LAB 3: Report

Team 1

Clementine Mitchell, Martina Stadler, Nick Villanueva, Samir Wadhwania, Jake Liguori, Jose Gomez

Overview - Clementine

The aim of this lab was to use the laser scan data from the racecar in order to design a controller which can detect walls and objects and then steer or stop according the information obtained. The laser scan data was used to detect the presence of an obstacle directly in front of the racecar (±30^o from the racecar's forward direction). The safety controller stops the racecar if there is an obstacle detected within some defined distance of the car (it is currently set at 0.5 meters). A regression was used in order to detect the distance and orientation of the racecar relative to the walls and a PD controller was implemented in order to adjust the steering angle of the racecar relative to the wall.

Approaches - Clementine and Martina

The team split up into subteams. There was one subteam that focused on organizing the nodes and ensuring that they were interacting correctly with each other and the racecar. The other subteams each worked on implementing one of the three different nodes:

- A linear regression node for wall detection relative to the racecar.
- A PD controller node to adjust the steering of the car.
- A safety node to stop the racecar if an obstacle was detected directly in front of it.

Subscribing and Publishing - Jake and Samir

One of the challenges we anticipated we would face was successfully having all nodes "talk" to each other, both on the simulation and racecar platforms. Although we had experience with remapping nodes and creating launch files in the previous lab, this time we had to use Python scripts to both create nodes and execute instructions for how to drive within these nodes. By addressing this potential problem from the beginning, we hoped to eliminate lag time between finishing programming in Python and successfully executing the code.

We wanted to create as modular of a system as possible for easy integration into future projects. Thus, we created a new node that would perform the RANSAC wall regression, as well as a controller node that outputs a drive function. Our reasoning for this setup was that a new mission objective would simply require swapping out the controller node, and slightly adjusting the information published from the Regression node.

Node Diagram - Samir

Below is a node diagram indicating our additions to the existing setup and their connections to it:

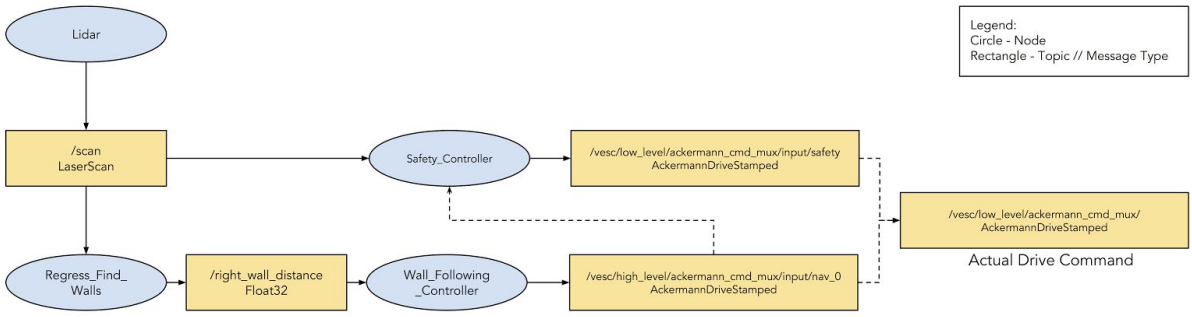

Figure 1: Node setup

Linear Regression Node - Nick and Jose

In order to be able to meet the lab requirements and have the racecar navigate by following the wall directly to its right, a wall detector was implemented. This wall detector worked by taking in the raw laser scan data, converting the values to cartesian coordinates, filtering out infinite values, and segmenting the points into different walls that were stored in a wall dictionary.

In the first of these steps, converting the raw data to cartesian coordinates, a for loop iterated through the raw data, calculated the angle corresponding with that step, and used this angle to generate an x and y coordinate as can be seen in figure x. At the same time that the coordinates were being generated these points were filtered to two regions, a left and right region, which were stored in the coordL and coordR array as can also be seen in Figure 2.

```
# turning lidar into x,y coordinates
for i in range(len(ranges)):
    angle = msg.angle_increment*i-(msg.angle_max-math.pi/2.0)
    x = ranges[i]*math.cos(angle)
    y = ranges[i]*math.sin(angle)
    # Seperating left and right data segments and ignoring
    if angle <= (math.pi/2.0 - math.pi/6.0):
        coordR.append((x,y))
    if angle >= (math.pi/2.0 + math.pi/6.0):
        coordL.append((x,y))
```

