# Lab #6 Report: Path Planning and Implementation with a Pure Pursuit Controller

Team #12

Evan Bell
Yasin Hamed
Artem Laptiev
Cruz Soto

Robotics Science and Systems

April 27, 2023

## 1 Introduction by Artem Laptiev

In this lab, we developed a framework for a racecar to plan and follow a path through a known environment. Given a map of the environment, the starting position of the racecar, and a target position to reach, the goal is to accurately navigate the racecar to the target location.

The approach involved implementing a BFS-based search and pure pursuit to accurately navigate the racecar to the target location.

The goal of path planning was to find the most optimal path to a goal position given a known occupancy grid map, representing the environment in a way that an optimal path can be found, and then finding an optimal path. Two methods of path planning were explored: search-based planning (SEP) and sample-based planning (SAP). While SAP was briefly explored, it was decided that SEP was more useful due to its ability to guarantee an optimal solution, generate a reusable map graph, use the occupancy grid format of the map, and avoid excessive searching when a valid path does not exist.

In addition to path planning, we also implemented and tuned the pure pursuit algorithm, which helps the racecar follow the planned path. We adjusted the lookahead distance and other parameters to improve the racecar's performance.

Overall, our framework provides a reliable and efficient solution for a racecar to navigate through a known small-scale environment.

## 2 Technical Approach

### 2.1 Path Planning by Evan Bell

The goal of path planning is to find the most optimal path to a goal position. We assume the map is known, so this becomes a problem of representing the map in a way that an optimal path can be found, then finding an optimal path.

In our situation, the map is given in the form of a known occupancy grid map. This occupancy grid includes values of 0 for known open space, values of 100 for known obstacles, and -1 for unknown spaces. Each value in this occupancy grid represents a square portion of the actual world frame which has length given by a resolution value ($m$/pixel).

Two methods of this path planning are **search-based planning (SEP)** and **sample-based planning (SAP)**. Search-based planning involves discretizing the map into a graph of points and searching for a shortest or least costly path in the graph. Sample-based planning does not require a discretized graph (although it can). Instead, possible paths are created by randomly adding points to a tree until a solution is found. While we explored sample-based planning briefly, it was decided that search-based planning is more useful for the following reasons:

1. SEP can guarantee an optimal solution. Sampling-based planners are fast, but can sometimes result in unusual-looking and possibly inefficient paths.

2. SEP generates a map graph. The map is not expected to change often (likely never), so the generated graph can be reused.

3. The map is already given in a occupancy grid format, which can be turned into a graph for SEP( or even used as the graph itself).

4. If a valid path doesn't exist, SEP will not spend excessive time searching, which SAP might.

The first step of SEP is to turn the occupancy grid into a graph. Although the grid can be used directly as a graph, there are qualities about the map we may wish to change.

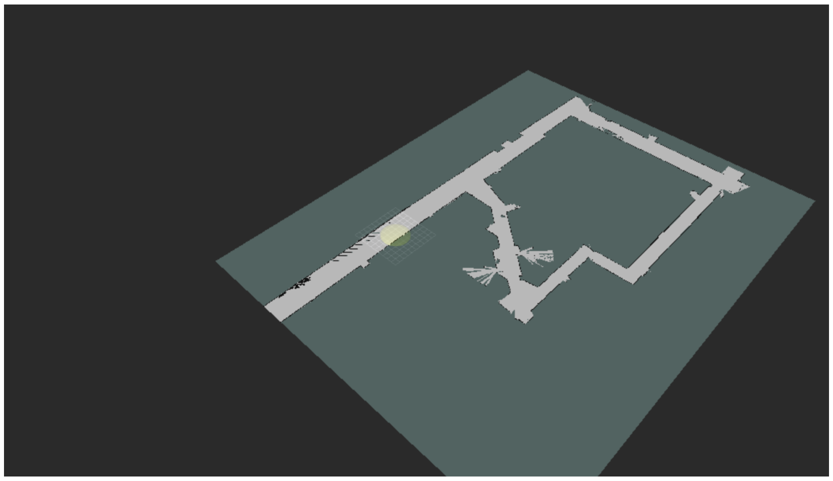


Figure 1: Stata basement map

This grid is large, about 1730×1300 values for the example map in Figure 1. Each value represents a 0.05 by 0.05 square of real space. The car is roughly 0.2 meters wide, so a path traversing a single grid value isn't actually feasible for the car. Also, the graph size will be much larger than needed and time will be wasted on unnecessarily high resolution paths. Turning groupings of values into a single node is more useful, so we turn each grouping of $4 times 4$ grid values into a node in the graph. This makes each node in the graph have roughly

$0.2 times 0.2$ meters of resolution to the real world.

