Lab #6 Report: Implementing Path-Planning and Pure Pursuit Algorithms for Robot Racecar

Team #9

Karen Guo
Yohan Guyomard
Michael Kraus
Jaclyn Thi
Carlos Villa

6.4200/16.405 - Robotics: Science Systems

April 26th, 2024

1 Introduction (Jaclyn Thi)

Path planning algorithms are a fundamental part of enabling autonomous robots to navigate through a known environment to reach a target destination without colliding with obstacles. A self-driving car might utilize path planning to navigate a known road network from its starting position to a destination, or calculate the motions necessary to maneuver into a parking space. A robotic arm might utilize a path planning algorithm to determine the best way to pick up or manipulate an object (e.g. screwing a threaded cap onto a bottle). For this lab, we observe both search-based (A*) and sampling-based (Rapidly-exploring Random Trees, or RRT) path-planning algorithms, which we describe in greater detail in later sections.

The path planning problem is challenging in nature due to various aspects. The values that we consider for this lab are reaching the destination and collision avoidance, computation time, and path distance. First, we must make sure that the robot can actually generate a path from the start to the end point, if a path exists. Environmental factors like obstacles, narrow hallways, and large distances between the start and end make this challenging; we have to guarantee that the car will not take an impossible path, while also not driving too close to the wall and crashing. In addition to reaching the destination, we also have to consider how long the computation takes. In our case, we must

consider speed in addition to accuracy as we are racing; the faster that we can compute a path, the faster we can start following it with our pure pursuit controller. One last important metric is how short the generated path is. The more optimal the path, the faster we can reach the end. Hence, we must balance the computational time versus the accuracy of the path, a difficult task.

The primary goal of this lab is to utilize a path planning algorithm to enable our racecar robot to determine a trajectory from a start point to an end point in the basement of the Stata Center. We assumed that a map of the basement would be provided, and that the environment would be a controlled setting with no moving obstacles. We also want our racecar to follow this calculated trajectory using a pure pursuit controller. Finally, we want to integrate our path planning algorithm and pure pursuit controller and combine it with our particle filter, which allows the racecar to localize its position in the real world using Light Detection and Ranging (LiDAR) scanning data.

2 Technical Approach

For this lab, we tackled the technical approach in three parts. First, we revised our Monte Carlo Localization (MCL) from the previous lab to get it to accurately localize in real life. Then, we approached path planning in two parts: path planning and path following. For path planning, we are given a map and a start and end point, which we must then generate a path between. These points are fed into A* and RRT, a search-based and sampling-based algorithm respectively, and the algorithms return a list of points that serves as a path. This path is fed into our pure pursuit controller, which then follows the path. The MCL localization then allows us to check the position of our robot with respect to the path.

2.1 Revised Monte Carlo Localization (Yohan Guyomard)

In our last lab, we attempted to implement localization on the race car; this allows it to determine its own position at all times and is the basis for both path planning and following. We opted for the Monte Carlo Localization (MCL) algorithm, which evaluates randomly chosen positions' likeliness by comparing observed LiDAR scans to simulated ones. At a glance, MCL consists of a motion and sensor model, where the former estimates the car's position using dead-reckoning with some added variability. Then, the sensor model computes a probability for each particle's new position and a resampling process is introduced to prune particles that are unlikely to be representative of the car's location. This process, given a reasonable starting position, will converge and continue to report the estimated pose of the car. Unfortunately, our implementation did not complete this goal and would have severely hindered our performance on subsequent assignments. Thus, we devoted part of the time spent on this lab towards finalizing the localization node in both simulation and

real life.

Ultimately, we discovered that our implementation was indeed correct but the launch files given to us were erroneous. Still, our attempts to improve the MCL algorithm towards something minimally viable resulted in a far more robust implementation than we would have had initially (once the launch files were corrected). Notable improvements include thorough usage of NumPy to increase performance and a new particle averaging approach. The latter involves computing the car's "final" estimated pose by taking the arithmetic mean of all the particles. Previously, this involved an unweighted mean of the particles' positions and unweighted circular mean of their rotations. We are now considering the probabilities computed by the sensor model to compute each particle's relative contribution to the car's pose; that is, we are taking a weighted mean.

$$\bar{x} = \frac{\sum_{i=1}^{n} w_i p_i^x}{\sum_{i=1}^{n} w_i} \tag{1}$$

$$\bar{y} = \frac{\sum_{i=1}^{n} w_i p_i^y}{\sum_{i=1}^{n} w_i} \tag{2}$$

$$\bar{\theta} = \arctan(\frac{\sum_{i=1}^{n} w_i \sin(p_i^y)}{\sum_{i=1}^{n} w_i \cos(p_i^x)}) \tag{3}$$

We qualitatively observed that this produced far better results at initialization (before MCL has had the time to converge) and during sharp turns. We have demonstrated that our localization algorithm can preserve the intricacies in the car's path through scenarios such as driving in a zig-zag, sharp U-turns and a complete loop of the Stata basement. Moreover, all of this is done at a relatively high refresh rate which gives our path follower more room to operate. Overall, this effort involved a complete rewrite of the MCL algorithm but resulted in a far more reliable localization node.

2.2 Path Planning (Karen Guo and Carlos Villa)

2.2.1 Overview

As mentioned earlier, we want our paths to have fast computation time, a short path, and no collisions with the wall or impossible routes. Hence, we looked at both search-based and sampling-based algorithms, specifically A* and RRT respectively. Search-based algorithms explore the map and possible paths systematically, which guarantees a solution if one exists, and an optimal or near optimal one. Sample-based algorithms randomly sample points in order to create a graph structure, and use heuristics to move towards the goal. They may or may not generate a path if it exists, and these paths are often not guaranteed to be optimal. In general, search-based algorithms are computationally more expensive, guarantee shorter paths, and may not be suitable to dynamic environments. On the other hand, sampling-based algorithms have fast computation, do not guarantee optimal paths, and more suitable to dynamic environments.

For both algorithms there were some pre-processing/post-processing steps that were needed. Our given map was in the form of an occupancy grid. We eroded this occupancy grid to give a "buffer" between any paths and the occupied points. The erosion sized was an arbitrary choice based on the size of the car, see figure 1. This helped generate paths that were farther away from walls and avoided tight spaces our racecar could not fit into. Additionally, we had to transform any poses given in the world frame to the frame of our occupancy grid before we started path planning because our path planning algorithms operated in the frame of the occupancy grid. After path planning we had to post-process our trajectory coordinates back into the world frame, so the racecar could follow them properly.

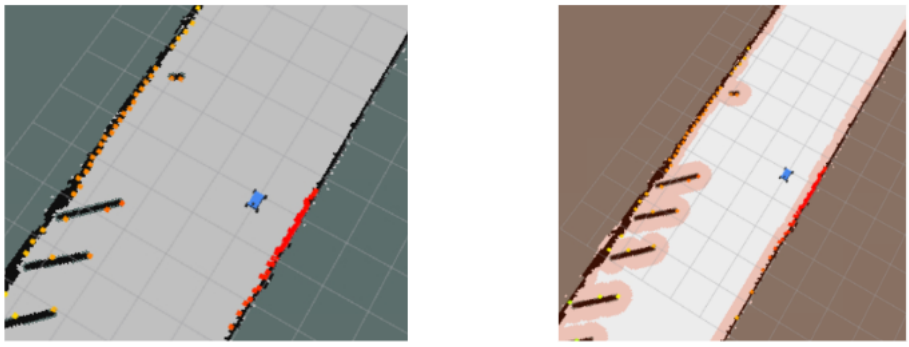


Figure 1: Free space of the map is modified by erosion. Free space is denoted by the light grey areas on the left and white areas on the right. Erosion helps paths stay away from walls and tight passages, and thus enables our algorithms to create more feasible paths for our racecar.

