

Lab #6 Report: Path Planning

Team #15

Lauren Carethers
An Bo Chen
Brian Li
Claire Lu
Rachel Lu

RSS

April 27, 2023

1 Introduction

Author(s): Lauren

1.1 Purpose and specifications

The purpose of this lab is to teach the robot how to drive. This is accomplished by implementing planning and control through the utilization of path planning algorithms and pure pursuit respectively. Specifically, our objectives are the following:

1. Plan trajectories in a known occupancy grid map from the car's current position to a goal pose using either a search-based or sampling-based motion planning method.
2. Program the car to follow a predefined trajectory in a known occupancy grid map using your particle filter and pure pursuit control.
3. Combine the above two goals to enable real-time path planning and execution in the racecar simulator.
4. Deploy the integrated system on our physical racecar.

1.2 Technical problem and solution

We investigated two different types of path planning for this lab: search-based planning and sample-based planning. Search-based planning employs graph search methods to determine trajectories over a discrete representation of the problem. Sample-based planning solves problems in a continuous space without graph representation, though certain methods discretize the space. The most notable difference between these two methods is their speed and efficiency; search-based methods can guarantee solution efficiency but require more computation time. In comparison, sample-based methods run faster, but can lead to the generation of possibly inefficient paths. To properly compare these two methods, we implemented and tested both.

To successfully implement the pure pursuit section of this lab, we will need to write and implement a pure pursuit algorithm and determine the necessary control to follow a provided path.

2 Technical Approach

2.1 Initial Setup

Author(s): Rachel

Building on the localization techniques explored in the previous lab, developing an efficient pathfinding algorithm is critical for achieving an effective and efficient autonomous racecar. Our team was tasked with determining a collision-free trajectory from the start position to the goal position, and we achieved this by utilizing the pathfinding algorithm. Once a collision-free trajectory was established, our team implemented a pure pursuit drive controller using localization to enable the racecar to follow the trajectory and travel from start to finish. In summary, by utilizing localization and a pathfinding algorithm, we successfully constructed a trajectory and a pure pursuit controller, enabling the racecar to autonomously travel from the start to end positions.

2.2 Path Planning

Author(s): Brian

For our implementation of path planning, we considered both sample-based and search-based methods. We considered several algorithms from both of these categories in terms of asymptotic optimality, single vs. multi-query, ability to capture dynamics, and runtime complexity as summarized in the table below.

Algorithm	Optimal	Single/Multi Query	Necessity of search after construction	Account for dynamics
A*	Yes	Single	-	Yes*
Dijkstra	Yes	Single	-	Yes*
DFS	No	Single	-	Yes*
RRT	No	Single	Yes	Yes
RRT*	Yes	Multi	Yes	Yes
PRM	No	Multi	Yes	Yes

*Can account for vehicle dynamics if we perform some post-processing on the returned path

Figure 1: A Comparison of Various Path Planning Algorithms

The complexity of different path planners is difficult to compare since their runtimes depend on the exact implementation and the scenario they are being applied to. Generally speaking, sample-based methods, like RRT, are faster but don't guarantee an optimal solution whereas search-based methods, such as A*, sacrifice speed in order to guarantee a solution that is optimal and/or feasible. Ultimately, we decided to implement both RRT and A* and compare them in order to see which one was the best for our needs.

2.2.1 A*

Author(s): Brian

A* is a search based algorithm, meaning it attempts to find the shortest path in a graph from some chosen start node to an end node by iteratively expanding its search space. Our implementation for A* centers on two key data structures: an open list and a closed list. The open list contains nodes that we are considering as possibilities for being part of our final path, while the closed list contains nodes we have already fully explored and don't need to reconsider. Each node, n_i , stores information about the corresponding $(x, y)_i$ coordinate it represents, the heuristic h_i associated with the node, the distance it takes to reach the node from the start node g_i , and a pointer back to the node's parent. The heuristic h_i is an estimate of how close the node is to the goal node. Although there are many types of heuristic functions, we chose an Euclidean distance heuristic eq. (1) as it is the simplest and works well for our path planning application with the car.