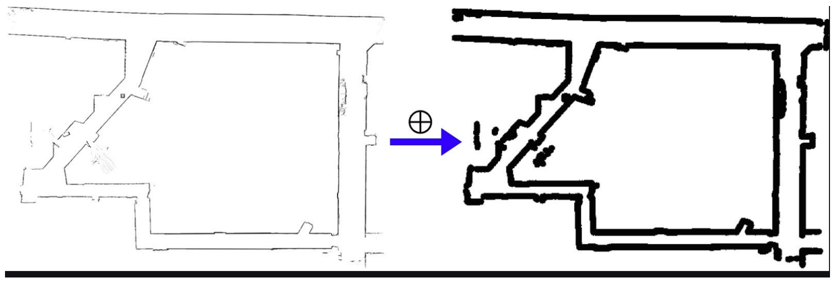


Figure 2: Wall thickening - Creating a buffer around all obstacles for better paths.

Secondly, the map only marks obstacles at their actual point measured. Search algorithms often tend to cut corners close since they are attempting to minimize distance. It is likely that the path chosen will be very close to walls in a situation where a turn is necessary because closeness to corners minimizes distance. However, the car has real dimensions which are not represented by the point in the graph, so the path may be collision free in the map space but not in reality. To solve this, we only want to consider open space beyond a buffer distance from each obstacle, so that any non-collision path chosen in the representation space is also collision free in the real world. This is called *wall-thickening*, shown in Figure 2.

Both these goals are accomplished by applying convolution on the occupancy grid, represented by a 2d numpy array. Choosing a stride of 4, we make sure the output graph has a single value representing 4x4 values in the original graph. Choosing a 15x15 filter of ones, we thicken the walls (now represented by a positive value in a new grid) by 7 pixel values in the original occupancy grid in each direction.

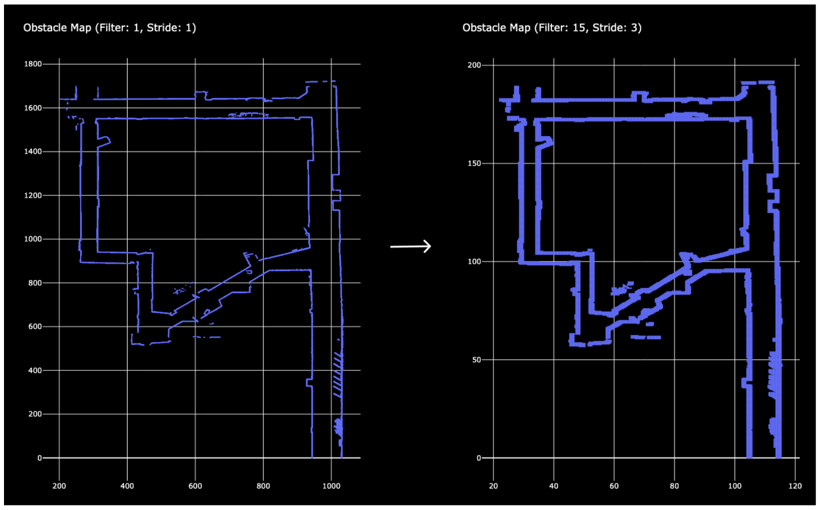


Figure 3: Application of convolution to the map. Axes are indices into the resulting grid.

Once convolution is done on the original map, we save the grid index of each value 0 open coordinate as a tuple of positions $(x, y)$. We save only these open coordinates because we can assume all other locations are not reachable (either a wall or unknown), although this is mainly for the purpose of reducing the space complexity of the graph we use. Thus we have a graph representation of the space.