2.2.2 A* Algorithm (Karen Guo)

The search-based algorithm that we tested was A*. What differentiates A* from Dijkstra's or BFS/DFS is that we are finding the shortest path to a singular node based on a heuristic function, while Dijkstra's and BFS/DFS don't care about the specific goal node or the heuristic. In this case, our heuristic includes Manhattan distance, shown in (4) added to the path length from the start to node of interest, which we will elaborate on later. Hence A* allows us to more efficiently find a path to the goal node.

$$h(x, y) = |x_1 - x_2| + |y_1 - y_2| \tag{4}$$

The steps for the algorithm are shown in Algorithm (1). Before beginning our search, we initialize a visited set (all the nodes that we have looked at before), a score dictionary (each node and their score), a parent node dictionary (each

node and its parent node), and a queue (which nodes will be looked at next).

A* begins the path searching from the start node. While we haven't reached the end node, we look at the neighbors of the current node we are at (in the beginning this will be the start node). For each neighbor, we calculate the f-score shown in (6), update the neighbors scores, set their parent node to the current node, and add them to the queue; the neighbors with the lower scores will be ahead in the queue. Equation (5) represents the distance of the current path from the start to the neighbor node, so we are considering how close the neighbor is to both the start and end nodes. After adding the neighbors to the queue, the current node is then added to the visited set. We end our algorithm when the current node is equal to the end goal. Then, we backtrack and find the parents of all the nodes, starting at the end node, and return the resulting path.

Our algorithm runs in $O(b^d)$ time, where b is the number of edges from each node, and d is the number of nodes in the shortest path. Hence, we expect our algorithm to run exponentially longer with a longer path.

$$g(x, y) = \text{start\_to\_current} + \text{current\_to\_neighbor} \tag{5}$$

$$f(x, y) = g(x, y) + h(x, y) \tag{6}$$

From Fig. 2, we can see that our algorithm is able to correctly generate a path. However, there are divots in the path when the wall caves in. This is due to the heuristic function that we used, so in addition to Manhattan distance, we tested other heuristics like Chebyshev and Euclidean distance, shown in (7) and (8), to try to fix this issue. Both Euclidean and Chebyshev were slower than Manhattan, and neither fixed the problem of divots; thus we stuck with the Manhattan heuristic.

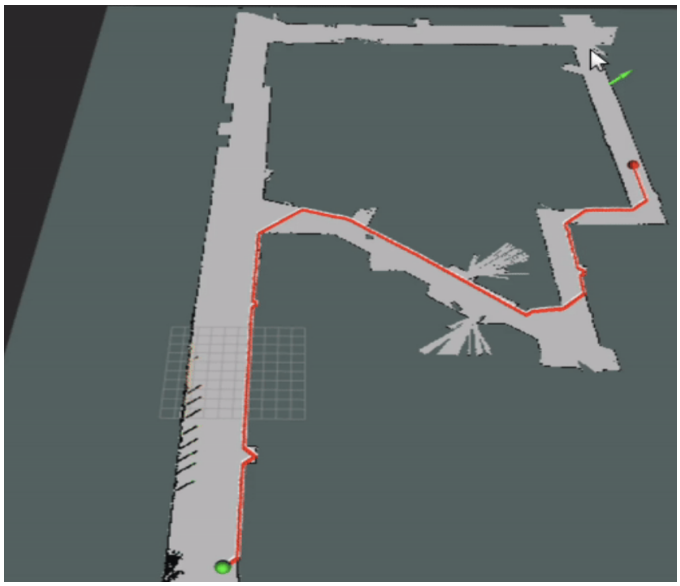


Figure 2: This is a path generated with the A* algorithm in Rviz shown in red, and the start and end points are in green and red respectively. The path is relatively optimal for the start and end points aside from the divots where the wall caves in. We can also see that the A* algorithm works for start and end points that are far from each other.

$$h(x, y) = \max(|x_2 - x_1|, |y_2 - y_1|) \tag{7}$$

$$h(x, y) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{8}$$

Algorithm 1 A* Search-Based Path Planning

```
function PLAN_PATH(start, end, map)
    if not IS_VALID_CELL(start) and not IS_VALID_CELL(end) then
        return                                  ▷ start and end points are invalid
    end if
    initialize visited, scores, parents, queue
    while queue is not empty do
        (current_score, current_node) ← pop(queue)          ▷ pop lowest node
        if current_node is end then
            return RECONSTRUCT_PATH(previous, start, end)
        end if
        visited.add(current_node)
        neighbors ← GET_NEIGHBORS(current_node)
        for neighbor in neighbors do                        ▷ score for each neighbor
            if neighbor not in visited then
                score = PATH_LENGTH(start, neighbor)+MANHATTAN(neighbor, end)
                if neighbor not in scores or score < scores[neighbor] then
                    scores[neighbor] ← score
                    parents[neighbor] ← current_node
                    queue.add(score, neighbor)
                end if
            end if
        end for
    end while
end function
```

2.2.3 RRT Algorithm (Carlos Villa)

We chose to implement the RRT algorithm for our sample-based planner due to its known variations and optimization methods and its inherent low computation time, $O(n \log(n))$ where n is number of samples taken. The basis of our implementation was informed by the provided paper "Sampling-Based Algorithms for Optimal Motion Planning." Starting with a basic implementation of RRT based on the pseudocode in "Sampling-Based Algorithms for Optimal Motion Planning" [1], see (2), we slowly iterated on this to come to our final implementation of the algorithm.

Algorithm 2 RRT

$V \leftarrow \{x_{init}\}; E \leftarrow \emptyset$
for i=1,...,n **do**
  $x_{rand} \leftarrow Sample\_Free\_Space$;
  $x_{nearest} \leftarrow Find\_Nearest(G = (V, E), x_{rand})$;
  $x_{new} \leftarrow Steer(x_{nearest}, x_{new})$;
  if ObstacleFree($x_{nearest}, x_{new}$) **then** $V \leftarrow V \cup \{x_{new}\}; E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
  end if
end for

After creating the framework for RRT we decided to attempt the RRT* algorithm, proposed in the given paper, to create more optimal paths. However, we soon learned that the "rewiring" portion of the algorithm, in which you alter the tree to maintain minimal cost paths, was very computationally expensive, so we ultimately abandoned these efforts.

In giving up on RRT* we made a conscious decision to focus on speed of generating paths, rather than the optimality of paths. Our next iterative step to increase speed was to take longer "steps" in each iteration of RRT. In order to do this we relied on the dubins packaged, also suggested by the lab instructions. The dubins package calculated Dubins curves, a series of curves that a vehicle with a given turning radius could follow to get from point A to point B. See 3 for illustration of this method. However, we maintained euclidean distance between nodes instead of curve length as our objective function for ease of computation.

