Lab #6 Report: Path Planning

Team 14

Michael Lu
Enrique Montas
Jon Stenger
Grey Sarmiento

6.141 Robotics: Science and Systems

April 15, 2022

1 Introduction (Jon S)

Planning and following a collision-free path is a very important skill in all of robotics, from robotic locomotion and navigation to robotic manipulation. It allows the robot to autonomously navigate from a start point to an end point.

In this lab, three main tasks were completed including: implementing a path planning algorithm, a pure pursuit controller, and finally integrating them together with the localization lab from before. These algorithms were tested in both the simulation and the physical racecar.

At a high level, the process is as follows. First, the path planning algorithm uses a search or sampling algorithm to determine a path for the robot to follow and will avoid collisions. Then the pure pursuit controller will control the speed and steering angle of the racecar so that it will follow the path as closely as possible.

2 Technical Approach (Jon S.)

To approach the technical challenge of implementing path planning, two software modules were developed to handle different components of the algorithm.

1. path_planning.py: A module that is responsible for searching for a path and formatting it correctly.
2. pure_pursuit.py A module that is responsible for following the generated path by controlling the car's steering angle and velocity.

2.1 Path Planning (Enrique M)

Given an occupancy grid map, a starting pose of the racecar, and goal position the module returns a trajectory that the racecar can follow, free of collisions. The two main parts of the module were first processing the input occupancy grid so it was fit for path searching, second the processed map was searched and a collision free path is found and converted to a trajectory for pure pursuit to follow.

2.1.1 Map

This section will break down how the map was processed and used in path planning. This section will also go into how starting and goal poses are received and set on the map.

Processing the Map

When the `"/map"` topic is published to the `map_cb` is called and the occupancy grid is passed in. For ease of computation later the map occupancy data is put into a two dimensional numpy array, the array used by the rest of the module as the map representation. After creating a numpy representation of the map the map gets dilated to avoid picking points close to objects where a wheel or antenna is in danger of crashing. To dilate the map the `scipy.ndimage` was useful providing the `binary_dilation, generate_binary_structure` and `iterate_structure` functions. The `generate_binary_structure(2,1)` output a unit structuring element that the other binary morphological operations use.

$$\texttt{generate\_binary\_structure(2,1)} \longrightarrow \begin{pmatrix} \text{False} & \text{True} & \text{False} \\ \text{True} & \text{True} & \text{True} \\ \text{False} & \text{True} & \text{False} \end{pmatrix}$$

The binary structure is then iterated $\texttt{int}(\frac{self.buffer}{map.info.resolution})$ times. This ratio of of the predetermined `self.buffer` to the `map.info.resolution`, as the buffer size increases more iterations are applied and a square occupied by an obstacle is spread to further squares. If the map resolution value goes down, that means each square of the grid relates to a smaller area in the real world, so to the same size buffer should occupy more squares, but the same area in the real world. If the map resolution value goes down, each square represents a greater area in the real world so a buffer of the same size should occupy less squares, but the same area.

$$\begin{pmatrix} \text{False} & \text{True} & \text{False} \\ \text{True} & \text{True} & \text{True} \\ \text{False} & \text{True} & \text{False} \end{pmatrix} \xrightarrow{\texttt{iterate\_binary\_structure(2)}} \begin{pmatrix} \text{False} & \text{False} & \text{True} & \text{False} & \text{False} \\ \text{False} & \text{True} & \text{True} & \text{True} & \text{False} \\ \text{True} & \text{True} & \text{True} & \text{True} & \text{True} \\ \text{False} & \text{True} & \text{True} & \text{True} & \text{False} \\ \text{False} & \text{False} & \text{True} & \text{False} & \text{False} \end{pmatrix}$$

After the binary structure is generated and iterated the `bianry_dilation(map_grid, structure=structure).astype(int)` a mathematical morphology operation that uses the structured element to for dilating the shapes within an image.

