Lab # 6 Report: Path Planning - Using Pure Pursuit to Follow Determined Trajectory from Path Planning Methods

Team 3

Wing Lam (Nicole) Chan
Shanti Mickens
Magnus-Tryggvi Kosoko-Thoroddsen
Sheng Huang
Jessica Zhang

RSS

April 26, 2023

1 Introduction - Magnus

In Lab 6, our team implemented path-planning and path-following to enable our car to determine the shortest path between user-given start and end points and follow it autonomously. In previous labs, we implemented a pure pursuit controller to follow a cone and line from a specified look-ahead distance, as well as Monte Carlo Localization to locate the car in a user-given map from sensor and motion models using probabilistic methods. Building from those previous modules, the car must use an existing map to locate itself in the world frame, then drive to the end point using pure pursuit to follow self-generated paths based on the given map. This enables our racecar to autonomously move from point A to point B, which is imperative for the final challenge.

This report will detail which approaches we explored for both path-planning, including sampling-based and search-based methods, and path-following, and how well our implementations worked in simulation and on the racecar.

2 Technical Approach - Magnus, Shanti, Sheng, Jessica, Nicole

For the car to plan and follow paths on the fly, two sub-modules had to be implemented. First was the path-planning script, which takes in the known map of the environment and the localized position of the car (based on the Monte Carlo Localization (MCL) module from Lab 5), and outputs a generated path between a given start and end point. Second was the path-following script, which receives the generated path and outputs *AckermannDriveStamped* instructions to the car to follow said path. These two sub-modules combined allow the car to independently navigate through a static map using higher-level instructions (start and end point) to generate a path and low-level instructions (steering angle) to follow the path. This section will provide an overview of how the paths are generated, and how the paths are processed and followed properly.

2.1 Path Planning

There are two main types of path planning: sampling-based and search-based. First, we will give an overview of these two methods, and then, we will detail what we chose to use for our path planning implementation.

2.1.1 Search-based Trajectory Planning Algorithms

Search-based planning uses graph search methods to compute paths over a discrete representation of the problem. For search-based algorithms, there were three main options we considered: Dijkstra's, Greedy Breadth First Search (Greedy BFS), and A* Search. In Fig. 1 below is a comparison between these path planning algorithms.

Fig. 1. A comparison of paths generated between Dijkstra's, Greedy BFS, and A* Search. The red star is the start point, the pink cross is the goal point, and then cells with a number in it have been explored and considered in the path planning process.

Starting from the left, Dijkstra's algorithm, goes from the start point and explores out from there based on the distance from the start point. This results in it finding the shortest path, but it wastes time exploring in not-so promising directions, as can be seen by the fact that it explores over 95% of the map before finding a path. On the other hand, Greedy BFS chooses what to explore next based on the distance to the goal point. This results in it finding a path much faster than Dijkstra's, but the path it finds is not always the shortest path. Lastly, A* Search is the best of both worlds. To decide where to explore it looks at the distance from the start point and a heuristic that estimates the distance to the goal, such as euclidean distance, to decide where to explore next. This results in it being faster than Dijkstra's and it always returns a shortest path, as long as the heuristic never overestimates the distance to the goal.

2.1.2 Sampling-based Trajectory Planning Algorithms

Sampling-based trajectory planning algorithms differ from search-based methods in the way that the planning begins. While search-based trajectory planning begins planning at a given start point and attempts to reach the goal point, sampling-based trajectory planning samples the configuration space and connect samples with possible trajectories to infer the connectivity of the free space. The two most common sampling-based methods are Probaility RoadMap (PRM) and Rapidly-exploring Random Trees (RRT).

PRM first samples N points from the configuration space uniformly at random. Then, it connects all of those points or nodes with straight-line trajectories. The vertices and edges that cross into obstacles in the map (which we check through looking up the corresponding location in the given occupancy map). The remaining roadmap will then include all of the vectors and edges that will make up the final trajectory. Lastly, we will evaluate the lengths of all the possible paths through the remaining graph and choose the one with the shortest length from the start to end points. Below, Fig. 2 and 3 show a graphical approach of how the PRM algorithm works.

