Lab #3 Report: Wall Follower

Team #12

Yasin Hamed
Artem Laptiev
Evan Bell
Cruz Soto

Robotics Science and Systems

March 12, 2023

1 Introduction
by Cruz Soto

The goal of this wall follower lab was to adapt the simulation developed in the previous lab to properly follow a wall in this lab. In particular, the wall follower was set to maintain a certain distance from a given wall and appropriately respond to environmental changes such as inner and outer turns, variations in the wall geometry, and initializing movement at different angles in respect to the wall. This is the first step in developing a robot that can detect and respond to its environment by way of an active extoreceptive sensor.

Using a Proportional Integral Derivative (PID) controller, a mathematical system through which error can be mapped to a successful output (where success is minimizing the error), the team was able to individually build their simulations where the car could respond to all of these parameters.

Adapting this to a real robot had its difficulties, however. Most prominently, the wall follower code depended on LaserScan data, a datatype built from a 2D Light Detection and Ranging (LiDaR) system, but now required the use of a 3D Velodyne Lidar® puck, which outputted 2D data in a different format for a different range of angles, meaning that the data needed to be re-patched, rotated, and re-adapted for practical use.

This lab also lays the groundwork for safety and operation of the car as it introduces the team to the workflow of setting up a router, checking power to the vehicle, secure shell (ssh) connecting to the car, and the code sharing between members of the team and the car itself. On the safety end, one of the core requirements here is to construct a safety controller. The cars are already in very few supply, and the difficulties of managing this with many technical issues on the vehicle itself will be discussed later in the report. As such, to prevent any significant damage or need to decommission the car, a node which can detect and stop the car from moving to avoid a collision, overriding any inputs to do otherwise, was implemented. This will be useful in stopping any autonomous code tested on the car in future from leading to significant damage while operating at high speeds in future labs.

2 Technical Approach

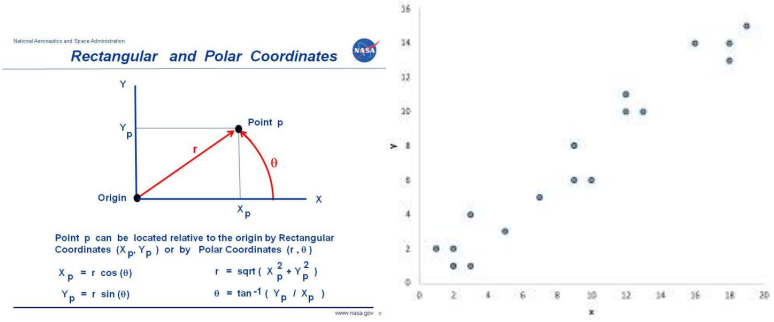
2.1 Wall Follower Considerations
by Evan Bell

A few items are key when considering how to implement a wall_follower

1. We need to define a wall. To do this, we use the Laserscan data. These will be a list of values representing distances seen by the robot at certain angles. Using the previously known scan angles, we can calculate which

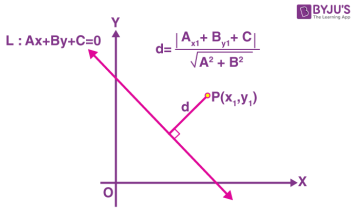
angle each point was taken at relative to the car. We can know choose sections of these values based on 'where' we want our wall_follower to look for walls.

2. The section that we look at will have a set of points, described as distances with known angles now. We can plot these values onto a standard 2 axis graph at points $(r\cos(\theta), r\sin(\theta))$



3. We now have these points graphed. To model the information more effectively, we perform a linear regression on these points. This will find the line which minimizes accumulated square error from all points. This is relatively efficient to calculate, and allows us to estimate the wall as a single function which we can manipulate easier. It is slightly more prone to being affected by incorrect data such as a improper reading for a certain distance, but the approximation is reliable enough.