$$h_i = \|(x, y)_i - (x, y)_{goal}\|_2 \quad (1)$$

As long as the open list has nodes remaining, we pop the node $n_{current}$ with the lowest cost from the list, where the cost is defined according to eq. (2).

$$cost_i = h_i + g_i \quad (2)$$

Then, we find all neighbors of $n_{current}$ as shown in Figure 2. These are the 8 nodes adjacent to the $n_{current}$ as defined in the search graph. The width of each grid in the graph is a parameter that affects the spatial resolution of our A* search: the smaller the width, the finer the detail in the returned path.

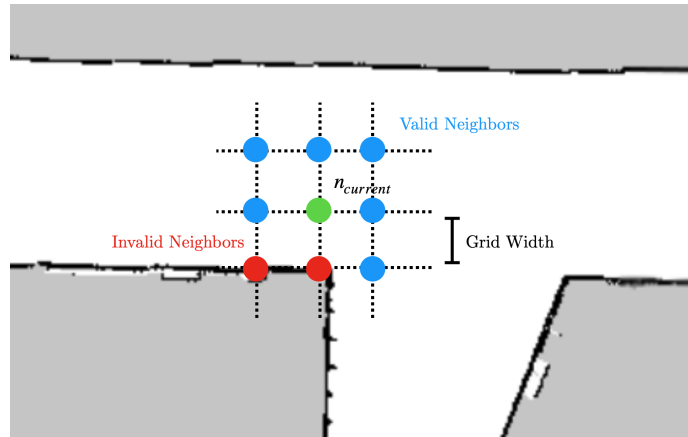


Figure 2: Neighbor nodes considered during A* search

For each neighbor node, if it lies outside of the valid search region or is in the closed list, we skip it. Otherwise, we calculate the heuristic and distance so far associated with the neighbor and add it to the open list so it can be included in our search for the optimal path. Finally, we add node $n_{current}$ to the closed list to ensure we don't consider it again.

2.2.2 RRT

Author(s): Rachel

The Rapidly Exploring Random Tree (RRT) algorithm is a sample-based path planning approach that involves exploring a domain space by randomly constructing a tree. The tree is built incrementally using random samples drawn from the search space, and the algorithm is capable of handling obstacles while efficiently identifying collision-free trajectories.

Initial Procedure

The RRT tree is rooted at the starting configuration and expands through the utilization of random samples from the search space. Our adaptation of the RRT algorithm involves the random sampling of collision-free poses from the search space, with a 10% probability of sampling the goal pose. As each new pose is generated, a connection between the new pose and the nearest pose in the tree is attempted, with the nearest pose determined using the Euclidean distance equation (1). In the standard RRT algorithm, the new pose is added to the tree if the direct path from the nearest pose in the tree to the new pose is collision-free and adheres to any constraints.

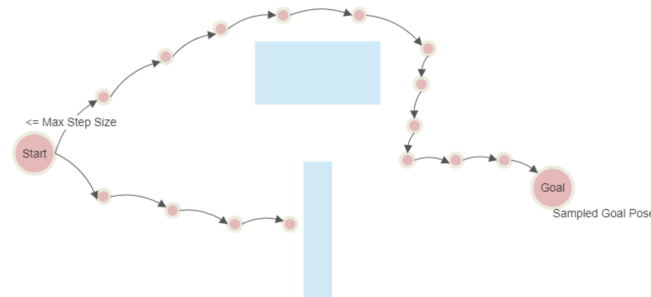


Figure 3: Example RRT Path without Shortcutting.