This graph is a representation relative to the map occupancy grid, not the real world frame. To use this graph for real world path planning we can convert the real world start/end coordinates into map frame coordinates, perform a search on the graph for an optimal path, then convert the path into world frame coordinates again. Because the graph is a essentially a set of coordinates from a grid, the path will be a set of adjacent points from start point to end point.

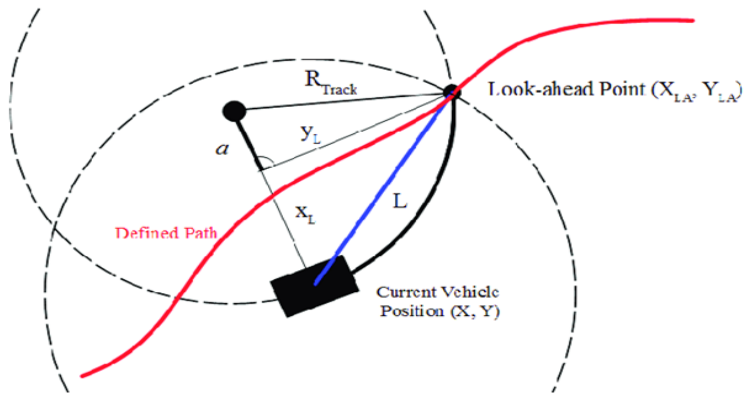
For an optimal path to be found, a path planning algorithm must be used on the graph. This graph represents real space, so we can an edge will only exist between adjacent coordinates, and each edge will be of the same length, so this graph can be treated as unweighted.

BFS or A* (a modification of BFS) is usually considered to be the best unweighted algorithm to find shortest path. BFS requires searching through every node of a distance $i$ from the source before searching any of distance $i+1$, meaning that BFS searches nearly the whole graph when the end node is far from the start. A* is a greedy version of BFS, meaning that a heuristic is used when deciding which nodes to consider first when searching the graph. The heuristic we used was the distance from a node to the goal. Because these nodes are coordinates we can find this distance fairly fast, and it provides an accurate way to measure the closest nodes to the goal. BFS guarantees a shortest path within the graph. So, if our graph is a sufficient representation of the map and the real world space, the path generated will be the shortest.

## 2.2 Pure Pursuit by Yasin Hamed

Once path planning has produced a viable trajectory between the source and the destination locations in a map, a method to have the robot track this trajectory must be developed and implemented. This can be accomplished via the use of pure pursuit. Pure pursuit is a theoretically simple yet very robust technique to have a robot follow a trajectory since it relies on very quick and reliable geometric calculations which allow the robot to have response times and tracking performance which is purely a function of other parameters integral to the technique. This leads to a much more simple and straightforward implementation which contrasts that of other algorithms such as the localization algorithm from a previous lab which was very computationally intensive and caused many other problematic considerations to arise throughout its implementation.

The actual algorithm relies on the principle of a look-ahead distance which is used to identify the next point which the car should direct itself towards. This look-ahead distance can be thought of as a radius which defines a circle around a car. Assuming that the the distance between the car and any point on the trajectory is within this look-ahead distance (which is a valid assumption given decent performance of the pure-pursuit controller), there will be at most 2 intersections between this circle around the car and the trajectory to be followed. Whichever point is closer to being in front of the car is the point that is chosen to be followed. Quantifying which point is "closer to being in front of the car" has a very natural mathematical interpretation which can be explained using a dot product between the vector defining the car's orientation and the vector defined by the car's position and either point of intersection. If the dot product between these two vectors is positive, then the angle made between the two vectors is less than $90°$, and this is closer to the front of the car than a vector that results in a negative dot product, indicating the car would need to turn more than $90°$ to follow that point. Figure 3 provides a geometric visualization of all of the different components of pure pursuit interact with each other.


R_Track
Look-ahead Point (X_LA, Y_LA)
y_L
a
x_L
L
Defined Path
Current Vehicle Position (X, Y)
Figure 3: The circle defined by the look-ahead radius, $L$, and the trajectory intersect at two points, but only the point closest to the front of the car is chosen to be followed. The arc traced in a solid black line is the path that the car projects it will need to take to reach the point at a constant steering angle.

