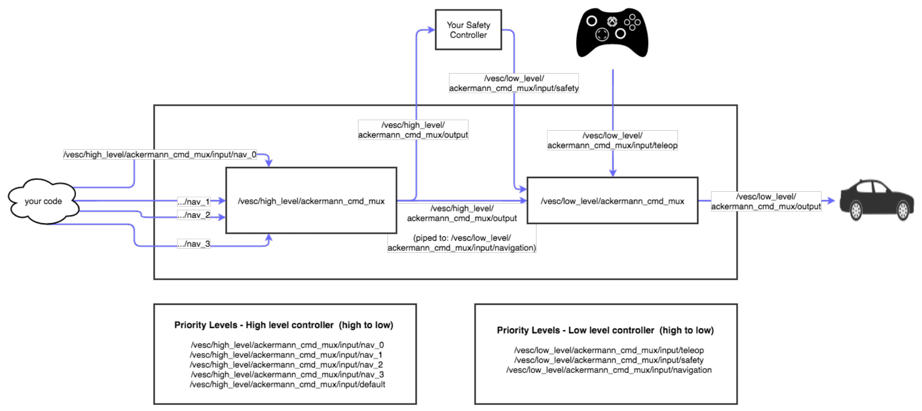
Your Safety Controller
/vesc/low_level/ ackermann_cmd_mux/input/safety
/vesc/high_level/ ackermann_cmd_mux/output
/vesc/low_level/ ackermann_cmd_mux/input/teleop
/vesc/high_level/ackermann_cmd_mux/input/nav_0
your code
.../nav_1
.../nav_2
.../nav_3
/vesc/high_level/ackermann_cmd_mux
/vesc/high_level/ ackermann_cmd_mux/output
(piped to: /vesc/low_level/ ackermann_cmd_mux/input/navigation)
/vesc/low_level/ackermann_cmd_mux
/vesc/low_level/ ackermann_cmd_mux/output
Priority Levels - High level controller (high to low)
/vesc/high_level/ackermann_cmd_mux/input/nav_0
/vesc/high_level/ackermann_cmd_mux/input/nav_1
/vesc/high_level/ackermann_cmd_mux/input/nav_2
/vesc/high_level/ackermann_cmd_mux/input/nav_3
/vesc/high_level/ackermann_cmd_mux/input/default
Priority Levels - Low level controller (high to low)
/vesc/low_level/ackermann_cmd_mux/input/teleop
/vesc/low_level/ackermann_cmd_mux/input/safety
/vesc/low_level/ackermann_cmd_mux/input/navigation

Figure 4: Pre-existing messaging infrastructure on the robot. A command multiplexer (mux) ensures that safety controller messages, when sent, take priority over navigational commands such as wall following

The existing architecture contains two command multiplexers (muxes), one for high-level control and one for low-level control. We take advantage of the low-level mux, which prioritizes safety controller messages, when they exist, over navigational messages such as the wall follower's. We were thus able to ensure that our wall following and safety controllers operate together by publishing safety controller messages to the topic

```
/vesc/low_level/ackermann_cmd_mux/input/safety
```

and publishing wall follower messages to the topic

```
/vesc/high_level/ackermann_cmd_mux/input/nav_0
```

3 Experimental evaluation

3.1 Wall detection and following

Author: Christina Jung

We used both observational and numerical data to evaluate the performance of our wall follower. After running the code on the physical racecar, we learned that values that worked well in simulation often led to unexpected errors in real life. Thus, we tested the robot in various situations and tuned the gains to ensure that our racecar performs robustly in all situations.

When collecting observational data, we focused on three main performance metrics. First, we looked at whether the robot consistently maintains the desired distance from the wall. In order to do so, we used tape to mark the path that the robot should ideally be following. Second, we tested how well the robot handles corners. We tested the robot in both convex and concave corners to ensure that the robot can handle different types of corners. Finally, we made sure that the robot is able to follow uneven walls.

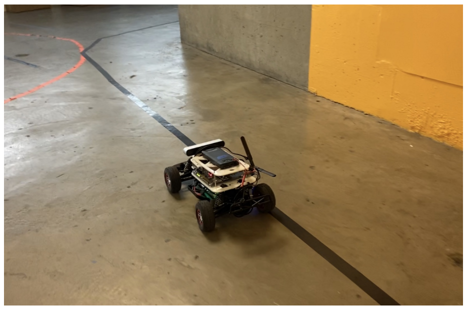

Figure 5: Demonstration of tape usage to observe the robot's distance away from the wall.

