Lab #3 Report: Wall Following on the Racecar

Team 14

Michael Lu
Enrique Montas
Jon Stenger
Grey Sarmiento

6.141 Robotics: Science and Systems

March 5, 2022

1 Introduction (Jon S.)

In this lab, our team completed four main tasks which included: setting up the hardware of the robot, testing the wall follower code from the previous lab, implementing a safety controller, and lastly setting up a team website on GitHub.

To make sure our robot was working properly, we first had to connect it to our computer and drive it manually with a joystick. The robot operated as expected in response to different steering inputs. This step was crucial because it meant that errors in the future would likely be from bugs in our code and not the hardware itself.

In the previous lab, we programmed and simulated a wall follower node that would allow the robot to drive autonomously by following a wall and keeping a certain desired distance away from the wall. Ideally, we would add our code from before and it would work on the physical robot. However, this is not always the case as there is often a gap between the real world and simulation. This can be due to various factors such as more simplistic models being used in the simulations or different variables being introduced. Ultimately, this was not a big issue as we only had to tune a few parameters to make the robot work as desired.

Next, we implemented a safety controller that could make the robot stop autonomously so that it would hopefully not crash. The robots use very expensive hardware and we want to avoid collisions whenever possible.

Finally, we set up a team website on GitHub. The website will be used for this lab and future labs to document our briefings and reports.

2 Technical Approach (Michael L., Enrique M.)

To approach the technical challenge of creating a car that would drive autonomously but also safely, the robot was driven by three ROS Node controllers, each with different priority levels:

1. /ackermann_cmd_mux/input/teleop: User input from a remote joystick controller.
2. /ackermann_cmd_mux/input/safety: A controller that prevented the robot from crashing into objects in front of the car.
3. /ackermann_cmd_mux/input/navigation: A wall-following algorithm that guided the car to steer alongside a wall.

The race car would listen to the active controller of highest priority to drive the car, meaning it would listen to any joystick teleop inputs before listening to the safety stopper controller before listening to the wall following algorithm.

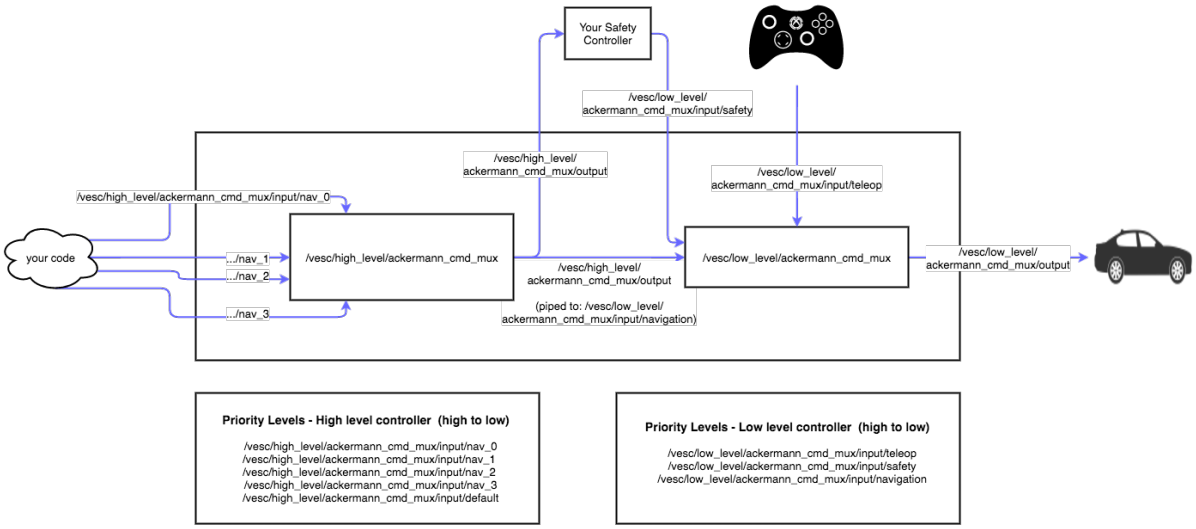

Figure 1: ROS Node Architecture: Robot car controlled by /ackermann_cmd_mux, which takes commands from /ackermann_cmd_mux/input/teleop, then /ackermann_cmd_mux/input/safety, and finally /ackermann_cmd_mux/input/navigation
.

