Lab 6 Report: Path Finding

Team 12

Angel Gomez
Ariel Fuchs
Gokul Kolady
Sharmi Shah

6.141

April 21, 2022

1 Introduction (Author: Gokul, Editor: Ariel)

In this lab we were tasked with giving our robot the ability to plan and follow a path from its current position to a goal location on a known map. Given information about the robot's area in the form of map occupancy data (which parts of the map are occupied with obstacles), we were to first create a path planning module that could create a trajectory of locations that the robot could follow to reach the goal location from its current position. Next, we had to implement a pure pursuit motion module that could navigate the robot through the trajectory in order to reach its desired point.

To implement our path planning module, we utilized map dilation to make the obstacles seem bigger than they are in order to make the robot's planned movement extra safe, along with graph creation to turn the map data into a graph of unoccupied locations and edges connecting them. We then tried two methods for path planning given this graph. The first was search-based planning, in which we used search algorithms such as BFS and A* to find a path on the graph to the goal location. The second was sample-based planning, in which random unoccupied locations are sampled and are used in the final path if they don't have obstacles between them. We also had to implement pure pursuit, which involved taking in a trajectory from the robot's position to the goal point and constantly steering the robot toward a point in the trajectory that was slightly further ahead than the robot.

Pure pursuit requires information about where the robot is on the map at any given time in order to steer it, however this ground truth information is not available to us. Thus, we had to build on the last lab by utilizing the localization module in order to maintain a real-time estimate of the robot's location during path following. Path planning and following is a very important ability for an autonomous robot to possess. Oftentimes, in real life deployments of autonomous robots, information is known about the area that the system needs to traverse, yet it may not be an option to have someone manually drive the robot in order to execute its tasks. Thus, the robot being able to travel to any location on a map is a very powerful skill.

2 Technical Approach

The goal of this lab was to be able to plan and follow a path from the robot's current position to some goal location given information about the map's layout and occupancy. This required the creation of two primary modules. The first was a path planning module that functions to plan a trajectory of locations for the robot to follow in order to get to the desired goal point. The second was a pure pursuit module that utilizes the pure pursuit motion algorithm to control the robot to follow the planned trajectory.

2.1 Path Planning

This module was created with the goal of taking an input map message (with information about occupancy), the robot's current estimated position provided by the localization module from the previous lab, and a goal location and planning a trajectory of locations that the robot can follow in order to reach the goal point. This involved a couple of pre-processing steps as described below. Additionally, we implemented two different types of path planning.

2.1.1 Map Dilation

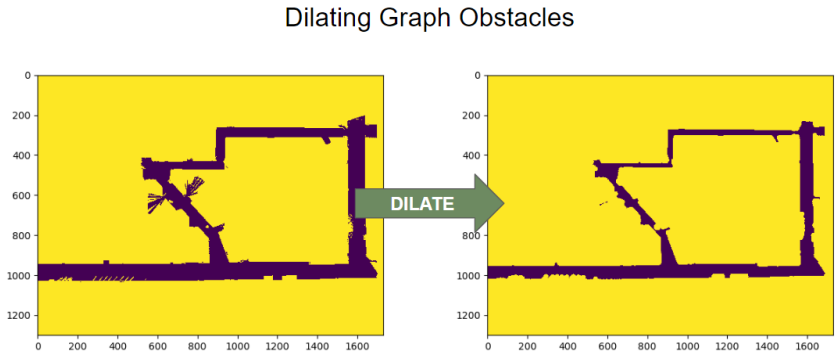


Figure 1: Shown here is the map image before and after dilation, where yellow represents obstacles. This dilation discourages the robot from planning a path that navigates right next to walls and obstacles.

When an OccupancyGrid message is received on the map topic (an object that contains information about which pixels of the map image are occupied with obstacles), it is first converted to a 2d matrix where element (v, u) represents pixel (u, v) on the map image. Next, it is dilated using the skimage.morphology.dilate function. Essentially, what this function does it take every occupied pixel in the map matrix and marks pixels around it within a specified radius as occupied. The idea is that if obstacles are dilated to be bigger than they are, then the planned path through them will avoid nearing obstacles to a higher degree, and thus the robot has a lower chance of crashing into obstacles even if it doesn't follow the planned path flawlessly.

2.1.2 Graph Creation

Now that the map has been dilated, an adjacency dictionary is created to represent a graph of unoccupied pixels. This dictionary has a key in the form of a (u, v) tuple for every unoccupied pixel in the dilated map matrix, and each

key's respective value is a set of adjacent pixels that are also unoccupied. This graph creation process is visualized in figure 2.

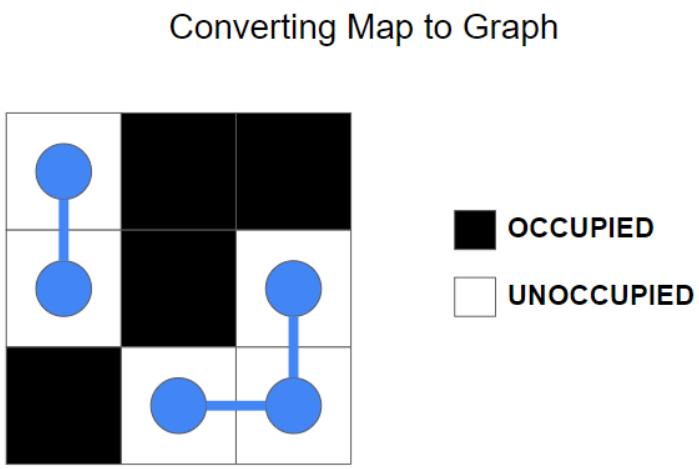


Figure 2: As shown in this figure, nodes are created at unoccupied pixels, and adjacent nodes are attached with edges.