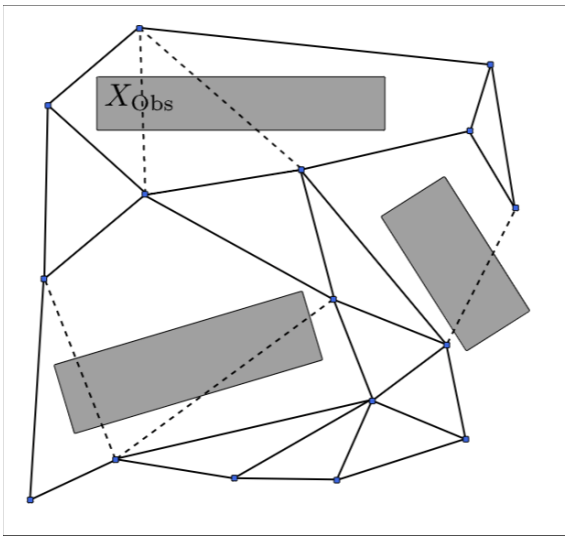

Fig. 2. All the remaining nodes are connected to each other through straight edges in the map. The dashed lines represent the eliminated edges since they overlap with obstacles in the map.

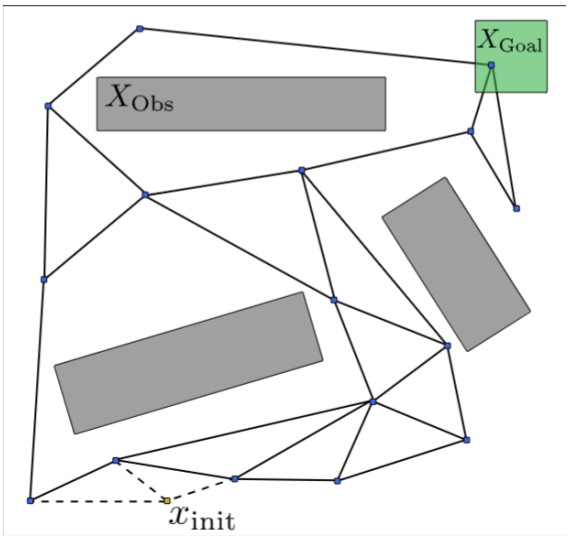

Fig. 3. The remaining valid paths are now solid black lines, with the x_{init} and x_{goal} displayed. There are multiple possible path but only one with the shortest length.

In RRT, the algorithm begins with the marked start point x_{init} and samples the next point from the configuration space uniformly at random. If the point collides with an obstacle, we would sample another point until it does not collide with an obstacle. As the new point is sampled, we connect it with an edge to the closest node that is already on the expanding trajectory. The algorithm repeatedly sample new points to grow the trajectory "tree" until a node reaches the x_{goal} location. Fig. 4 and 5 display a more graphical explanation of what RRT does.

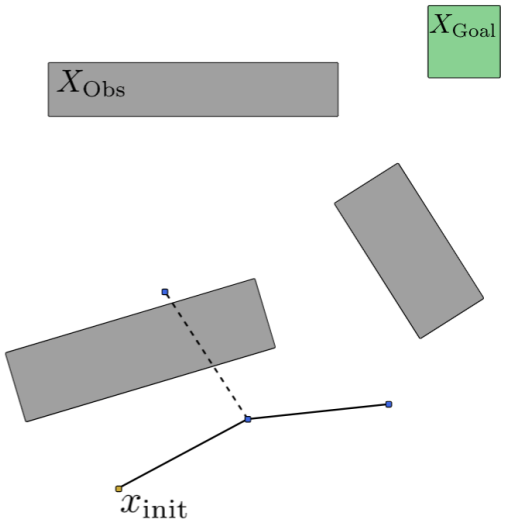

Fig. 4. The newly sampled point is behind the trajectory, with the dotted line showing the edge that overlaps with an obstacle in the map. Therefore, this edge is removed and the trajectory doesn't include this node.