2.1 Safety Controller

2.1.1 Design

The Safety Controller prevents the robot from crashing. The obstacle can be either stationary or moving, if it is within a stopping distance the robot will stop. Before the lab, there was no safety controller so the team had full control of the design and implementation. The Lab 3 handout had a spec including expected behaviors: prevent crashing while not hindering safe motion.

The safety controller was designed to mirror the simplicity of its spec, "stop if you're going to crash".

$scan \leftarrow lidar$
if car will crash **then**
  stop
else
  drive
end if

Algorithm 1: Safety Stop Algorithm

The above pseudocode is the logic within the callback function. To determine if the robot is in danger of crashing the LIDAR data is used.

$distances, angles \leftarrow scan \leftarrow lidar$
$front\_distances, front\_angles \leftarrow distances, angles$
if $\min(front\_distances) <=$ stop threshold **then**
  return will crash
else
  return no crash
end if

Algorithm 2: Safety Stop Algorithm Stop Condition

The front of the robot is defined as the sector of the scans within the two front tires. In this lab, the only way for the robot to crash into something is for it to drive its front into the obstacle and crash. This means we can ignore obstacles outside this sector even if they're close to the robot. We can find the perpendicular distance of the relevant obstacles to the robot, or how far the front bumper is from hitting it. Basic kinematics states that as our robot's velocity increases the time and space it needs to stop also increases, so the safety controller should adjust itself to the velocity of the robot. The LIDAR isn't on the front bumper of the robot so it is important to account for this offset as well. After finding the distance of potentially dangerous obstacles the safety controller should check if it is within a safety threshold and evaluate if the robot needs to begin to stop or not. In the case the robot needs to stop the safety controller should publish a command to the mux, since it has higher priority than the wall follower the

robot should stop. In the case where a stop is unnecessary the stopper need not publish anything to the mux and the robot will continue to follow the wall follower's commands. The safety controller is always running but only publishes when the robot needs to stop, mirroring the spec's "stop if you're going to crash" requirement.

2.1.2 Implementation

In this section, specific implementation details of the safety controller will be explained.
The Safety Controller is written within safety_controller.py in the SafetyController class. The controller subscribes to the *LaserScan* LIDAR data to find the distance and angle of objects with respect to the robot. The class also initializes a publisher that publishes to the safety topic so the mux knows when the safety controller is taking over. To isolate the distances in front of the robot, the sector bounded the two front wheels. On the robot, it corresponded to the middle $\frac{1}{5}$ or the third $\frac{1}{5}$ segment of scanned distances. Similarly, we find the corresponding $\frac{1}{5}$ angles, they're necessary to find the perpendicular distance. To get the perpendicular distance multiply the distances by the cosine of their respective angles. Take the minimum value, the first point of contact with the car, and compare it to the stop threshold. To calculate the stop threshold we take 10% of the robot's velocity and add .2m for the mentioned LIDAR offset.

$$stop\_threshold = \frac{robot.velocity}{10} + .2$$

If the closest relevant object is within this threshold the robot needs to stop and the controller publishes an AckermannDriveStamped with zero velocity. The priority of the safety controller is higher than the wall follower so it stops. If the object is outside the threshold the safety controller doesn't publish anything and the robot continues to follow the wall follower. Both the wall follower script and safety controller script are launched with the wall follower launch file so the user doesn't accidentally launch the robot without safety running.
After the safety controller was written params.yaml was updated to ensure everything was published to the mux so the priority of controllers was reflected in the robot itself.

2.2 Wall Follower

The wall-following algorithm implemented on our robot would take in laser scan data and steer the robot to follow a wall on either its left or right side. The algorithm consisted of three main steps:

1. Use the robot car's laser scan data to model the wall with a best-fit line.
2. Calculate the perpendicular distance from the car to the wall's best fit line.

3. Use a PD controller to adjust the robot car's steering based on the error between the desired distance to the wall and calculated perpendicular distance.