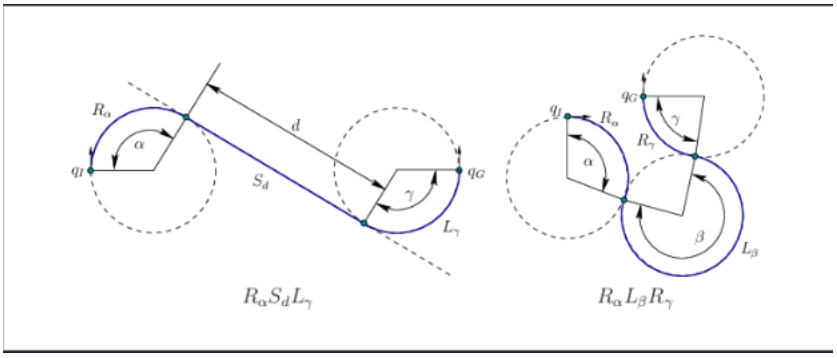


Figure 3: Dubins curves determine motion between points that are possible for a robot with a given turning radius. Adapted from [2]

The dubins package was beneficial in two ways, we now were able to make bigger steps at every iteration, while still maintaining collision checks, and our generated paths would be more accessible to our racecar. This benefit can be seen in the generation of paths that create u-turn trajectories, see figure 11, for the racecar to follow in order to reach goals behind the car instead of just assuming the car can turn on a dime and follow a straight line trajectory behind it.

Our final iteration revolved around steps per iteration instead of step size. Everytime we sampled a random point we decided to go in the direction of that point, in step size increments, until we no longer could. This allowed our tree to branch out quicker and enabled our algorithm to finish faster. Finally, we had come to our final version of our implementation of RRT,which we'll call RRTNew, described by 3.

Algorithm 3 RRTNew

$V \leftarrow \{x_{init}\}; E \leftarrow \emptyset$
for i=1,...,n **do**
    $x_{rand} \leftarrow Sample\_Free\_Space$;
    $x_{nearest} \leftarrow Find\_Nearest(G = (V, E), x_{rand})$;
    $x_{news} \leftarrow DubinSample(x_{nearest}, x_{new})$;
    prev_point $\leftarrow x_{nearest}$
    for point in x_{news} **do**
        if ObstacleFree(prev_point, point) **then**
            $V \leftarrow V \cup \{x_{new}\}; E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
            prev_point $\leftarrow$ point
        end if
    end for
end for

Ultimately, our implementation of RRT was able to generate paths very quickly and incorporated the dynamics of our racecar, but had noticeable drawbacks. While the algorithm was quick it was not asymptotically optimal, so even if we let it run forever it could not generate an optimal path unlike other path planning algorithms. It may never even generate a path in some cases. It is a single query algorithm which means it focuses just on our immediate task of finding a path from our start point to our end point, but that means we can't use any gained information from one search to inform another. Additionally, it is a search after construction algorithm wherein the path is traced back from the goal, but this is ultimately a neutral feature because our search-based algorithm A* must also do this.

2.3 Path Following (Michael Kraus)

2.3.1 Pure Pursuit

Now that we have a path, we need a way to follow that path. Our starting point was a simple pure pursuit algorithm. As shown in Fig.4, the pure pursuit algorithm has a set "look ahead" distance, L_1. As long as the distance from the vehicle to the path is less than L_1, there will be exactly two points on the path that are exactly L_1 meters away from the robot; one in front of the robot and one behind the robot. The point in front of the robot is the reverence point, as shown in Fig. 4. Once this point is determined, the algorithm calculates what steering angle is required for the robot to hit this point. To do this, a vector from the robot to the reference point forms an angle, η, with the velocity vector of the car. Once this angle is obtained, the steering angle, δ can be determined with (9):

$$\delta = \arctan\left(\frac{2L\sin\eta}{L_1}\right) \tag{9}$$

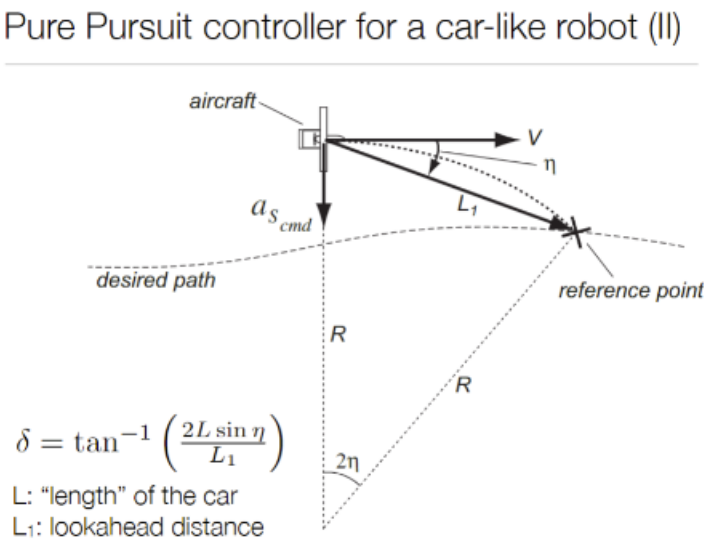


Figure 4: The pure pursuit algorithm works by finding a point on a given segment that is exactly the lookahead distance, L1, away from the car. The steering angle of the car is adjusted such that the car would hit this point with a constant steering angle. Adapted from [3].

In general, pure pursuit is a simple and easy-to-implement algorithm. However, there are several important edge cases and challenges that need to be considered while trying to follow a segmented path. These challenges are mainly,

1. Identifying the segment on the path that is nearest to the car

2. Going around curves smoothly

2.4 Nearest Segment Identification

The path-planning algorithm produces a series of segments that makes up the whole path, as shown in Fig. 5.

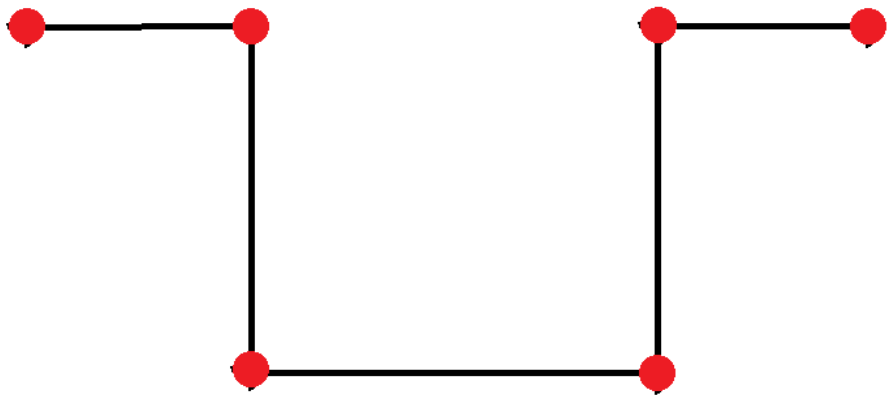


Figure 5: The path planning algorithms will produce a segmented path. In this case, there are 6 points and 5 segments. In order to function properly, the pure pursuit algorithm needs to identify which segment the car is on or nearest to.

One simple way to find the nearest segment is to use a dot product formula, as shown in (10) and illustrated in Fig. 6 to find the normal distance to each segment.

$$d = \frac{|\vec{QP} \cdot \vec{n}|}{||\vec{n}||} \tag{10}$$

The issue with this method is that it assumes that the segment is infinitely long. Moreover, this method will always calculate the *perpendicular* distance to the *centerline* of the segment (ie, it will always calculate d in Fig. 6 even if the car is past the endpoint of the segment).