4. It is important to note that the origin is the car's position relative to the scan information, and the x axis is the forward direction of the car. Given a wall line of the equation $y = mx + b$ we can calculate the car's distance to a line as $\frac{m+b-1}{\sqrt{m^2+1}}$. This will be the shortest distance to the wall, and can be used as the perceived distance from the wall.



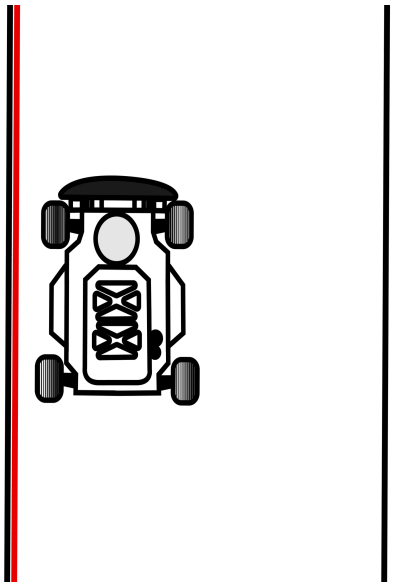
5. The error can be estimated as a function $f(x)$ based on this distance value, and the steering controls can depend on that.

2.2 Wall Follower Implementation by Yasin Hamed

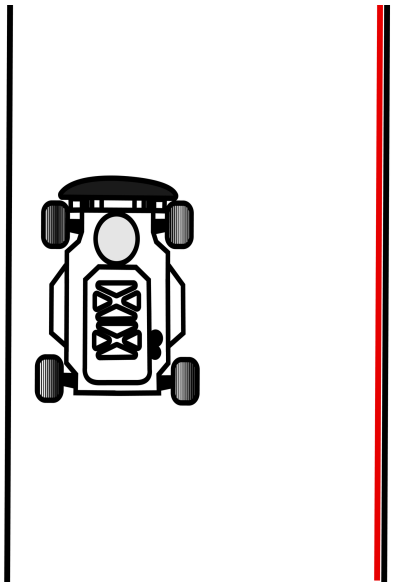
The first thing to note about this implementation of the wall follower is that the wall selection for the robot can be made static or dynamic which significantly affects the performance of the robot in different situations.

In the dynamic wall selection mode, the robot takes the LIDAR data, splitting it up into left and right sections. It takes an average of the ranges of the points on each side and decides to follow the side which it is closest to.

In the static wall selection mode, the car will always use the data on the prespecified side of the robot to follow the wall – that is to say that it will always try to follow either the right or left wall but will never switch from one to the other.



Here, the car has dynamic wall selection enabled, and is thus following the wall closest to it – the left wall.



Here, the car has static wall selection enabled, with the code set to follow the right wall. It is following the right wall despite that wall being further from the car than the left wall.

Once a line representing the wall relative to the robot has been calculated – as described in the previous section – the error signal is taken to be the difference between the desired distance (setpoint) and the actual distance of the car from the wall (shortest distance from the car to the generated line). This signal is then passed into the PD controller to control the steering angle of the car, driving it towards the setpoint. This method works very well for following generally straight walls without corners.

For corners, the wall following algorithm distinguishes between 2 cases: inside corners and outside corners. During the approach of an inside corner, the robot will be approaching a wall directly in front of it. Once the robot detects the wall in front of is within a critical distance, the generated line is shifted to be directly on top of the robot, so the robot will think that it is way too close to the wall it is following. This will result in the car turning away from the wall it is following, which is the desired behavior in anticipation of the inside corner that is approaching. The generated line will continue to be "on top of" the robot until the robot no longer detects a wall in front of it, i.e. when it has completed the turn.

The car deals with outside turns the same way it deals with following a straight wall. The points in the car's data set will simply become further and further away as the car approaches the outside turn, shifting the generated line to be further and further away, increasing the magnitude of the error and causing the

car to start gradually driving towards the wall during its approach to an outside corner, which is the desired behavior.