2.2.1 Wall Modeling with Best-Fit Line

The data from the robot car's laser scan would be delivered as an array of distances, with each element corresponding to a detected distance at a specific angle away from the car. The laser scan data was thus represented as two arrays R and Θ of equal length where a given pair of elements i corresponded to a point a certain angle θ_i away from the car at a distance r_i. These pairs of elements were effectively polar coordinates measured from the car (which was treated as the origin). This data is graphically represented in Figure 2.

$$R = [r_1, r_2, \ldots, r_n]$$
$$\Theta = [\theta_1, \theta_2, \ldots, \theta_n]$$

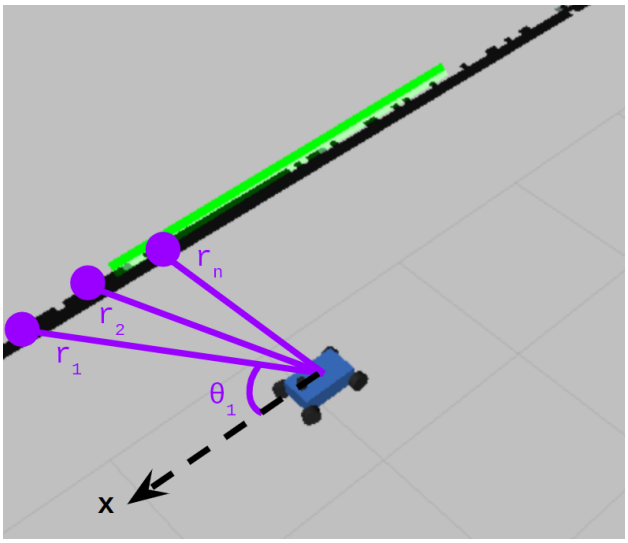

Figure 2: Diagram of laser scan data: Each purple point is a polar coordinate measured from the car, which is the origin. A point is represented by distance r_i and at angle θ_i measured with respect to the x-axis represented by the dotted black line. These points were converted into Cartesian coordinates so that a best-fit line, pictured in green, could be applied.

To first remove any unnecessary data processed in our algorithm that would compromise its correctness, we first filtered out half of the scan data that corresponded to the side of the car where no wall exited and also removed any remaining data points that were more than thrice the desired distance. This

filtering process ensured that the data points remaining were nearby points on the wall on our car's desired side. Otherwise, any unnecessary points would result in an inaccurate best-fit line to model the wall.

Using the newly filtered R and Θ arrays, each pair of polar-coordinate laser scan data (θ_i, r_i) was converted to a Cartesian-coordinate pair (x_i, y_i) and stored in two arrays X and Y using the following equations:

$$x_i = r_i \cos\theta_i$$
$$y_i = r_i \sin\theta_i$$
$$X = [x_1, x_2, \ldots, x_n]$$
$$Y = [y_1, y_2, \ldots, y_n]$$

With a set of Cartesian coordinates, a best-fit line of the form $y' = mx' + b$ was applied to model the wall.

2.2.2 Perpendicular Distance to Wall

With a best-fit line of the form $y' = mx' + b$ modeling the wall, the perpendicular line going through the wall and the car was calculated to be $y_p = -\frac{1}{m}x_p$ where the slope $-\frac{1}{m}$ was the negative reciprocal of that of the best-fit line.

Since the perpendicular distance from the car to the wall would be along this line, calculating that distance involved finding the Euclidean distance between two points on that perpendicular line, one being the car and the other being the intersection point of the perpendicular line and best-fit line for the wall. Since the point along the perpendicular line on the car was at the origin, the problem reduced to finding the intersection point (x^*, y^*) between the perpendicular line and the best-fit line:

$$y^* = y_p^*$$
$$mx^* + b = -\frac{1}{m}x^*$$
$$mx^* - \frac{1}{m}x^* = -b$$
$$x^*(m + \frac{1}{m}) = -b$$
$$x^* = \frac{-b}{m + \frac{1}{m}} = -\frac{bm}{m^2 + 1}$$
$$y^* = mx^* + b$$

