Lab 3 Report: Wall Following

Team 17

Alicia Zang
Ari Grayzel
Robert Cato III
Sebastian Garcia
Shreya Gupta

RSS

March 5, 2022

1 Introduction

by Alicia Zang

As robots become more mobile and complex, the tasks they can accomplish also increase, such as navigating an unknown environment that humans cannot enter. Autonomous navigation is needed, as robots cannot rely on humans to plan the path. In addition, a robot would also need to detect and avoid obstacles so that it does not crash.

Wall follower addresses both of these requirements, as it will need to observe the environment for a wall to follow while also avoiding crashing into walls. These features would be useful in future labs for obstacle avoidance. It also serves as a basis for any autonomous navigation that would be implemented in the future.

The problem was to have a robot be a certain distance from the wall on a specified side. The robot is moving at a target velocity and would turn along with the wall. We approached the wall follower in three parts: connecting the robot to a manual controller, implementing a safety controller, then finally the wall follower algorithm.

We first connect the robot to a manual controller. The controller functions as a safety precaution that will stop the robot from running code in case of bugs. We can also drive it around to get an idea of what messages and information the robot is sending.

Next, we implemented a safety controller. When the robot is within a certain distance to the wall that we consider to be too close, it will stop moving to prevent the robot from crashing. The robot achieves this by checking the distance in front and the sides with LIDAR.

Finally, we apply the wall follower controller. The robot uses a subset of the scan data to detect a wall and the distance it is from it, then uses a PD controller to adjust the trajectory. We then test the algorithm on a variety of walls and adjust the values accordingly.

To handle sharp turns without triggering the safety controller, we independently implemented a method outside of the regular wall follower controller. This method would cause the robot to turn at the maximum angle when it detected anything within a tested range in front of it.

2 Technical Approach

2.1 Teleop

Section 2.1 By Ari Grayzel

2.1.1 Problem

One of the first tasks involving the physical robot was to connect and operate the robot remotely using the teleop feature.

2.1.2 Set-up and Approach

The implementation of the remote operation was trivial, as we used teleop, a prebuilt package which we simply had to run and gave us full remote control of the robot.

2.1.3 Implementation

For the teleop functionality we simply used the prebuilt teleop package, with it operating at our highest priority mux (mux 0). This ensures that any manual commands which are sent from the joystick will override any other commands the robot may be trying to execute autonomously.

2.2 Safety Controller

Section 2.2 by Ari Grayzel

2.2.1 Problem

We were also tasked with creating a safety controller process operating on the robot which would, when needed, override the velocity commanded by a lower priority process on the robot. This process should take over with enough time to prevent a collision regardless of the speed at which the car is traveling.

2.2.2 Set-up and Approach

The base of our safety controller was a range checker. That is to say, the robot ought to check the LIDAR scan to ensure that it doesn't detect any obstacles within a certain range considered to be "dangerously close". In order to determine what would be "dangerously close" we ran a series of experiments to determine how quickly the robot could safely come to a stop. From this investigation we devised a safety control algorithm which determines the safe stopping distance as a function of the vehicle speed.

2.2.3 Implementation

On our next highest priority mux (mux 1) we constructed our safety controller. The safety controller node subscribes to two topics. One is the laser scan topic and the other is the Ackermann drive command produced by the priority mux system. The safety node has one publisher which publishes to the priority mux safety input.
The safety controller algorithm has two main components. The first determines what the safe stopping distance is based on the current velocity of the robot, and the second part determines if the robot is currently violating that limit and needs to perform an emergency stop.

```
def drive_command_callback(data):
  v = data.drive.speed
  self.min_dist_front = max((self.default_min_dist, v*0.5))

def laser_scan_callback(ranges):
  min_distance_front = min(ranges[front_angles])
  min_dist_sides = min(ranges[side_angles])
  if min_distance_front < self.min_dist_front and min_distance_sides < self.min
    stop()
  else:
    PD_controller()
```

The code which checks if an obstacle has intruded within the minimum distance checks two different safety zones, one for the front of the vehicle and one for the sides. The front detection zone has a much higher minimum distance that will trigger a stop than the sides. The idea with this setup is that the front zone is in front of the robot and is in the direction of travel of the robot, thus it is meant primarily to guard against the robot crashing into something in the course of its

commanded movements. The side zones are mainly checked to see if a dynamic obstacle is approaching the vehicle such as someone walking towards it.