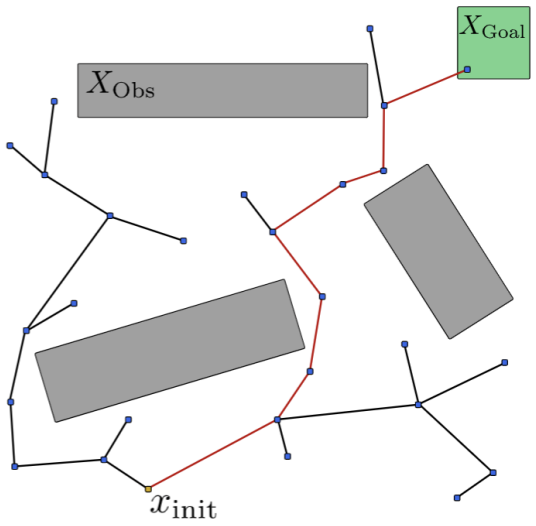

Fig. 5. The tree continues expanding and exploring the map until a node reaches the x_{goal} point, then, the completed trajectory is formed (colored in red).

From these two sampling-based methods, RRT uses less time since it does not continue exploring trajectories that will intersect with an obstacle. PRM has the drawback of blindly exploring and adding edges to the

trajectory that may not be feasible or optimal to commit to. One upside of sampling-based methods is that they do not require constructing the configuration space while search-based methods do.

2.1.3 Our Implementation

In the end, we chose to implement search-based path planning algorithms due to time constraints, but for the final challenge, we plan to explore how sampling-based path planning works compared to our current strategy. Based on the comparison of search-based methods, we decided A* Search would be the most suitable for our lab's objectives, and we also chose to stick with the grid space search domain. Through the search-based method of A* Search, the script can generate efficient paths through the map of Stata's basement. However, such generated trajectories purely consider the width of the generated path, not the actual width of the car, meaning collisions against the walls are more likely to occur. To improve the accuracy and safety of the trajectories, we performed the morphological operation of dilation on the map to create more space for the car to accurately follow the path while avoiding walls due to physical limitations. In Fig. 6 and Fig. 7, it shows the difference in the paths generated with and without dilation being used.

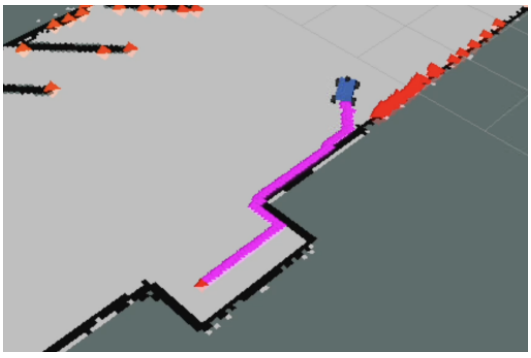

Fig. 6. A path generated in RVIZ simulation without the use of dilation on the map.

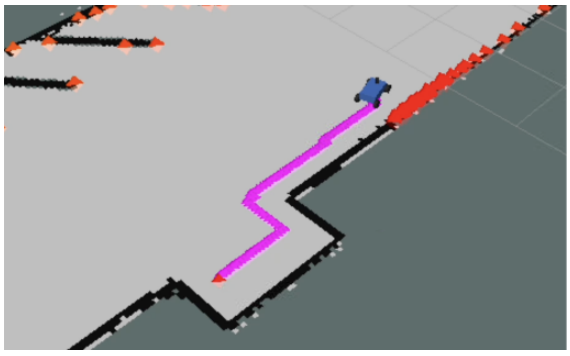

Fig. 7. A path generated in RVIZ simulation with the use of dilation on the map

2.2 Pure Pursuit Trajectory

Our pure pursuit trajectory loads in a defined trajectory from the Path Planning module, implements a pure pursuit algorithm, and steers the car in the direction of the trajectory by sending Ackermann Drive Packets to the car with a defined steering angle.