With the calculated intersection point at the wall, the perpendicular distance could be obtained by taking the Euclidean distance between the intersection point and origin, which reduced to:

$$d = \sqrt{x^{*2} + y^{*2}}$$

2.2.3 PD Steering Control

The calculated perpendicular distance could then be compared to the desired distance d^*. An error term could then be calculated, and a PD controller with a proportional gain term K_p and derivative gain term K_d could be applied to come up with a steering angle ϕ:

$$e = d^* - d$$
$$\phi = K_p e + K_d \frac{de}{dt}$$

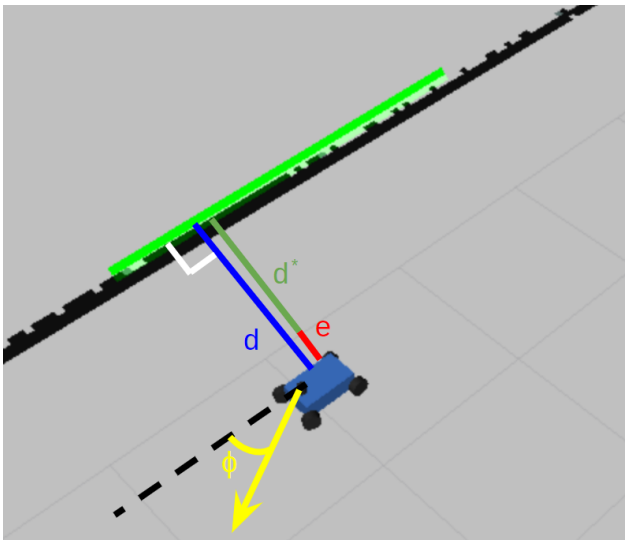

Figure 3: The difference between the perpendicular distance and the desired distance was saved as an error, and a steering angle ϕ was applied based on that error to guide the car towards the desired distance.

The K_p term meant that more steering was used when the car deviated from the desired distance more, meaning that if the car was extremely close to the wall, i.e. $d << d^*$, then $|e| >> 0$, so ϕ increased to aggressively steer the car away from the wall.

The K_d term meant that more steering was used if the car's error was changing quickly, meaning that more aggressive steering was applied if the car was quickly deviating from the wall. Intuitively, the K_d term helped keep the car parallel to the wall and removed many of the oscillatory and overshooting behavior characteristics of a purely proportional controller.

Figure 4 graphically explains the effects of each of these gain terms on the system response.

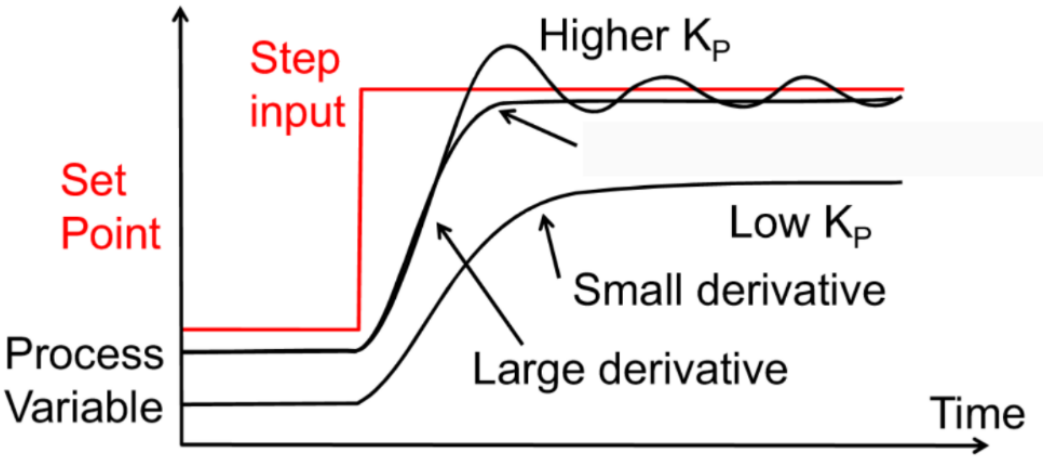

Figure 4: Different gains in a PD controller affect the system response. A higher K_p causes more aggressive compensation but leads to overshoot and oscillation. A higher K_d fixes this but can still result in some steady-state error.

The K_i term was not used for the wall-following algorithm to simplify our controller, as this would be one less parameter to tune and maintain. This simplification was especially beneficial given the tight timeline of our project. However, this might have come at the expense of some steady-state error observed in testing our PD controller.