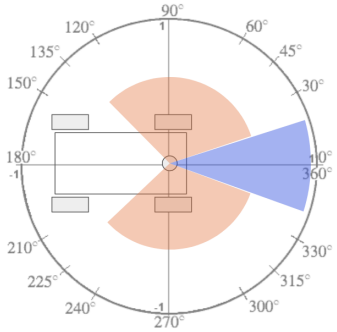

Figure 1: Safety detection zones

The decision to make the safety controller operate as a function of the vehicle velocity was based on our testing results of what constituted a safe stopping distance. Our empirical testing showed that as the velocity increased a larger stopping distance was required to maintain the ability to avoid collision. Our data showed some minimum distance that was needed, even for very low speeds, which we determined to be 0.3m and is the absolute smallest forward stopping zone our algorithm uses. The other trend we observed in our stopping data was that as the speed increased the required stopping distance increased linearly with speed. This can be seen in the figure 2.

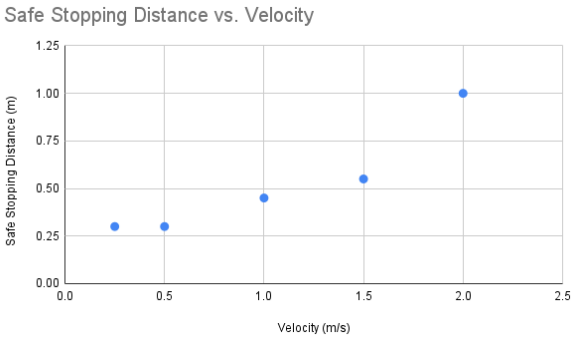

Figure 2: Recorded safe stopping distance for various robot speeds

The methodology for how these tests were conducted can be found in the Experimental Evaluation section.

Our intention with the safety controller is to have the controller only take over when it is absolutely necessary to prevent a collision, i.e. we want as conservative a controller as possible while still preventing collisions. The reasoning behind this is that a safety controller which is too aggressive in preventing collisions may impede vehicle performance and trigger an emergency stop when it is not really necessary. For this reason we opted for a function to approximately match our experimentally found minimum safe stopping distances. The goal and result with this approach is to have a controller which is more aggressive when needed and doesn't unnecessarily interfere with the robot processes in safe circumstances.

–By Ari Grayzel–

2.3 Feedback Control System to Follow Walls

Section 2.3 by Robert Cato III

2.3.1 Problem

The foundation of this lab was to ensure that the robot car follows a wall, parameterized by the side of the robot, the car's velocity, and the desired distance from the wall. The robot's initial state would be set up for "success" in that it would be placed relatively near the wall. However, the wall would not be a simple straight line. Our wall following implementation had to effectively follow noisy and dynamic walls as well as corners using only range data from the LIDAR.

2.3.2 Set-up and Approach

Understanding that the robot shouldn't need any other exteroceptive outside of the LIDAR to follow noisy walls allowed the team to scope the implementation to a relatively simple feedback control system to autonomously control the robot's distance from the wall. Feedback control systems work well to reject disturbances and model uncertainties in order to drive a parameter towards zero. At a high level, after processing the LIDAR data, we compute the distance the robot is from the wall and the angle between its heading and the wall. From these two values, we calculate how far the robot is from the desired distance and how quickly that error changes. Inputting the distance error and error rate into the proportional-derivative control law gives the steering angle we send to the robot.

2.3.3 Implementation

Our wall follower implementation has several steps to process the scan data before it is ready to use the feedback controller. The LIDAR sensor reports detected ranges from a 270 degree semicircle ($\pm$ 135 degrees from the forward direction of the car) in polar coordinates. Detected points are first filtered out

if they are farther than twice the desired distance and then if they are not on the side the robot is supposed to follow. The remaining points are converted from polar to Cartesian coordinates using:

$$X = R \cdot \cos(\theta)$$
$$Y = R \cdot \sin(\theta)$$

We then use Numpy's polyfit function with the Cartesian points to approximate the location of the desired wall using a linear polynomial fit with each point weighed by $\frac{1}{R}$. This approximation helps reject the noise in the range data, smooths out dynamic walls, and prioritizes closer points. We leverage geometry to determine the distance and angle of the wall with respect to the robot's LIDAR.
The wall approximation from Numpy polyfit is a line defined by

$$y = m \cdot x + b$$

The smallest vector from a point to a line is along the perpendicular of that line so the distance vector, $\overrightarrow{d}$, is along the line: $y = -\frac{1}{m} \cdot x + b$. The robot's LIDAR is at $(0, 0)$ so the intercept must be zero and the line is: $y = -\frac{1}{m} \cdot x$.