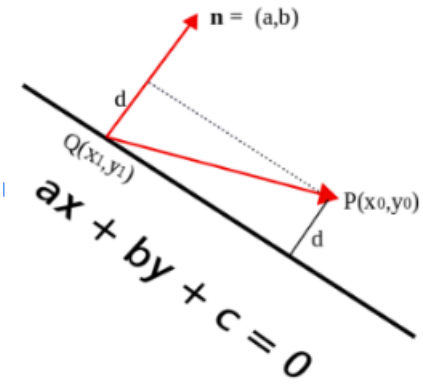


Figure 6: To get the shortest distance from a point to a line, one can simply project the point's position relative to a point on the line onto the normal vector. This will give the perpendicular distance between the line and the point.

When the car passes the end of the segment the shortest distance to the segment is the Euclidian distance from the car to the endpoint of the segment. Thus, we need a way to tell when we pass the end of the segment.

To accomplish this we can create a parameter called t. This parameter is calculated with (11) and the visual representation is shown in Fig. 7. Once t is calculated there are three cases to consider:

1. $t < 0$**:** In this case the car has not reached the beginning of the segment. **The shortest distance to the segment is the Euclidian distance from the car to the *start point* of the segment.**
2. $0 < t < 1$**:** In this case the car is somewhere on the segment. **The distance to the segment should be calculated using (10).**
3. $t > 1$**:** In this case the car has passed the end of the segment. **The shortest distance to the segment is the Euclidian distance from the car to the *endpoint* of the segment**

$$t = \frac{|\vec{QP} \cdot \vec{A}|}{||\vec{A}||} \tag{11}$$

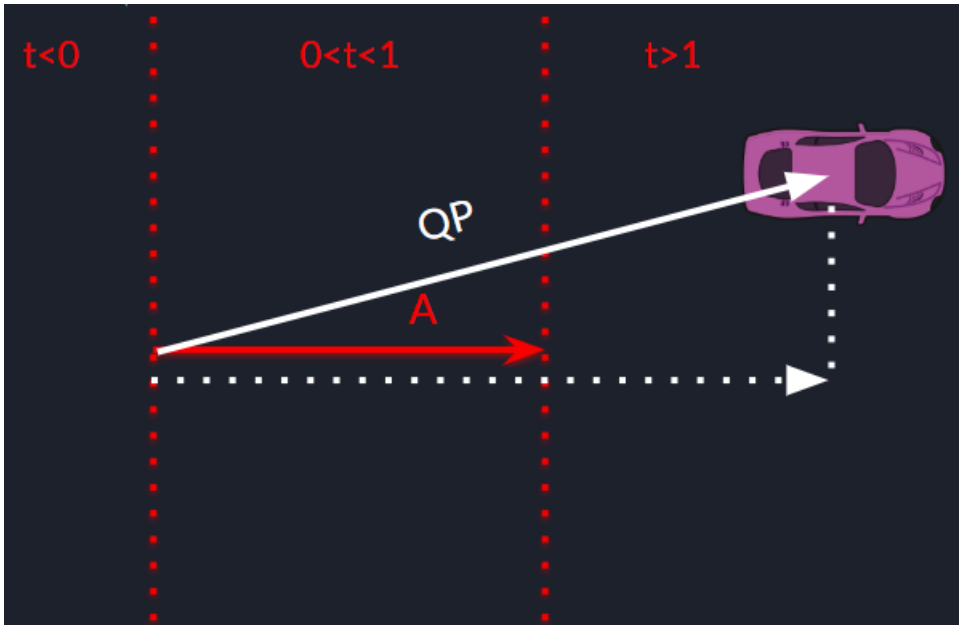


Figure 7: We want to get a parameter that tells us if we are on the segment, past the segment, or have yet to reach the segment. By projecting the car's position onto the segment with a dot product and dividing by the length of the segment we will get a parameter, t. If $t < 0$ we have not reached the segment, if $0 < t < 1$ we are on the segment, and if $t > 1$ we are past the segment.

2.4.1 Looking Through the Turn

By using the nearest segment and the lookahead distance to calculate a reference or target point, the car is able to follow a straight line path. However, the car still has a tendency to overshoot turns, as shown in Fig. 8.

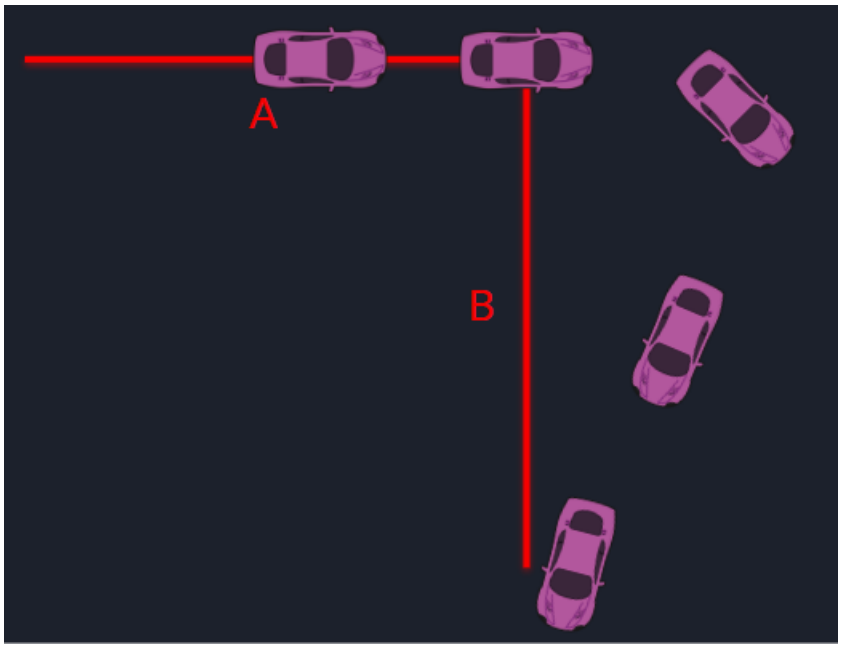


Figure 8: If the car starts on segment A, the car will obviously calculate that segment A is the closest segment and will calculate the reference point such that it lies on segment A. Then, when the car gets close to the end of segment A, the car still wants to put the reference point on segment A! As a result, the car does not "see" the turn and will not start turning until it passes the endpoint of A and calculates that B is the closest segment!

Fig. 8, shows why this happens: as the car approaches the turn, the distance to segment A is always zero. Thus, segment A is the closest segment and the pure pursuit algorithm tries to place the reference point such that it lies on segment A or an extension of segment A. When this happens, the car will keep driving in a straight line until it reaches the endpoint of segment A. Once this happens, segment B becomes the closest segment and the car finally begins to turn - but it's much too late!

To fix this, the car simply has to look at the next segment. This is easily implemented with a simple while loop:

```
dist = distance to end of segment
```

```
while (dist <= lookahead) and (not on last
    segment)
    nearest_segment = the next segment on the
        path
    dist = distance to end of nearest_segment
```

Once this is implemented, the car will track smoothly through all turns.