Our RRT algorithm modifies the standard check and add process by including a feature that allows for the addition of partial paths. Using a predetermined step size, the algorithm attempts to identify a mini collision-free trajectory from the nearest pose to the new pose. Selecting an appropriate step size is crucial, as an excessively large step size could result in collisions being missed, while a very small step size could significantly slow down the RRT algorithm. If the path is not entirely collision-free, only the portions of the trajectory from the nearest pose to the last collision-free pose are added to the tree. This adaptation significantly enhances the efficiency of the algorithm in identifying collision-free paths in the search space. Figure 3 illustrates an RRT-generated path. Once a collision-free path from the start pose to the goal pose is established in the tree, the algorithm proceeds to the Shortcutting process.

Shortcutting

Shortcutting is a post-processing technique that enhances trajectory optimality and path quality. The algorithm operates by repeatedly applying *shortcuts* to the path. It functions by selecting two random points, p_1 and p_2 , along the path and attempting to connect them using a straight segment. If the straight segment is shorter in distance than the original section and is collision-free, then the original portion can be replaced with the straight segment. Figure 4 depicts

this process.

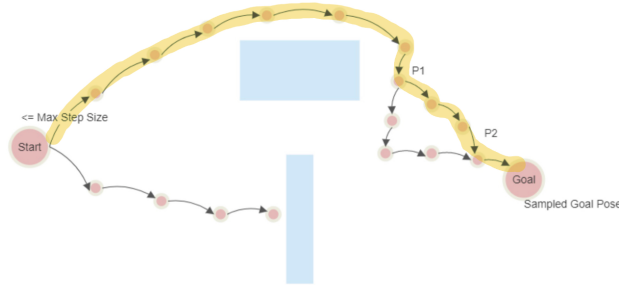


Figure 4: Shortcutted RRT Path with the highlighted path as the final trajectory.

The RRT algorithm does not guarantee optimal paths from the start pose to the goal pose. The RRT* algorithm is the optimized version of RRT. Although RRT* is capable of finding optimal paths, it is also slower than the standard RRT algorithm. RRT with Shortcutting, on the other hand, can generate optimal trajectories while maintaining the fast execution speed of the standard RRT algorithm.

2.2.3 Dilation and Erosion

Author(s): An Bo

Search algorithms often try to optimize for the shortest distance or time, which can result in cutting corners. However, this can lead to infeasible paths in the real world since a car has physical dimensions and is not just a point mass. Moreover, sharp corners in the path can pose a challenge for the pure pursuit controller, which also aims to cut corners. To prevent potential collisions, we can modify our map to mark walls and nearby obstacles as off-limits for the planning algorithms. This way, the algorithms can generate safe and feasible paths that consider the physical constraints of the vehicle.

To modify the map, we applied a binary dilation operation onto the incoming occupancy map in the path planner. This operation expanded the boundaries of obstacles and ensured that the generated plan trajectory no longer stuck to the walls. By doing so, we created a buffer zone around obstacles that allowed the car to navigate safely without getting too close. As a result, the modification improved the overall safety of the path planning algorithm and prevented potential collisions with nearby obstacles. The result of our morphological diation on the occupancy map is shown below:

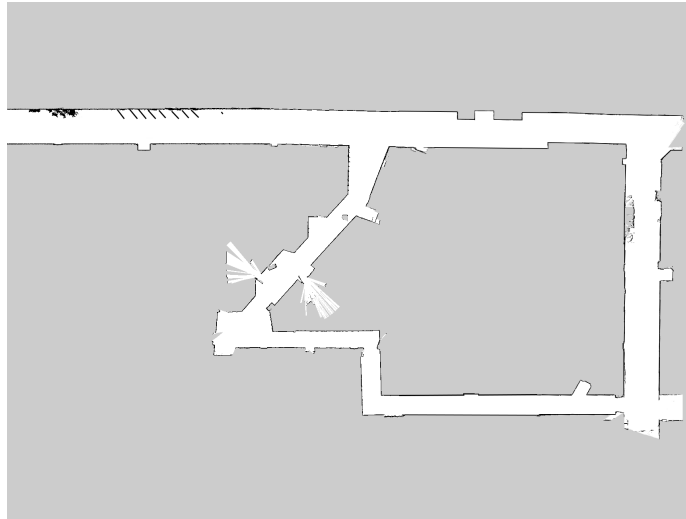


Figure 5: Original Stata Basement Map



Figure 6: Dilated Stata Basement Map

2.3 Pure Pursuit

Author(s): An Bo

To ensure that our car follows the planned trajectory, we use a pure pursuit algorithm. This algorithm involves preprocessing the path before the car follows it. First, we identify the closest point on the trajectory to the car's current position. Then, from that point, we find the target point that is a given lookahead distance away from the car. Finally, we calculate the steering angle using the current position and orientation of the car, such that it reaches the target point. By using this algorithm, we are able to make the car follow the planned trajectory accurately and smoothly.

Preprocessing the planned trajectory

The planned trajectory is represented as a sequence of (x,y) points, where each adjacent pair of points forms an abstract line segment. To preprocess the planned trajectory, we first calculate the distance between every two adjacent points. If the Euclidean distance between two adjacent points is greater than a fixed factor, such as 0.2 meters, we divide the path between these two points into 0.2 meter segments, creating and adding new points along the path. This approach allows us to ensure that the car follows the planned trajectory accurately, even if the segments are long and curvy.

Finding the Closest Point

By dividing the planned trajectory into line segments of roughly equal length, we can easily find the closest point on the path to the car. If we had long line segments, we may end up with a closest point that is far away from the car. To find the closest point on the planned trajectory to the car, we simply go through the sequence of points in the path and find the one that has the minimum Euclidean distance to the car's position.

Identifying the Target Point

After finding the closest point on the planned trajectory to the car, we iterate through the subsequent points along the path and calculate the accumulated distance between each adjacent point until we reach a point that is a given lookahead distance away. This point becomes our target point, which the car follows. By continuously updating the target point, the car can track the planned trajectory, even if there are sudden changes in the environment. Finally, we compute the steering angle between the car's current position and orientation and the target point. To prevent sharp turns, we bound the steering angle. This steering angle command is then sent to the car's controller, which guides the car along the planned trajectory.

Tuning the Pure Pursuit Controller

Tuning the pure pursuit controller is an essential step in ensuring that the car accurately and safely follows the planned trajectory. The speed of the car plays a crucial role in determining how well the controller performs. If the speed is too high, the car may overshoot the target point, leading to unstable and jerky

movements. On the other hand, if the speed is too low, the car may not be able to follow the path in a timely and efficient manner.

Another important parameter to tune is the given lookahead distance. This distance determines how far ahead the controller should aim to reach along the planned trajectory. If the distance is too small, the car may not have enough time to react to upcoming turns or obstacles, leading to potential collisions. Conversely, if the distance is too large, the car may end up cutting corners or taking unsafe turns, jeopardizing its stability and safety.

The fixed factor used to divide up each line segment in the planned trajectory is also an important tuning parameter. This factor determines the resolution of the path, with smaller factors resulting in more fine-grained trajectories. However, smaller factors also increase the computational cost of the algorithm and may not be necessary for all scenarios.

3 Experimental Evaluation

3.1 Planning Algorithm Comparison

Author(s): Brian

In order to numerically evaluate our RRT* and A* algorithms, we ran them with the same start and end points in simulation and recorded several statistics including time to complete the algorithm as well as total path length. These results are summarized in the Table 1 and 2.

3.1.1 Evaluating A*

Author(s): Brian

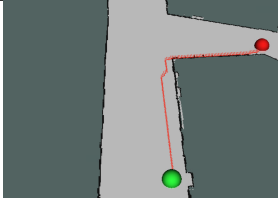
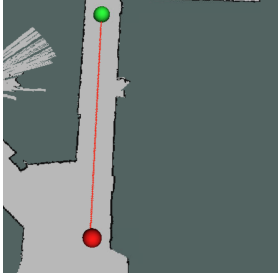
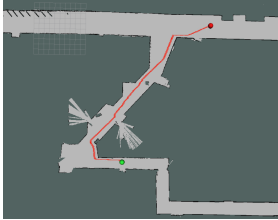
Test	Returned Path	Runtime	Path Length	Optimal Path Length
Corner		145.5 s	16.59 m	13.11 m
Straight-Line		0.5 s	13.94 m	13.73 m
Long Track		221.1 s	43.9 m	33.77 m

