Lab #6 Report: Search and Sampling-Based Path Planning on Autonomous Robots

Team 2

Esha Ranade
Ian Pérez
Jason Salmon
Ritika Jeloka
Sydney Chun

6.4200 Robotics: Science and Systems

April 27, 2023

Edited by: Esha Ranade

1 Introduction

Esha Ranade

The goal for this lab was to enhance planning and control operations by determining a path to a desired destination and enabling the robot to follow it, further developing our racecar's autonomous capabilities. In our previous labs, we were able to program our robot to follow walls and lines, move in accordance with objects within its environment through visual servoing, and determine its orientation and position within a known environment through Monte Carlo Localization (MCL) using scans from a Light Detection And Ranging (LiDAR) sensor and wheel odometry data. However, our robot had no sense of how to navigate to a destination given its current position. To address this limitation, we implemented, tested, and tuned several path-planning algorithms—namely rapidly-exploring random trees (RRT), RRT^*, and A^*—to find trajectories to a desired location and a pure pursuit controller to determine drive commands to follow these trajectories. With this approach, the robot can now navigate from its current location to a desired destination on a map while avoiding obstacles and minimizing collisions. This knowledge is essential to achieve autonomous driving, which will aid us as we attempt to take on the city driving components of the final challenge. Path-planning is commonly used in robots across industry, particularly in the heavily-researched field of self-driving vehicles. Companies

like iRobot incorporate this feature into products like the Roomba, an automated vacuum and mop, to enable robot actions like automatically returning to its charging dock once a cleaning job is complete or when its battery is low.

2 Technical Approach

2.1 Problem Statement

Esha Ranade

This lab's primary objective was to enable our racecar to autonomously navigate to a goal position given its current location and information about its surroundings. To achieve this goal, our team subdivided our path-planning implementation into individual deliverables, each of which is described in further detail below, to effectively modularize our code structure and testing process:

1. **Path Planning.**

   The Path Planning module plans 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. **Pure Pursuit.**

   The Pure Pursuit module provides driving commands to the car to follow a predefined trajectory in a known occupancy grid map using particle filter localization and pure pursuit control.

3. **Integration.**

   The Path Planning and Pure Pursuit modules are combined to enable real-time path planning and execution. Deploying particle filter localization, path planning and following modules, and a modified safety controller on our physical racecar allows us to determine our robot's current position, plan trajectories to a set destination, and enable autonomous driving to that point without obstacle collision.

2.2 Full process

2.2.1 Path Planning

Ian Perez

Path planning is the process of finding a feasible path from an initial position to a goal position while avoiding obstacles in the environment. Our team im-

(a) Undilated map (b) Dilated map

Figure 1: Effect of dilating the map

plemented two different types of path planning algorithms: A* (search based) and RRT/RRT* (sample based).

2.2.1.1 Map Dilation

Before planning a path on the map, we have to ensure that the planned paths are feasible and safe for the robot. To achieve this, we dilate the obstacles in the map. Dilation involves expanding obstacles by a given amount in order to create a safe margin around them. This ensures the robot has enough clearance when navigating through the environment so that it does not collide with anything.

2.2.1.2 Map to Image Coordinates

In order to plan a path on the map, we need to convert the map coordinates into coordinates that we can use in the occupancy grid. In order to do so, we subtract each component of the pose from the map origin and divide the result by the map resolution. The result of this is the corresponding point of the map in the grid.

2.2.1.3 A*

A* is a search-based algorithm that uses a heuristic function to guide the search towards the goal. The algorithm works by maintaining a priority queue of nodes to explore. The priority of each node is determined by the sum of the cost of the path from the start node to the current node and the estimated cost from the current node to the goal node.