3 Integration (Yohan Guyomard)

Our localization, path planner and follower are all implemented as ROS2 nodes. We make use of RViz (ROS' visualization package) to submit user inputs and render the computed path. The following graph provides a complete picture of our system:

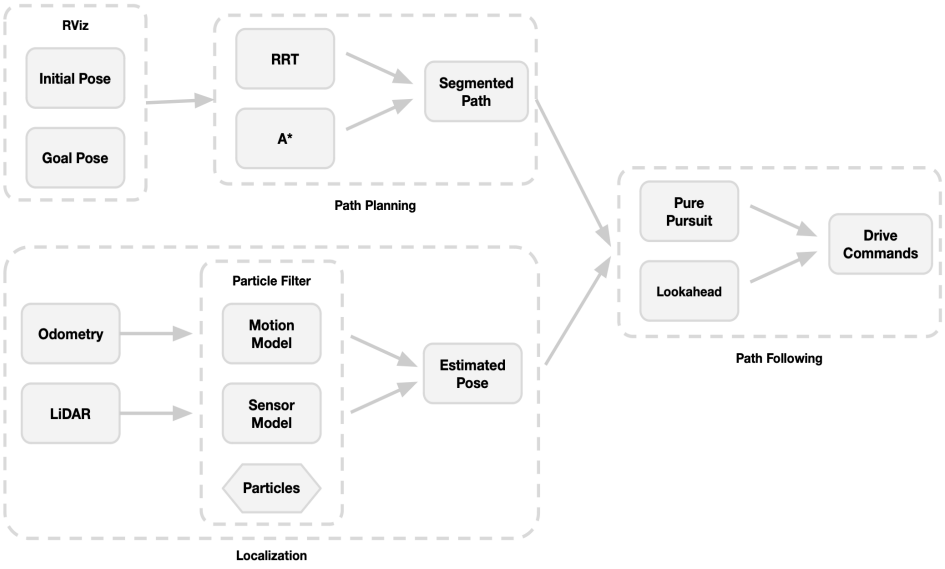


Figure 9: High-level overview of the ROS2 nodes that constitute our path following routine, from user input to navigation commands. We can see that our path planning and localization feed into our path following.

4 Experimental Evaluation

4.1 Path Planning (Karen Guo and Carlos Villa)

4.1.1 Overview

We evaluate the A* and RRT generated graphs against each other to compare the computation time, distance of path, and how drivable the path is. Overall, we found that RRT will be significantly faster in long paths, while A* has more optimal paths. RRT also generates paths that our car can follow when the end point is behind the initial orientation of the car. However, RRT may not always generate a path, hence we plan to implement a new controller that uses both RRT and A* for robustness.

In order cover a wide scope, we have five different test cases: a short, straight line, a backwards line, a narrow hallway, a far endpoint, and sharp turns. We chose these cases to test the different scenarios that are possible on the map of the stata basement. While we ran out of time to test in real life, we believe that running in real life would result in a slower computation time, but no difference to the length of the path.

4.1.2 A* Algorithm (Karen Guo)

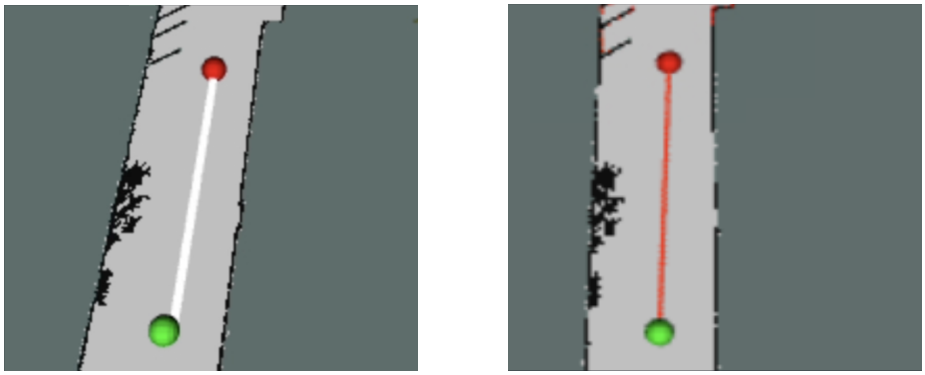
For the A* times, refer to Table 1. From Fig. 10a we see that A* generates a near optimal straight line from start to finish in 0.3s. This is a fast and optimal path. In Fig. 11a, we see that we once again have an optimal line in 0.4s. However, because we are using a pure pursuit controller, it will not be able to follow this path. In Fig. 12a, A* successfully generates a path through the narrow hallway in 1.6s. Aside from the wall divot, this is a relatively optimal path that is also drivable. Fig. 13a is our slowest test at 3.2s, because the ending point is so much farther from the start; as we mentioned before, the farther the points, the longer A* will take. The path remains near optimal. Lastly, Fig. 14a shows A* can handle sharp turns in 0.6s with a near optimal path.

From these results we can see that A* performs consistently, returning a near optimal path each time. However, if the orientation of the car is not facing the end point, our pure pursuit controller will not be able to follow the path. Modifications can be made to both the controller and the algorithm to either generate a feasible path or have the car spin around when it can't find a path. Additionally, the A* algorithm runs much slower than RRT, times shown in Table 2, for the far endpoint case. To quicken runtime, we can try downsampling the map to allow for more efficient searching of the space.

4.1.3 RRT Algorithm (Carlos Villa)

Based on the results summarized in tables 1 and 2 which demonstrate its lower runtime on average compared to our implementation of A* we decided to employ

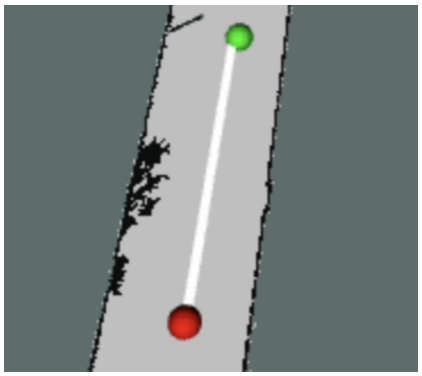
our implementation of RRT as our main path planning algorithm. However, the numbers don't tell the full story of RRT being somewhat unreliable in certain situations. For instance, A* is also a better fit for some use cases like sharp turns as demonstrated by figure 14 and very narrow hallways like in figure 12. Thus, we decided that a master controller that utilized both would be the best compromise. First we would run our RRT implementation with low samples, if it can find a path, it will find it very quickly, if not, we will employ our A* implementation to find the path if it exists. We believe our two stage master controller would give us the best of both worlds we have seen from our path planners.



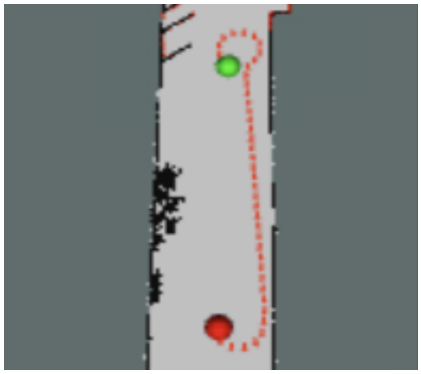
(a) A* Test 1 Path

(b) RRT Test 1 Path

Figure 10: Here is the straight line test, where the green is the start and the red is the end. A* and RRT both perform well when tested against a straight line: optimality, compubtability, and feasibility are all achieved. The runtime for RRT from Table 2 is much faster, however, due to the fact that we return early if we can reach the end point in a straight line.