To optimize the PD controller, the K_p term was first deduced through manual tests in the simulation until the car could follow the wall with some oscillations. Then, the K_d term was applied to remove these oscillations. Upon using the real car, manual tuning of the same process was used to finalize the PD controller.

2.2.4 Wall Follower ROS Implementation

Our wall-following algorithm was implemented in a Python $WallFollower$ class, which contained a subscriber to the robot car's $LaserScan$ data to find the wall and an $AckermannDriveStamped$ drive topic publisher to control the car.

After initializing the subscriber and publisher, the ROS node pulled parameters from a *params.yaml* file, which contained the robot driving speed, wall distance, side with the wall, and the ROS topic names.

Every time the robot received a new set of laser scan data, a callback was triggered, executing the wall-following algorithm illustrated in the Algorithm

$R \leftarrow lidar\ scan\ data$
if first lidar scan R **then**
  $\Theta \leftarrow theta_{min} + \theta_{inc} | 0 \leq i \leq \frac{\theta_{max} - \theta_{min}}{n}]$
  $n_{side} \leftarrow 0.55n$
  $\Theta_L \leftarrow Theta[n - n_{side} : n]$
  $\Theta_R \leftarrow Theta[0 : n_{side}]$
end if

$R_L \leftarrow R[n - n_{side} : n]$
$R_R \leftarrow R[0 : n_{side}]$

if track left wall **then**
  $R_L \leftarrow R[r_i | r_i < 3d^*]$
  $\Theta_L \leftarrow \Theta_L[\theta_i | r_i < 3d^*]$
  $X \leftarrow [r_{Li} \cos\theta_{Li} | 0 \leq i \leq n_{side}]$
  $Y \leftarrow [r_{Li} \sin\theta_{Li} | 0 \leq i \leq n_{side}]$
  $m, b \leftarrow best\_fit\_line(X, Y)$
  $x^* \leftarrow -\frac{bm}{m^2+1}$
  $y^* \leftarrow mx^* + b$
  $d \leftarrow \sqrt{x^{*2} + y^{*2}}$
else if track right wall **then**
  $R_R \leftarrow R[r_i | r_i < 3d^*]$
  $\Theta_R \leftarrow \Theta_R[\theta_i | r_i < 3d^*]$
  $X \leftarrow [r_{Ri} \cos\theta_{Ri} | 0 \leq i \leq n_{side}]$
  $Y \leftarrow [r_{Ri} \sin\theta_{Ri} | 0 \leq i \leq n_{side}]$
  $m, b \leftarrow best\_fit\_line(X, Y)$
  $x^* \leftarrow -\frac{bm}{m^2+1}$
  $y^* \leftarrow mx^* + b$
  $d \leftarrow \sqrt{x^{*2} + y^{*2}}$
end if

$e \leftarrow d^* - d$
$t \leftarrow time\_now()$
$\frac{de}{dt} \leftarrow \frac{e - e_0}{t - t_0}$

$\phi \leftarrow K_p e + K_d \frac{de}{dt}$

$e_0 \leftarrow e$
$t_0 \leftarrow t$

Drive car at velocity v
Steer car at steering angle ϕ

Algorithm 3: Wall-Following Algorithm Pseudocode

3 pseudocode. In lines 2-7, the first set of data is received, so an array of all possible scan angles was created and partitioned into the set of left and right-side scan angles. After setting up the angles, the scan array of distances was partitioned into the left and right-side scan distances as shown in lines 9 and 10.

Then, based on whether the wall to follow is on the left or right, either lines 13-20 or 22-29 run to create the best-fit line for the wall and calculate the perpendicular distance to the wall. In the implementation for this part of the algorithm, the data was first cast as a NumPy array, which would prevent the need for slow Python loops and make the algorithm faster. The data was then filtered and converted to a Cartesian representation, as described above. Then, the NumPy polyfit function was run on the data to calculate the slope and y-intercept parameters of the best fit line. Finally, the perpendicular line parameters were calculated, and the perpendicular distance to the wall was calculated.

Using the calculated distance, the error was calculated on line 32. Then, the derivative of the error was calculated to be the change in error from the last algorithm iteration divided by the change in time from the last algorithm iteration. Times were measured using the ROS *get_time*() function. Finally, the calculated error and its derivative were applied to the PD gains to generate the steering angle. Finally, the car was driven at the provided velocity and steering angle.