2.2.1 Getting Trajectory

First the module receives a message from the Path Planning module that contains information about the planned out path. Inside of the Pure Pursuit module, we have a $self.trajectory$ which is made from the LineTrajectory class. When we receive the message from the Path Planning module, we first decompose the pose messages down from Pose objects to a simple list of points made up of x and y coordinates by calling the $LineTrajectory.fromPoseArray()$ function. Then, the list of points can be retrieved in a list shown below:

$$self.points = [(x_1, y_1), (x_2, y_2), (x_3, y_3)...]$$

2.2.2 Localization

The Pure Pursuit module also subscribes to the Localization module that we had made in Lab 5 in order for the car to constantly update itself with the correct position inside of the Stata Basement map. We implemented this subscriber inside of the Pure Pursuit controller and it continuously listens to the Localization Module and updates itself with the robot's current position.

2.2.3 Pure Pursuit

After we have received both the points on the path and the robot's starting position, the Pure Pursuit module is ready to start calculating steering angles and sending drive messages to the robot.

There are a couple steps that we need to do before the correct steering angle can be applied to the car.

1. We calculate the closest point on the trajectory to the car. The closest point to the car is defined as the next point on the sent trajectory that has coordinate in front of the car. As we were writing a function for this, we have come to realize that sometimes, the car will "see" the point that it had just passed as being the closest point and try to drive toward that. As a solution, we have set a constant lookahead distance in front of the car such that only if the point is further along in the path and has distance greater than the lookahead will the car consider that point as the next closest point.

2. After we have retrieved the closest (x, y) of the closest point, we combine this data with the car's current position: $x, y,$ *initial angle* from the localization module to calculate the steering angle that the car needs to turn in order to stay on path.

   We first use this equation to calculate the *heading* that the car needs to be pointed at in order to make it to the next closest point.

   $$heading = math.atan2\left(\frac{\Delta y}{\Delta X}\right) - initial\ angle$$

   Next, after we have calculated heading of the car, we then use the pure pursuit formula that we had learned in class to calculate how much the car needs to turn by using this formula:

   $$turn\ angle = 2 * lookahead\ distance * \sin(heading)$$

   After this, we have to make sure that we are not publishing an angle that is larger than the max steering angle of the car, so we chose the max steering angle if the turning angle we had calculated was out of bounds.

3. The final step is to wrap up this calculated steering angle inside of an AckermannDriveStamp which will be sent to the car's navigation channel which will navigate the car as calculated.

3 Experimental Evaluation - Nicole

3.1 Testing Performance in Simulation

3.1.1 Generating Trajectories

In order to test the our trajectory planning algorithm, we visualized Stata Basement virtually and marked the start and goal points in multiple test cases. When selecting the goal points, we try to select paths of different complexities to test the algorithm more exhaustively to find a failure case. From our testing, we did not find any. Below, Fig. 8 and Fig. 9 are multiple snapshots of the generated path as well as markers indicating the start and end positions of each path.

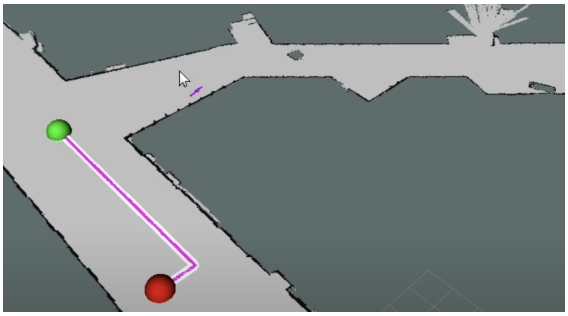

Fig. 8. The magenta line shows the car's planned trajectory using A* Search algorithm, with the red sphere being the start point and the green sphere being the end point.

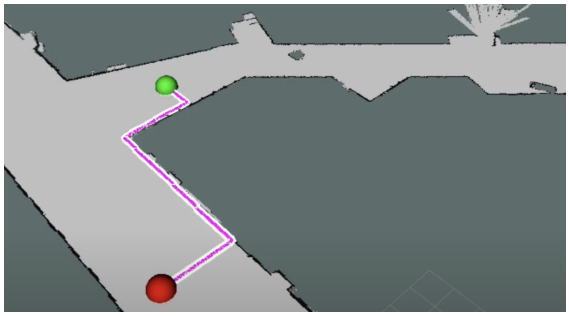

Fig. 9. The magenta now shows the trajectory planning around the hallway corner and not entering the greyed out spaces that represent the walls in the basement map.