In the case of outside turns, the distinction between static and dynamic wall selection. Our algorithm works for outside corners only works for static wall selection in all cases. If dynamic wall selection is enabled, and the robot were trying to follow a wall into an outside corner at a T-intersection, it will begin to drive past the corner, turning slightly as the generated line adjusts to be further and further away. This works well until the average of the points on the opposite wall becomes smaller, and the robot then switches targets and begins following the opposite wall, which is not the desired behavior for the car. This problem is solved by enabling static wall selection on the robot since it will follow the desired wall regardless of how far it is from the car (given that it is still within the range of the LIDAR, of course).

As a side note, dynamic wall selection exists as an option in our code to allow for the robot to drive down the exact middle of a hallway, regardless of the width of the hallway.

2.3 Reformatting Velodyne Laserscans by Yasin Hamed

The simulation that we were provided with to test our wall follower code assumed a format for the laser data messages which provided an array of ranges which was symmetric about the front of the car. The total range of the LIDAR sensor, which spanned from $-120°$ to $120°$ was provided in a single laser message in the simulation.

Unfortunately, our team had to deal with the Velodyne LIDAR sensors which used the laser messages in ROS to represent their data very differently from the simulation. The Velodyne sensors published not one, but *two* laser messages for a single $360°$ sweep. Additionally, it used the full length of an array representing the $360°$ in each of these 2 messages, filling half of each message with "Inf" values. It is possible the Velodyne does this to publish data at a higher frequency, albeit incomplete data. Additionally, the ranges array always had a constant section of "Inf" values representing the back of the LIDAR where the actual car was. To make things even more confusing, the ranges data was not even symmetric about the front of the car. All of these novelties in the data representation led our team to write a separate node which subscribed to this data, reformatted the angles and ranges arrays to be exactly the same format as the laser messages in the simulation, and publish a new, reformatted laser message to a different topic. This new topic is where the wall follower node gets its LIDAR data from so that changes did not have to be made to the code which already worked in simulation.

2.4 Writing a Data Recorder by Evan Bell

The goal is to record information about testing and operation of the robot for analysis later. We aimed to record two sources of data: the scan data perceived by the robot and the estimated error of the robot.

For the scan data, we have reformatting of the Velodyne Laserscans to correct for the angle offset and piece together to two halfs of published data. This means we can use the published reformatted scan to visualize the corrected scan as would be seen in front of the robot.

The error is generated inside of wall_follower.py, so we created a publisher to publish those values as a Float32.

We created a new node for data recording, and we added it to wall_follower.launch. It will create a new directory titled data in the src folder, and it will create two sub-directories err_data and scan_data in there. Each time the node is run, it will generate a bag file and a text file which both have a timestamped title. Two subscribers listen to reformatted_scan and error_log topics. The robot error is continuously written to the text file, and the scan data is written to the bag file 3 times per second to save on memory use.

2.5 Implementing a Safety Controller by Cruz Soto

Now that the robot has a configurable wall follower and is collecting data to properly document the functionality of said wall follower, it is necessary to implement a safety controller to prevent any damage coming to the robot in the case of a malfunction. The controller must satisfy a few requirements. The controller must:

1. Recognize the distance of the wall in front of it.
2. Intercept the drive commands outputted by other nodes.
3. Publish a movement-killing drive command to override any existing AckermannDriveStamped data.

To begin this implementation, the team wrote up an alogorithm to sort through only the portion of laserscan data that applied to a range

$$\theta$$

.
Getting the data would require an implementation that collects data as follows, sifting through an existing laser scan.