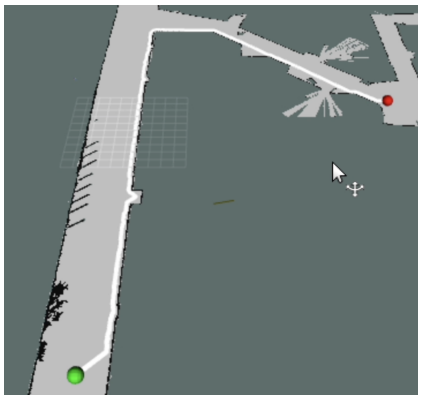


(a) A* Test 2 Path

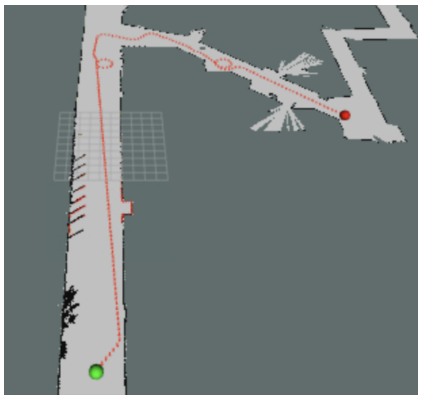


(b) RRT Test 2 Path

Figure 11: This shows the paths from the backwards straight line test, where the start is green and end is red. From Table 1 and Table 2 we can see that A* runs faster and has a shorter path. However, pure pursuit cannot follow the path without modification. Hence, the RRT path is more feasible.

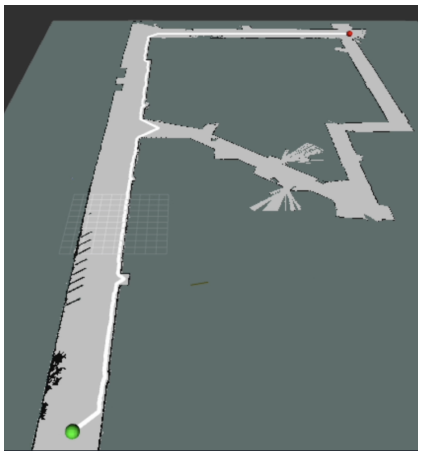


(a) A* Test 3 Path

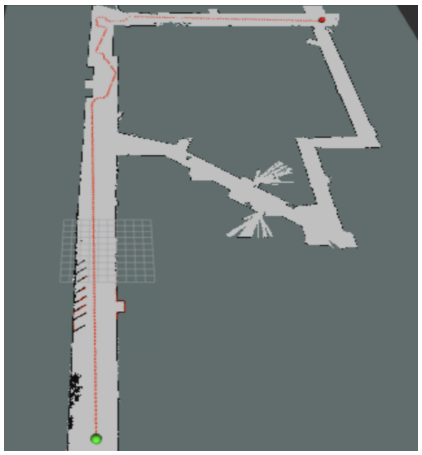


(b) RRT Test 3 Path

Figure 12: Both RRT and A* are able to generate a path in a narrow hallway, with similar times. From the paths we can see that A*'s path is more feasible and optimal, because RRT has loops inside. These loops could potentially hinder the pure pursuit controller, so we would prefer A* in this case.



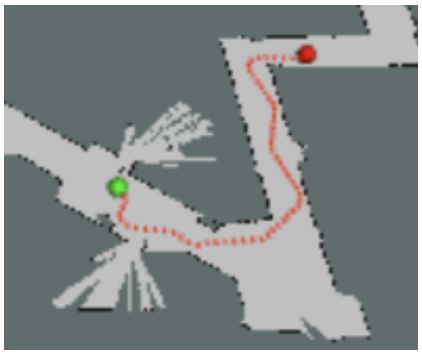
(a) A* Test 4 Path



(b) RRT Test 4 Path

Figure 13: These are the results for the far endpoint test. A* and RRT are both able to generate paths, but RRT's path generation is much faster. While A*'s path is shorter, the different in computation time is a few seconds, which means RRT would be better suited for this case. However, RRT doesn't consistently generate a path, so we would have to rerun the algorithm is the times that it fails.

(a) A* Test 5 Path



(b) RRT Test 5 Path

Figure 14: This is our last test for sharp turns. A* generates slightly better paths at a slightly faster rate. Also, RRT occasionally fails for this case as well, so A* would be a better fit.

Table 1: A* Computation Times and Path Lengths

Test Case	Computation Time (s)	Path Length (m)
(1) Straight Line	0.2979	12.0588
(2) Straight Line Behind	0.3609	12.0461
(3) Narrow Hallway	1.5725	71.8649
(4) Far Endpoint	3.2286	117.9590
(5) Sharp Turns	0.6104	25.2121

Table 2: RRT Computation Times and Path Lengths

Test Case	Computation Time (s)	Path Length (m)
(1) Straight Line	0.0640	11.9330
(2) Straight Line Behind	3.7017	24.7377
(3) Narrow Hallway	1.5537	77.3080
(4) Far Endpoint	0.3583	118.9053
(5) Sharp Turns	0.6473	31.6437

4.2 Pure Pursuit (Michael Kraus)

4.2.1 Overview

A good path follower has two major performance metrics:

1. Experience minimal path deviation when the car encounters a turn in the path
2. Return to the path quickly when a deviation occurs

The objective of this section will be to explain how we measured these performance metrics and how we used those measurements to choose the optimal lookahead distance.

Note that for these tests we are not using the particle filter; we are using the true odometry so that we are only evaluating the quality of the pure pursuit algorithm.

4.2.2 Test #1: Deviation Correction

To test how well our car recovers from a deviation, we used a hand-drawn trajectory. This trajectory was a straight line, and the car was given an initial position of about 0.55m off of the centerline, as shown in Fig. 15

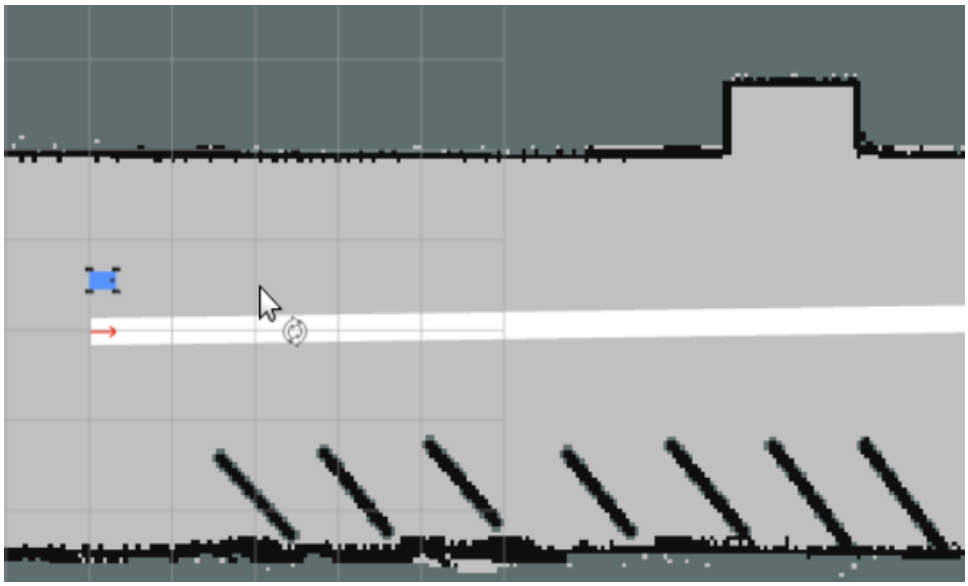


Figure 15: To test deviation correction, the car was given a straight trajectory and was initialized about 0.55 m off the centerline of the trajectory.