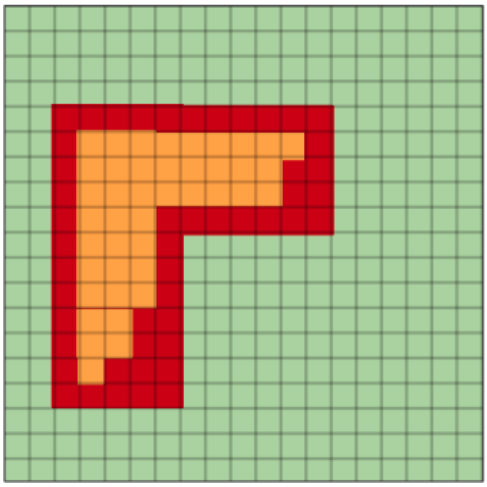

Figure 1: The Dilated Discretized Map. In this example, the green represents unoccupied squares or free space that the racecar can drive in. The orange is the obstacle so the squares are occupied and the racecar can't drive in those squares. The red squares are considered occupied after the map was dilated, these squares are considered occupied and the racecar must find a safer path further from the obstacles.

Establishing a Start and Goal

The path planning module uses odometry data from the localization particle filter to set the start pose of the racecar. When an odometry message is published, the x and y coordinates are converted to coordinates on the map frame. The search algorithm uses the start coordinate as the starting vertex of the graph to search. Similar to the start pose, when a message is published to `/move_base_simple/goal` the x and y coordinate are also converted to the map frame, if the start pose was already set path finding begins.

2.1.2 Path Finding

Path finding breaks into two main parts, first using a search algorithm to find a path from the start to the goal. Next the search algorithm returns a potential path the nodes on the path are smoothed to give smooth trajectories for the racecar to follow with pure pursuit.

A* Algorithm A* is an informed search algorithm, using a best-first approach from a start node to the goal node. A* guarantees finding an optimal

solution if one exists, but it is memory intensive since it maintains a set of nodes the algorithm has explored or used.
While iterating through the queue, the current node is compared to the goal if they match A* returns the current path. Otherwise, the node is explored and its neighbors are added to the queue. Neighbors are found by exploring the eight adjacent spots on the discretized map. If the adjacent square is off the map or occupied by an (dilated) obstacle it isn't considered further. The queue is a priority queue, based on the cost of reaching the end node from start plus a heuristic on the current node compared to the goal. The queue takes the minimum score in an attempt to save some unnecessary iterations and exploring nodes moving away from the goal. After A* finds a path to the goal, it returns a path in the real world coordinate frame making it easier to build trajectories.

Heuristic Function

The Heuristic function used was based on distance from a node to the goal. The distance function used was a projection of the racecar's pose compared to the navigation goal. The heuristic is combined with the cost to get an estimate of how close the node gets to the goal, the cost takes the current distance traveled from the start node and adds the euclidean distance of traveling to the next node.

2.1.3 Constructing Trajectories

After A* returns a path it is converted to a trajectory. Initially each coordinate on the path was put directly onto the trajectory, but this caused trajectories to have many elements and come out choppy. To remedy the choppy trajectory a trajectory smoothing was implemented. To smooth the trajectory the average was taken within a sliding window. Taking local averages and using the average as the trajectory smooths the trajectory by giving the racecar less points where it may need to turn, but the resulting path followed by the reacecar may be slightly longer. After the path points are converted to trajectories, they are published to the pure pursuit controller.

2.2 Pure Pursuit (Michael L.)

Given a list of points for a trajectory to follow, the pure pursuit module calculated the required steering angle and driving velocity at each time step to drive the car along this path. This algorithm involved several steps:

1. Receive the list of points that comprise the trajectory, and precompute matrices that represent the trajectory for pure pursuit to use.

2. Find the closest point on each segment of the trajectory, and choose the segment with that nearest point.

3. Find a point on the trajectory that is a certain distance away from the car. This distance, the lookahead distance, is a key hyperparameter of

pure pursuit, and determines how far along the trajectory the car should be looking to find trajectory points to follow.

4. Compute the necessary steering angle and driving velocity to land on that lookahead point.

5. Vary the velocity and lookahead distance based on the curvature of the path to improve the accuracy of turns and speed of path completion.

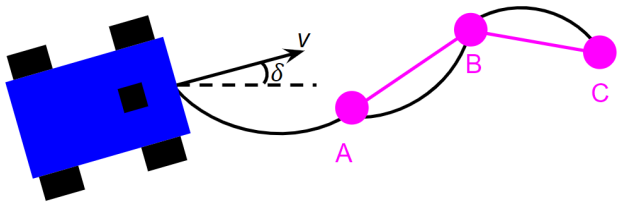

Figure 2: An overview of pure pursuit. Given a list of points on a trajectory, pure pursuit calculates a driving velocity v and steering angle δ for points along the trajectory to steer the car to follow the trajectory.

