# Lab 3: Implementation of Wall Follower and Safety Controller

6.141/16.405 - Robotics: Science and Systems

Spring 2020

Aaron Makikalli,[1] Aaron Garza,[2] Edward Rivera,[1,3] Kuan Wei Huang,[3] and Kevin Sun[3]

[1]Department of Aeronautical and Astronautical Engineering, MIT

[2]Department of Mechanical Engineering, MIT

[3]Department of Electrical Engineering and Computer Science, MIT

**Introduction (Kuan Wei Huang)**

***Lab Overview***

A robot is an intelligent system that can interact with its environment. In the most fundamental sense, it should be able to move safely with intent, which are the main deliverables of this lab. In particular, we designed a racecar robot that follows the wall autonomously using its LIDAR to observe its surroundings and a proportional-integral-derivative (PID) controller to determine the desired steering angle. We also implemented a safety mechanism that reduces the racecar velocity to zero when it detects objects within a certain threshold.

***Goals and Motivations***

The goals and motivations of this lab could be categorized into hardware and software components.

**Hardware**: Our hardware goals were to become familiar with the racecar and the equipment onboard. This included being able to ssh into the racecar, manually control the racecar with a controller, and visualize the LIDAR scans with Rviz. These tasks helped us design and debug our algorithm in this lab and will prepare us for more complicated assignments in the future.

**Software**: Our software goals were to develop algorithms that allow the robot to follow a particular path while navigating safely. The ability to follow a desired trajectory or wall is crucial, especially for autonomous cars, as we would like a reliable outcome every time. More importantly, we would not want the car to damage itself or its surroundings while it moves, which was the motivation behind the safety mechanism. By combining a robust wall-following algorithm and safety controller, we have created a strong foundation for the racecar that we could build on for future projects.

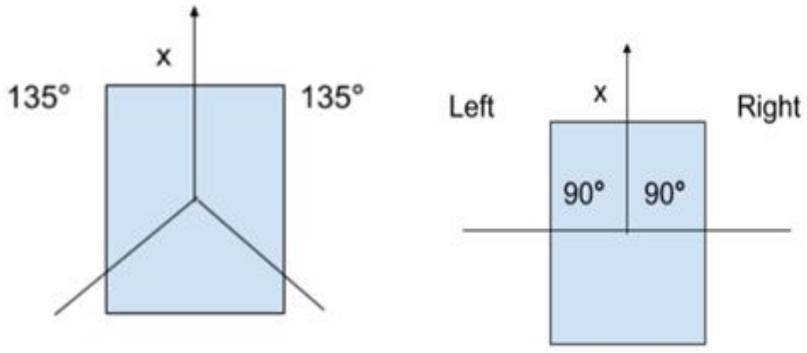
**Technical Approach - Wall Follower (Aaron Garza)**

The first part of this lab involved developing an algorithm that would allow the racecar to autonomously follow a wall on the left or right side of the car. Through a progression of four main computational methods, we were able to develop a successful wall following algorithm. At a high level, the algorithm processes and separates the lidar data based on the specified side, performs a linear regression on the data points to calculate the wall to follow, uses a PID controller to calculate a correction for the car to move toward the desired distance, and finally publishes the corrected drive command. An outline of each step is provided below.

The first step in the algorithm involves processing the lidar data received from the "/scan" topic. Scanning from right to left, the lidar covers a 270-degree range with 135 degrees on each side from the x-axis which points in the forward direction of the car. Depending on which side is specified, a 90-degree range of the scan data is extracted from the total range of

values. For example, if the side corresponds to the left, then the data corresponding to the 90 degree range from the x-axis of the car and moving toward the left-hand side is extracted for analysis. A diagram of the scanning range and range corresponding to each side is shown below:

**Figure 1:** Scanning range of lidar and scan range for the left and right side



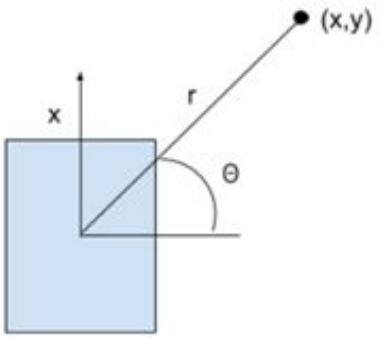

Each point in the lidar data corresponds to a distance from the car to the surface that the lidar detected and the angle at which this distance was measured. In order for this information to be useful for us in detecting a wall, we had to transform each of these distances to Cartesian coordinates using the following formulas where x and y correspond to the point's position in cartesian space, r corresponds to the distance measured, and theta corresponds to the angle at which this distance was measured:

$$x = r * cos(\theta)$$

$$y = r * cos(\theta)$$

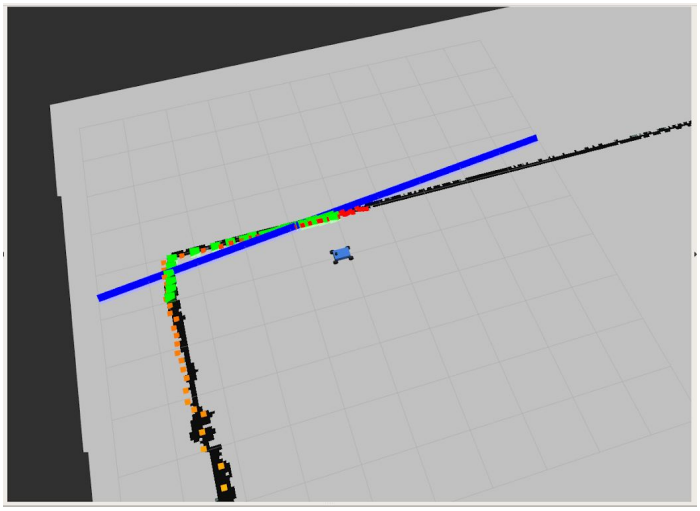
A visualization of how the calculations were performed is shown below:

**Figure 2:** Illustration of the conversion from polar to cartesian coordinates for a data point




After the data has been converted to Cartesian coordinates, the code performs a least squares regression on the selected range of points using scipy's "linregress" function. This corresponds to the line or the "wall" that the robot is supposed to follow. An illustration of the car calculating the line to follow is shown below in a snapshot of the simulation environment:

**Figure 3:** Visualization of the car following the calculated "wall"



Using the calculated line's slope and y-intercept, the algorithm uses this information to begin the adjustment process of moving the robot to the desired distance from the wall. The main point of the algorithm is to follow the wall, but it must do so while maintaining a constant distance from that wall. To maintain this distance, the code uses a PID controller, where PID stands for proportional, integral, and derivative control respectively. The PID controller works by monitoring a desired parameter and comparing the current value of that parameter to a setpoint or desired value. In the case of this lab, the controller monitors the distance of the car from the wall and compares it to the desired distance from the wall which corresponds to the

setpoint. This discrepancy corresponds to the error in our controller and is defined by the following mathematical relationship:

$$error = distance\_desired - distance\_current$$

The current distance of the car relative to the calculated wall is determined from the slope and y-intercept using the following equation for the distance between a point and a line where m represents the slope and b represents the y-intercept:

$$distance = \sqrt{b^2/(m^2 + 1)}$$

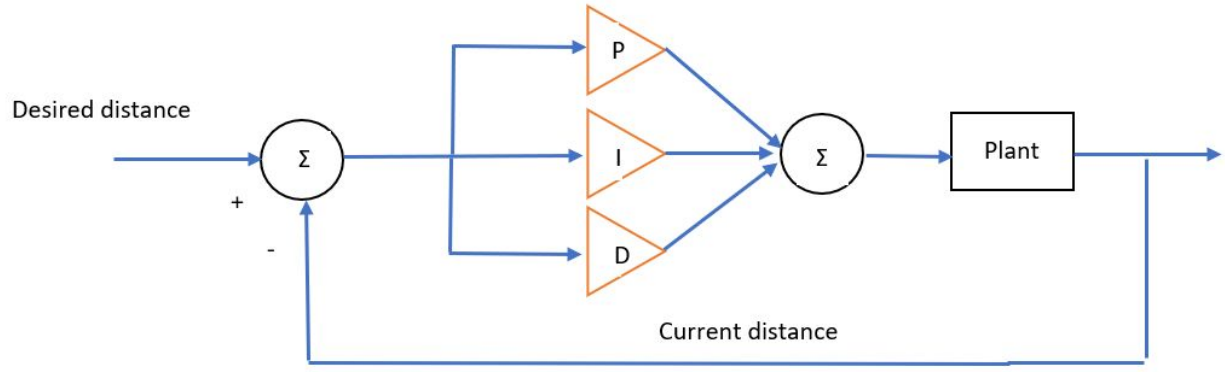
The PID controller consists of three main components which are proportional, integral, and derivative control. The proportional control keeps track of the current error and makes sure the system reaches the desired setpoint. Integral control keeps track of the history of the error and reduces oscillations and steady-state error when the system is at the setpoint. Finally, derivative control keeps track of how quickly the error is changing and helps reduce the settling time of the system to the setpoint. In each of the three components, there are gains, or constants that tune the sensitivity of the controller. These are defined as Kp, Ki, and Kd for proportional, integral, and derivative control respectively. The output of the PID controller consists of a sum of the following equations corresponding to the proportional, integral, and derivative components specified in the block diagram shown below where “dt” corresponds to the time between subsequent laser scans, and “sum error” is the sum of the errors up to the current time.

$$P = K\_p * error$$

$$I = K\_I * sum\ error$$

$$D = \frac{K\_d*((error - previous\ error)}{dt}$$

**Figure 4**: Block diagram of the algorithm’s PID controller




The plant process in the diagram usually consists of a physical model for the system. In this case, we did not have one and processed the output of the PID in a different way using a constant to process the output of the PID. Another parameter the algorithm monitors is the angle of the car with respect to the angle of the wall to follow. In order to follow the wall, the car's positioning angle must be parallel to the calculated wall. The desired angle of the car is calculated using the slope of the linear regression to the selected points using the following equation where m represents the slope:

$$desired\ angle\ =\ arctan(m)$$

This desired angle paired with the output of the PID controller was used to tune the steering angle of the car to point toward and move closer to the desired distance from the wall. As previously mentioned, a plant process was not used to process the PID output. Rather, an experimentally determined constant was used to tune the PID output to be fed into the steering angle of the car. After all of the aforementioned steps have been performed, the steering angle is adjusted using the following equation where $\alpha$ represents the tuning constant.

$$steering\ angle = \alpha * PID\_output + desired\ angle$$

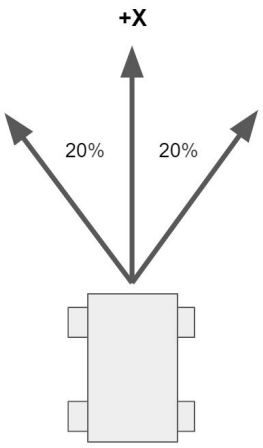
Keeping track of the desired steering angle and the PID output allows the racecar to adjust itself parallel to the wall while moving in the direction of the desired distance from the wall. The tuning constant $\alpha$ was determined by running different wall following scenarios in the race car simulator and finding the value that allowed the car to adequately adjust its position rather than overshooting or undershooting the steering angle. Finally, the steering angle is specified in the AckermannDrive message published to the drive topic that commands the car to move. These steps are continuously performed as the car receives laser scan data.

**Technical Approach- Safety Controller (Edward Rivera)**

In order to prevent potential crashes, a safety controller was added to the car. The safety controller was designed as a node inside a separate ROS package within the racecar catkin workspace, which publishes to a rostopic that takes priority over the wall_follower controller commands. In other words, if the safety controller, also known as "Smokey", publishes a message, then it will override the current drive commands published by the wall_follower, and if Smokey doesn't publish anything, the car continues following the wall.

Fundamentally, there are two possible approaches to implementing Smokey. The first involves directly controlling the car's steering angle. One possible implementation of said approach could introduce another PID controller that adjusts the steering angle according to the velocity in order to achieve a particular distance. Notice there is no optimization for the angle in this scenario as the priority becomes increasing the distance between the car and the potentially dangerous obstacle. This approach would likely require a multitude of tests and a vast program susceptible to bugs. The other approach is to directly control the velocity of the car independent of the steering angle. One possible implementation known as a “hard stop” is to set the velocity to 0 with a single command after some threshold of distance is breached. In fact, this is the approach our team chose as it is less prone to computational bugs and is easier to visualize in a physical test.