Table 1: A* Performance in Various Testing Scenarios

As we can see, A* runs very quickly in the straight line case. In addition, in all 3 test cases, the path returned by A* is close to the ‘optimal’ path (note that we define the optimal path length as the Euclidean distance between the start and end positions of the path). This can be attributed to the use of the Euclidean distance heuristic, which guides A* towards the optimal solution.

3.1.2 Evaluating RRT

Author(s): Rachel

The RRT algorithm has a time complexity of $O(N \log(N))$ for a data size of N samples. Our algorithm allows for 120 seconds of pathfinding and 5 seconds of shortcutting when given a start and goal pose. However, as shown in Table 2, the RRT algorithm rarely uses the full 120 seconds to find the path. For corner paths, the algorithm takes an average of about 2 seconds to find the path, plus the 5 seconds of shortcutting, resulting in an average total runtime of 8 seconds. Even for longer paths, the pathfinding algorithm takes roughly 32.40 seconds in total to find a collision-free path. While the average time required is not

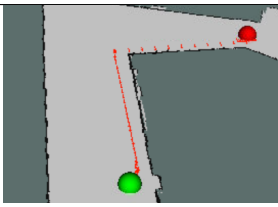
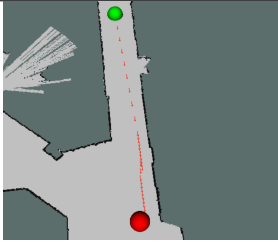

Test	Returned Path	Runtime	Path Length	Optimal Path Length
Corner		8.11 s	18.61 m	13.11 m
Straight-Line		7.73 s	14.76 m	13.73 m
Long Track		32.40 s	45.51 m	33.77 m

Table 2: RRT Performance in Various Testing Scenarios

significant, the path generated is not necessarily optimal. As seen in Table 2, the generated path length is greater than the optimal path length, which is calculated using Eq 1. Sampling-based pathfinding algorithms, such as RRT, typically generate paths that are not optimal. However, by using shortcutting, we can improve the path's quality and efficiency.

The RRT algorithm is also a probabilistic algorithm, which means that a path can always be found given enough time and that there exists a feasible path in the environment. However, despite its benefits, the RRT algorithm also has some limitations.

Shortcomings of RRT

Author(s): Rachel



Figure 7: Not optimal RRT Path

In addition to the inefficiency of RRT in finding optimal paths, another limitation of the algorithm is its inherent bias towards growing in large unsearched areas of the search space as shown in Figure 7. This means that the algorithm may prioritize exploring areas with fewer obstacles, potentially leading to paths that are not the most optimal. However, if this happens, the trajectory can always be replanned and the RRT algorithm will eventually find a more optimal path.

3.2 RRT vs A*

Author(s): Rachel

After implementing and testing both algorithms, our team decided to use the RRT algorithm over the A* algorithm. Although the A* algorithm can find more optimal paths than the RRT algorithm, it takes significantly longer time to find paths for long trajectories, making RRT the better choice. Furthermore, the addition of shortcutting also makes the paths generated by the RRT algorithm more optimal, resulting in path lengths that are fairly close to those of the A* algorithm. In summary, based on the results in Table 1 and Table 2, RRT is a better choice than A* due to the significantly shorter runtime for long paths and the small difference in path lengths between the two algorithms.

3.3 Pure Pursuit Testing

Author(s): Claire

We used cross-track error to evaluate the performance of our pure pursuit controller. For this lab, we defined the error at any given time as the absolute difference between the car's current position and the car's target location at that point in time. The target location on the trajectory was identified from the algorithm mentioned above, and the car's current position was taken from our localization readings. We tested our pure pursuit controller on three different trajectories in simulation: straight down a hallway, around an open corner, and on the labeled track in Stata basement (Figure 8).