Figure 2: Cartesian Coordinate Transformation Code Snippet

After having transformed the LiDAR data to cartesian coordinates and separated it into a left and right region, the next step was to filter the infinite values in the data which correspond to LiDAR points that did not hit an object within the LiDAR range of 10 meters. These same infinite points, given that they represented regions with no walls or objects were then used as flags to

partition points into individual wall arrays to be stored in a walls dictionary containing all detected walls as shown in Figure 3.

```
# Iterating through all points in right region
for xy_point in coordR:
    # Filtering out infinite points, storing points in temporary x and y arrays, and flagging start of region with a wall.
    if (counter<len(coordR) and counter>0 and float(xy_point[0])!= -float('Inf') and float(xy_point[1])!= -float('Inf') and float(xy_point[0]
        x.append(float(xy_point[0]))
        y.append(float(xy_point[1]))
        new_wall_flag = True
        coord_counter+=1
    # Flagging start of infinite region / region with not walls detected
    else:
        new_wall_flag = False
    # Separating individual walls and storing walls in a walls dictionary
    if prev_wall_flag == True and new_wall_flag == False and (counter<len(coordR)+1 and counter>0):
        self.walls_dict['Wall_%d'% wall_counter] = [list(x[index_break:coord_counter]), list(y[index_break:coord_counter])]
        index_break = int(coord_counter)
        wall_counter+=1
    # Edge case: Separating individual walls and storing walls in a walls dictionary
    elif (counter == len(coordR)-1):
        self.walls_dict['Wall_%d'% wall_counter] = [list(x[index_break:coord_counter]), list(y[index_break:coord_counter])]
        index_break = int(coord_counter)
        wall_counter+=1
    counter+=1
    prev_wall_flag = new_wall_flag
```

Figure 3: Cartesian Coordinate Transformation Code Snippet

After the data was sorted to determine which sections represented walls, we were able to fit lines to the data. Throughout this process we implemented two line fitting methods; least squares regression and RANSAC. Both methods developed similar results in fitting lines to the data. While implementing both methods we also decided to expand our wall detection to detecting all walls around the car. To do this, the wall data was further segmented to fit multiple lines to different sections. This allowed us to detect more features apart from just one straight wall. There were some issues with the solvers for the line fitting methods when dealing with entirely vertical walls. Since we knew we would ideally be handling vertical walls relative to the car, we decided to rotate the walls such that vertical walls would be represented as horizontal walls.

In our final implementation to find the distance of the car from the wall and the orientation of the car relative to the wall, we did not use our line fitting methods that looked at all the surrounding walls. Even though we didn't use this functionality, it gave us insight on how we can take these measurements. Our method to find the car's distance and orientation relative to the wall involved looking at a small section of the wall directly to the right of the car. We were then able to use the RANSAC method to fit a line to the data to find the slope of the wall and calculate the orientation of the car. We were also able to find the distance of the car from the wall by averaging the distance of a small section of the wall to the right of the car. The distance and orientation were then sent to the Wall Following Node.

Wall Following Node - Martina

The wall following node is used to implement a simple PID controller that regulates the distance from the robot to the wall.

The node subscribes to the topic /right_wall_distance, the current estimated perpendicular distance between the robot and the wall. It publishes an Ackermann Drive Stamped, message, which includes a speed and a steering angle, to the high level command input topic (/vsec/high_level/ackermann_cmd_mux/input/nav_0), which in turn sends commands to the robot's actuators. In the current implementation, the robot speed is constant, and the steering angle is adjusted by the PID controller.

The PID Controller uses a standard PID design. The input to the controller is a difference, or error, in actual performance compared to desired performance. In the team's case, this was the difference in the current distance from the wall and the desired distance from the wall, where the desired distance was hard-coded into our wall follow algorithm at 1 meter. The error is then scaled by three terms, the proportional, integral, and derivative terms. Each of the terms tunes a specific parameter about the car's response to the error. The proportional term adds a component to the response that is proportional to the current error. The integral term adds a component to the response that is proportional to the accumulated error of the robot over time; this influences the steady-state error of the responses. The derivative term adds a component to the response that is proportional to the change in error over time; this influences the settling time and overshoot of the response.