**Figure 5.**



The following procedure describes the design of the Smokey. As shown by Figure 5, Smokey takes 20% of the laser scans on either side of the x-axis or direction the car is facing as an indicator of oncoming points. This laser scan range was chosen as 20 points on either side of the x-axis in simulation proved to be reasonable after a few tests, hence 40 of the 100 points in simulation translated to 40% of the points in the Hokuyo lidar. Then, Smokey determines whether 25% of the points are below a certain threshold distance. A quarter of the points were arbitrarily set as a metric and then empirically proved by running several successful tests outlined below in the experiments section. If 25% of the points are below the certain threshold, then Smokey publishes an Ackermann message that sets the car velocity to 0. The aforesaid threshold distance is also determined by Smokey.

Since there is a time from which the car receives the "stop" message and when the car comes to a complete stop, our team decided to run a few tests to determine a function that mapped the relation between velocity and the drift distance traveled during this period. The details of these experiments and graphs are outlined in the next section. Smokey constantly calculates the threshold distance as the velocity of the car changes and makes sure that the car is not approaching a potentially dangerous obstacle. Although this calculation is conducted at every timestep, we have not seen any impeded performance experimentally with both controllers running.

**Experimental Evaluation (Aaron Makikalli)**

***Wall Follower***

**Wall Follower Simulation.** Before implementing the wall follower code on our robot, we tested it in a ROS simulated environment as part of Lab 2. The test suite provided to us in Lab 2 enabled us to test a range of variables, including the robot's ability to follow either wall as specified, correct itself from various start angles, and turn around both convex and concave corners. The code we transferred to our robot passed this test suite with 97% accuracy. Even though we anticipated needing to adjust the PID coefficients and LIDAR data computations for our hardware, the simulation test suite enabled us to feel confident that our code's logic was successful.

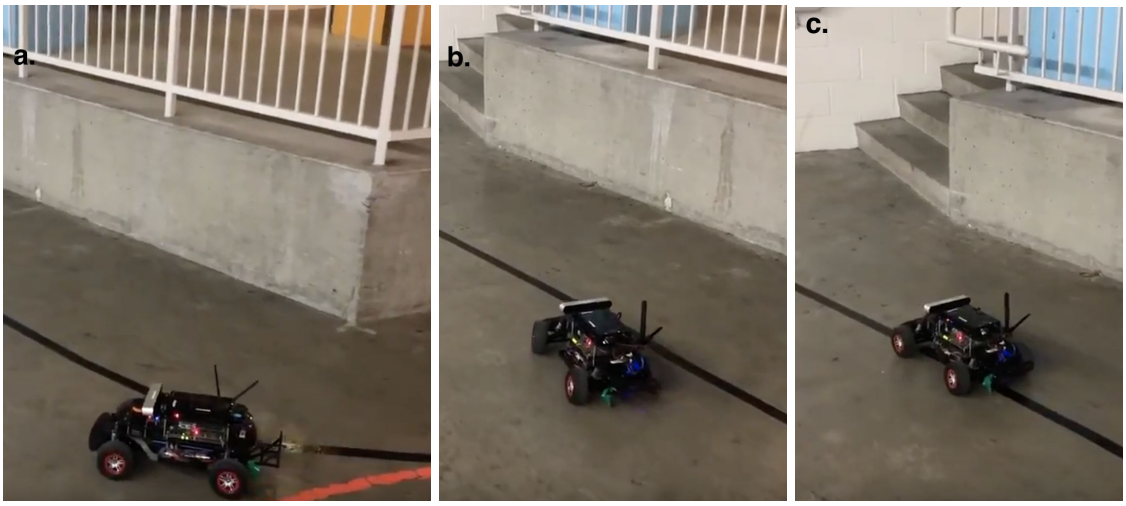
**Black Line Test.** After transferring our wall follower program to the robot, we evaluated the program's accuracy by comparing the robot's path to a black tape line on the ground 1 meter from the wall. The black line test enabled us to visualize the wall follower's accuracy while testing variables such as starting angle, convex and concave corners, and environment. Initially, we tested only the right side; later, testing the left side exposed our most prominent bug. This test exposed critical issues regarding the LIDAR scan's direction and list length.