We also looked at numerical and graphical data to gain a better understanding of controller performance. Specifically, we recorded the robot's distances from the wall and compared the values to the desired distance. We collected data of the robot following straight walls and corners separately, then calculated the average percentage errors of the distances. We then graphed the data to take a closer look at the oscillations and damping.

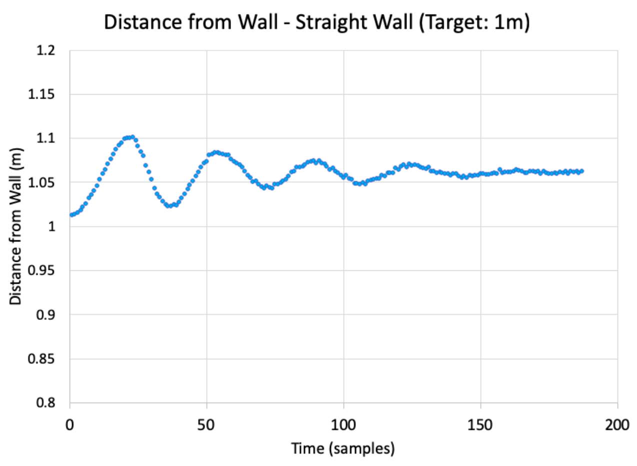

Figure 6: Recorded distances from the wall when the racecar follows a straight wall with a desired distance of $1m$. The average percent error was 5.96%.

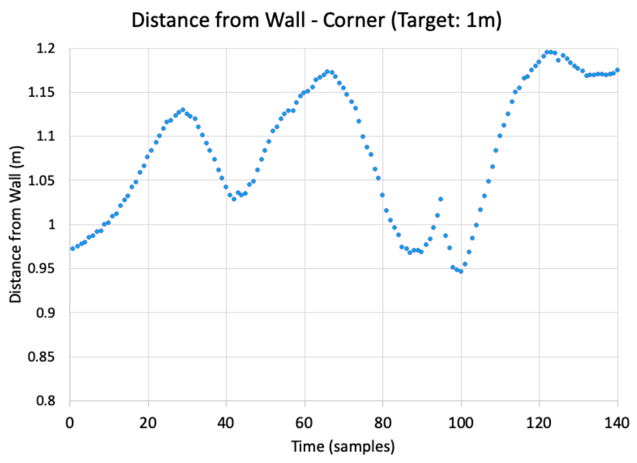

Figure 7: Recorded distances from the wall when the racecar follows a corner with a desired distance of $1m$. The average percent error was 9.26%.

The robot followed walls and corners with reasonable percent errors of 5.96% and 9.26%, respectively. As figures 5 and 6 indicate, there is some oscillation near the beginning and more oscillation when going around corners compared to when following straight walls. The robot is also consistently farther away from the wall than its desired distance. Such factors are likely due to the lack of PID control in our wall following algorithm. Still, the robot quickly stabilized its path when following straight walls and after turning corners.

Based on observational and numerical data, we demonstrated that our wall follower is robust. The robot maintains the desired distance from the wall, handles convex and concave corners, and follows uneven walls. Analysis of the percent errors further confirm that the robot is able to follow the wall at the desired distance.

3.2 Safety controller

Author: Diego Mendieta

We evaluated our safety controller by observing how the racecar reacted to different potential obstacles, both sudden and anticipated. We turned off the wall-following mechanism when testing the robot with faraway stationary objects or else it would interpret them as a wall and avoid them without using the safety controller. For these cases, we created a script that would allow the robot to move forwards at a constant velocity. When we made the robot drive straight towards a wall at velocities of 1 m/s, 2 m/s and 3 m/s, it repeatedly stopped at a distance of 25 cm from the wall measured from the LIDAR casing.

The robot also stopped successfully for several unexpected objects. We tested the safety controller on these obstacles with and without the wall follower. We created sudden obstructions by asking a teammate to step in front of the car while in motion and by dropping a backpack on its path. For both cases, the robot successfully braked without crashing. However, for velocities greater than 3 m/s, the front bumper of the robot did make contact with the obstacle, albeit lightly.