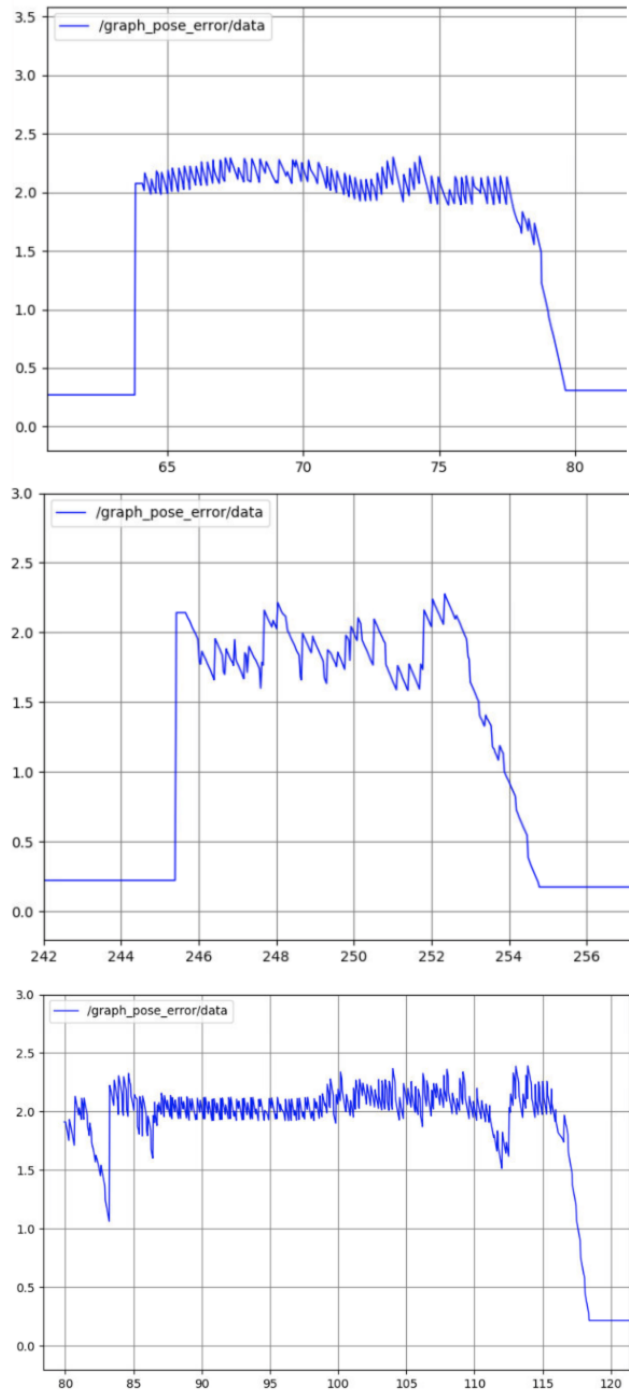


Figure 8: Cross-track error over time on a straight trajectory (top), open corner (middle), and on a path following the Stata track (bottom).

Finally, we also measured cross-track error on the car (Figure 9).

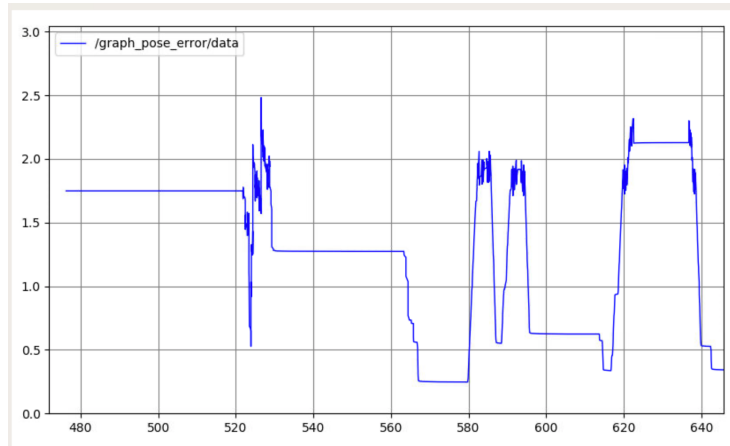


Figure 9: Cross-track error while running pure pursuit on the car.