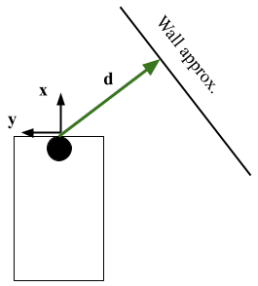

Figure 3: finding the distance vector

Then, $\overrightarrow{d}$ describes the intersection point between these two lines. We can find the intersection point by setting the two equations to be equal:

$$m \cdot x + b = -\frac{1}{m} \cdot x$$
$$x = \frac{-mb}{m^2 - 1}$$

Plug x back into one of the equations to get

$$y = \frac{b}{m^2 - 1}$$

giving

$$\overrightarrow{d} = < \frac{-mb}{m^2-1}, \frac{b}{m^2-1} >$$

As represented in Figure 3, the magnitude of $\overrightarrow{d}$ is the distance from the robot to the approximated wall.
From here we want to determine how quickly the robot approaches the wall. If we assume the total velocity, $\overrightarrow{v}$, goes directly forward with respect to the robot, we can decompose the velocity into a component going along the wall and a component perpendicular to the wall. Using the slope of the linear approximation of the wall, we can form a right triangle with the x-axis, $y = mx$, and $x = m$.

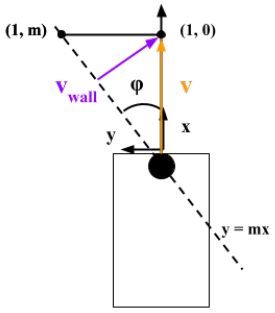

Figure 4: Vector decomposition to find the robot's velocity perpendicular to the wall

In Figure 4, the velocity component perpendicular to the wall is given as:

$$v_{wall} = v \cdot \sin(\phi)$$

We can get $\sin(\phi)$ from the larger triangle using the definition of $\sin(\cdot)$:

$$\sin(\phi) = \frac{m}{\sqrt{m^2+1}}$$

Then

$$\frac{d(\text{distance})}{dt} = v_{wall} = v \cdot \frac{m}{\sqrt{m^2+1}}$$

This is equivalent to the rate of change of the distance error through time if the wall is on the right and the negative error rate if the wall is on the left.
Processing the range data from the LIDAR , we can determine the distance error and how that changes:

$$error = ||\overrightarrow{d}|| - \text{desired distance}$$
$$\frac{d(error)}{dt} = \frac{d(distance)}{dt} = v \cdot \frac{m}{\sqrt{m^2+1}}$$

Using the error and error rate equations above, we can use a feedback control system to drive the distance error to zero effectively; the robot can follow a robot autonomously. The controller implementation uses the following pseudocode:

```
def get_steer_angle(X, Y):
    m, b = np.polyfit(X, Y, 1, w = 1/use_ranges)
    d_vect = np.array([-m*b/(m**2-1), b/(m**2-1)])

    distance = np.linalg.norm(d_vect)
    d_distance = velocity * m / np.sqrt(m**2 + 1)

    error = distance - desired_distance

    if SIDE is LEFT:
        d_distance *= -1

    steer_angle = SIDE * (KP * error + KD * d_distance)
    return steer_angle
```

The gains were initialized using the Ziegler Nichols method for PD tuning. However, as we increased the velocity of the robot, we found that performance decreased dramatically. Through several iterations at velocities ranging from 0.75 m/s to 2 m/s, effective K_P and K_D gains were found to be 2.0 and 1.0, respectively.
–By Robby Cato –

2.4 Front Turning Method

Section 2.4 by Shreya Gupta

2.4.1 Problem

The front turn component of our code is designed to address the problem of the PD controller not handling sharp turns well. Despite tuning the PD controller as described above, the robot still made sharp turns with high errors; more specifically, the position after the robot completed the turn was very close to the wall initially in front of it. In some cases, the robot was not able to complete the turn because the safety controller was triggered during the turn. Due to this, we formulated a sub-problem within the lab: building a heuristic method to make sharp turns independent of the PD controller.