3.1.2 Navigating the Trajectory

After testing the algorithm's ability to generate valid trajectories, we then combined it with our pure pursuit implementation for the car to follow the trajectory. A crucial part of the lab was using the localization code from Lab 5 for the car to estimate its pose in the environment. We visualized this pose estimate using red arrow markers as in the previous lab. In Fig. 10, we can see that the car is able to accurately localize and follow the trajectory with minimal error qualitatively.

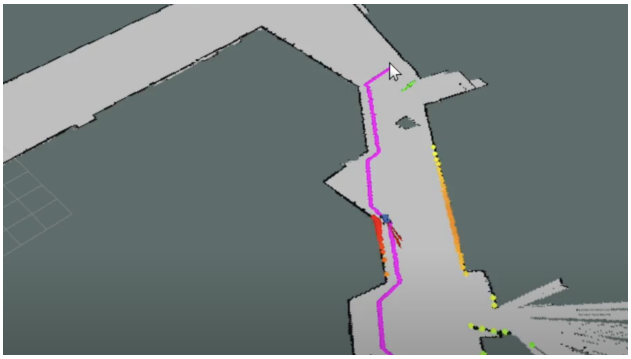

Fig. 10. The car accurately follows the magenta trajectory along the wall and never gets too close to the wall.

3.2 Testing Procedure on Racecar

When implementing this code to the racecar, there was an option to use the staff solution of the localization code from Lab 5 instead of our own localization code. After some experimentation and qualitative checks on the visualization software RVIZ, we decided to use our own localization program due to some inconsistencies between the virtual and real environment when localizing. After that, we then tested the car's ability to plan a simple and complex trajectory. In the simple case, the car was placed in the straight and wide hallway of the Stata Basement (with the start point being the origin of the map [0, 0]) and told to travel to a goal position 10 meters away in the positive x direction (with the coordinate being [10, 0]). We specifically chose these start and end points because we would predict that the car would plan a straight trajectory in the x-direction between those two points, which minimizes the car's planning algorithm's points of failure.

In the more complex case, we had the car travel around the corner and make a right turn. The car moved from [2, 5.5] to [3, 0]. Since the trajectory and start and end points are more complex, we were not able to come up with a robust and accurate way to calculate and graph the position error from the planned trajectory. Instead, we qualitatively evaluated the car's ability to follow the path.

3.3 Racecar Trajectory Following Performance

In the simple case, we graphed the error of the car's estimated pose with the ground truth of the trajectory. Since the car is expected to move strictly in the x-direction, we can simply track the position's y-error over time. Fig. 11 shows the car following the trajectory with minimal error (the plot on the right side plots the difference between the ground truth and the estimated y-error of the car's position over time). There is around 5% error as the car oscillates along the straight line trajectory.

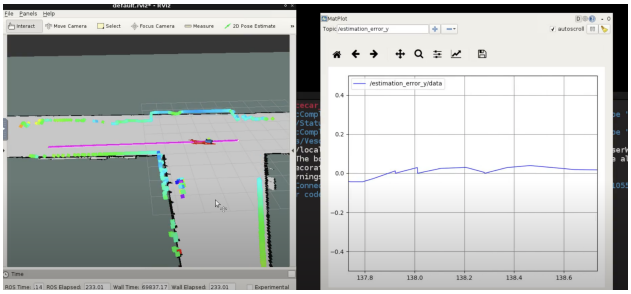

Fig. 11. The car successfully plans a safe trajectory around the corner bend, with the path marked in magenta. The red arrows indicate the car's estimated pose. The plot of the car's relative y-error is displayed on the right.

In the more complex case, the car planned the trajectory around the right corner turn. Fig. 12 and Fig. 13 are snapshots of the car planning a trajectory around the corner and following the trajectory at different times. We see that there is some deviation as the car makes the right turn, mainly that the car turned too "wide." However, since the car was able to make it to the goal position with no collisions while following the general shape of the planned trajectory, we considered this trial a success.