2.1.3 Conversion Between Coordinate Types

Both of the following path planning methods require the ability to convert between (u, v) pixel coordinates on the map image and real-world (x, y) coordinates that the robot can utilize. Given the map message's position, rotation, and resolution information, the equations for these conversions are shown below.

Pixel (u, v) to Real (x, y):

$$(x, y, \_) = (u * r_{map}, v * r_{map}, 0) \cdot R_{map} + Translation_{map} \tag{1}$$

Real (x, y) to Pixel (u, v):

$$(u, v, \_) = \frac{((x, y, 0) - Translation_{map}) \cdot R_{map}^{-1}}{r_{map}} \tag{2}$$

where r_{map} is the map's resolution, R_{map} is the map origin's rotation, and $Translation_{map}$ is the map origin's translation. The _ symbols represent irrelevant/arbitrary thetas received from the calculation.

2.1.4 Search-Based Planning

Once a goal point is received, the path planning module aims to create an efficient and clean path from the robot's current estimated position to the goal

point. Because these start and end points are received as (x, y) coordinates, they are first converted to pixel coordinates using equation 2. Next, a graph search is performed on the map pixel graph in order to find a path from the start pixel to the end pixel.

We tried implementing both BFS and A* search. A* was implemented by restricting BFS to extend potential nodes based on a hueristic. Essentially, for a given node v, its heuristic value is the sum of the length of the shortest known path from the start node to v and the euclidean distance from v to the end node. Finally, once a path is found, each pixel in the path is converted back to (x, y) space using equation 1 and published as a trajectory. We chose not to incorporate dynamics into our search-based planning, as dilation proved to be able to keep the planned paths at a safe distance from the walls, and we tuned our pure pursuit so that it had no issues following even sharper turns quite closely.

We chose to utilize A* in our final implementation, primarily because it cuts down on run time compared to BFS. The asymptotic time complexity of BFS is $O(V + E)$, where V is the number of vertices (or unoccupied pixels) in our map graph and E is the number of edges. However, There are a maximum of 4 outgoing edges per vertex in our graph (as they can only connect to adjacent pixels), thus the run-time is effectively $O(V + 4V) = O(V)$. Theoretically, A* search should cut down on this run-time, as it is essentially BFS but it uses a heuristic to avoid extending inefficient paths, so we decided to use this algorithm for the final robot implementation. However, after comparing the simulation run-times of these algorithms for different path lengths, we realized that our BFS implementation was actually significantly faster:

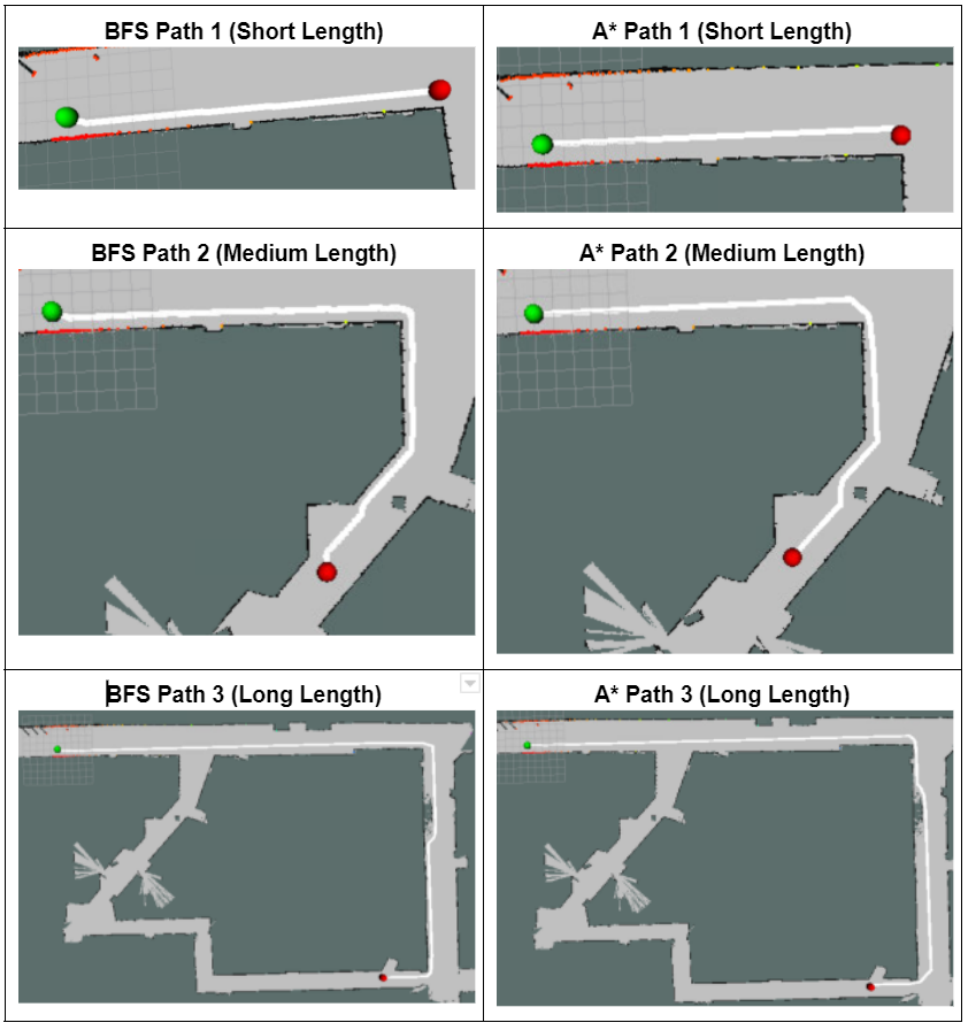


Figure 3: These images show the paths planned by both BFS and A* for increasingly farther start and end points.