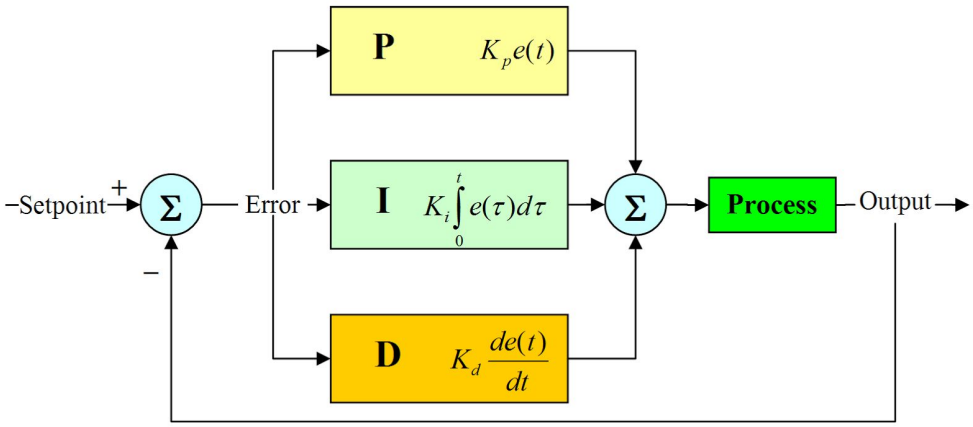

Figure 4: PID Controller Block Diagram
Source: https://radhesh.files.wordpress.com/2008/05/pid.jpg

In every PID controller, gains are used to adjust the relative influence of each of the above terms in the final output of the controller. While these gains can be determined analytically based on the desired controller response, this requires an accurate model of the system dynamics and well-defined requirements. Because the team had neither of these, we decided to tune the PID controller manually. To manually tune a vehicle, the gains are adjusted in the

following order: proportional, derivative, integral. At each step, the specified gain is modified until the vehicle is as stable as possible for the given gain. However, after the team set the proportional gain and the derivative gain, the car's response was considered acceptable, and no integral gain was used. The final gains were:

- Proportional: 4.0
- Integral: 0.0
- Derivative: 0.0005

Safety Node - Clementine

The safety node was designed to take in range data and output a stop message if there was an obstacle directly in front of the racecar (within 0.5 meters of it).

The safety node takes in a message from the laser scanner data. It uses the values of the distance of the points from the racecar from -30º to +30º (`msg.ranges[360:601]`), which are of type float32. The correct values from the large array of laser scanner data were calculate by observing that the laser scanner sweeps out a range from -120º to +120º and finding that there were four data points per degree swept (the angular increment was 0.25º). The data that was outside of the range of the sensor was discarded. The minimum and maximum range values were part of the original message from the laser scanner. From the hardware specifications, the minimum range is 0.1 meters and the maximum range is 30 meters.

In order to be sure that noise within the data set did not cause the racecar to unnecessarily stop in the absence of an obstacle, the data was smoothed (and noise normalized) through a lowpass filter:

```
for j in range(1,len(msg_cut)):
      msg_cut[j] = low_pass_filter * msg_cut[j] + (1 - low_pass_filter) *
msg_cut[j-1]
```

A low pass filter works by essentially averaging out the values with the filter ratio (a weighted average), which can be customized to achieve the desired sensitivity, to eliminate the high frequency oscillations in a signal.

Then if there was still a point in front of the racecar within 0.5 meters, then a stop message was published and output from the node.

Debugging - Jake and Samir

Once we had confirmed the success of our node architecture and node files in the simulation, we moved on to performing tests on the actual hardware. However, we ran into a multitude of issues that required several hours of team debugging to resolve.

The first issue we noticed was the mismatch between our published nodes and the command output nodes. More specifically, we were missing a couple of the *ackermann_cmd_mux* nodes and joy_teleop was not being routed correctly to the input. We realized that the errors were contained within the */racecar/launch/teleop.launch* file, but were unsure of how to fix it. Our intuition was to compare it to the node graph generated while testing in the *racecar_gazebo* package and manually remap nodes - fortunately, we quickly discovered the notice sent by the LAs to update our workspace before we progressed too far.