3 Experimental Evaluation (Grey S.)

3.1 Localization Error Tests

For the wall following task, localization error is a comprehensive evaluation metric to quantify how well the robot was able to minimize error and follow the wall during a trial. The wall follower was able to successfully maintain error within a margin of 40 centimeters, and the PD controller adjusted error in the right direction as it changed over time.

3.1.1 Setup

Besides the ROS topics that published drive commands, one topic was dedicated to perceived localization error of the robot. The published error over the duration of a run was recorded via a rosbag file. Graphing this error over time using `rqt_plot` allowed us to visualize the performance of the system.

3.1.2 Results

The algorithm successfully maintained a low error within an error margin of 40 centimeters. As seen in Figure 5, the PD controller was able to adjust the robot's steering angle as the error value increased away from the wall or decreased toward the wall. Particularly, "zig-zagging" of the error plot

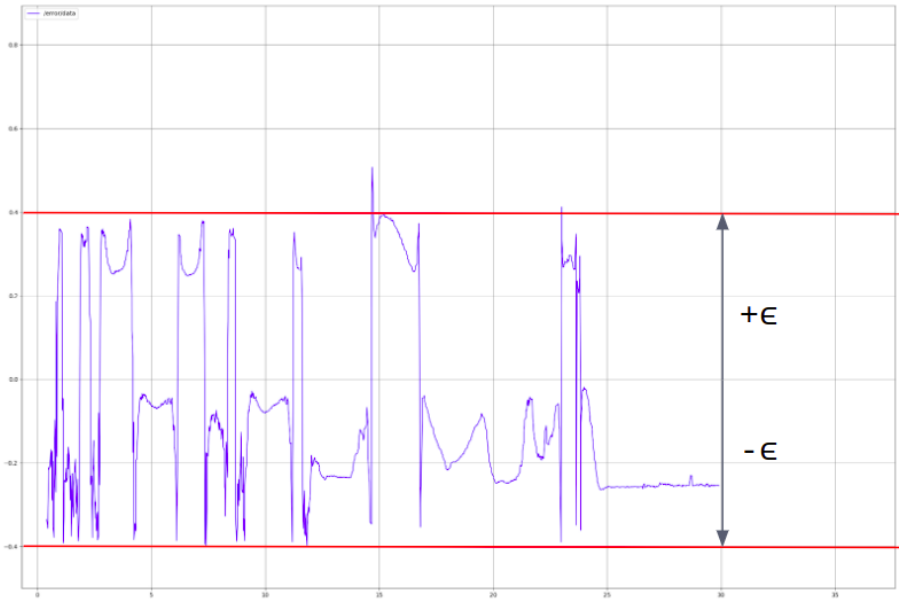

Figure 5: Error as graphed by rqt_plot. Localization error (meters) over time (seconds). Positive and negative error margin thresholds shown in red.

signifies that the controller is actively maintaining the steering so that error decreases.

3.2 Safety Controller

The safety controller was implemented for use as a practical tool while developing the solution to this challenge, as well as to demonstrate command over the robot MUX. Keeping the robot safe will continue to be relevant throughout the course, and establishing a basis for a hierarchical ROS node structure has the potential to extend beyond safety applications.

The safety controller was found to be successful on all of our test cases, including in crowded scenarios with multiple obstacles that are both moving and stationary.

3.2.1 Setup

The safety controller was evaluated on two metrics: its **capacity to stop the robot when an obstacle enters its safety margin**, and its **robustness to both moving and stationary obstacles**.

Capacity to stop was evaluated by placing the robot on a box so that it was unable to drive, and moving a panel toward and away from the lidar

sensor. From here, the range at which the wheels stop moving was validated, along with the verification that the wheels will start moving again once the range increases beyond the safety minimum. By shifting the panel to both sides of the robot slowly, the horizontal range of the robot front view was also verified.

Robustness to obstacles was evaluated by using team members and other volunteers to act as obstacles in the robot's path as it followed the hallway wall. By varying the number of people in the hall, the extent to which these people are moving about, and the way in which people enter the robot's range of view, a qualitative assessment of the safety controller's functionality was achieved.