BFS vs. A* Simulation Run-time Comparison	BFS Run-time (s)	A* Run-time (s)
Path 1 (Short Length)	0.2336	1.9608
Path 2 (Medium Length)	0.5602	4.4062
Path 3 (Long Length)	1.3686	10.1881

Table 1: This table compares the run-times of BFS and A* when planning paths on increasingly far start and end point pairs in simulation.

Table 1 shows the run-times of BFS and A* in simulation when executed on increasingly farther start and end points. Figure 3 shows what these paths looked like after being planned by the respective algorithms. Based on these results, our A* implementation was much slower than BFS overall, and we suspect this was because of the time-inefficient method that we used to maintain a queue of heuristically sorted paths to extend (we used numpy sorted list search and insertion methods to maintain the sorting of this queue, which are both quite time-consuming when the queue is large). Thus, for future use of this path planning module, we would likely use BFS or re-construct our A* search to use a more efficient queue-maintenance method (ie. a min heap).

2.1.5 Sample-Based Planning (Author: Sharmi, Editor: Gokul)

Sample-based planning is another class of algorithms that can be used to plan a trajectory. There are several different types of sample-based planning algorithms, but the one this report focuses on is RRT (Rapidly-Expanding Random Trees).

The steps of the RRT algorithm are explained below:

1. Randomly sample a location from the configuration space. For this application a random, unoccupied pixel was chosen from the stata basement map. Furthermore, the sampling was biased so that there was a $\frac{1}{20}$ chance that the goal point would be chosen so that the expansion would veer towards the goal.

2. Find a node already in the tree that is nearest to the random sample.

3. If the distance between the nearest point and random sample is greater than some threshold, find the direction of the line segment from the nearest point to the random sample and define a new leaf 20 pixels away from the nearest point in that direction as shown in equations 3 and 4.

$$direction = \frac{random - nearest}{\|random - nearest\|} \tag{3}$$

$$newleaf = nearest + direction * 20pixels \tag{4}$$

4. Check if there is a collision. For every line segment check if there is an occupied space where the robot would traverse. To implement this, a bounding box was made between the nearest point and the new leaf. Figure 4 shows this collision checking process. The left image in Figure 4 shows an instance when there would be no collision between the two points and the right image in Figure 4 shows when there would be a collision. In this case, a new point would be sampled and the process would be restarted.

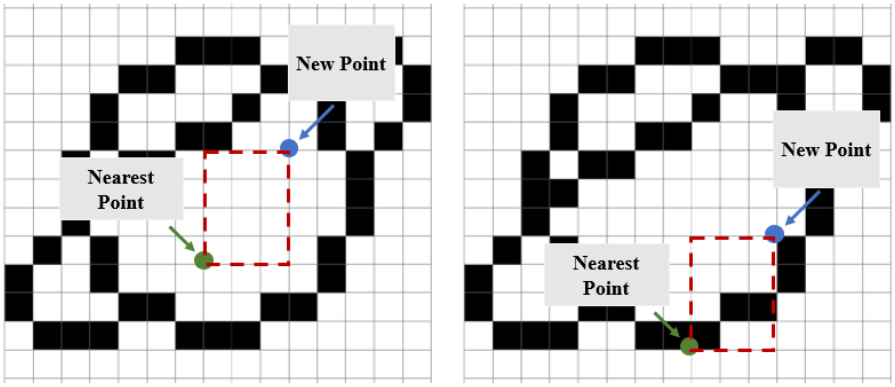


Figure 4: Collision checking. Left image shows an iteration where no collision would be detected and the right image shows when a collision would be detected.

5. Repeat this process until a node is sampled within a certain range of the goal. The image on the left side of Figure 5 shows how the tree expands as more points are sampled and added to the tree.

6. Once a node is sampled within a certain range of the goal, back propagate to define a trajectory from start to end as seen on the right side of Figure 5.

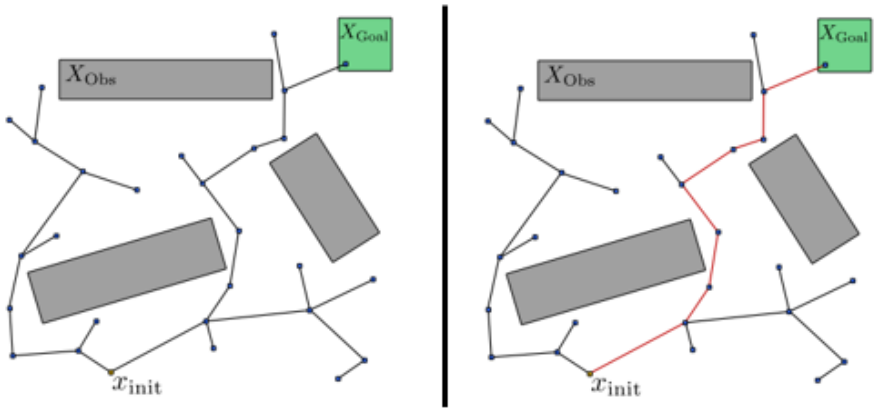


Figure 5: Tree expanding through free space of the map after many iterations (left). Trajectory acquired through back propogation once final node is found (right).

This video shows how the tree in RRT expands with each green point representing a new node in the tree until the final goal is reached!

2.1.6 Single vs Multi-Query (Author: Sharmi, Editor: Gokul)

A single-query algorithm is one that can evaluate a single path when two points are selected. On the other hand, a multi-query algorithm is one that can be used to find multiple paths from start to goal. RRT is a single-query algorithm because it only returns one path after it forms its tree-like structure. RRT does guarantee a feasible path, but at the expense of optimality.