2.3 Precomputing Trajectory Matrices

The pure pursuit algorithm would first precompute several matrices containing trajectory information that would be useful for the rest of the algorithm, thus saving the need to recompute any trajectory information at every iteration. After receiving the list of points along the trajectory, the algorithm would create an $n \times 2$ matrix T where the i-th point (x_i, y_i) along the trajectory was stored in $T[i] = [x_i, y_i]$.

In addition, a matrix representing the change in x and y between every point ΔT was created, where $\Delta T[i] = T[i+1] - T[i]$ for all $0 \leq i < n$. To store the distance between every point on the trajectory, matrix D was precomputed where $D[i] = \|T[i+1] - T[i]\|$ for all $0 \leq i < n$.

2.3.1 Finding the Nearest Trajectory Segment

After precomputing relevant trajectory matrices $T, \Delta T$, and D and receiving the robot position $P = (x, y)$ and orientation θ in the world frame, the pure pursuit algorithm then proceeded to find the nearest segment along the trajectory:

For a given segment comprising of points $A = T[i]$ and $B = T[i+1]$, where $\|AB\| = D[i]$ and the robot's position $P = (x, y)$, vectors $\overrightarrow{AP}$ and $\overrightarrow{AB}$ were

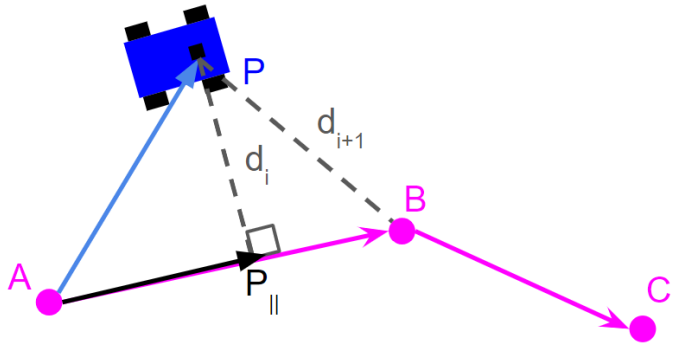

Figure 3: Calculation of closest trajectory segment. For a segment AB on the trajectory, vector projections were used to calculate shortest distance d_i from the segment AB to the car position P. Reference: https://stackoverflow.com/questions/849211/shortest-distance-between-a-point-and-a-line-segment/15017251501725

created. Then, vector $\overrightarrow{AP}$ was projected onto vector $\overrightarrow{AB}$ to create vector $\overrightarrow{AP_{||}}$ containing point $P_{||}$ with distance d_i from P:

$$t = \max(0, \min(1, \frac{\overrightarrow{AP} \cdot \overrightarrow{AB}}{\|\overrightarrow{AB}\|^2}))$$

$$P_{||} = A + t\overrightarrow{AB}$$

$$d_i = \|P - P_{||}\|$$

Because points A and B formed a segment, it was necessary to clip the parameter t between 0 and 1 as to ensure $P_{||}$ remained on $\overrightarrow{AB}$. Then, distance d_i was calculated as the euclidean distance between point $P_{||}$ and robot position P, which would be the nearest point on the segment to the car. A noted edge case was that in the event that t was either 0 or 1, $P_{||}$ would turn out to be either A or B, meaning the distance d_i would be the distance from P to one of the segment endpoints.

After these distances were calculated for each segment, the pure pursuit algorithm chose the minimum d_i and recorded the index i.This index i corresponded to the index of the segment/point along the trajectory that was closest to the car.

