Lab Report #6: Path Planning: Using RRT, A*, and Pure Pursuit to Autonomously Drive a Racecar Around A Known Environment

Team #5

Waly Ndiaye
Cole Paulin
Preston Tower
Erina Yamaguchi

6.4200/16.405: RSS

April 26, 2023

1 Introduction

1.1 Overview and Motivations

This lab dealt with designing path planning and path tracking algorithms to be embedded into our team's racecar. The purpose of designing the path planning algorithms was for our racecar to be able to find the fastest path through a known environment, in our case the Stata Building's basement. The purpose of designing pure pursuit, our path tracking algorithm, was to enable our car to follow a path we created or another, already-provided, path with accuracy and safety being kept in mind. This lab builds on the preceding labs during which we designed a wall follower, safety controller, line follower, parking controller, and particle filter. The lab serves as the penultimate step before the final challenge. It will allow our racecar to drive more autonomously as it plans and optimizes its own routes from a start location to an end location. This lab's added functionality can also be paired with previous labs, such as line following, for running our racecar on the Johnson Track for the Final Challenge. In order to evaluate the success of these systems, we set our goals to be minimizing the runtime of A*, ensuring RRT creates optimal solutions, and minimizing the distance error from our trajectory during pure pursuit. The deliverables for

Lab 6 were split up into four sections: designing the RRT algorithm, designing the A* algorithm, designing the pure pursuit algorithm, and testing code in simulation.

1.2 Goals and Implementation

For the path planner, we started developing our RRT algorithm. Since RRT is a sample-based algorithm that can run quickly but doesn't necessarily produce the most optimal solution, we aimed to fine-tune our approach to get better final paths. Alongside this, we worked on implementing a search-based algorithm in the form of A*. A* guarantees an optimal solution, but we set a goal of minimizing the amount of time it takes since A* can run for a long time. Limiting run-time will make it easier to test code on our car going forward.

For the path follower, we worked on a pure pursuit method using a lookahead circle. To find a goal position, a position on the trajectory to be driven towards, we calculated where the lookahead circle intersected with our planned trajectory. The steering angle of the car was changed accordingly. The primary goal for this stage of the lab was to limit the distance error from our trajectory, which was calculated by finding the robot's distance from the line segment it was currently tracking. Limiting distance error will provide better functionality as we complete tasks such as the Johnson Race Track challenge.

In addition, we had many ROS nodes we dealt with and had to figure out how to configure them to work with our implementation. In general, the robot subscribed to the map, odometry data, and goal location as inputs for path planning. For pure pursuit, the robot was subscribed to the odometry data and the current trajectory. Planned trajectories, drive commands, and distance error publishers were all implemented.

During simulation testing, we collected data to analyze our path planning/following performance. With this information, we will be able to make educated decisions as we deploy our code onto our racecar and prepare for the final challenge's many parts.

2 Technical Approach

For autonomously driving the racecar to a desired position, there are two core steps. The first is to create a path from the starting position to the final desired position. Next, is to implement a control function to follow the desired trajectory. For Path Planning, two algorithms were tested, Rapidly-exploring Random Tree (RRT) and A*. Then, the produced trajectories were followed using pure pursuit.

2.1 Path Planning Search Algorithms

For path planning, generally, there are two ways to approach the problem. The first, is a sample based search approach. The main idea for a sample based approach is that the algorithm randomly selects parts of the map to explore and using the explored locations, creates a path to the goal state. This approach is used by the RRT algorithm. The state space search approach, used by A*, however, discretizes the map into nodes and visits all the nodes in increasing order of increasing cost functions until a path to the goal is found.
Both approaches have their benefits and drawbacks. For state-space search like A*, as long as a path exists, the approach guarantees completeness and optimality. However, it takes up memory because of the need to store all nodes in memory. In comparison, the sample based approach like RRT does not guarantee optimality and could take time as it samples unnecessary nodes that may be explored. However, it does not require as much memory because fewer states need to be explored.

2.1.1 Preparing the Environment

2.1.1.1 Modifying the Map

The robot must have a map of the world it is in to be able to localize and plan its path. Our tests are in the basement of the Stata building, which is why we are using the map of Stata, given in Figure 1. However, there are breaks or thinner outlines of the wall in some locations. Additionally, to account for the racecar's width, the provided map was eroded and dilated. The final map that was used for path planning is the pixel data from Figure 2.

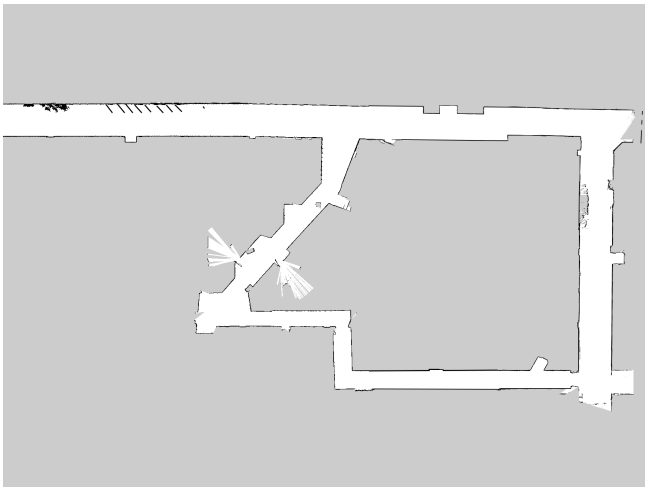


Figure 1: Original Stata Map image

Figure 2: Stata map after dilation

2.1.1.2 Real World and Pixel Conversions

Another key component for this lab is the conversion from the real world frame to the map frame. Since the map is given by an image, given a position in the world frame, it is necessary to represent that position in the pixel frame, and in the opposite direction. Taking advantage of the Occupancy Grid Subscriber in ROS, we could use information about the rotation and position with respect to the world frame to compute the rigid transformations. From there, the points would have had to be scaled by the resolution.