2.2 Search-Based vs Sample-Based Algorithms (Author: Sharmi, Editor: Gokul)

There are inherent pros and cons when comparing search based and sampling based algorithms. Generally, search based algorithms guarantee an optimal solution but at the expense of runtime and memory. On the other hand, sampling based algorithms may not necessarily provide an optimal solution, but have to store less as they sample different points in the configuration space instead of storing every single pixel and its edges. The table below indicates some other differences between the two kinds of algorithms.

For the racecar the theory suggests that it may be best to utilize a search-based algorithm. The only aspect that would give the sample-based algorithm superiority is if the sample space was incredibly large and there were memory constraints. The testing environment for the racecar is not large enough to meet this constraint, at least not in Stata basement and likely not at the location of

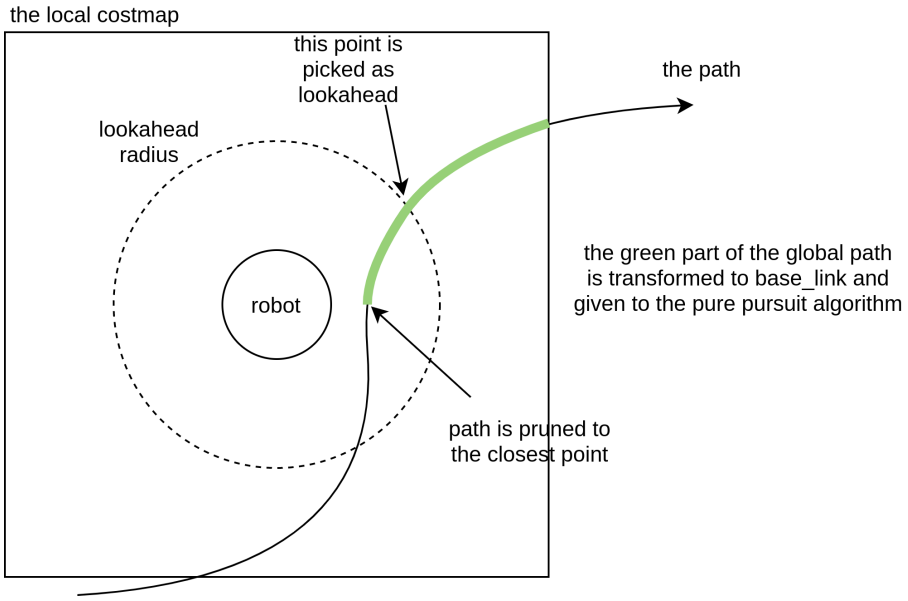
Topic	Search-Based	Sample-Based
Configuration Space	Requires discretization, can become complex	Sample space and connect samples
Time	Exponential in degrees of freedom	Not well known
Completeness	Resolution Complete	Probabilistically Complete
Asymptotic Optimality	Asymptotically optimal	Not asymptotically optimal

the final race. Furthermore, there are only 2 degrees of freedom in this case which allows the search-based algorithm to run faster than if there were more degrees of freedom, in which case a sample-based algorithm may have to be used.

3 Pure Pursuit (Author: Ariel, Editor: Angel)

3.1 Algorithm

The ideal trajectory for the robot can be mapped, however, in the real world the robot will not be able to always perfectly follow this path. This is because there will be uncertainty in the car's odometry and sensor measurements, which means that the assumed position of the robot will be an estimate of its actual position. Since the robot's actual path will not be exact, we need to account for these errors within our path following algorithm. Pure Pursuit is the solution to this problem, as it allows the robot to take in the estimate position as it follows the path and correct any deviations in real time. It is an algorithm that allows autonomous vehicles to calculate the trajectory it needs to travel to reach its target point. This algorithm generates the curvature path well even while an object is in motion.



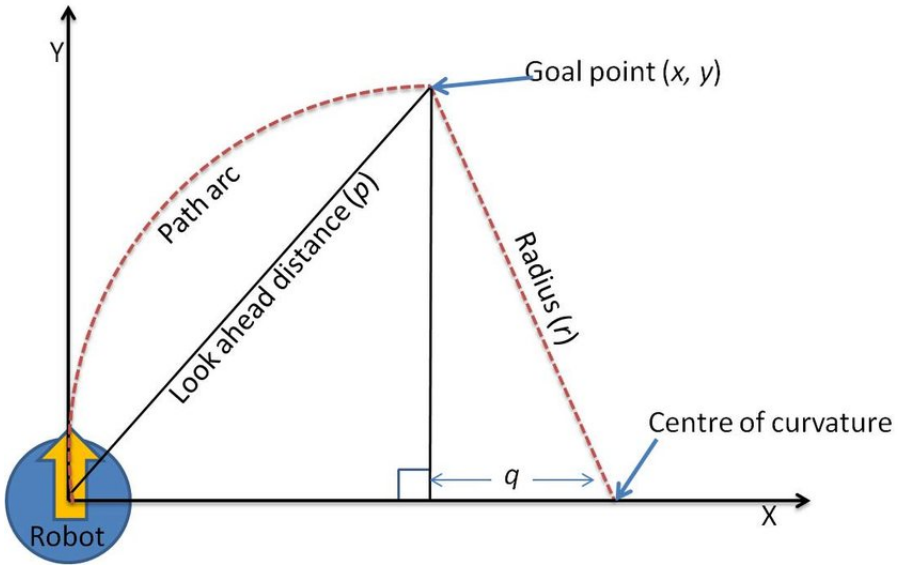
3.2 Implementation

3.2.1 Find the point on the trajectory nearest to the car

The first task was to locate the closest trajectory point to the robot's current estimate position. This was done by first creating an array of trajectory points and calculating the distances between each segment.