To implement A*, we first decided to represent our occupancy grid as a graph where every pixel was connected to its 4 neighbors. The only time a pixel was not connected to a neighbor was if the neighbor corresponded to a pixel with an obstacle. We then initialized the start and goal nodes and added the start node to the priority queue. We then looped through the priority queue, expanding the node with the lowest priority. We add the neighbors of the current node to

the priority queue and updated their priorities. We continued this process until we reach the goal node or the priority queue is empty.

2.2.1.4 Rapidly-exploring Random Trees (RRT)

RRT is a sample-based algorithm that works by incrementally building a tree of nodes in the configuration space of the robot. At each time step, the algorithm randomly selects a configuration in the space. We bias towards the goal such that there is a 10% chance that the configuration selected is the goal position. A new node is added in the direction of the randomly selected one at a distance specified by the step size parameter only if there is no obstacle between the two nodes. We continue this process until we find a node within a given tolerance to the goal. We then find a path from the goal node to the start node using the parent pointers of each node. The tree generated by this algorithm can be visualized in Figure 2.

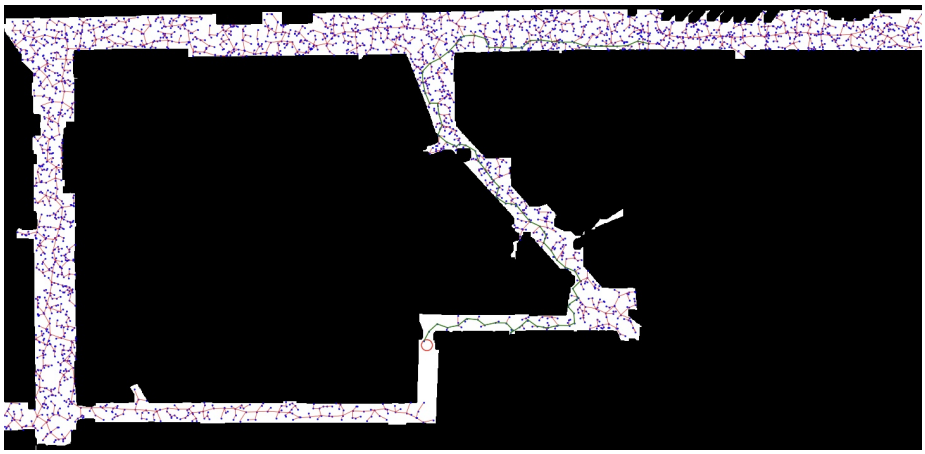


Figure 2: Tree generated by RRT

2.2.1.5 RRT*

RRT* is an extension of RRT that aims to improve optimality of the generated paths. It does so by using distance as a cost function to guide the search towards the goal and re-wiring the tree to reduce the cost of the path.

In order to implement RRT*, we first initialize the tree with the start node. At each timestep, we select a random configuration and find the nearest node in the tree. We add a new node in the direction of the random configuration if it does not collide with any obstacles in the environment. We then rewire the tree around a given radius of the new node to reduce the cost of paths. We continue this process until we reach a node within the goal tolerance.

We implemented a couple of optimizations for this algorithm. One of them was a shrinking ball radius given by the following equation $r = 100 \cdot \sqrt{\frac{\log n}{n}}$ where n is the number of nodes added to the tree at that given point. This balances exploration and exploitation in the search. When the tree is small, the radius will be larger, allowing the algorithm to explore a larger area of the space. As the tree grows and becomes more dense, the radius will shrink, allowing the algorithm to exploit existing nodes and connect to them more efficiently. The other optimization was the use of the RTrees to store nodes. This allowed for faster nearest-neighbor searches.