2.4.2 Set-up and Approach

Since the front turn component is made to handle sharp turns, we decided to make a simple controller that would turn whenever the robot detected anything

within a region in front of it and based our design process on simplicity and clean design. After making this initial design decision, we had several free variables to work with: the steering angle we were going to send to the robot, the side bounds of the region, and the front bound of the region. The bounds of the region is shown below in Figure 5.

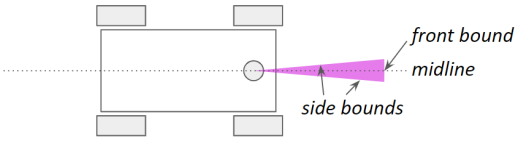

Figure 5: Visualization of the bounds used in the front turn component

We decided to approach this problem by testing a variety of variable combinations on a sharp 90 degree corner and visually assessing the performance. Keeping in mind that our goal was to make sharp turns, we set the steering angle to the maximum in order to ensure the turns were as tight as possible. After that, we created an experimental setup where we set the robot approximately 0.5 meters away from the wall with a desired distance of 0.5 meters and began testing it with a speed of 0.5 meters per second. The distance away from the forward wall was not noted, but it was consistent between runs. We began by setting the front bound to 0.5 meters and modulating the side bounds of the region. The reason we set the front bound to 0.5 meters (exactly the desired distance) was because the robot was going slow enough that the time between it detecting the wall in front of it and beginning the turn was negligible. We started with side bounds of 15 degrees on either side of the robot's mid-line and reduced it until the robot was able to perform a turn successfully. We arrived at the result of approximately 2.5 degrees on either side of the robot's mid-line. After determining the side bounds, we came to the conclusion that the front bound had to be dependent on both desired distance and velocity because when the car traveled at a faster velocity, it traveled a further distance between wall detection and turning. Our final front bound took the form of $c_V * \text{VELOCITY} + c_D * \text{DESIRED DISTANCE}$, where c_V and c_D are constants. We pulled c_D as 1.5 from prior experimentation done in the simulation. We then tuned c_V by making the robot take a sharp right turn at a speed of 1 m/s at a desired distance of 0.5 m and finding a value where the safety controller wasn't triggered and the turn looked good visually. Our metric for a good turn was the robot being approximately the desired distance away from the wall after

it finished the turn, but we did not use quantitative methods.

2.4.3 Implementation

After using the methodology referenced above, we had the following parameters and pseudo-code:

```
# parameters
steering_angle = -1 * SIDE * .34  # radians
side_bound = approximately 2.5 degrees
front_bound = 0.75*VELOCITY + 1.5*DESIRED_DISTANCE

# polar
zone_ranges, zone_angles =
splice scan data using side_bound and front_bound

zone_cartesian = polar_to_cartesian(zone_ranges, zone_angles)

if there are any LIDAR points within zone_cartesian :
        take turn using steering_angle
else:
        run PD controller
```

As shown above, the front turn component code is extremely simple and prioritizes computational speed by minimizing operations. First, the range and angle data gathered from the LIDAR scan is spliced to include only points within the zone designed by the parameters. Then this zone is converted from polar to cartesian coordinates. For as long as it detects any LIDAR points within the designed zone, it will make a turn at the maximum possible steering angle. If there are no LIDAR points in front of it, it will proceed with the PD controller. This code is called in the callback function so the robot makes a check for objects in front of it every time it gathers LIDAR scan data.

3 Experimental Evaluation

3.1 PD Values Tuning

PD Values Tuning by Sebastian Garcia

The process of tuning our PD controller to have an effective proportional and derivative constant started by using Ziegler Nichols method initially, and then through repeated testing along a similar environment each time.

Our starting implementation for our wall follower had its own parameters in place for our simulation, but we had to start from scratch due to moving from

a virtual car to a real car with different specifications and hardware.

First, we used the Ziegler-Nichols method of tuning the PD controller, since we wanted a baseline for our values. This was effective in eliminating many possible combinations of parameters, since we learned that the derivative constant had to be greater than the proportional constant.