$i \leftarrow length(LaserScan)$

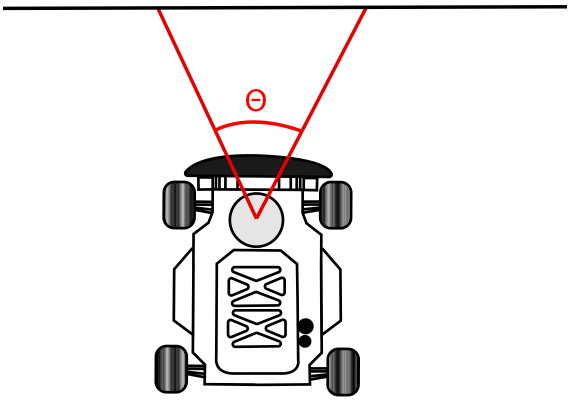


Figure 1: The robot detecting a wall with sweep angle θ

$L \leftarrow 0.3 * i$
$R \leftarrow 0.6 * i$
$LaserScan[L : R]$
for $k \in LaserScan$ **do**
  if $k = \infty$ **then**
    $k = d_{max}$
  end if
end for

The code which takes a laserscan of the front 90 degrees (30%) of the data Following this, the average distance to the wall can be calculated by taking the mean of these values. This official distance can then be printed to the console as well as plugged into a loop to prevent the robot from breaching a minimum distance to the wall. However, the scan itself has an issue with the velodyne LIDAR, as the Velodyne's minimum distance buffers around hte 0.5-0.4 meter mark, making it difficult to tell if the device is any clsoer as this will incur generation of the "inf" values seen in the psuedocode. To alleviate this, if the total quantity of "infs" in a scan exceeds a certain amount, the net distance to the wall will be assumed zero and the car will stopped.

while *alive* **do**
  if $num_{infs}/size_{front} > 0.4$ **then**
    distance_in_front = 0;
  end
end

Algorithm 1: A loop that sets he distance equal to zero if the number of "infs" is more than 40% of the total array size

In order to intercept the drive topic and publish a new command, the node needs to subscribe to drive topic at the highest level. This new command will now be spit out at the safety topic level, creating the following workflow.

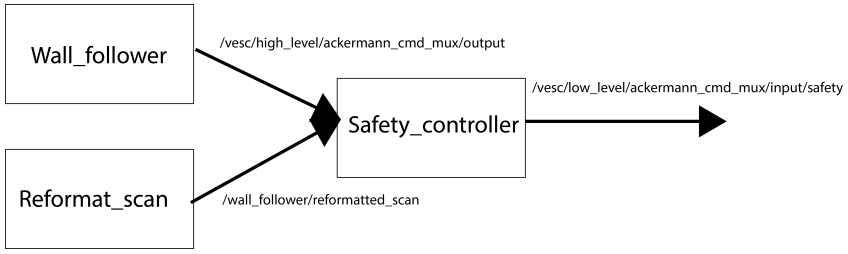


Figure 2: A block diagram dictating the publishing workflow for each node.

These were chosen as the safety topic is below the joystick topic, which we like to use to override commands in case the robot is difficult to collect, but above all other active nodes. This controller proved good at stopping with distances further from the wall at great accuracy, to be discussed in the experiment section.

2.6 Improving Development Experience: Dynamic Reconfiguration by Artem Laptiev

One of the problems we have forseen while working on implementing Wall Follower is the time-consuming nature of recompiling our code for ROS. Generally, making any changes to the code requires us to re-upload the node on the robot, restart the node, and potentially the simulation. This system works relatively efficiently when we only update code after writing large features. However, it causes too much of a lag when we need to update code to test small changes and tweaks in the parameters of our code, such as when picking parameters for PID control system. In this case, we would like to have a robust system that allows us to dynamically reconfigure our robot without having to go through the process of rebooting the entire node.

To prevent this problem from occurring in the future, we designed an implementation of ROS' "dynamic reconfigure" package. This package allowed us to create an easy graphic interface into the main dynamic parameters of our bot.

We have yet to refactor our robot code to put dynamic configuration to practice.