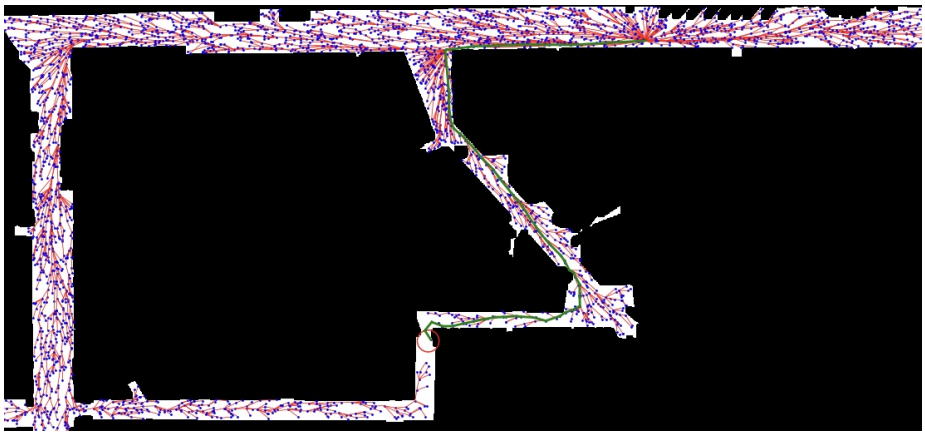


Figure 3: Tree generated by RRT*

2.2.1.6 Search-based vs. Sampling-based

Search-based methods such as A* and sampling-based methods such as RRT/RRT* each come with their own strengths and weaknesses. For example, search-based methods are guaranteed to find a path if one exists. It will also find the shortest path to the goal if one exists. However, it can be slow when the search space is very large and memory-intensive as it needs to store all the nodes it has visited.

On the other hand, sampling-based methods such as RRT/RRT* can quickly explore the search space and find a solution when one exists. Because of this, they are well-suited for dynamic environments where obstacles move. On the other hand, they are not guaranteed to find a solution and if they find one it is not guaranteed to be the shortest one. This is less likely with algorithms like RRT* that take cost into account. As will be explained in Section 3, we decided to go with RRT* since it provides the shortest path most of the time a lot faster than A*.

2.2.2 Pure Pursuit

Jason Salmon

For this lab, the robot should be able to safely traverse an arbitrary trajectory between some start and end points. To take full advantage of the specificity of our pre-planned routes, we used a pure pursuit controller to automatically adjust the steering angle of our robot. Although pure pursuit can technically be thought of as a niche implementation of PID control, there are key differences that make it advantageous in this situation.

- **The controller anticipates the path.** As opposed to our wall follower, where we had some line as an estimate of a wall and the projected distance of our car's location onto this line became our error signal, pure pursuit controllers are more nuanced. The concept relies on some known trajectory, as well as a specified distance between the car and the point we want it to follow. This is called the lookahead distance - the car is quite literally *looking ahead* of its own position, and using the trajectory at that distant location instead of its current location to inform its movement. This means that, while our wall follower would fail to start turning until it gets right next to a corner, the pure pursuit controller would begin to adapt ahead of time. This makes tracking for more general cases far simpler. This point in particular ties into the second key advantage of pure pursuit.
- **The controller tracks complex trajectories.** This is partially to do with its aforementioned anticipatory behavior. However, the second dimension to consider is the amount of information the controller considers. Previously, we have implemented PID controllers which handle one-dimensional streams of data - like the distance in exactly one direction, or the bearing of exactly one point in the robot frame. Pure pursuit, on the other hand, handles a lookahead *point* - some location at some distance and bearing relative to the robot. In essence, we have access to the full polar coordinates of the point we want to follow in the x-y plane. This is a useful and compact representation for our purposes since the nonholonomic kinematics of the racecar allow us to minimize error in x and y using only the bearing, assuming a fixed lookahead distance. This phenomenon can create even richer control schemes if we allow the fixed parameters to vary — like having a smaller lookahead distance for sharper turns.

With these thoughts in mind, we used a pure pursuit controller with a fixed lookahead distance for our first-pass solution.