Once a point has been identified to reach, the steering angle must be computed for the car such that it can drive forward with this constant steering angle and the arc that it travels on will intersect the desired point to be reached. This angle can be simply computed using equation 1, using the already known angle between the vector from the car to the desired point and the vector defining the car's orientation.

$$\delta = \arctan \frac{2L \sin \eta}{L_1} \tag{1}$$

The angle $\eta$ used in equation 1 is visualized in figure 4.

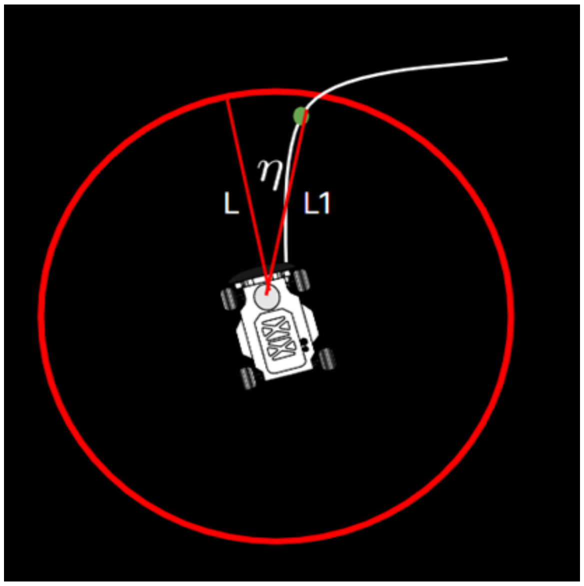


Figure 4: $L_1$ is the distance between the car and the desired point and $L$ is the look-ahead distance for the pure pursuit algorithm.

This process of "looking ahead" and adjusting the angle of the car to try to intersect the trajectory is done continuously throughout the trajectory-tracking of the robot, and this continuously computed nature of the algorithm provides the potential for very quick responses by the algorithm to changes in the trajectory. Additionally, since the car is always looking ahead to find the next point to follow, it is less sensitive to perturbations present directly in front of it, and for this reason, the actual path that is traced by the car tends to be a slightly more smoothed out version of the actual trajectory (this is assuming a long look-ahead distance relative to the frequency of perturbations in the trajectory), and this type of smoothing behavior is very desirable in our application since it minimizes the risk of damage to the car.

## 3 Experimental Evaluation

### 3.1 Choosing Search Algorithm by Artem Laptiev

During our experimental evaluation, we compared the performance of BFS and greedy search algorithms for finding the shortest path in a graph of 17K nodes.

BFS, or breadth-first search, is a search algorithm that explores all the vertices at the current depth level before moving on to the next depth level. It starts at the source node and explores all the neighbors of that node before moving on to the neighbors of those neighbors, and so on, until it finds the destination node (or determines that it doesn't exist). BFS guarantees that it will find the shortest path between the source and destination nodes, as long as all edges have the same weight.

Greedy search, on the other hand, is a search algorithm that chooses the node that looks the most promising at each step, based on some heuristic or rule. For example, it might choose the neighbor that is closest to the destination node, or the neighbor with the lowest weight. Greedy search does not guarantee that it will find the shortest path between the source and destination nodes, but it can be much faster than BFS if the heuristic is well-chosen and the graph is large.

We found that BFS produced a more optimal path and ran efficiently in less than 1 second. On the other hand, we observed that greedy search often produced suboptimal paths, which could lead to less stable driving of the car, as shown in Figure 4. Therefore, we chose BFS.