In our first round of tests, the robot seemed to not follow the wall at all – it turned away from the wall when we tested both the right and left walls. Since the robot's actions were

consistently opposite from what we expected, we determined that the onboard LIDAR's scanning direction was opposite the simulation's LIDAR. Once we adjusted our code to reflect this change, the robot began to approximately follow the wall.

Even after resolving the LIDAR direction, we noticed that the robot turned convex corners at a very wide angle when following the right wall. In addition, the robot exhibited prominent oscillations when correcting its path. At first, we assumed that imperfect PID coefficients induced the high overshoot and oscillations. However, when testing the wall follower with the left wall, the robot seemed to make no attempt to follow the wall at all. The difference in behavior between sides led us to investigate the length of the LIDAR data list, and we discovered that we were only indexing over approximately 10% of the data. Once we adjusted our code to be compatible with any list length, the wall follower followed both walls with significantly less overshoot and oscillation (Figure 6).

**Figure 6.** After correcting our LIDAR issues, our maximum overshoot was less than 15 cm (a). When an overshoot occurred, the robot self-corrected with minimal oscillations (b). At steady state, the robot followed the 1 m line closely ©.




The simple black line test enabled us to quickly visualize our robot's performance and isolate issues. Initially, we used the robot's behavior as it deviated from the line to diagnose coding errors such as LIDAR direction and length. Once we corrected these issues, the robot's path closely matched the black line in both directions, in both an open area and a narrow hallway, and around both convex and concave corners.

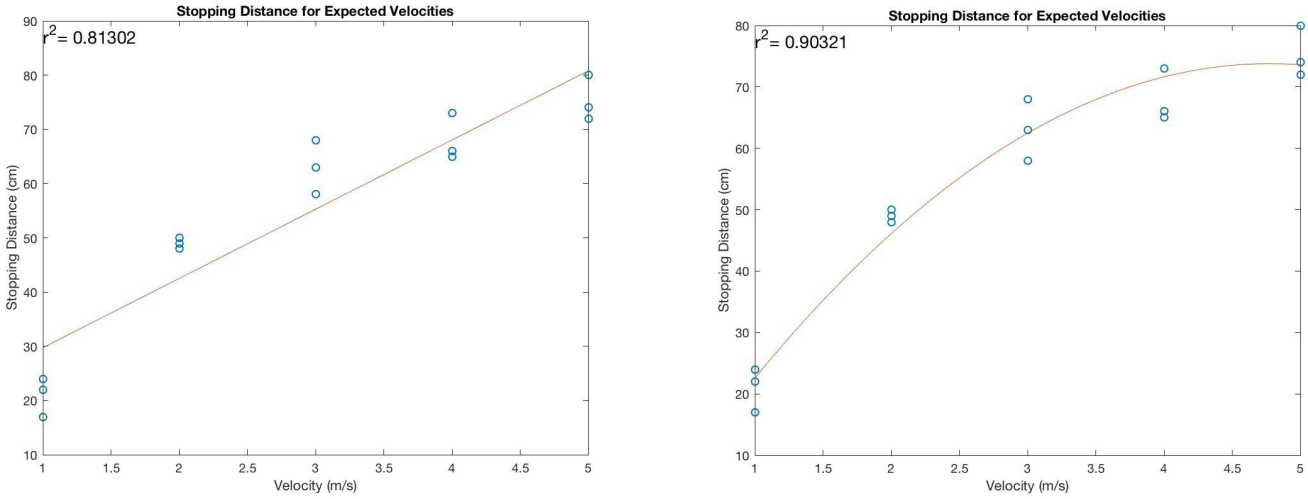
### *Safety Controller*

**Stopping Distance Test.** While implementing our safety controller, we performed an experiment to determine when the robot should execute the "stop" command. After executing

the stop command, the robot drifts some distance before coming to a complete stop; we call this distance the *stopping distance*. To evaluate the stopping distance, we drove the robot in a straight line at full speed, then released the joystick and measured the distance the robot drifted before stopping completely using a meterstick. We repeated this procedure for incremental velocity values up to 5 m/s (according to a TA, 5 m/s is the maximum velocity we should expect our robot to travel during the competition). At each velocity value, we completed three trials to reduce experimental error.