3.2.2 Find the goal/look ahead point

The look ahead value is the distance ahead of the robot that is scanned for an ideal goal point to drive to. Once a look ahead point is chosen, an arc is calculated from the robots current position to the goal point. This process of locating the robots position and look ahead point is done repeatedly, and allows for the robot to closely follow the planned trajectory.



3.3 Further Adjustments

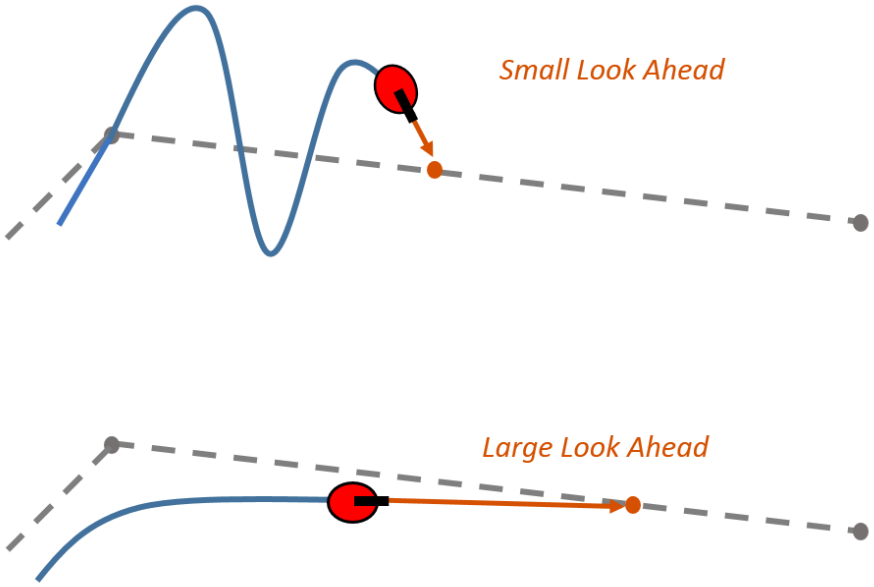
Watch out for different edge cases (for example, when a segment has two intersection points on your circle!!) These edge cases can be discovered intuitively, and should also be apparent in implementing the above steps.

3.3.1 Edge Cases

Since the look ahead distance can have multiple intersections with trajectory points, we needed to be sure that our Pure Pursuit was taking the point that followed the robots current trajectory. This meant that we wouldn't accidentally take a point that we had already passed and would make the robot go backwards. To resolve this, we would ignore trajectory points we already passed and kept the relevant points that still needed to be traversed.

3.3.2 Tuning Look Ahead

In order to create an efficient algorithm for our autonomous vehicle, we needed to have a variable look ahead distance. Having a short look ahead distance is beneficial when there are many obstacles and tight turns. Thus, short driving arcs allows the robot to navigate smoothly while maintaining the intended path. Long look ahead distances are necessary when the distance between trajectory points are greater. The arc will then be smooth and not deviate much from the ideal path. To create a variable look ahead distance we made it proportional to the distance between trajectory points, and then controlled the speed proportional to the calculated steering angle.



4 Experimental Evaluation (Author: Angel, Editor: Sharmi)

4.1 RVIZ Visualization

Once we had both implementations of the path planning algorithm and the pure pursuit modules set up, the next step was to test these implementations within the simulated Stata basement utilizing Rviz and the Racecar_Simulator package, as well as the instructor's Localization solution. To do so, for the path planning side of the equation we visualized the starting point of the path to be followed in the /planned_trajectory/start_point topic, the goal point to be reached in the /planned_trajectory/end_pose, and the path generated by our path planner function in the /planned_trajectory/path topic. On the other hand, to test that our pure pursuiter worked, we published the lookahead point that the robot will utilize to calculate its turning angle into the /goal_pt topic. The result of these functionalities can be seen below in Figure 6.

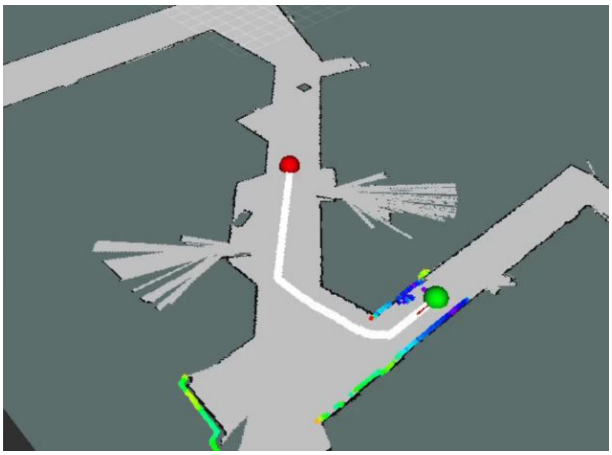


Figure 6: Rviz visualization of start, goal, and path.

4.2 Path Planner and Pure Pursuit in Simulation

The first goal we had for these modules was to determine whether our car should utilize a sample-based planning (RRT) or a search-based planning (A*) algorithm, which was done to have a quantitative (time) and qualitative (visual shortness of path) measure to inform our path forward. The second goal within this section was, after choosing an approach for our path planner, to demonstrate the simulated success of the path planner/ pure pursuiter combination in simulation, as the proof-of-concept within simulation of the pure pursuiter is key to determine whether it will work within the real life racecar framework. In the following sections, we'll see the results from the research of these two tasks.