2.2.2.1 Theory

Imagine a robot at some location with some pose in the global frame, $\begin{bmatrix} x & y & \theta \end{bmatrix}^T$. We determine the lookahead point by finding the intersection between the given

trajectory and the circle centred at $\overline{\alpha} = \begin{bmatrix} x & y \end{bmatrix}^T$ with radius equal to the lookahead distance L_1. This point will have bearing η relative to the robot. Using the Ackermann Steering Model, the desired steering angle for pure pursuit is given by Equation 1, where L is the wheelbase length of the car.

$$\delta = -\tan^{-1}\left(\frac{2L\sin\eta}{L_1}\right) \tag{1}$$

Assuming all values but η are independent of the lookahead point, this calculation by itself is straight-forward. However, because of the discretized nature of the trajectory, the calculation of the lookahead point is slightly more involved.

2.2.2.2 Sorting Trajectory Segments

The trajectories we worked with were given as a list of waypoints. The first step to finding potential intersections is to convert this list of points into the corresponding trajectory segments between consecutive pairs, then find the segment closest to the robot's current position. We consider an ordered set of position vectors which represents our waypoints:

$$T = \begin{bmatrix} \overline{p}_1 & \overline{p}_2 & \overline{p}_3 & \cdots & \overline{p}_n \end{bmatrix}$$

Our segment starting points are every position vector but the last, and our ending points are every vector but the first. Therefore, the corresponding ordered set of segments is given by

$$S = \begin{bmatrix} \overline{p}_2 - \overline{p}_1 & \overline{p}_3 - \overline{p}_2 & \overline{p}_4 - \overline{p}_3 & \cdots & \overline{p}_n - \overline{p}_{n-1} \end{bmatrix}$$

The closest point as a vector, $\overline{p}_{c,k}$, would ordinarily be the projection of the robot's position onto each segment. However, since the line segments have finite length, the eventual vector must be clipped to the endpoints of its segment. In other words, we have the following:

$$\overline{p}_{c,k} = \overline{p}_k + t \cdot \left(\overline{p}_{k+1}\overline{p}_k\right) \tag{2}$$

$$t = \min\left(\max\left(\frac{\left(\overline{\alpha} - \overline{p}_k\right) \cdot \left(\overline{p}_{k+1} - \overline{p}_k\right)}{\|\overline{p}_{k+1} - \overline{p}_k\|^2}, 0\right), 1\right) \tag{3}$$

The 2-norm of the vector between the result and $\overline{\alpha}$ is the shortest distance for segment k. We can then compare these values across segments to order the list.

2.2.2.3 Finding Intersections

This process proceeds in the newly-sorted segments. Firstly, we exploit the directionality of the trajectory to reject intersections with backwards segments (we do not want to go backwards, so we don't care about intersections behind the car). In other words, if we last had an intersection with section k, we will only look for an intersection with segment m if $m \geq k$. Overall, the question

is simply whether the distance between a hypothetical lookahead point on the segment and the position of the car is equal to the lookahead distance. In other words, we would like the following:

$$\left\|\overline{p}_k + l \cdot \left(\overline{p}_{k+1} - \overline{p}_k\right) - \overline{\alpha}\right\|^2 = L_1^2 \tag{4}$$

This becomes a quadratic in terms of l, and one of the roots corresponds to our lookahead point. We must also handle ambigious/edge cases:

- **There are no intersections, or l is not between 0 and 1 (the intersection is not within the segment)** - the intersection is rejected.
- **There are two valid intersections on the segment** - we choose the intersection ahead of the robot by selecting the larger of the two l values.

If no valid intersections are found, we proceed to the next segment in the ordering. If no intersections are found for any of the segments, the lookahead point is temporarily considered to be the lookahead point from the previous time step.

2.2.2.4 Determining η

To find η, we compare the heading of the robot in the global frame to the angle of the vector from the robot to the lookahead in the global frame. To accomplish this while retaining sign information (we need to know if the path is to the left or right of the robot), we determine the four-quadrant arctangent for both cases.