To ensure that the robot would not stop unnecessarily, we also performed negative testing on the safety controller. To test whether we would be able to navigate through narrow openings without interference from the safety controller, we had the robot go straight through the legs of a chair. As seen in figure 8, the robot passed below the chair successfully. When we rotated the chair 45°, however, the robot detected a threat of collision and stopped as expected.

A summary of the testing methods and results from the safety controller evaluation is shown in table 1.

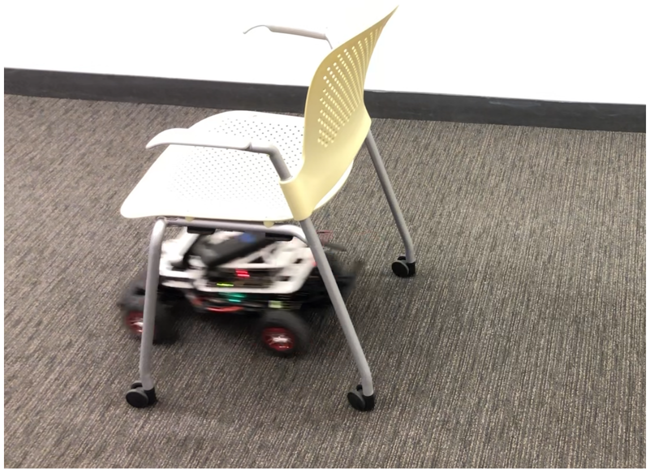

Figure 8: Negative testing performed on the safety controller by having the robot drive between the legs of a chair. The safety controller did not detect a threat of collision and did not stop the car.

Table 1: Summary of methods and results from safety controller evaluation.

Method	Stopped	Did not stop
Drive straight into wall	✓	
Sudden step	✓	
Backpack falling	✓	
Chair (narrow pass)		✓
Rotated chair (no pass)	✓	

4 Conclusion

Author: Silvia Knappe

In this lab, we successfully designed and implemented a wall follower as well as a safety controller for our robot. We found that a pure pursuit wall following controller behaved robustly. Using the pure pursuit controller, our robot was able to turn around both convex and concave corners as well as follow uneven walls or walls with gaps. We were able to tune the gain for the pure pursuit controller to a value that is robust for many scenarios, such as the ones mentioned above, by testing the robot in a variety of situations. In response to any undesired behavior, such as crashing into walls, we adjusted the gain. We took the effect of the gain on the pure pursuit steering angle equation (18) into account when adjusting the gain, so as not to blindly guess at a gain.

Our safety controller was able to detect obstacles and stop the robot before it got the chance to crash. We improved the responsiveness of the safety controller by having it take into account the velocity of the robot when deciding how far away from potential obstacles to stop. By considering the velocity, our robot is now able to detect and avoid crashes even when moving at a higher velocity.

In the future, we would like to increase the effectiveness of our wall following controller by adding PID control to it. While our wall follower behaves well, it is slightly underdamped, which can be resolved by adding derivative and integral control. We would also like to have bug free code on our robot, so we will be adding unit tests to increase the robustness of our code. Lastly, we would like to increase the performance of our robot, so we would like to increase the speed of our robot and still have our code working.

5 Lessons Learned

William:

- Learned about the theory and practical implementation of pure pursuit.
- Furthered understanding of Overleaf formatting structures.
- Improved ability to deliver short technical talks, learned suggested hand placements while idle and speaking.

Henry:

- Reality differs greatly from simulation.
- Theory doesn't go very far with our robot. Most of the time required is for tuning parameters empirically, not performing calculations.
- The highest priority in briefings is knowing the content exactly. In timing emergencies, paring down content to the essentials is essential.
- I personally value concision higher than I previously believed.

Christina:

- Learned how to tune parameters through trial and error and various rounds of testing.
- Become much more familiar with using GitHub and the software used on the racecar, although still in the process of learning.
- Realized that I should work more on timing and keeping my speech succinct. Learned the importance of practicing along with actual slides, especially when showing videos.

Silvia:

- Learned differences between simulation and real life: gains and parameters were often different on the real robot.
- Learned how to use Github better for version control
- Became more aware of filler words in presentation and worked to remove them

Diego:

- Got acquainted with the software used to interact with the robot.
- Learned to overcome unforeseen complications such as having unexpected noise in the LIDAR data or miscalibrated hardware on the racecar.
- Learned that it is important to practice a presentation beforehand and prepare for difficulties that might arise while presenting.
- Got to know my team and started to develop better strategies for collaboration together.