3.2.2 Results

Capacity to stop: In all driving test cases, the robot was able to stop within the desired distance to an obstacle. No cases were observed where the robot failed to stop soon enough and collided with the environment.

Robustness to different obstacles: All test cases were observed to be successful and can be summarized in the following table.

- Stationary: While not perfectly stationary, people stay standing in roughly the same spot in front of the robot. A successful trial of this obstacle type entails navigation around the stationary obstacle unless the person becomes a "dynamic" object.
- Dynamic: One person steps in front of the robot while the robot is running, entering the field of view "unexpectedly" from the side. If in a group, people may arbitrarily step in front of the robot one after the other while the other people act as "stationary" obstacles.

Number of obstacles	Stationary	Dynamic
1	✓	✓
2	✓	✓
4	✓	✓

3.3 Options for Further Evaluation

As this design was finalized in a residence hall, there was not an opportunity to conduct all desired performance tests in the hallway (which was frequented often by students). In particular, our team acknowledges the capacity for further documentation of:

1. A larger variety of drive speeds
2. Various types of wall corners
3. A larger variety of distances to maintain from the wall, especially distances that are smaller than the minimum front-range distance for the safety controller

Although some of these scenarios were tested qualitatively as the hallway setting changed, quantitative documentation of each variable in isolation would be valuable. These tests would further validate the success of the solution and demonstrate robustness to a wider range of environments and scenarios.

4 Conclusion (Grey S.)

In this project, our team was able to successfully implement a wall following algorithm and safety controller in ROS on a real robotic system by applying basic PD control. Our system was experimentally evaluated to be successful on a variety of settings and obstacle scenarios, although the door remains open for further evaluation. Through the development of this system, our team established a basic understanding of the ROS platform, PD control, and using a MUX hierarchy to control robotic navigation, which will continue to be relevant as we proceed through the course and beyond.

Further improvements on the current system could include optimizations on the wall following algorithm, such as the inclusion of an integration term in the P(I)D controller, or implementation of a noise reduction algorithm such as RANSAC to the range scan data to improve wall estimation.

Beyond the technical milestones of this lab, our team was also able to establish a communication system and expectations, a git organization to maintain version control of our work, and a team website.

5 Lessons Learned

5.1 Jon Stenger

This lab was very important to help up learn more about ROS and implementing control theory to develop autonomous systems. One thing that became clear from very early on was how much more time and effort it took to test our physical robot compared to the simulations. Since this lab was our first lab completed as a team, it also taught us how to work together collaboratively. We often had to do work asynchronously and then later make sure everything worked all together.

5.2 Michael Lu

This lab helped us sharpen our skills working with ROS, especially in real life, where many simulation tools are not available. We learned how to test, debug, and validate our software without the convenience of simulation visualizers and simulation test cases. We also refined our abilities to tune PD controllers, especially by qualitatively analyzing robot behavior in real life.

Finally, this lab provided an opportunity to collaborate with others in a technical and interpersonal setting, learning how to code collaboratively and determine logistics for working together.

5.3 Enrique Montas

This lab gave us each more experience troubleshooting and debugging in ROS, whether it be joystick connecting issues, bugs in our controllers, or setting up credentials on the robot to use git. We also got to work together using collaborative tools such as google slides, git and overleaf.
This lab reinforced the value of organization in teams. This is applicable to communication, team conventions, code, and arranging for time outside lab to work. Reflecting I think this lab helped make the team more comfortable together and gave us a foundation with one successful lab together. From here we can iterate and smooth out how we work together, I can imagine more asynchronous work as we are more familiar with ROS, the robot, how each other work, and where each other's strengths lie.

5.4 Grey Sarmiento

I think one important lesson that this lab taught us was the challenge of adapting a system from simulation to the real robot. In getting all the hardware components to work together, we were able to get experience using essential ROS debugging tools and practices, such as listing and echoing ROS topics and nodes at different points and recording output with the `rosbag` command - which I think will continue to be important to master as our assignments become more complex. I was excited to get started working on a physical robot!

On the collaboration side, I think that in the future we could benefit from setting a more detailed timeline of when we want to work throughout the week and when we want certain things to be done by (which was understandably more shaky this week because of all the start up that was happening). Overall, our teamwork seems to be pretty smooth, and I anticipate that we will only become more streamlined as the class continues.