The heading of the robot is given by the unit vector $\begin{bmatrix}\cos\theta & \sin\theta\end{bmatrix}^T$, while the relative vector to the intersection is given by $\overline{p}_k + l \cdot \left(\overline{p}_{k+1} - \overline{p}_k\right) - \overline{\alpha}$. η is the difference between these angles.

2.2.2.5 Operation

In general, to operate pure pursuit, the car was placed at the start point of a loaded trajectory in the Stata basement map. In simulation, this was first done using the absolute odometry of the robot to test our code while using a less noisy signal. In practice, odometry data was obtained from the most likely pose determined via the particle filter. The marks of a successful implementation were following a high percentage of the given path and minimizing cross-track error.

2.2.3 Safety Controller Upgrade

Sydney Chun
Before testing the code in the robot and running it at full speed, we decided to update the safety controller that we implemented during the first lab: wall follower. In that safety controller, the robot would only stop for a limited range in front of the robot. In this lab, we were going to be turning corners at a fast rate, using particle filter and localization, and following a planned path, so

we needed a better safety controller to prevent our robot from breaking due to unplanned incidents.

To prevent the robot from running into a wall, obstacle, or clipping the side, we expanded the robot's view to a full 180 degrees around the side and front of the robot. We took the LiDAR data to find the distance of the robot from obstacles around it and sampled the min distances. If we had more than 5 samples a threshold distance of 0.5 meters from the closest obstacle, we triggered the stop-driving command.

The stop-driving command was also triggered if the particle filter was failing or the robot deviated too far from the path. We implemented this by adding a publisher to both the particle filter and pure pursuit that signaled an error message if something went wrong or the robot was too far from the path. Our safety controller then subscribed to the publishers in the particle filter and pure pursuit and activated the stop-driving command when a message was published.

2.2.4 Integration

Sydney Chun
After implementing and testing the path planner and pure pursuit in simulation, we started the integration of the code into the robot. The integration into the robot required a series of sequential steps. The first step was to run teleop to connect the robot to the joystick. The next step was to run the localization in the real environment. This allowed the car to know where it was positioned within Stata basement. We were able to visualize this in RViz through pose arrays and an odometry arrow. After the localization was running, we ran the code to follow trajectory, which activated our pure pursuit code. From pure pursuit, we were able to visualize the marker for the path and end goal on RViz. Finally, we ran the plan trajectory which implemented path planning and published our end goal. Once the path was found, our robot was able to autonomously follow the path it was given using pure pursuit and reach the end goal.

We ran into several issues during this process. The first issue was the map topic not being found when we ran the localization in the real environment. At first, the issue was simply because we forgot to publish the initial state of the robot; however, there were times when even publishing the robot's initial position did not find the map topic. Instead, we took the on-off approach where we reset our operations in a sequential manner from high level to low level until it worked. Starting with RViz and working our way down to the individual launch files, the robot, then the docker, we were able to relatively successfully resolve most issues by using this method. The other major issue we ran into was our inability to visualize the planned path on RViz. Without the visualized path, our robot could drive in any direction and we would not know whether the robot was following a path or just guessing where to go. The resolution to

this issue was our sequential steps to run the launch files in the robot. When running the path planning launch file first, the marker topics never visualized the robot even though the robot showed it received the message for the planned path. Finally, our last problem involved the drive topic. The drive topic in pure pursuit was called using get params on rospy; however, the default value for the drive topic was for the simulation drive rather than the robot's drive topic. Once we resolved that issue, we were able to get the robot to drive and successfully test our robot's path planning.