Then the car was allowed to drive and we recorded the distance from the centerline with respect to time. Our findings are summarized in Fig. 16. From these findings, we want to pay attention to three things:

1. **Response Time:** We want the car to correct itself as fast as possible.
2. **Overshoot:** When the car tries to turn back to the path, it might overshoot the path. We want this overshoot to be as small as possible.
3. **Settling Time:** Simply put, we do not want the car to oscillate around the path. Generally, this can be quantified but for our purposes, we analyzed this metric in a qualitative manner.

We tested the following lookahead distances: 0.6 m, 0.9 m, 1.2 m, and 1.5 m. Clearly, smaller lookahead distances produce a faster response. However, smaller lookahead distances have a larger overshoot and more oscillatory tendencies. However, the difference in overshoot in our test cases was negligible, and only a look ahead of 0.6 meters produced slight oscillations. Since we want a fast response, we should choose the smallest lookahead possible (note that we are only concluding this because all test cases had similar overshoots and settling times). Our car's ability to smoothly negotiate turns takes priority over our car's step response, so a final decision requires further analysis, performed in the next section.

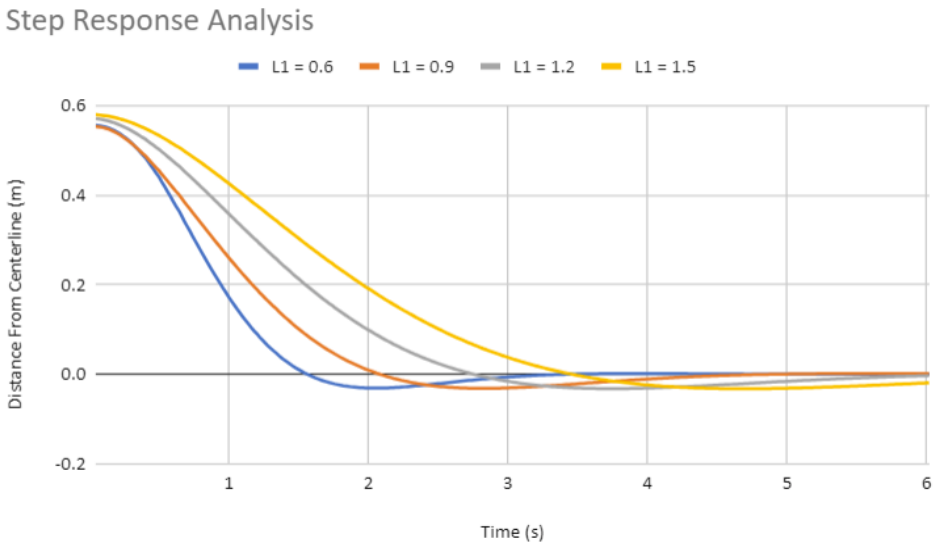


Figure 16: From the responses to similar deviation, we see that smaller lookahead distances have faster response and more overshoot and oscillatory behavior. However, in the ranges that we tested, the increase in overshoot and oscillatory behavior was very small (all lookahead distances tested had an overshoot of 5%) By inspection, it appears that all of the values we tested also had similar settling times.

4.2.3 Test #2: Sharp Turns

To test how well our car handles sharp turns, we used a hand-drawn trajectory. This trajectory was a 90-degree turn. The car was allowed to start on the centerline of the path, as shown in Fig. 17.

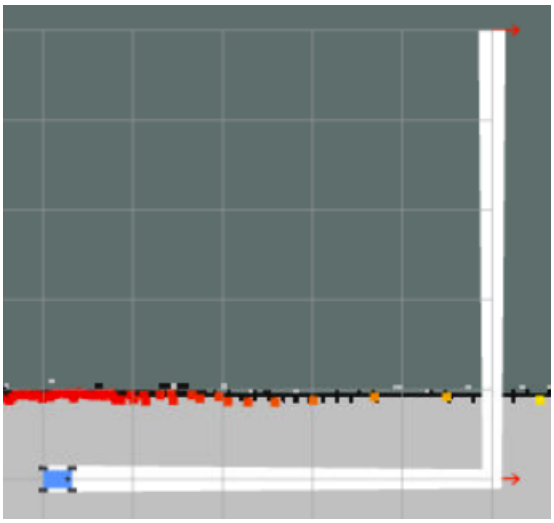


Figure 17: This is the test that the car runs to judge its turning ability. We made a simple 90-degree turn and allowed the car to start on the centerline.

Again, the car is allowed to drive and the distance to the centerline is recorded. The idea is to analyze how much the car **overshoots** or **undershoots** the turn. Our findings are shown in Fig. 18.

From Fig. 18, we see that short lookahead distances cut the corner less and larger lookahead distances cut the corner more. On the other end, small lookahead distances have large overshoots. The overshoot will decrease as lookahead is increased *up to about 0.9 meters.* After that, the overshoot begins to increase again. Based on this, it appears that a lookahead of 0.9 meters gives the best turning performance - it has the smallest overshoot error and relatively small corner-cutting error.

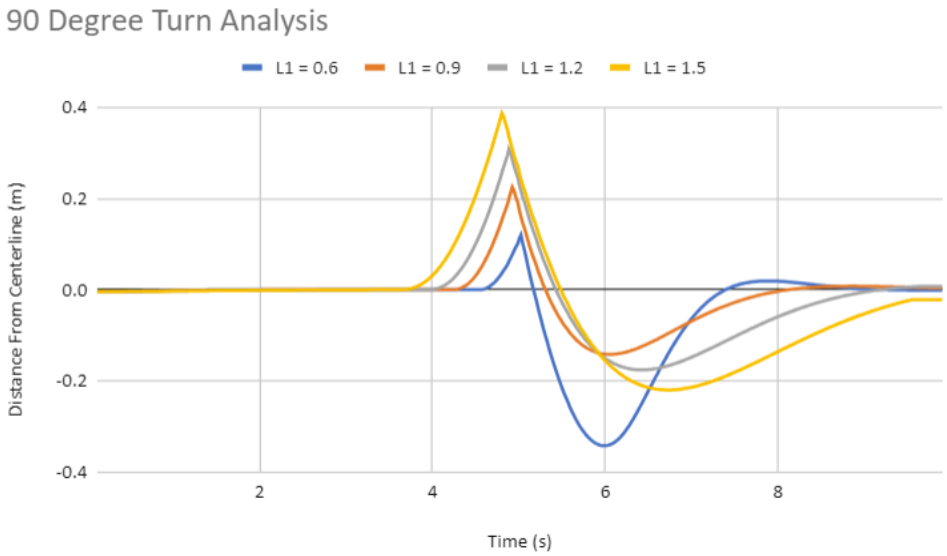


Figure 18: This represents the distance from the centerline as the car goes through the 90-degree turn. The first peak represents how much the car "cuts the corner" and the second peak (or valley) represents how much the car overshoots the turn. From this graph, we can see that lookahead of 0.9 performs the best.