We ran our code and realized that this configuration was not effective with the speed that we were using at the time, which was 0.5 m/s. Given that this is a relatively low speed than what we could test after, we knew our parameters had to be tuned to be much better in order to handle higher velocities, and uncommon environments.

To finalize our tuning, we did repeated trials of our car testing slightly different parameters each time. We did these trials with the car starting at a greater offset from the wall than the desired distance, and at a slightly off-center angle. The wall setup we tested was a right wall leading into a corner. The purpose of this was to test whether the robot could get properly positioned from an imperfect starting position, maintain a distance from the wall without oscillating too much, and turn a corner without being inconsistent.

Once we got an effective set of constants from these experiments, we used a step response test, where we drove the car along a wall that had a section sticking out, then reverted back to a normal wall. This was done to start testing how well our PD controller handled uneven walls.

We attempted this multiple times and recorded the controller error by publishing it to a topic. We then saved these trials as bag files to retrieve the errors later to display them graphically using rqt plot. These displays used matlab to show graphical data. An example of the displays we used to visually show and compare errors is shown below.

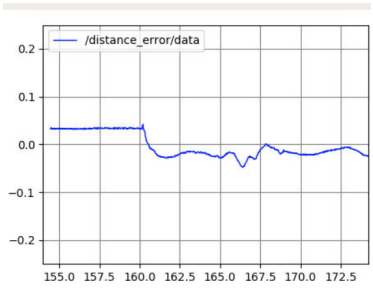

Figure 6: Error data visual

Once we were satisfied with our results after many repeated trials, we stopped changing the PD constants and focused on the other parts of our implementation.

–By Sebastian Garcia–

3.2 Safety Controller

Safety Controller By Ari Grayzel

The safety controller was developed using an empirical testing process. Our goal was to determine how much stopping distance was required for the robot to safely come to a stop. We did this by first setting the robot speed very low (0.25 m/s), setting a conservative bound of the needed stopping distance, and then verifying that the robot could reliably stop in such a short distance. Once we determined that the robot could successfully stop at such a distance, we iteratively shortened the distance until we determined the robot could no longer reliably stop safely.
We tested the safely stopping function by programming the robot to drive in a straight line. When the robot detected an object within its minimum acceptable distance range it would immediately write a speed command of 0m/s via our safety control mux. We took three primary actions to protect the robot hardware to ensure we did not damage it during this testing process. The first mechanism was the fact that we started at very low speeds and slowly increased them as we gained confidence in the safety controller. Similarly, we always began our tests with more stopping distance than we though was needed and then reduced the distance from there. This resulted in any collisions which did occur happening at very low speeds since the robot was almost always nearly completely stopped. The final protective measure we took was using a cardboard brick as the detected obstacle. This obstacle was incredibly light but still easily detectable by the robot. On the few occasions that the robot impacted the brick at low speeds, the foam bumper of the robot simply knocked the brick over and the robot suffered no damage.
Our finalized data from these experiments can be found in figure 2, and these data were used to define the final parameters on our safety controller.

–By Ari Grayzel–

3.3 Front Turning Method

Front Turning Method by Shreya Gupta

In order to test the front turn component, we also had the robot navigate around hallways that required it to use the front turn component. The performance of the robot was assessed visually, and we noted that the safety controller was not triggered and after the turn was made, the robot was well set up to use

the PD controller because it was close to the desired distance from the wall on the other side. We realize that neither of these procedures result in quantitative measures, nor are replicable, so we propose the following setup for further testing. We have code written that prints out error calculated as the quantity |desired distance - minimum distance from wall| divided by time passed once every n timestamps, with n being a free variable. For extremely accurate data, n can be set to 1. The setup of the test would involve placing the robot 3 meters away from the front wall of right hand turn and the desired distance away from the right wall. This is shown in Figure 7. Then, form a data collection grid with two "axes", one axis for desired distance and the other axis for velocity. Test the robot at desired distances of 0.25 m, 0.5 m, 0.75 m, 1.0 m, 1.25 m, and 1.5 m. Test the robot at velocities of 0.5 m/s, 1.0 m/s, 1.5 m/s, and 2 m/s. Do each test twice and note the average of the cumulative error or whether it triggered the safety controller (and therefore completely failed the test) in the respective box. Although we were not able to complete this test due to time constraint's, we have all the infrastructure necessary to do so and believe that it would provide a comprehensive assessment of the front turn component.