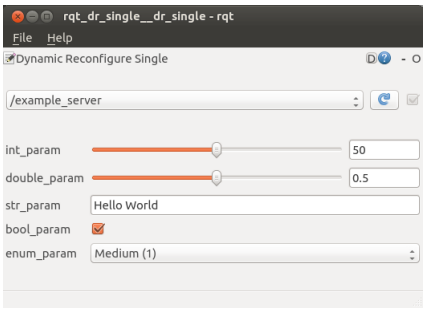


Figure 3: Graphic interface into bot's parameters using Dynamic Reconfigure

3 Experimental Evaluation

3.1 Testing of the Safety Controller by Cruz Soto and Evan Bell

The safety controller setup was tested at a set of multiple distances with a single speed running directly at a wall. Evan controlled the car and measured the distances, while Cruz manipulated the rosparameters and set testing values. This car was always started at a distance of ten meters and the distance decreased until the robot hit the wall (as the distance, or base_link is not at the perfect distance zero). Distances were also checked by the Velodyne LIDAR as well as physically to it with a tape measure. This yielded the following results:

Stopping Distance (m)	**Actual stopped distance (m)**
3	1.4478
2.5	0.7874
2	1.3208
1.75	1.0414
1.5	0.889
1.25	0.6096
1	0.381

Which had a surprising degree of error and showed that the controller was malfunctioning in some way.
After inspecting the LIDAR and looking at data generation in RViz, it was found that the robot was actually running slightly skew to the wall by 30°, and so the angle adjustment in the reformatted scan node was adjusted to reflect this. Following this adjustment, as well as increasing the sample collection size for each distance from two to three samples and averaging them, the error

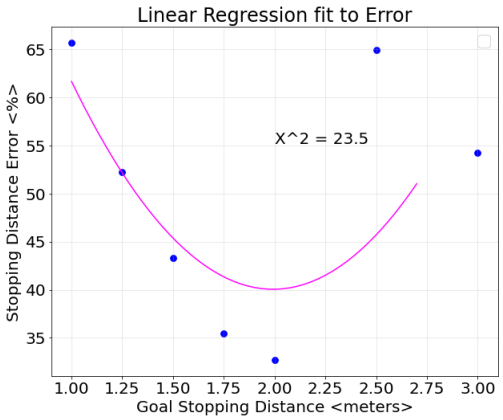


Figure 4: A block diagram dictating the publishing workflow for each node.

greatly decreased and demonstrated a clear inverse relationship between stopping distance and the error in said distance. The χ^2 value for the exponential fit also greatly decreased, indicated a low probability of better fit parameters being generated. This gives a high probability that there is a direct causation between stopping distance increasing and proportional stopping distance error decreasing. In the future, the safety controller will be tested with wider viewing

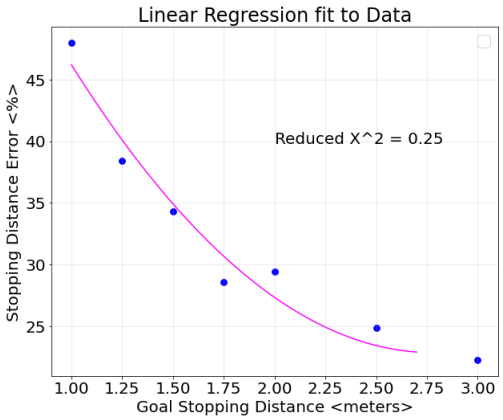


Figure 5: A block diagram dictating the publishing workflow for each node.

angles for the LIDAR sensor (to sense walls nearly perpendicular to the robot) as well as test at speeds above 1 meters per second (to make sure fast testing doesn't lead to incurred damage). It should also be run at the wall with a greater spread of angles to ensure that damage is prevented even when the car is skewed towards the obstruction. The controller should also be tested with the fast introduction of obstructions (such as team members jumping in front

of it) to make sure nobody within or without the team is injured while code is running.

4 Conclusion
by Artem Laptiev

As such, we successfully finished both major goals for the lab, as well as built tools for our development and analytics work in the process.