After collecting our data, we used the method of least squares to generate a line of best fit for our observations (Figure 7). At first glance, we considered fitting our data to a quadratic curve, since the quadratic curve elicited a higher correlation ($R^2$ = .903) than a linear curve ($R^2$ = .813). However, the quadratic curve's y-intercept was well below zero, which would cause the robot to stop moving too early for slow velocities. For this reason, we chose to use the linear curve to enable our robot to drive at all expected velocities.

**Figure 7.** The quadratic curve fits our data points better than the linear curve, but the linear curve is positive for all positive velocities.




Finally, we applied a factor of safety (FOS) of 1.3 to our safety controller – in other words, the robot stops moving if the actual forward distance to the wall is less than 1.3 times the calculated stopping distance. We chose the value 1.3 for two reasons: (1) the aerospace industry FOS standard is 1.5; since our robot poses no threat of falling out of the sky, we feel comfortable with a slightly lower FOS. (2) Furthermore, an FOS of 1.3 captures all of our data from the stopping distance test. By adding this FOS to our velocity-based calculated stopping

distance, we are confident that our safety controller will be robust without being excessively cautious.

**Obstacle Avoidance Test.** We tested the safety controller in three incremental phases to ensure that it functioned as expected without inhibiting existing software. In the first phase, we tested to make sure that the safety controller does not hinder our wall follower program during normal operation. To accomplish this, we manually set the safety controller's threshold stopping distance to 0 m – in other words, the robot would execute the stop command only after colliding into an obstacle. Once we were convinced that the two files can coexist, we moved on to the second phase. In this phase, we implemented our stopping distance function and tested the robot's stopping ability by introducing obstacles into its path. We placed a red brick, a human foot, and a clear plastic bin in the robot's path at .5 m/s and 1.0 m/s to confirm that the robot stops once it reaches its threshold stopping distance. In the third phase, we confirmed our safety controller's robustness by removing obstacles from its path and observing the robot continuing its path. From these three qualitative metrics, we feel confident that our safety controller will dynamically respond to obstacles in an unpredictable environment.

One shortcoming of this test is that we only tested our safety controller at low speeds. To ensure that our linear stopping distance calculation works as expected, we will need to test the safety controller at higher velocities. Additionally, we can improve our test procedure by including quantitative measurements of stopping distance. By measuring the actual stopping distance of the robot in various scenarios, we can gauge the accuracy and true FOS of our stopping distance calculation. In future labs, we will make sure to emphasize quantitative data collection in our performance metrics.

**Conclusion (Kevin Sun)**

Robots need to be able to make sense of their surroundings and make use of this information to achieve their goals. In this project, we were able to design and implement an algorithm for a race car robot to locate itself relative to a wall and use this information to maintain a constant distance from the wall without further information on its absolute position or orientation. We tested this in some scenarios and have shown our robot is able to follow relatively simple walls at a set distance as well as make turns at corners, eventually going back to the desired distance. We also designed a safety mechanism to protect the robot from possible errant control. Independently of the programs or code running on the robot, a hard-stop signal is given when the robot detects that there will be an unavoidable obstacle in its current direction. This safety mechanism was also tested and shown to prevent crashes with walls and temporary obstacles even when the control given by the wall follower program would lead to a crash.

In the future, we plan to improve our wall follower to navigate around general obstacles, instead of just computing a single wall, which would decrease our reliance on the

safety mechanism and possibly give smoother trajectories where obstacles are not just straight walls. We also would like to evaluate the robustness of our code in a more principled and thorough manner by testing our method in more scenarios and conditions, as well as determine a more quantitative metric for how accurately a robot follows a wall, if possible.