2.3.2 Calculating the Lookahead Point

With the index i of the closest trajectory segment, the pure pursuit algorithm scanned the rest of the trajectory from index i, onwards to find a point on the

trajectory to drive towards that is the lookahead distance away. This point was computed by finding the intersection between a circle of radius lookahead distance R centered at robot position $P = (x, y)$ and each segment AB (where $A = T[i]$ and $B = T[i + 1]$) along the trajectory. Computing this intersection point was done by solving a quadratic equation for parameter t and P', which satisfied the equation of the lookahead circle and trajectory line segment:

$$\text{Equation of Lookahead Circle: } R = \|P - X\|$$

$$\text{Equation of Trajectory Segment: } R = \|A + t\overrightarrow{AB} - P\|$$

$$R^2 = \|A + t\overrightarrow{AB} - P\|^2 = (A + t\overrightarrow{AB} - P) \cdot (A + t\overrightarrow{AB} - P)$$

$$\implies t^2(\overrightarrow{AB} \cdot \overrightarrow{AB}) + 2(\overrightarrow{AB} \cdot \overrightarrow{PA})t + (A \cdot A + P \cdot P - 2A \cdot P - R^2) = at^2 + bt + c = 0$$

$$\implies t = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ if } b^2 - 4ac \geq 0$$

$$\implies P' = A + t\overrightarrow{AB} \text{ if } 0 \leq t \leq 1$$

If the quadratic equation had a valid solution, parameter t, which was between 0 and 1, indicating an intersection within the segment bounds, a valid lookahead point P' had been found. While the quadratic could have yielded two solutions, only one solution (the negative solution) was used to guarantee the lookahead point was in front of the car and along the trajectory. In addition, if R were to be increased, the lookahead point would be pushed further along the trajectory, yielding a different P' solution.

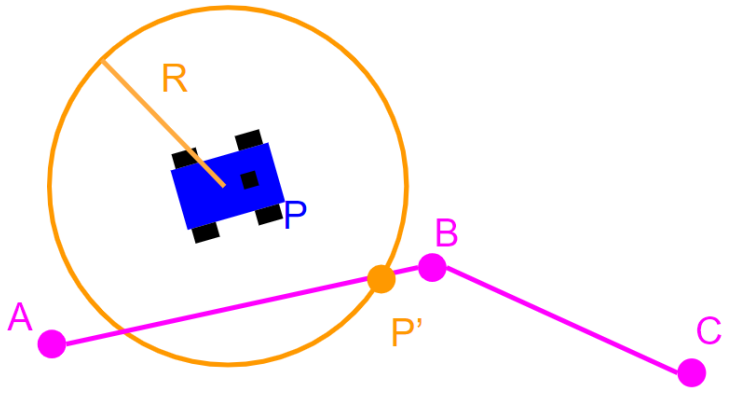

Figure 4: Calculation of lookahead point. For a segment AB on the trajectory and the car's position P with lookahead distance R, the intersection of the circle and segment is the desired lookahead point. Increasing lookahead distance R pushes this point further along the trajectory. Reference: https://codereview.stackexchange.com/questions/86421/line-segment-to-circle-collision-algorithm/86428

2.3.3 Computing Steering Angle and Velocity

To calculate the necessary steering angle to hit the lookahead point, the lookahead point is first transformed into the robot frame via a translation and rotation using the robot's position $P = (x, y)$ and orientation θ:

$$P' = \begin{bmatrix} \cos -\theta & -\sin -\theta \\ \sin -\theta & \cos -\theta \end{bmatrix} (P' - P)$$

Using the transformed point P', the euclidean distance L_1 and angle η from the car to P' is calculated:

$$L_1 = \|P'\|$$
$$\eta = \arctan \frac{P'_y}{P'_x}$$

The steering angle is thus calculated using these parameters as well as the distance between the car's wheels L:

$$\delta = \arctan \frac{2L \sin \eta}{L_1}$$

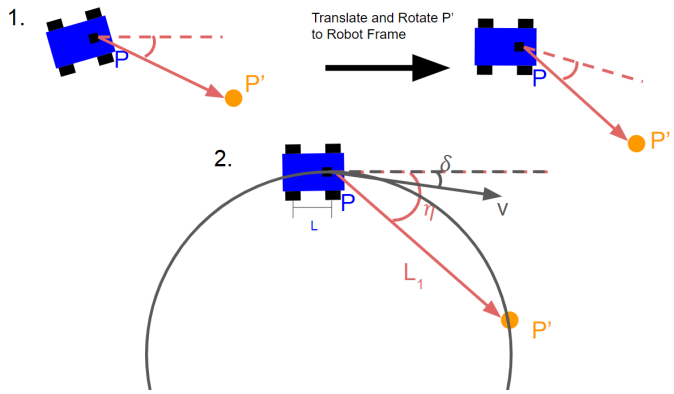

Figure 5: Calculation of steering angle. The lookahead point is first transformed into the robot frame, and then the distance and angle to the transformed point is calculated. These ensure that the steering angle δ can be computed to drive the car along an arc to hit the lookahead point.