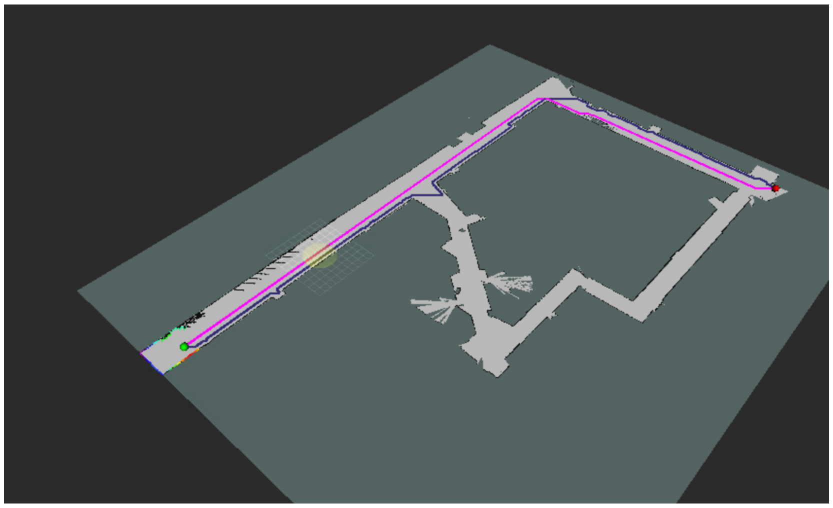
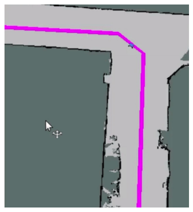


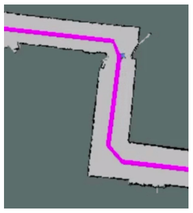
Figure 4: Paths by BFS in pink and by greedy search in blue.

### 3.2 Pure Pursuit Controller by Cruz Soto

Like the other controllers from previous work, the pure pursuit algorithm has a tuning parameter that should be set for optimal performance. For testing, the group established testing of the controller at two different corners of the default trajectory. The second curve in particular provided a straight segment with two bends followed by another straight section, providing a regular system by which to assess the system.



(a) The first curve on the track



(b) The second, curved track segment.

Figure 5: The two testing tracks used to assess the lookahead distance

Error was defined as the shortest distance between the localized pose of the robot and the closest trajectory point. Despite having relative success with the localization in the previous lab both in simulation and in real life, the team opted to go with the staff solution due to the team's submission specifically failing to localize the car at the corner pictured in Figure 6 in simulation (although not in real life).

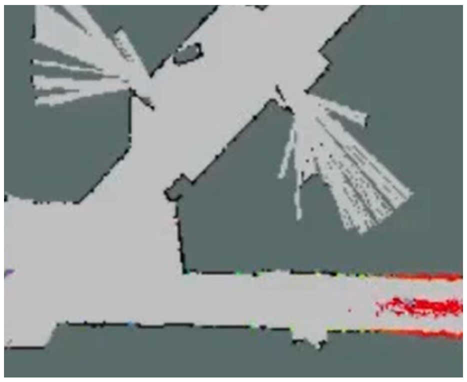


Figure 6: The corner at which the localization failed, with the trail jumping to the other side of the map.

The error distance was calculated with minimal memory usage, as it is just the closest point on the line between the previous and current lookahead points (cross-track error). The error was outputted to rostopic /error whenever the robot's pose updated and was also echoed as a loginfo that was copied to a script in python due to the team often forgetting to record rosbags (and this method was easier to execute with a single person than recording rosbags and changing ros parameters).

We first varied the lookahead at five different distances, plotting a simplified transfer function with the small angle approximation for five distances. The speed was kept constant at 1.25 meters per second, which is the fastest speed that was implemented during the localization and visual servoing labs. However, the speeds on the final pure pursuit controller, when tuned, were able to go faster (as discussed later in this section).

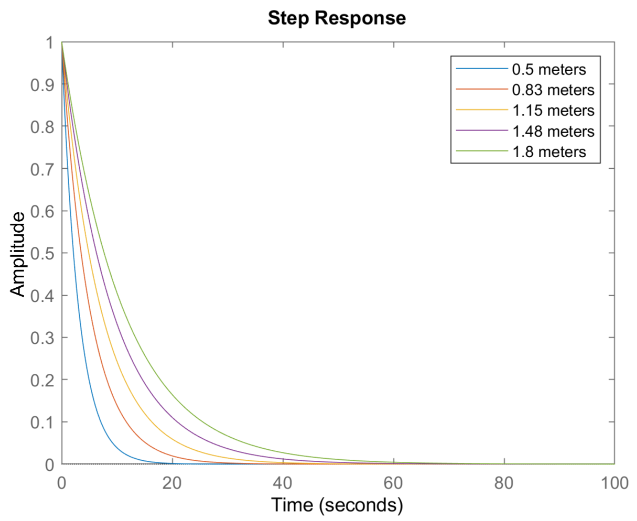


Figure 7: A series of step responses as given by a Matlab simulation and RViz. As will be found later, this is representative of a step response in real life!