3.3.1 Tracking Trajectories

Author(s): Claire

From the three graphs in Figure 8, we can see that our pure pursuit controller was very accurate in simulation. Before and after the car begins following the trajectory, we see that cross-track error stays at around 0.25. This non-zero baseline is likely because the lookahead distance assures that the target point is always ahead of the car, so the cross-track error will always be non-zero. We can also see that while the car is following, the error stays very consistently around 2 for all conditions. The cross-track error is tightest for the straight path and varies the most on the open corner. For the third graph, the big dip at the beginning is because of the open corner at the start of the Stata track, and the error remains close to 2 for the rest of the path. Open corners exhibit the largest variation in cross-track error because the trajectory is made up of line segments of different angles. The angling in the trajectory segments on open corners result in an initial large error before the car has angled onto that specific segment, then a decrease in error as the car adjusts, before another large increase in error from adjusting to the angle of the next trajectory segment.

All three graphs exhibit a sawtooth shape because of the constant recalculation of the target point. Once the car gets close to one target point and the error decreases, the lookahead distance ensures that there is a new target point further away, so the error jumps back up again.

On the car, we do not see errors that are as smooth (Figure 9). The error

looks much more like a sequence of square waves that jump up and down. This could be due to several factors, but the most likely cause is that our localization was slightly inaccurate. Since our localization was off, the errors in localization contributed to the errors from pure pursuit itself, so the amplitude of differences in errors is much larger. In addition, the errors in localization would accumulate over time since each new prediction is based on the prior one, which is why we see huge spikes later on in the error graph. We still see the sharp dip at the beginning from first rounding the open corner, and then plateaus at varying magnitudes as the car continues to turn along the trajectory. The square wave pattern is very interesting since it only appears on the car, but this is likely due to errors in localization increasing variation in the graph.

3.3.2 Shortcomings

Author(s): Claire

The main shortcomings of our pure pursuit controller is that it still cuts through the wall for trajectories that go around corners. This could be because the lookahead distance causes the target point to be far along the curve, so the car turns earlier to reach this further point and ends up cutting across the wall. One major change that we could make to the pure pursuit controller is to decrease the lookahead distance when going around corners so that we make smaller adjustments to our steering angle. In general, dynamically adjusting our lookahead distance could result in more gradual and appropriate steering angle changes while the car is following the trajectory.

4 Conclusion

Author(s): Lauren

4.1 Accomplishments

Through this lab we have successfully taught our car how to drive. Specifically, we planned trajectories and programmed our car to follow these paths by accomplishing the following:

1. Implementing a sample-based planning method to determine trajectories in combination with morphological dilation to avoid collisions
2. Incorporating pure pursuit control to follow these trajectories
3. Combining planning and control to allow our racecar to drive successfully in simulation and in real life

4.2 Next Steps

Before proceeding to the final challenge our team has the following tasks:

1. Continue tuning the look ahead distance for our pure pursuit controller
2. Test pure pursuit on different racecar speeds to find maximum tolerable speed

Once our team has accomplished these tasks, our team will be prepared to move onto the final challenge where we will synthesize everything we have learned and streamline everything we have created to defeat the evil RACECAR Em-tire!