2.3 ROS Implementation

Sydney Chun

The ROS implementation involved 4 subscribers, 3 path marker publishers, and 2 message publishers. For path planning, our code listened to the location of obstacles from the OccupancyGrid message on the /map channel. This message listed the occupancy values of map pixels so that we could check the occupancy of a real-world point for an obstacle so the path planning algorithm could avoid it. The next subscriber listened to by path planning is the Odometry message in the /odom channel. This was used to determine the robot's current position which would serve as the start point of the path planning algorithm. The final subscriber is the PoseStamped message in the /move_base_simple/goal channel which was the end goal of the robot. Combined, these three subscribers provided our code with the necessary data to produce an efficient and optimal path using our path-finding algorithms. Once the path was created, our first message publisher published the path to the channel /trajectory/current.

The last subscriber in our pure pursuit code used the published message from path planning to retrieve the trajectory the robot was to follow. The second message publisher was the drive topic which published an AckermannDriveStamped message to instruct the car on where to steer to follow the planned trajectory. Besides publishing markers that showed the path, start point, and end point, our pure pursuit published four debugging markers. The first debugging marker was a buoy marker which highlighted the segments of the path the robot was trying to follow. The second debugging marker was a dot that represented the point where the car was looking, which was on a spot along the circumference of the circle marker which showed how far ahead the car was looking. Finally, the fourth debugging marker was a line that showed which direction the robot is trying to turn. These debugging markers provided us with visual feedback about the state of our system and allowed us to address any unexpected behavior.

3 Experimental Evaluation

Ritika Jeloka

3.1 Path Planning Evaluation

After coding our path planning and pure pursuit modules and integrating them onto the robot as outlined above, we decided to test our implementation.
We first did this in simulation. We tested our three path planning algorithms: A* (search-based), RRT (sample-based), and RRT* (sample-based).

(a) A* Path (b) RRT Path (c) RRT* Path

Figure 4: A* vs RRT vs RRT* Path Planning

Figure 4a shows the path planned by A* and Figure 4b shows the path planned by RRT with the given start and stop locations. All three algorithms found a potential path, but the path computed by RRT was much longer with a length of 124.45 whereas the path computed by A* was less than half of the length with a measure of 47.23 and RRT* was 48.84. However, it is important to note that RRT found this path much quicker with a time of 5.27 seconds as opposed to RRT* which found it after 5.87 seconds and A* which found it at a time of 50.03 seconds. In the end, our team decided that RRT* outputted good quality paths in a much shorter timeframe that A*, therefore we decided to use RRT* as out path planning algorithm.

While modifying our RRT* path planning algorithm, we submitted our trajectories to Gradescope and visualized the path as shown in Figure 5 to compare our calculated/submitted trajectory to the solution trajectory.

Table 1: Adjusted lookahead distance values and scores

Lookahead Distance	Total Gradescope Score
0.6	6.8739
0.75	9.3601
0.85	**9.3603**
1	7.04
1.5	6.9417
3	5.2013