Updating our workspace was another issue! When we connected to the internet and attempted to run *wstool update*, there were several git issues being run into. Piazza did not provide any advice. After analyzing the error messages, we concluded that git was having trouble updating and merging - in response, we went into each package and manually pulled from the master branch (*/racecar, /vesc, /racecar_simulation, /lab3*).

Once we were able to push our code onto the racecar, we attempted to run *wall-following* and *safety* tests. Our *wall_regression* node quickly threw an error: *failed to import sklearn*. We needed to install the necessary packages for the RANSAC regression. Initially, we simply did *pip install sklearn* just as we had done on the VM. However, the install process started hanging well beyond a reasonable time for installation. Upon further research, we attempted to install with *sudo apt-get install python-sklearn*. This worked, and we were finally able to get all of our nodes published and running correctly.

Our code was designed to follow the right wall at a distance of one meter. However, if we started at this set point and ran ROS, our car started turning to the left and continued to hold this steering angle, moving in a counterclockwise circle. We tried using different initial set points and turning the car to follow the left wall, but we achieved the same result. However, we noticed right from the beginning that our safety code was quite effective. Thus, we reduced the safety threshold to 0.5m (because we were in a relatively confined space) and continued to test wall following abilities. While some team members were examining the wall-following algorithm, others were manually moving the car closer and further from the wall while the car was running to see if distance to the right wall was correctly being detected. We noticed that the car was attempting to hold a steering angle of 15 degrees. Thus, we shifted the steering angle by the value it was currently holding. However, this did little to help us.

We reset the steering angle initial point and spent some time re-examining our code and moving the car manually some more. While doing so, we realized that if we moved the car beyond the desired distance from wall, the wheels would continue to turn in that direction. Thus, we came

to the conclusion that the error may need to be inverted. We simply flipped the sign of the error and our car started working as expected (although we still had to tune the gains). The methodical process of debugging we followed allowed us to find a solution rather quickly.

Future Work - Martina

For future labs, the team would like to implement a controller that combines both the distance and heading data we have relative to the wall. We are in the process of examining the bicycle model. We would also like to be able to use data from all detected walls in the controller.

Another thing to take into consideration in the future is to feed information from the regression node into the safety node in order to be able to differentiate between a wall and an obstacle.

Important Parts of our Code:

The important parts of our code for this lab were:

i. Creating the launch file
ii. Linear regression for wall detection
iii. PD controller for steering
iv. Low pass filter for obstacle detection

For the full code, see the team GitHub.

Lessons Learned
Description about what we learned about working as a team:

Samir: Not all tasks require equal amounts of time. If we naïvely split tasks into 5 sections and assign people, some people will end up doing a lot more work than others. Better communication about who needs help where would be conducive to working more efficiently as a team.

Martina: Some tasks take longer than expected, and if the entire team is not up to speed with the task, it is challenging to parallelize and involve other team members in these more time intensive tasks. We should work to make sure every team member has a better idea of what other team members are doing, so they can jump in and help out if one task is taking longer than another.

Jake: Solving problems that arise in implementation is yet another task that can be accomplished in parallel. When debugging the racecar, we were able to quickly arrive at solutions to problems we encountered by splitting up into groups. Each group attempted to tackle different solving methods, and usually someone was able to find the issue. This only works if the team whole understands it operates as a single unit and solving a problem wasn't just the result of a single individual's work.

Jose: Splitting up tasks at the beginning was a good strategy to make decent progress on multiple fronts, but as we progress and get a better sense of actual work required per task we should consider shifting people from task to task as needed. Also being able to physically meet up, drastically improves the team's rate of progress, so developing a regular meeting schedule should be considered.

Nick: While much of the work could be done independently, the time that we got to work as a whole team was invaluable. We were able to tackle problems together and use all of our expertise and knowledge of our section of responsibility. Unfortunately, we all have busy schedules and finding time when we can all meet is difficult. This also means that we have to use the time when we can meet efficiently.

Clementine: Perhaps we should have secondary tasks for when each sub-task takes a different amount of time so can more equally divide workload. This could be achieved from gathering as a group earlier and more often.