5 Lessons Learned

5.1 An Bo

Throughout this lab, I gained valuable insights into the process of planning trajectories for the race car. We leveraged a sampling-based planning method to generate a path from the car's current position to a designated goal pose. As well, it was fascinating to witness the car following a predefined trajectory using our particle filter and pure pursuit controller. By merging these two goals, we were able to achieve real-time path planning and execution on our physical racecar. One of the most critical aspects of this lab was the emphasis on team communication. Working as a team, we had to ensure that everyone understood their roles and responsibilities to accomplish our goals effectively. Through open communication, we were able to distribute the workload effectively, ensuring that every team member contributed their best efforts towards achieving our objectives. Moreover, I realized that effective communication goes beyond delegating tasks. It also involves actively listening to one another, providing constructive feedback, and offering solutions to any issues that arise. By working together as a team, we were able to overcome any challenges we faced and achieve our objectives in a timely and efficient manner.

5.2 Brian

One of the lessons I took away from this lab is the importance of separating issues with our racecar from issues with our code. While testing our path planning implementation on the racecar, I noticed that the LiDAR scans from RViz were behaving. The scans flickered erratically and didn't match up with the corresponding physical environment. At first, I thought we had an incorrect transformation or there was something going on with Rviz. However, none of these

proved to be the source of the issue. After a lot of head scratching and the help of a TA, we isolated the issue to the LiDAR scanner on the car. By restarting the car, we were able to set the LiDAR scans back to normal. This lab also made me realize the importance of having good standard practices and workflows, especially when it comes to testing. For example, in order to run the path planner on the racecar, I had to run 4 different launch files in 4 different terminal windows and run the pose sub/pub files in order to manually set the initial and goal poses for the car. Having to rerun all these files whenever we wanted to test the car ended up taking a significant chunk of time that we could have used for testing. Going forward, one thing we can do to fix this is to put all our launch files into a single launch file in order to streamline the launch process.

5.3 Claire

I learned a lot about pure pursuit and geometry in this lab - mainly, to actually try to keep things as simple as possible. I wrote the logic in pure pursuit in one sitting, going off of the given StackOverflow posts, but I ended up with code that was very messy, had many different parts, and was filled with formulas that I didn't actually understand. This became a major issue when it didn't work on the car and other teammates would try to debug my code, only for it to be too confusing. In the end, Rachel ended up massively rewriting my original code for it to have worked, and greatly simplified the way in which we were even going about pure pursuit. This taught me a lot about truly understanding the problem, developing an easy approach by myself, and working on implementing that, rather than blindly following the staff instructions. I also learned the importance of asking for other teammates' help and involvement early on in the implementation process, as the idea of handing off code to each other becomes impractical when the code is hard to understand. I also really felt the importance of getting everyone to start early - I think we were all tired and or burnt out from racing to meet the deadline for lab 5, so we didn't start this lab until a few days afterwards, but it meant that we were also racing to meet the deadline for this lab. For the final project, we'll try to set ambitious goals to keep ourselves on time or ahead of time.

5.4 Lauren

Given the complexity of this lab, the biggest lesson that I learned was the importance of timing and coordination. While we were given two weeks to work on this lab I believe that the earlier due date of the briefing pushed us to complete a large portion of the lab in the first week. Overall, I enjoyed learning more about pure pursuit and pros and cons of search-based and sample-based motion path planning.

5.5 Rachel

As with previous labs, it was crucial for us to effectively divide the work into modules and work on them synchronously. We began by deciding to implement both sample-based and search-based path planning algorithms, with me working on the RRT and Brian working on A*. After careful consideration, the team agreed to implement both algorithms and perform a cost analysis to determine the best one to use. Meanwhile, other team members worked on Pure Pursuit. This experience taught me the importance of iterative programming and the significance of thoroughly testing each component of an algorithm. Given the complexity of the algorithms, it was crucial to ensure that each component functioned correctly before integrating it into the larger algorithm. Although I had prior experience working with the RRT algorithm, collision detection remained a challenge. However, after careful consideration of the graph and conversions needed, I was able to implement the RRT algorithm on the racecar. This project reinforced the importance of effective communication and teamwork to successfully integrate multiple modules into a cohesive final project. We effectively communicated the issues we encountered and asked for help when necessary.