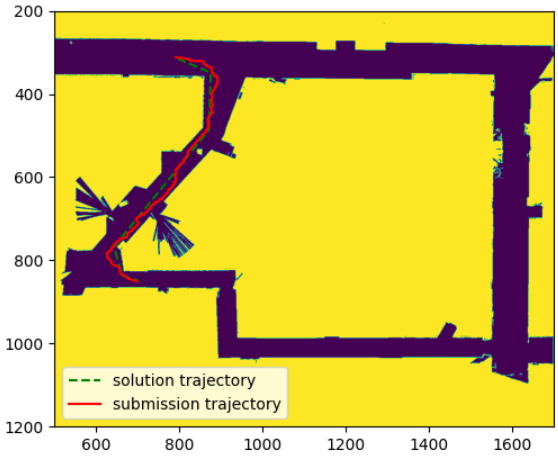


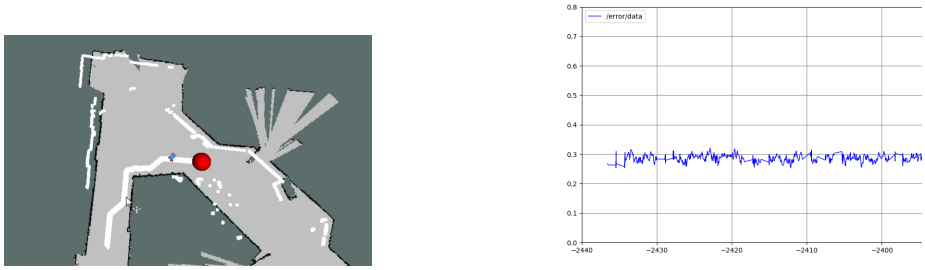
Figure 5: Path Visualization RRT*

3.2 Pure Pursuit Evaluation

Once we received a score of 2.97/3 on out path planning algorithm, we wrote our pure pursuit controller and submitted this code to Gradescope as well. We, unfortunately, received a very low score at first and had to tune our parameters by adjusting the lookahead distance as shown in Table 1 below to achieve our optimal lookahead distance of 0.85 which resulted in a score of 9.3603.

Next, we moved to implement our code from simulation to the robot. After noticing that parameters needed to be adjusted, we tried several different lookahead distances. We started with 0.75 which resulted in the car successfully following the given path (as shown in Figure 6a). Figure 6b shows the error

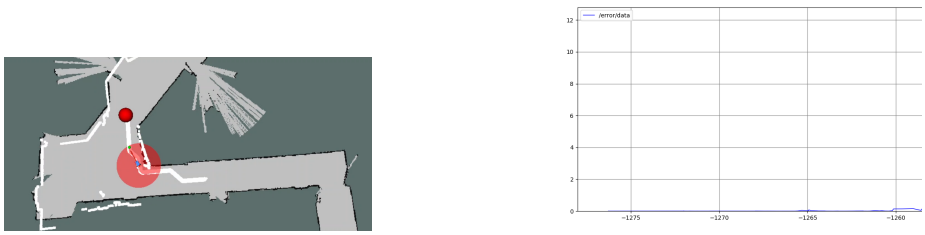
calculated as the difference between the position of the car and the planned trajectory position on the path. As shown, the error remains consistently around 0.275 for the run.



(a) Visualization (b) Error Plot

Figure 6: 0.75 lookahead value Visualization + Error Plot

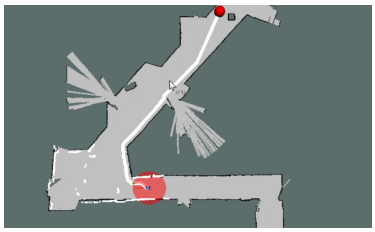
To reduce our error, we then tried a lookahead distance of 1.5. The run as shown in Figure 7a and the error plot as shown in Figure 7b show that the error was almost 0 during this run and the car stayed almost perfectly on the path.



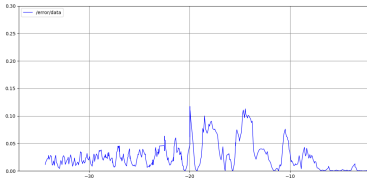
(a) Visualization (b) Error Plot

Figure 7: 1.5 lookahead value Visualization + Error Plot

Since this path was relatively short, we tried again but with a longer path as shown in Figure 8a. As shown in Figure 8b, the error stayed pretty low, ranging from 0.025 to 0.11.



(a) Visualization



(b) Error Plot

Figure 8: 1.5 lookahead value Longer Visualization + Error Plot

For verification, we tried a higher bound lookahead distance of 3. As expected and shown in Figure 9, the car veers completely off the path and continues to turn in circles for the remainder of the trial. We could immediately verify that this lookahead distance is an upper bound.

Figure 9: Lookahead Distance of 3 Visualization

We ultimately settled on a lookahead distance of 1.5 in our final pure pursuit approach as this value resulted in the least error.

As seen in the start and end points that our team picked, we avoided certain areas around the Stata basement as the robot consistently had problems there. We believe that near the water fountain, our localization code failed to pick up as many lidar scans and therefore beams, causing our robot to lose perfect sight of where it is when it pases this area. To avoid this, we created paths that did not pass by the water fountain.