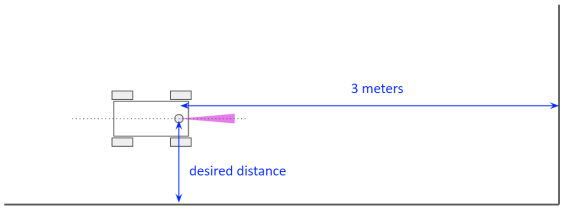

Figure 7: The setup of the proposed testing configuration for front turns

–By Shreya Gupta–

4 Conclusion

Looking back at this lab, we successfully implemented a functional wall follower that works in common environments such as hallways, and uncluttered rooms. We had also implemented our safety control and are able to detect clear obstacles and stop a reasonable distance away, along with sudden obstacles such as someone walking in front of the car.

Due to our implementation strategy of placing the safety controller higher up on the priority list, we realized that there were tradeoffs between safety, and

robustness of our actual wall following algorithm. Since they both take into consideration distances from a detected wall or obstacle, certain environments such as sharp turns would prove to be a challenge.

Another huge point of interest was the concept of velocity as a parameter for tuning and improving our safety controller. Since we initially had general stopping guidelines for when we wanted our robot to stop, we had to adapt to different speeds, and therefore devised a way for our safety controller to activate at different distances for different speeds.

In the future, we would hope to devise a strategy to better balance caution and effectiveness in order for our race car to drive in even more complicated environments.
–By Sebastian–

5 Lessons Learned

Throughout this lab we learned a lot about ROS, PD control, and how to implement a safe and effective wall following algorithm. We also improved our communication skills and learned a lot about working a team. Here are some of our personal reflections for this lab.

5.1 Alicia Zang

One technical lesson I learned this lab was that even though algorithms can work well in simulations, it may not work well with the physical robot. As someone who used to only deal with software, learning that the hardware would have a big impact on the controller itself, rather than the other way around, was enlightening.
The CI lessons I learned in this lab was that it was very important to coordinate and discuss what needed to be done at a specific meeting, especially since the robot couldn't have been split up for each person to work individually on their part. It was also important for everyone to be up to date on what parts were done and to explain how it was implemented so other people could work on it.

5.2 Ari Grayzel

My biggest technical lessons from this lab are mostly just getting more comfortable with the technical tools. I was already decently familiar with SSH but the collection of ROSBAGs, scp-ing them off the robot, plotting with RVIZ and also interfacing with the actual controller on the robot through the mux interface were all good learning experiences for me. Additionally, documenting everything we do when testing any parameter is a very important lesson learned

for this lab. I feel like my default mode previously has been "fiddle with things until it works" but now that we need to formally justify all decisions with data we need to at least record how we fiddle with things.
My main CI lessons are to communicate better with teammates while writing with and modifying shared code. Planning ahead with how we want the different pieces of the code to interact and also notifying when we change things are both very important lessons I have taken away from this lab.

5.3 Robert Cato III

The most important technical lessons I learned from the lab are performing and documenting tests regularly as well as how to work more comfortably with ROS to run the physical robot. I don't think extensive documenting would have been productive but having some record of changes would have better justified some of the floating values used in the lab. On the communication front, I learned how important it is to have update meetings and use tasks lists. It's hard to know how much work we need to get done for each deliverable without updates.

5.4 Sebastian Garcia

Technical lessons I learned in this lab involved getting more experience with the visualization tools in RViz, which helped immensely in recording, using, and displaying data collected during testing. Also, from my teammates, I saw how they made visuals that helped communicate our methods clearly, and am looking forward to working more on that aspect with help from them. Some communication lessons I learned were to outline and have a structure for addressing tasks, since working in a team got a little chaotic by not doing so.

5.5 Shreya Gupta

One of the main technical lessons I learned during this lab is how to work with physical systems and being able to debug problems that aren't entirely software based. Along with this, I learned a lot about the importance of acquiring good data and focusing on that, which is also something I need to improve on going forward. In terms of communication, I learned the importance of breaking up a problem into smaller tasks and defining meeting objectives. Finally, for collaboration, I learned and am continuing to learn how to better match the energy of a group and work effectively in a team where everyone may not have the same work style.