2.3.4 Varying Driving Velocity and Lookahead

For the pure pursuit module, the look-ahead distance parameter and the drive speed of the robot affected how quickly the robot was able to converge onto the planned trajectory. If the look-ahead distance was too small or the speed too great, the robot would oscillate as it would over correct its direction. If

the look-ahead distance was too large or the speed is too low, the robot would approach the planned trajectory too slowly and may have collided with the environment because it was not on the planned path. In addition, because the paths the pure pursuit algorithm would follow would contain turns of various curvatures and directions, keeping the same driving parameters would prevent the robot car from adapting to the path for optimal path following.

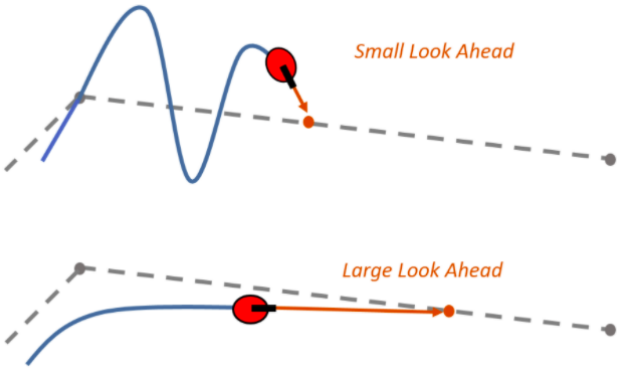

Figure 6: Effect of look ahead distance. Smaller lookahead distances cause large oscillations and slow convergences while larger lookahead distances cause few oscillations, faster convergences, but overshooting and undershooting of a path. A similar phenomenon occurs with high and low speeds, respectively.

To address this, the pure pursuit algorithm would decrease its speed and lookahead distance for sharp turns, allowing it to not overshoot a turn and also be more granular in its trajectory lookahead to detect path changes. The driving parameters were modified as follows, where $v_{max} = 8, R_{max} = 4$:

$$\delta' = min(0.5, |\delta|)$$
$$v = max(0.5, v_{max}(1 - \delta'))$$
$$R = R_{max}(1 - \delta')$$

3 Experimental Evaluation (Grey S., Jon S.)

3.1 Look-ahead Distance and Speed Experiments

In order for the racecar to follow the path quickly and effectively, the lookahead distance and speed needed to be dynamically implemented. The overall idea once again is that the car will slow down on turns and decrease the lookahead, but on straightaways will speed up and increase the lookahead. As such, the

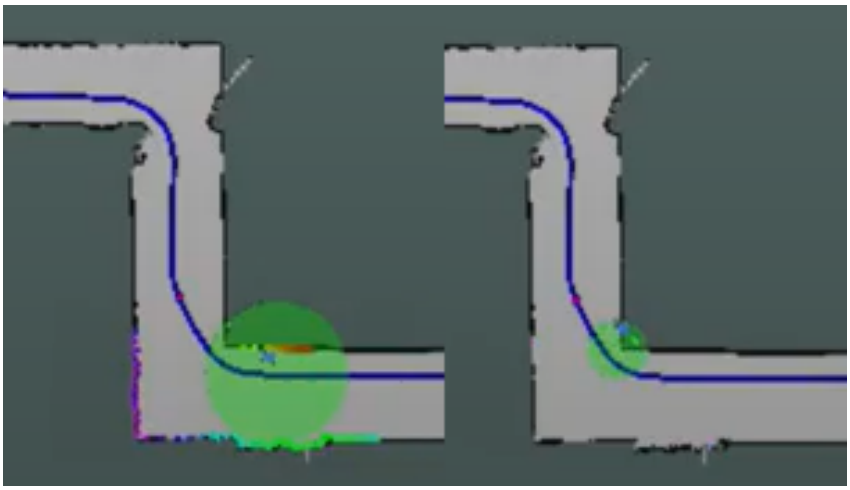

Figure 7: Dynamic lookahead distance. Based on how sharp of a turn the car was making, the lookahead distance, pictured by the green circle, would change. When going along a straightaway, the car's lookahead circle would be larger to anticipate upcoming path changes. When going along a turn, the car's lookahead circle would shrink to catch any sudden path changes that might otherwise be missed with a large lookahead radius.

dynamic parameter schemes outlined in Section 2 provided a good compromise between accuracy and efficiency.