From the step plots seen in Figure 7, it would make sense to pick the smallest lookahead distance, as this would provide the fastest convergence rate. This was the mentality testing was approached with, providing room for very low lookahead distances. However, it emerged quickly that the lookahead distances that were too close to the car had an issue not represented in the step function above, as the peak overshoot isn't encapsulated with the trigonometric approximations made ($sin(\theta) = \theta$ and $tan(\theta) = \theta$). As can be seen below, however, there is significant overshoot and oscillation in cases with a low lookahead distance, meaning that while the convergence rate appears fast, it takes a long time to approach steady state error.

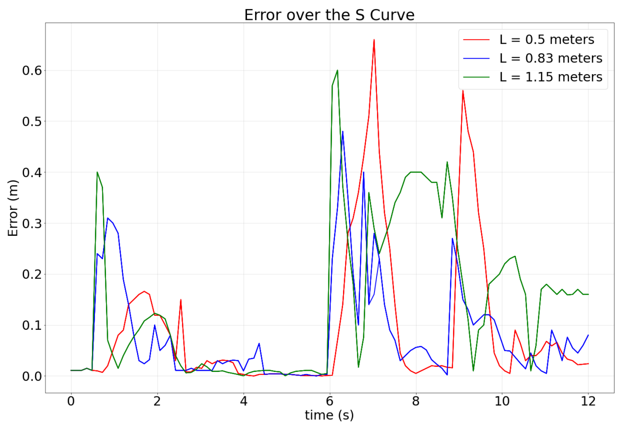


Figure 8: The response to different lookahead distances on the s-curve.

Another issue is apparent with long distances as well. Because of how far the lookahead point is, it has a large steady state error (or takes a long time to converge to the true $e_{ss}$). This is most pertinent in the green graph (L=1.15 meters) in Figure 8.

As a result, it seems that long lookahead distances work best on long tracks, where there is no need for high adjustment, while the short lookahead distance works best on straight sections of track with no high lookahead distance. This is again supported by the conclusion that the lowest demonstrated (average) error is across the red segments of the graph. Even if there are higher peaks, there is the fastest correction with a small radius of the lookahead circle. On the contrary, if a small lookahead is used when the car is on a straight section, it will induce oscillation (Notice there is nose between three and six seconds for in 8 for the short lookaheads but not the longest). As a result, the team used a medium of $1.1\frac{m}{s}$ in implementation, but a program that can recognize curved sections and reduce lookahead may also be implemented for the final challenge. The team also explored speed relations in the real life implementation.

### 3.3 Pure Pursuit Controller by Cruz Soto

A real life implementation on the default path proved that speed has a good relationship with the turning parameter on the car. Testing at two different speed settings, the error went up with speed keeping the same lookahead distance.

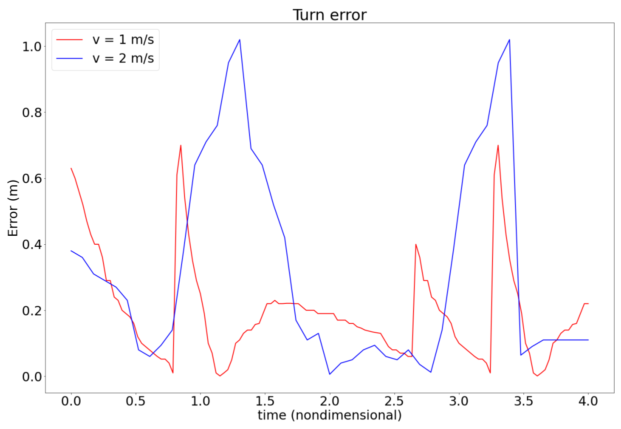


Figure 9: The time here is normalized to the respective time it took to traverse the curve. In this way it is easier to see the upscaling of respective peaks as the velocity increases.