4.2.1 Testing Plan and Metrics

For our first goal, four distinct (start,end) tuples of points were chosen to test the speed and efficacy of the paths created by both approaches of path planning algorithms, as shown in Table 2 below. For these trials, the time (in seconds) it took for each algorithm to generate the path was recorded and, utilizing the /planned_trajectory/path topic, the generated paths were qualitatively evaluated and compared. The time serves as a quantitative. For our second goal, we took the instructor's pre-built loop track and path-planner+pure pursuit track to evaluate the pure pursuit and, afterwards, how our two modules together. As a quantitative measurement for the efficacy of our modules, we took the percentage difference between the optimal track (solution) and ours. If the robot was able to perform at desired standards in this simulated environment, then theoretically it could run well within the actual racecar platform in real time. Below we see the results of this testing.

Table 2: List of tests performed for RRT vs A*

Test	Start	End	Type of Path
1	471//992	1158//998	curved1
2	461//984	1169//972	straight1
3	471//992	1608//796	curved2
4	471//992	918//389	far

4.2.2 Testing

All of the trials for the algorithm comparison were ran in the region shown below in the Stata Basement map;

The first trial shown in the two figures below are those made to do a simple turn in the bike rack area in the basement. We see that A* results in a path that more closely hugs the wall, and as such makes the car travel less distance.

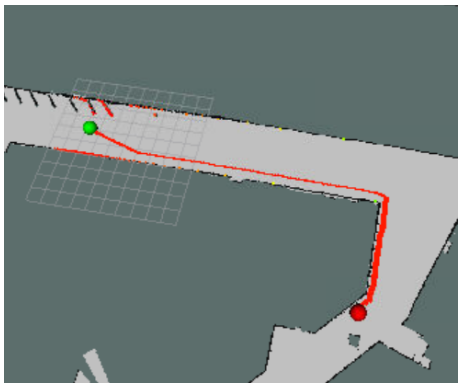


Figure 7: Path generated on curve by A*

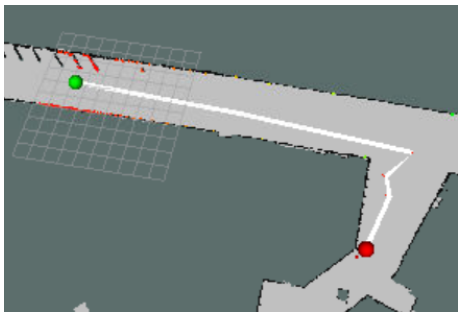


Figure 8: Path generated on curve by RRT

The second trial shown in the two figures below are those made to follow a

straight line path in the bike rack area in the basement. We see that A* results in a path that is a straight line, as expected, but the RRT method results in an unnecessary turn.

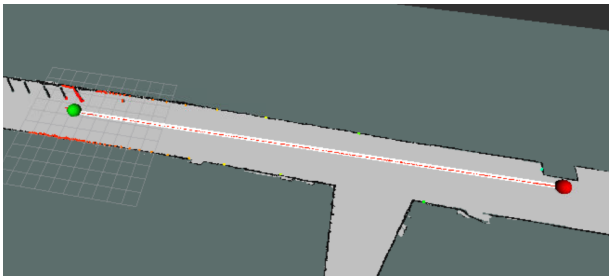


Figure 9: Path generated on straight line by A*

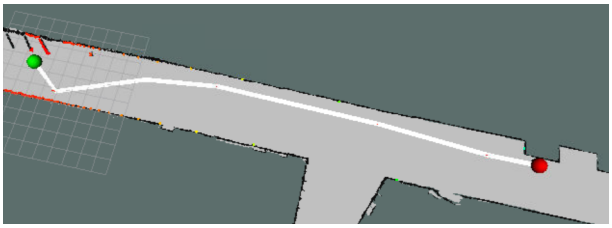


Figure 10: Path generated on straight line by RRT

The third trial shown in the two figures below is a distinct curve from that in the first trial, also starting from the bike rack area of the basement. We see that, again, A* results in a path that is more streamlined than that generated by RRT.

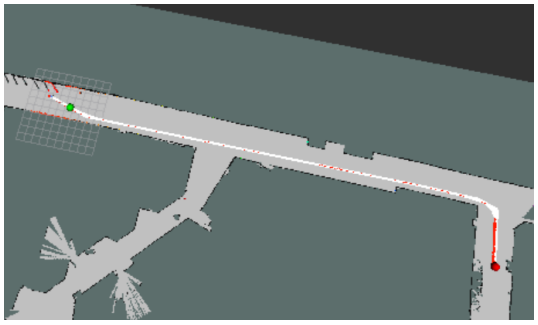


Figure 11: Path generated on curve by A*

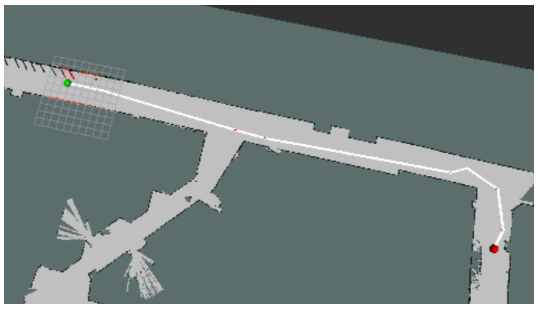


Figure 12: Path generated on curve by RRT

Finally, the last trial was chosen as a far goal for the car to get to, starting from the bike rack area of the basement. In this case, we see that RRT found a solution that's much longer than the one found by A*, choosing to go around the basement instead of through the 32-082 hallway.