The dynamic parameter schemes for look-ahead distance and drive speed experimentally outperformed the trials where these parameters were set to constant values.

3.1.1 Setup

To assess the performance of different parameter combinations, the pure pursuit module was tested in simulation (RViz) with a fixed published trajectory while completion of the course was timed. In each trial, a different combination of dynamic and constant-valued schemes were tried, for a total of 4 combinations for the two parameters.

3.1.2 Results

Dynamic parameter schemes outperformed the constant parameter schemes in all trials.

Reasonably equivalent values were tested between dynamic and constant parameter schemes. The tested values are as in the table below, where S denotes

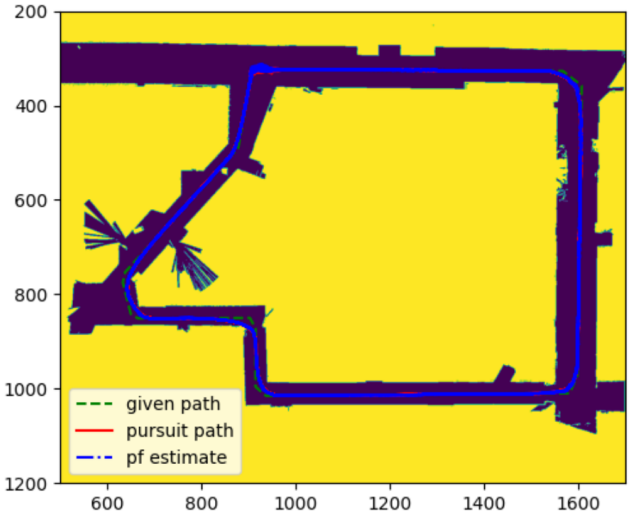

Figure 8: Pure pursuit test trajectory. With a dynamic driving velocity and lookahead distance, the car was able to follow a given path accurately in simulation.

the speed parameter and L denotes the look-ahead distance parameter.

	Speed (m/s)	Look-ahead(m)	Time (s)
Constant S, Constant L	3	2	49.3
	3	1.5	50.1
	3	1	52.9
	3	4	N/A
Constant S, Dynamic L	3	[1.5-4]	50.1
Dynamic S, Dynamic L	[0.5, 6]	[1.5, 4]	45.7
	[0.5, 8]	[1.5, 4]	41.2
	[0.5, 10]	[1.5, 4]	40.6

Dynamic schemes for both parameters outperformed the combinations with constant values in lap time by roughly 10 seconds.

In the first iteration of testing, only the lookahead distance was varied. The optimal range was found to be 1.5 to 4 meters. A lookahead of 1.5 worked well for the turns, however any distance smaller would often cause the car to go off the path or get stuck. Similarly, a lookahead of 4 worked well for the straightaways and kept the car on the path.

With these results, the next iteration of testing involved using a dynamic lookahead distance such that it would increase on straightaways and decrease on

turns. The range from before [1.5-4] overall worked well enough to follow the path, but there were still a few issues that needed to be fixed. For a few of the turns, especially the lower left turn near the wall, the car would sometimes collide with or clip the wall in that section. This meant that the car could benefit from slowing down on turns as well.

For the last iteration of testing, both the lookahead distance and speed were implemented dynamically. Ultimately, this was able to reduce the overall lap time by about 8 seconds. Although the maximum speed of 10 m/s allowed for a faster lap time, the racecar was capped at 8 m/s as it was able to transition into the turns more smoothly.

3.2 Real World Path Quality

When implementing the planning and pursuit pipeline on the real robot, drive command smoothness and qualitative observation of the robot's performance demonstrates that the pipeline was successful in creating and executing a smooth, collision free path through the environment.

To assess the quality of the pure pursuit in the real world, drive commands were recorded in a rosbag file and graphed over time (Figure 9). As speed commands stayed relatively constant with no sudden variations in value while the pure pursuit module was running, it qualitatively concluded that pursuit was smooth with no oscillations.