4.2.4 Final Lookahead Selection

Test one tells us that a look ahead of 0.6 meters gives us the fastest response (although it does oscillate slightly), but test 2 tells us that a look ahead distance of 0.9 meters gives the best turning performance.

That being said, when the path planner publishes a plan, the car should already be on the centerline of the generated path. Thus, the ability of our car to smoothly handle turns should take priority over its ability to quickly correct deviations. **Therefore, we have decided to us a lookahead distance of 0.9 meters.**

4.2.5 Verification of performance

To verify that a lookahead of 0.9 meters would give satisfactory performance, we turned the particle filter back on and allowed the path planning algorithms to generate a path. Again we recorded the distance from the centerline as the car drove. Our findings are shown in Fig. 19. The generated path that the car was following was created with the RRT algorithm and is shown in Fig. 20.

Our maximum error on this path was about 10cm - about 1/3 of our car's track width. For reference, this would be roughly equivalent to a real car being about 6-12 inches from the edge of a standard 12-foot lane.

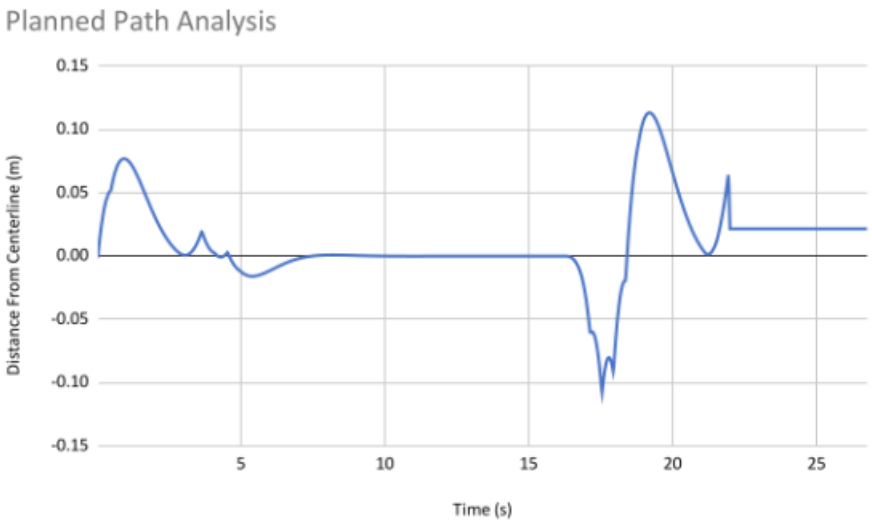


Figure 19: This graph summarizes how far the car is from the centerline of the path produced by our path planning algorithm. The largest error is about 10cm - just under 1/3 of the car's track width. For reference, if you scaled our max error and our car up to the size of a real car, the edge of the car would be about 1 foot from the edge of a standard 12-foot lane.

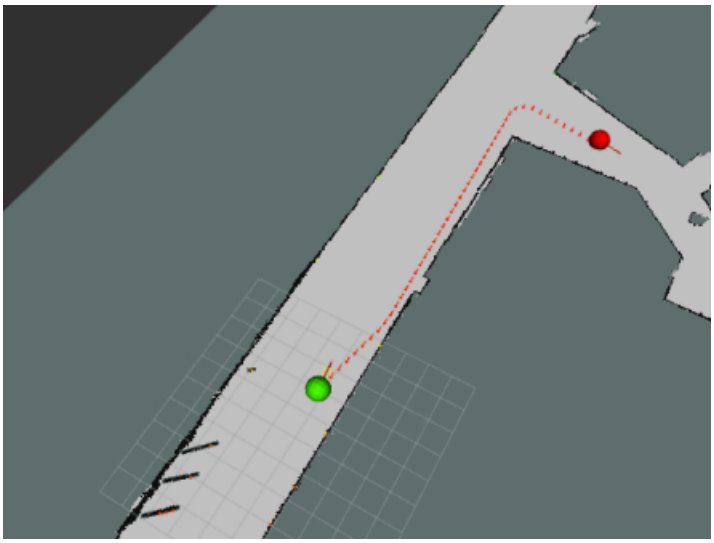


Figure 20: This path was generated with the RRT algorithm. It incorporates straight paths and a 90-degree turn, similar to the tests that we ran with the hand-drawn paths. The path is near optimal and drivable due to its distance from the wall.

5 Conclusion (Jaclyn Thi)

Although we experienced issues with getting our racecar to localize in the previous lab, we were able to successfully implement two different types of path-planning algorithms as well as a pure-pursuit controller, which allows our racecar to successfully calculate and follow a trajectory from its starting position to a target point in the basement of the Stata Center.

For path-planning, we found that A* is slower but computes a more optimal path, while RRT returns a less optimal path within a shorter computation time. In the context of autonomous racing for our racecar robot, we want to be able to balance the tradeoffs between computation time and path optimality.

Since RRT can compute a path more quickly but does not guarantee optimal paths or convergence at smaller iterations, one approach we might take in a racing environment is to attempt RRT for a set number of iterations, after which we might utilize A* which is guaranteed to converge if the path exists.

In the final challenge, we will combine all the components we developed for our racecar to compete in a racing challenge around the Johnson Track as well as an urban driving challenge in the basement of the Stata Center. As such, we might want to utilize a combination of path-planning and computer vision for

the urban challenge to navigate to target locations while obeying traffic laws dictated by stop signs and traffic lights.

6 Lessons Learned

6.1 Yohan

I primarily worked on fixing and improving the last lab (MCL), where I learned how frustrating debugging ROS can be. However, this forced me to become far more acquainted with both ROS' and Linux's tools to troubleshoot and iterate quickly.

6.2 Michael

In this lab we learned that the pure pursuit algorithm should not always "look" at the nearest segment of the path to generate a reference point. "Looking ahead" produces a much smoother turning behavior.

6.3 Jaclyn

We learned that eroding the provided map can help generate paths that don't lead the robot too close to the wall. We also learned to improve the professionalism of our reports and briefings by contextualizing the technical problem, using visual storytelling, and following Institute of Electrical and Electronics Engineers (IEEE) formatting standards.

6.4 Karen

I learned that transforming coordinates is very important; I had an issue where the path would fly off the screen because the coordinates were in the wrong frame. Additionally, having the correct step size for getting neighbors affects the A* algorithm a lot. Lastly, I strengthened my understanding of the hierarchy to introduce points in a presentation and report. We should state the values that guide our algorithm design and testing at the beginning.

6.5 Carlos

I learned the importance of understanding an algorithm fully. It was easier to optimize the RRT algorithm when I thought of its behavior, rather than just the technical implementation. My group members and our CI instructor also helped me learn how to illustrate ideas in briefings more succinctly through visual storytelling.

References

[1]S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," The International Journal of Robotics Research, vol. 30, no. 7, pp. 846–894, Jun. 2011, doi: https://doi.org/10.1177/0278364911406761. Available: http://roboticsproceedings.org/rss06/p34.pdf

[2]Cloudfront.net, 2024. Available: https://d3i71xaburhd42.cloudfront.net/4176af5e2059cde003ab1a85c982a74 Figure2-1.png. [Accessed: Apr. 24, 2024]

[3] Carlone, L., "Lecture 5-6: Embedded Control Systems," Feb 20 2024.