In the implementation of the Simulator Wall Follower into a real-life autonomous racecar, our most notable challenge was studying, clearing, and reformatting the Velodyne LaserScan data into the LaserScan messages, expected by our Wall Follower. We strongly suggest that instructors and TA's pay careful attention to the future class cohort of students using Velodyne LIDARs to help them save time for the more exciting parts of the lab. The output of our work is a standalone node that performs the LaserScan transformation. In addition to that, we made a series of minor design choices in order to marry our Simulation Wall Follower code with the real environment, such as creating a distinction between "static" and "dynamic" wall side detection, preferring the former for better outside-corner navigation.

In building the Safety Controller, we designed a custom node that reflects the specificity of working with the Velodyne LaserScan data, specifically managing the near-view "blindness" of the sensor, as well as established a publishing workflow that overrides the driving node commands in case of the safety emergency. We also carried out the initial safety controller tests and planned additional tests on more edge-case scenarios.

We also built a solution to record the data, produced by our bot, for further analysis and evaluation purposes.

Finally, as we are looking forward to future labs, we implemented a dynamic reconfiguration module in our bot, which we expect will help us speed up our future development.

As for the next steps, we plan to refactor the code to better reflect the multi-module nature of future labs. We also consider writing additional bash scripts to automate commands in the process of development environment setup.

5 Lessons Learned

5.1 Cruz Lessons Learned

Managing time with the robot is difficult, and since the schedule of the team during the day was hectic, the best choice was to meet in the evenings and at

the night, leading to a lot of difficulty if and when the robot refused to work. Troubleshooting the controller, batteries, and, most importantly, the Velodyne LIDAR, took a significant toll on our team.

Converting the Velodyne scans to a readable format took three days, significantly depleting our team's ability to implement and test the wall follower. Even with the wall follower implemented, tuning the controller was hard at close distances owing to the maximum range of the device itself capping out within 0.5 meters.

In order to better collaborate with peers, starting on the technical report even when the project isn't done (parallel performance) will be crucial in future labs for proofreading and reorganizing both the briefing and report. Presents individually authored self-reflections on technical, communication, and collaboration lessons you have learned in the course of this lab.

5.2 Yasin Lessons Learned

The biggest takeaway from this lab was that real life implementations of ideas will never work as they did in simulation – not just due to inaccuracies in the physical models used and idealizations made – but also because data will not always come into the system in the same way that it was assumed it would be received in the simulation.

This goes hand in hand with another lesson learned – time management. I realized that we as a team should plan to get the project done days in advance of the actual deadline since several factors (including the sharing system which severely limits our time with the car) are certain to inhibit the progress that we plan to make and postpone our termination of the project.

5.3 Evan Bell Lessons Learned

We took a lot of time correcting technical issues such as troubleshooting the controller, batteries, the Velodyne LIDAR data, and connection issues with swapping robots. Our team also has many different schedules which made it difficult to meet together too often. While I enjoy the application of the ideas we've been learning in class to a real system, there is often a lot of frustrating things that can interfere with smooth progress.

Having to coordinate with others effectively means I will have to be much more intentional about being prepared for things not to go smoothly. Doing the parts of work early in the future will be helpful. We mostly planned to work early in the week respective to the release of labs, which is a good start, but it didn't guarantee the success I thought it would. I could also improve my communication. In summary, better time management and planning more thoroughly will be very useful.

5.4 Artem Laptiev Lessons Learned

My practical technical learnings from the lab are those of using ROS system on real-life robots, external packages use, and code refactoring. We also all learned the importance of proper version control the hard way, losing code in the progress.

As for writing reports, I learned how important it is to keep track of all the developments in the code, and most importantly, record visual checkpoints of the progress. In the future, I will try to work on the documentation while doing the lab instead of starting it after the work is finished.

On the teamwork side, the lesson is the modularization of work. We spent a lot of time working on a single feature together. However, when the development of a feature meets a bottleneck that needs only one person to be solved, the rest of the team has to just "sit around" and wait for the resolution, which is a waste of both time and motivation. The solution here is to modularize work as much as possible.