This conclusion is also in agreement with real world observations. The robot was observed anecdotally to successfully avoid collisions with the environment during test time and did not visually oscillate during every trial.

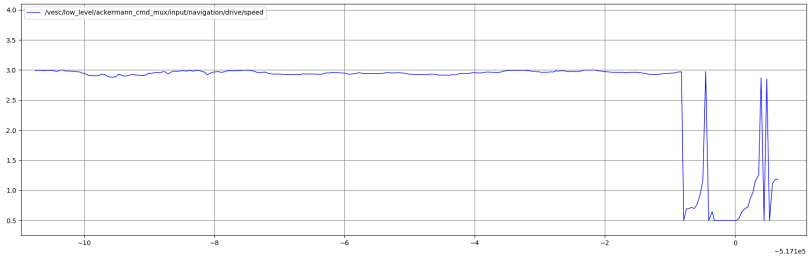

Figure 9: An example of speed commands in a real world demonstration. The spikes on the right reflect the time after the trajectory was complete and pure pursuit was no longer running.

3.3 Options for Further Evaluation

For performance, there is still room to evaluate real world performance more rigorously on a variety of different trajectories that more comprehensively include turns and weaving maneuvers. In this vein, it would also be informative to experiment with different dilation radii of the map for path planning. Because speed is set dynamically, turns are potentially less favorable for efficiency and could be minimized by increasing the dilation radii.

For path planning, a rigorous comparison of the differences in computation time and path quality between RRT and A* may be beneficial, as well as experimentation of different heuristic functions for A* beyond what was implemented in this lab.

4 Conclusion (Jon S.)

Overall, our team was able to successfully implement a path planning algorithm and pure pursuit controller and then integrate them both into the system. The system was tested with various experiments by tuning the pure pursuit controller to effectively handle both turns and straightaways. Overall this lab is very important and a big foundation for the final competition.

In the future it will be useful to further evaluate the path planning algorithms for efficiency and time complexity. It also might be useful to combine different search and sampling algorithms such that we can defer to speed or optimality depending on what is needed.

5 Lessons Learned

5.1 Jon Stenger

This lab helped us learn about path planning and pure pursuit algorithms. Similarly to previous labs we were able to divide up the modules and then combine the modules in the end. We have also improved at this, because the majority of our bugs this time came from within each module and not so much integrating them. This lab also used knowledge from previous labs such as working with the localization module, recording data with rosbags, and debugging/making sure we are using the correct subscriber/publisher topics.

5.2 Michael Lu

This lab helped us sharpen our skills working with ROS and high-level algorithmic thinking with the heuristics behind path planning and mathematics behind pure pursuit. We anticipated, observed, and noted the discrepancies between

theory and implementation as well as simulation and real life. We learned how to make the jump from simulation to test, debug, and validate our software in the real world without the convenience of simulation visualizers and simulation test cases. We also refined our abilities to record real life data and play it in simulation through the use of rosbags and rviz as well as to generate error graphs. This ability to analyze real-time data was crucial in debugging issues with our path planning and pure pursuit algorithm.

Finally, this lab provided an opportunity to collaborate with others in a technical and interpersonal setting, learning how to code collaboratively and determine logistics for working together. This was especially important for this lab, where the modularity of software was conducive to dividing work among group members and learning how to safely combine all components for a functioning final product.

5.3 Enrique Montas

This lab enforced principals of ROS and teamwork similar to the previous labs. This lab in particular involved a classic algorithmic problem path finding, while reinforcing what we've done with controls in pure pursuit. The modules in the lab made splitting work not difficult since we had to combine parts without having to change much within each module, further reinforcing teamwork and good software engineering practices. Tools we used in previous labs came back to help us, from rosbags, cleverly placed print statements, error graphs and jumping between simulation and real life for sanity checks.

5.4 Grey Sarmiento

This lab helped continue to contextualize localization with the implementation of the pure pursuit and path planning. I think that pure pursuit was a more in depth application on controls similar to our wall following lab, and the path planning module was good practice of algorithm implementation. Teamwork went smoothly this lab, as both modules could be developed relatively independently and one split did not rely too much on the other. I feel like we are becoming increasingly comfortable with switching between simulation and real life, and the use of rosbags in both helped us troubleshoot real world problems.