Figure 13: Path generated on far by A*

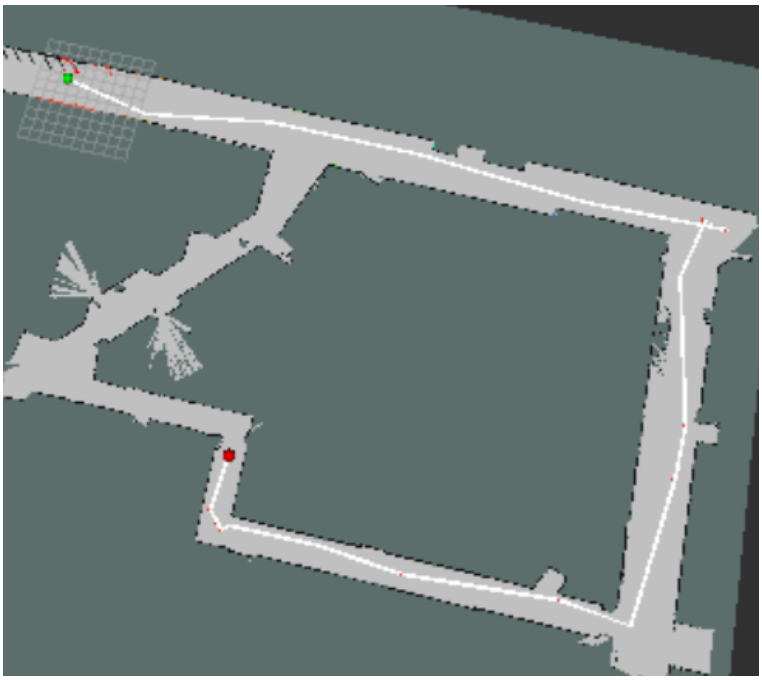


Figure 14: Path generated on far by RRT

Below on table 3, we see the resulting times it took to evaluate and publish the time for each of the methods. On all runs, A* was much faster in generating the path than RRT.

Table 3: List of tests performed for RRT vs A*

Test	RRT Time	A* Time	Type of Path
1	2.347	1158//998	curved1
2	3.72	1169//972	straight1
3	6.34	1608//796	curved2
4	82.44	918//389	far

Onto the results of our second objective, we see below from Figure 15 that our Pure Pursuiter was effectively able to follow the given track with an efficiency of around 95% , as detailed by Gradescope. Also, as seen from Figure 16, when we proceed to combine our path planner and pure pursuiter modules, the car is able to effectively get from its start point to its goal with an efficiency of 98% according to Gradescope.

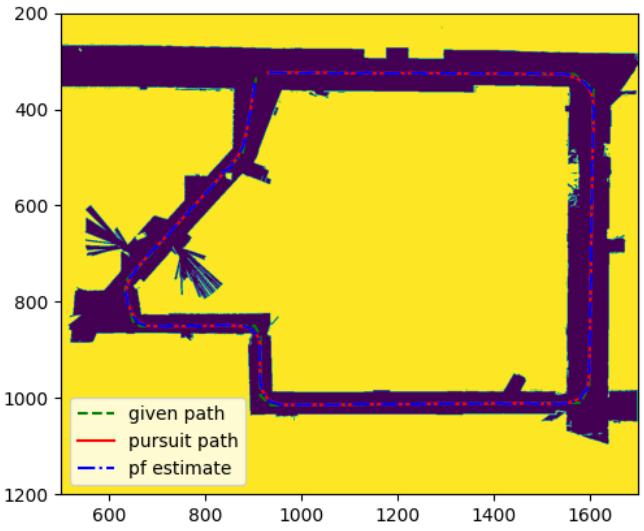


Figure 15: Pure Pursuit on Pre-built track

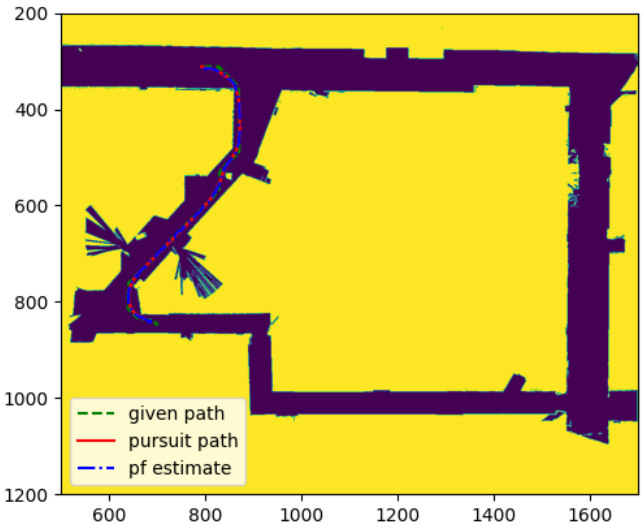


Figure 16: Path Planner and Pure Pursuit Sim vs. Solution

4.2.3 Performance Assessment

From these results, we can conclude that A* is the algorithm of choice to continue for our purposes, as it resulted in both quicker calculation times and more streamlined/efficient paths for our car to take. We can also see that both of our modules work effectively in tandem to produce the start/end path planning and following required for this lab. With this, we can move onto implementing it into our Racecar.

4.3 Path Planner and Pure Pursuit in Racecar

After obtaining these results within the simulated environment, it was time to proceed to test the implementation in the racecar platform itself in the Stata basement. Given the fact that the racecar tended to hug the walls way too closely with our A* implementation, we had to increase the dilation of the obstacles on the map for it to more safely round corners, and play around with the speed and lookahead distance parameters. With a change of topics and some debugging, we proceeded to drive our car around.

4.3.1 Testing Plan Metrics

A few trials were performed to evaluate the success of our modules in the actual robot, with one being recorded as we'll see in the following section. Given the fact that we had no ground truth for our racecar, the efficacy of our solutions was evaluated by comparing the generated path for our robot and the position published at /pf/pose/odom by the instructor's localization module. We also qualitatively evaluated the results by looking at the robot's Rviz path taken from a rosbag of the robot.