**Lessons Learned**

***Technical***

- It is important to perform multiple experiments and test as many conditions as possible before assuming all components are working as needed (Aaron Garza)
- We should come up with quantitative metrics to evaluate our robot's performance as we implement more complicated programs. (Aaron Makikalli)
- As a computer science major, I was always told to not believe a bug is fixed simply because a test case started passing, and in this lab I experienced it first hand. Now I never assume that a bug is fixed because a particular subset of desired behavior is exhibited. (Jack)
- Even when given a task with a clear objective, it can still be important to consider how a solution generalizes as well as how best to demonstrate completion of the task. (Kevin)
- Not make any assumptions. Simulation conditions and results could be very different from physical conditions and results. Should run multiple tests to ensure robustness of code. (Kuan Wei)

***Communication***

- Listening to each team member's ideas and evaluating them rather than focusing on our own personal ideas throughout a discussion of a problem enabled us to move forward with our ideas and implementation quickly and efficiently. (Aaron Garza)
- We can save time and improve overall efficiency by dividing key tasks among team members at the beginning of a session rather than all working through the same problem at once. (Aaron Makikalli)
- Planning out and dividing tasks between teammates helped us tackle more problems in less time. This will be especially important in the upcoming task. (Jack)
- Be clear about your own shortcomings and what still needs work in a project. (Kevin)
- Open communication and collaboration allowed us to resolve bugs quicker and come up with creative solutions, ultimately maximizing our work efficiency. (Kuan Wei)

**Appendix A. Debugging Lessons Learned (Edward Rivera)**

As a computer science major, I was always told to not believe a bug is fixed simply because a test case started passing, and in this lab I experienced it first hand. The first Saturday after the lab was announced the team convened in the Stata basement to port the wall_follower code into the car. Using ssh connections and a joystick we executed the code on the car and to our surprise the car turned away from the wall. Perhaps it was set to follow the opposite wall, so we tried following the wall in the other direction, but it also turned away from the wall. Both of these tests were conducted in an open hallway at an intersection, thus it was postulated that it was the noise in an open environment--the calculated line was inaccurate due to multiple objects redirecting the focus of the lidar. We moved to a narrow hallway to minimize noise in the lidar points, but it still wasn't working. We reconnected everything and brought the router closer to rule out the bug was due to network connectivity. Still nothing. If the car exhibits the exact opposite behaviour in both directions and the only change from simulation to hardware was the physical lidar, then perhaps one of our underlying assumptions was incorrect. We postulated that the Hokuyo lidar scanned from left to right instead of right to left and thus we needed to change our code to reflect the reverse splicing of the lidar range. After making this change, the car followed the right wall perfectly! We moved back to the bigger hallway and it followed the right wall again even while turning a complex corner.

Notice that I said right wall. After achieving success, we assumed the bug was gone and we had a reasonable explanation for it. However, we did not immediately test that explanation by following the left wall. Later in the week, we printed out the length of a laser scan and to our surprise found over 1000 points or 10x the number of lidar points in simulation. Normally, this would have been fine, but the wall_following algorithm at the time had hard coded values, meaning that depending on the wall, the algorithm will choose either 16:50 or 51:85 as the range of points. If this was the case, then the car was only using a fraction of the lidar points (several degrees) and was still able to follow the right wall! If both ranges, however, are less than 500, then the car was using the first half of the range, and if the car only followed the right wall, then the Hokuyo lidar actually scans from right to left like the simulation. This realization falsified our original hypothesis and to confirm this new discovery, we checked the online specifications for the Hokuyo Lidar to confirm it scanned from right to left. To fix the bug and subsequently follow walls in the other direction, we changed the code to use variables instead of hard coding the range.

Ultimately, it is important to test hypotheses thoroughly in a variety of scenarios. For example, testing on the left wall or double checking the specifications online on that fateful Saturday could have shed light on the real bug much earlier. Now I never assume that a bug is fixed because a particular subset of desired behavior is exhibited. Finally, as an interesting note, it was odd that switching the hard coded range from one small slice to another small slice on the right half allowed the car to follow the right wall, but my postulation is that one range was yielded points too far back and was this too inaccurate whereas it was lucky that the other range was horizontal enough to the car to accurately detect the wall.