4 Conclusion

Esha Ranade
Through this lab, we were able to achieve our goal of enabling our racecar to plan a path to a desired destination and autonomously drive to it. This involved building off of the previous lab on Monte Carlo Localization, which determined the robot's current pose using data from the robot's LiDAR sensor scans, a provided map of the environment, and odometry data collected from the car's movements. Knowing the robot's position and orientation informed our next steps in determining how to navigate to another point on the map. We projected several trajectories using search and sampling-based path planning and chose the optimal one that avoided all obstacles to prevent collisions while driving. We implemented a pure pursuit controller to provide the corresponding motion commands to the racecar for it to efficiently follow the chosen trajectory. With this update in functionality, our robot is now able to autonomously create and follow a path from its current position to a desired destination on a given map. This development in our robot's capabilities puts our team one step closer to addressing future objectives, such as driving through the cityscape in the final challenge.

5 Lessons Learned

5.1 Ritika

This lab showed me that some times editing parameters can work some days and not others. We spent countless hours debugging, when all it took was a reset or working on another day with the same code for it to work. This taught me patience and resilience. This lab also showed me the importance of teamwork as my team came together many days in a row, all pitching in where we can to get the job done.

5.2 Ian

During this lab I learned several technical skills related to path planning, control, and localization. Specifically, how to use different motion planning algorithms such as search-based (A*) and sampling-based methods (RRT/RRT*). I also learned the importance of clear and concise communication with my team which was essential for collaborating on Part C.

5.3 Esha

Working on this lab involved parsing through code from several different sources, which I found required a great attention to detail to properly integrate. It also taught me that the "process" really matters. We ran the same few commands in what seemed like every permutation of orders to figure out what would solve

our problems. After figuring it out once, we were very organized in how we performed each step, from writing down "what worked" and having the commands in ordered tabs of our terminal to be run in succession. This allowed us to be very efficient in our iterating and testing process. We also learned that "turning it off and turning it on again" often works, both for the robot/simulation and also for our own minds. Our team was stuck trying to make the visualization appear on our RViz when connected to the robot for many hours, leading to some degree of frustration. After ultimately deciding that we shouldn't sink too much more time into a single issue and instead wait to get help, we each mentally took a break and found ourselves to be much more productive upon returning to the task the next day.

5.4 Jason

An important takeaway for me was that, even though you can visualize something, code that thing, and confirm that thing should work, there is no guarantee that it will actually work. There were several points during this lab where we felt that we had done everything correctly, but were still somehow unable to get pieces of our code working on the robot. For the better part of a week, We were stuck with our pure pursuit controller not working on the robot, with no hope of visualizing the trajectory or debugging through other means. This was despite our best efforts to scour the lab documentation /Internet for answers and consult the TAs. After all, it worked in simulation, so why wouldn't it work on the robot? Finding out that there will be things that could happen without any real hope of resolution (we still have no idea how our code is working now) was a tough pill to swallow. In the future, we'll hopefully find that these situations are rare, and just require us to be patient and persistent to get through them.

5.5 Sydney

During this lab, the biggest struggle was the visualization of the path. Although our team faced other issues, they were resolved by discussing among team members about different strategies on how to approach them. The visualization of the path, however, was particularly troublesome none of us could find the issue, none of the piazza posts or TA suggestions resolved the issue, and all of the TAs were not in stata to help us resolve the issue during OH or our team was not available during stat OH. In the end, I learned that I needed to fully understand the issue and discuss it properly with others to resolve the issue. The most valuable lesson I learned is that a solution can be found even if the most conventional ones are not always available. This lab made me appreciate my teammates tremendously as we all put in so much effort in figuring out a solution despite it taking a few days and several hours of attacking the same issue. These lessons will serve me well in the upcoming final challenge.