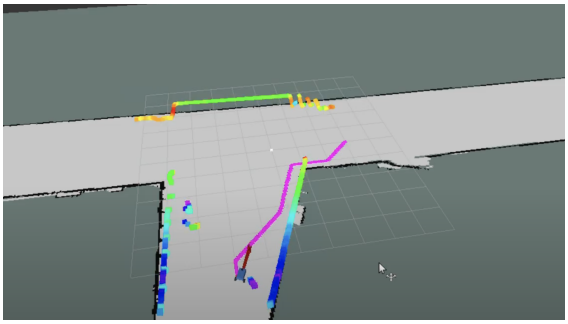

Fig. 12. The car successfully plans a safe trajectory around the corner bend, with the path marked in magenta. The red arrows indicate the car's estimated pose.

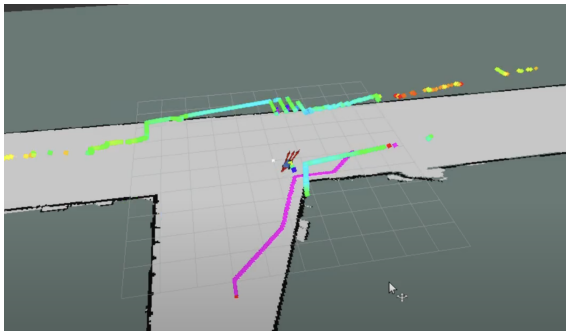

Fig. 13. The car deviates from the planned path because the long featureless hallway in the basement is interfering with the car's ability to localize.

4 Conclusion - Magnus

Over the course of Lab 6, we successfully implemented a path-planning and path-following algorithm to navigate through a map of Stata's basement in simulation and the real world. We learned about how different path-planning algorithms have different strengths and weaknesses, and how A* Search satisfied our criteria for speed and accuracy. However, the path planning and following can be improved on a few fronts. Since we used our Lab 5 solution for localization rather than the staff's solution, there may be lag time during processing that causes the LIDAR scan to oscillate and not match up with the map, or the localization may be imprecise enough to cause inaccuracies and therefore oscillations. Additionally, the pure pursuit controller's oscillations could either be attributed to the speed of the path-following script or the quality of

the controller itself.

These sources of error are potential points for future work, which we will have to tackle for the final challenge when racing along the Johnson Track and driving through the cardboard city. Our goal is to smoothly and safely drive in each environment as fast as possible.

5 Lessons Learned

Below we each included our own self-reflection on the technical, communication, and collaboration lessons we learned in the course of this lab.

5.1 Nicole

In this lab, I learned about different path planning methods and the drawbacks of each. It was interesting to finally have the car have more autonomy over deciding what path to follow instead of following a feature of the environment like line and wall following in previous labs.

5.2 Shanti

In Lab 6, I learned about different path-planning methods, both sampling and search-based. I also researched more into various search-based methods, such as Dijkstra's, Greedy BFS, and A*, and decided to implement A* search algorithm for our path planning. I also learned about how when you have a larger team it can be very helpful to divide up the work even between labs when deadlines overlap to ensure things are completed on time.

5.3 Magnus

In Lab 6, I learned how path-planning algorithms process the environment and how well each method can work in our lab setting. Enabling the car to handle low-level decisions through our past modules and follow our high-level commands makes me feel like the progress we've made along this semester has amounted to a much smarter car. As for communication, it was interesting to see how diagrams for path-planning can almost completely explain the problem and solution, while text explanations can easily lose me.

5.4 Sheng

It was interesting to implement the pure pursuit controller and think about the math behind how to calculate the steering angle. It was also pretty cool to see how all of the modules all communciate with each other about different parts of the car to make the entire car realize its surrounding.

5.5 Jessica

In this lab, I learned about how different search algorithms can be used for path planning, and the benefits and drawbacks of each method. Figuring out how to integrate this lab with the last lab(localization) was also an interesting challenge that felt very rewarding when everything finally worked.