4.3.2 Testing Performance Assessment

Below, we see the results:

Pure Pursuit+Planner in Robot

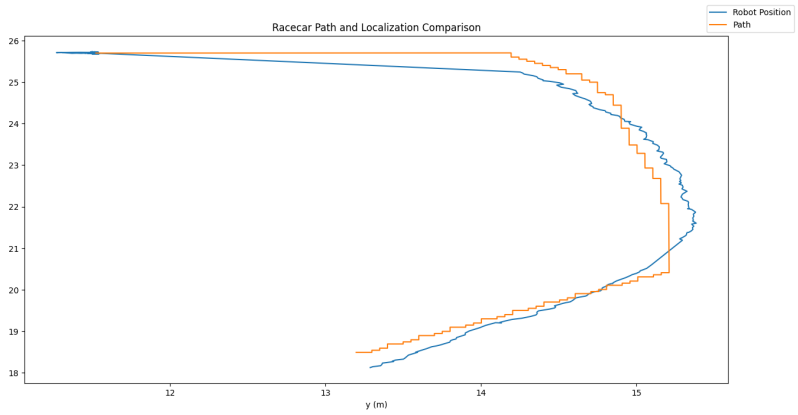


Figure 17: Car Localization Pose vs. Path

From this trial, we can see from both the actual video and the Rviz video how the robot starts at around (13.5,18) and is able to accurately enough plan and follow a trajectory to its end point, where it ends up stopping. Some of the errors seen could be due to errors in our /initalpose placement, as this was all estimation on the orientation/position of the robot within the basement, as well as from the fact that /pf/pose/odom is but an estimation of where the robot is currently located. We can also see some turning beforehand from Figure 17, which is most likely due to a larger-than-needed lookahead distance. Further work could be done in tuning the lookahead and speed parameters, but this current solution serves our purpose well.

5 Conclusion (Author: Sharmi, Editor: Gokul)

Overall this lab was an exciting addition to the rest of the functionality developed for the robot thus far. The various path planning algorithms and their relative strengths and weaknesses were exciting to learn about and implement.

We implemented two search-based path planning algorithm, A* and BFS, and one sample-based path planner, RRT. While A* and BFS comb through all the nodes and their corresponding edges to find the most optimal path, RRT randomly samples the configuration space and connects these points to map a tree in the space from which a path can be found. In our experiments we decided on using A* to path plan the trajectory for the robot because it was theoretically quicker and more optimal (although we later found that our implementation of BFS was more efficient in reality). We also implemented pure pursuit and were able to demonstrate it following a path generated by the path planning algorithms. Finally, we used localization in combination with the path planning algorithm and pure pursuit to accurately plan and navigate a path in Stata

basement!

It was exciting to see all the parts of this lab and parts of the previous lab come together to implement path planning and following. There were some challenges that came with this lab. Initially we had an issue where the path planning and following modules worked well together, but just the path following alone did not work. A key to overcoming this problem was visualizing how the pure pursuit was behaving in Rviz and this opened up a large issues with the pure pursuit algorithm that were corrected afterwards. Another problem that came up while implementing the path following on the robot was that it occasionally bumped into the wall as it followed the pure pursuit algorithm. This may have had to do with the localization code - the robot was closer to the wall than it thought. A simple fix was to increase the dilation in the path planner to create paths farther from the walls! In the end the robot performed successful path planning and following in real-time!

6 Lessons Learned

Ariel: I really appreciated how the lab was broken down into its different components because it made the final algorithm easier to understand. Splitting up pure pursuit with Angel allowed me to really understand how the look ahead distance would efficiently keep the robot on track. Then when combining Angel's component, it really came together. It was also cool to revisit search algorithms that I have learned in previous course 6 classes. Although I always implemented them virtually, it is much cooler to see the algorithm be implemented in our physical robot. The last lesson learned was that we had one of our teammates present virtually while the rest of the team presented in person. Because we practiced our presentation many times before, we were able to move through slides and transition seamlessly. This was a great reminder to be to practice presentations so that even if something unexpected happens, you can navigate around it if you know the material.

Angel: This lab, like for others, really showed me how important clear communication and collaboration is needed when parallelism is at play in a project. We were stuck debugging the pure pursuit module for some time, and given that I was one of the ones who initially created the architecture, really showed me how important it is to keep track of information on how your code works in order to be able to explain to others what might possibly be wrong with it, and for them to more effectively help you in solving any issues that could come up. It also showed me the value of visualizing/ focusing on parts of a code at a time, as it was in this way that we were able to finally pinpoint where the bug was within our pure pursuit code.

Gokul: On the communication and collaboration side of things, this lab taught me about the importance of communication regarding technical work. We were

stuck on debugging the pure pursuit module for quite some time, but what made it easier was sitting down with the team members who implemented it initially and having them explain the ins and outs of their implementation in detail. This allowed debugging to be much more focused and informed. On the technical side of things, I learned that consistency and planning are very important in the experimental testing process. In order to run path planning and following, there were a lot of moving parts that needed to be carefully launched and set up in a particular order. By the end of the lab, we thought very strategically about how we needed to navigate back-to-back runs of the path-follower based on the constraints that we had to work with, and this made it much quicker to execute test runs quickly.

Sharmi: For this lab I learned how much visualizing what is happening in my code is important to debugging. When I first implemented RRT I was combing through the code to figure out where the bug was but I couldn't see how the random sampling as taking place which made it hard to understand what the problem was in the first place. I then started visualizing the particles that were being sampled and not very long after I had successful piece of running code. I will definitely carry this lesson for the remainder of the class and the future. This lab was also difficult in terms of teamwork since a few of our members were sick unfortunately, but we were able to split the deliverable up into pieces so we could accomplish as much as possible individually before putting it together!