After optimizing for the best lookahead at the new speed, an ideal proportion of 0.75 times the velocity was the best to implement, and is looking to implement it at faster speeds on straight sections.

## 4 Conclusion by Cruz Soto

In the lab, the team implemented an A* Breadth First Search path planning algorithm after comparing its performance with that of the Rapidly Exploring Random Tree. While this algorithm definitely runs faster than A*, the BFS algorithm the team went with did a better job of selecting optimal trajectories. In a real-time implementation the shortest path will almost always be the fastest (excluding paths that are far curvier than straight due to difficulty in traversing them with a pursuit algorithm). For the pure pursuit controller, the team needed to tune the lookahead distance, or how far of a radius from center the robot can look for points on the trajectory to follow. This ended up being approximately 1.1 meters in simulation. However, when varying the speed of the car the team ultimately found an almost linear relationship where the car lookahead was ideally approximately 0.75 times the car's speed. The team also found that the speed should be different between curves and straight sections, as the long paths generally should have longer lookaheads to reduce oscillation and curves should have short lookaheads to not turn too early or not induce a strong correction in the turning angle. This outlines the next steps the team would take in optimizing the car's performance in the State basement and other environments where pathfinding with localization will be implemented.

## 5 Lessons Learned

### 5.1 Evan Bell Lessons Learned

The four members separated into pairs for path-planning and pure pursuit, which worked out well. We spent far more time thinking about path planning before actually implementing anything which helped a lot in actual development. We also made sure to communicate overall to make sure that the path planning and pure pursuit implementations could be joined easily because they use the same path format. The implementation for path planning went pretty smoothly and we were generally able to find bugs quickly.

I am also happy with the implementation. Having a simulated system perform equally well in the real world can be very challenging, and I think we experienced far less issues with it because of a better design overall. Coming into the final project I feel prepared to make a strong well-designed approach to each challenge before implementation.

### 5.2 Yasin Hamed Lessons Learned

With regards to the technical implementation of this lab, I learned a lot of new techniques surrounding the debugging process when dealing with programs running in ROS. Chief among these lessons was learning to modularize testing environments in ROS using several different launch files where each file is tailored towards a different set of testing parameters. The result of doing this is a much quicker, smoother, and less frustrating debugging experience which allows our team to get the implementation of the car working much more efficiently.
I also believe that this lab facilitated the development of my communication skills with my teammates even further. We realized that the most efficient working model and schedule did not revolve around having all of us together at the same time, but rather trying to break our team up to meet with different people at different times depending on the tasks that needed to get done. This led to a significant increase in our efficieny and understanding of the material and we will continue to implement this team model going foward into the final project.

### 5.3 Artem Laptiev Lessons Learned

Splitting into groups of two proved to be a highly effective strategy for team-management because we managed to subdivide the tasks on more granular level and didn't have to face the challenges of merging the schedules of four people at the same time. We will continue with the same strategy for the last part of the class.
With regard to the technical material, it was very rewarding to see how my knowledge from other classes came in very handy in this lab. We also even further developed a framework for pulling out data from ROS to analyze through

other tools, which helped immensely in both development, debugging, and visualization for stakeholders.

## 5.4 Cruz Soto Lessons Learned

In terms of technical understanding, I worked far more in the debugging phase than any other lab, working tirelessly from the first version of the controller to the final one. I was also able to work less intensely on this lab than the previous, where I was involved in the development of all the components. Instead I delved purely into the pursuit algorithm for this lab.

To improve communication and organization, I grew very accustomed to testing the robot and finally have a solid understanding of how to select and test evaluation metrics on both the car and in simulation. I know how to communicate these findings visually in stylized graphs with appropriate captions in a way that makes it immensely concise for the reader. I've also started understanding how noise and other factors can greatly deviate results between real life and simulation, something we luckily didn't experience too much, as the simulated pure pursuit model is functional enough in real life.