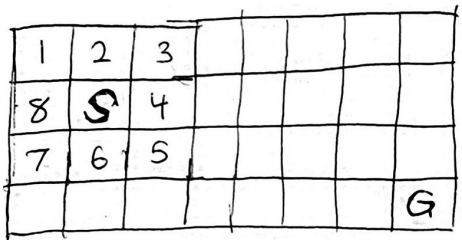
2.1.2 Implementing A*

Our implementation of the A* path planning algorithm used much of the same graph structuring code as the RRT algorithm. It also gave a similar output in that it produces a path consisting of an ordered sequence of segments. The way it differs from RRT is that while RRT is a sampling-based planning method, A* is a search-based planning method. This difference is expected to make A* globally optimal at the expense of computation time and space.

The A* path-planning algorithm is guaranteed to return the shortest path between two points, in finite time, if one exists. It is a variation of best-first search which uses the straight-line distance to the goal plus the distance traveled thus far as the heuristic by which to compare path options. This heuristic is always equal to, or less than, the actual shortest path distance to the goal, so it will never have the opportunity to complete a sub-optimal path.

Our implementation of A* segmented its search space using the pixels in the

dilated occupancy map described above. starting at the start pixel, it adds the paths to the 8 neighboring pixels to a queue (as long as they are free space). From these paths, it selects the one which has the lowest heuristic value and adds to the queue the paths to its neighboring pixels which haven't yet been visited. The algorithm always expands the path in the queue with the smallest heuristic until it intersects the goal pixel. The first path which finds the goal is returned as the shortest path. If the algorithm explores all available space without finding the goal, it reports that there is no path to the goal.



Above is an example of a simple grid space like the map our car explores, only smaller. S is the starting pixel and G is the goal pixel. The algorithm begins by expanding paths to each of the neighboring pixels, numbered 1-8. It adds these paths to the queue, along with their heuristic values, which are shown below:

Queue:

Path	Path Length	Dist to Goal	Heuristic Value
S→1	1.41	7.62	9.03
S→2	1.0	6.71	7.71
S→3	1.41	5.83	7.24
S→4	1.0	5.39	6.39
S→5	1.41	5.09	6.50
S→6	1.0	6.08	7.08
S→7	1.41	7.07	8.48
S→8	1.0	7.28	8.28

The algorithm then repeats the process using the end of the path which has the smallest heuristic value in the queue, which in this case would be from S to the neighbor labeled 4. Interestingly, pixel 4 isn't the neighbor which is closest to the goal, pixel 5 is. But pixel 4 makes the most direct progress toward the goal because the vector from S to 4 is more aligned with the direction to the goal than is the vector from S to 5. So it turns out that the shortest path from S to G starts by visiting pixel 4.

2.1.3 A* Practical Considerations

We completed the rest of this lab using RRT in favor of A* because we developed it earlier and we needed to budget our limited available time. If we were to complete path following using an A* path, we would need to make some

additional considerations.

First, when A* is applied to the dilated occupancy map generated earlier, it produces the shortest possible path between points within the free space. This will cause it to tightly hug corners. Because of this, it is imperative that the occupancy map be dilated to provide sufficient clearance that the path follower following the A* path does not collide with the wall.

Next, A* takes no account of the car's minimum turning radius, nor the car's heading at any time, so space which is defined on the occupancy map to be reachable may not actually be reachable from the car's start pose, especially points which are very near to the car. This problem cannot be addressed without deviating completely from A* and implementing some consideration of the car's dynamics, possibly via Dubins Curves. We did not pursue this option, rather we kept this behavior in mind as a limitation. We can avoid this problem by placing the car at the start point facing in the correct direction to follow the path. This way, there will be no major initial turns to align with the path, and the car can closely follow most paths from that point onward.

2.1.4 Implementing RRT

Our implementation of RRT in simulation finds the desired path. However, because we do not use techniques like RRT* to produce straighter and smoother lines, some of the paths that are produced are jagged. Additionally, due to the nature of this implementation method being a sampling method, when facing goal points that are further away tended to take a longer time to compute. In conclusion, in a smaller environment like the Stata basement, the memory cost of keeping track of all of the states is a faster and more optimal approach, especially due to the lack of obstacles in the environment.
With a completed A* algortithm, it would be beneficial to determine the time it takes for each of the algorithm to find a path, given the same initial and ending states. Additionally, another good comparison metric for comparing the path planning algorithms would be the total distance traveled.

2.2 Pure Pursuit

The purpose of the Pure Pursuit controller is to follow a trajectory that is returned by our Path Planning algorithm. This process has a similar overall goal as our Wall Follower Controller. The robot determines its location with respect to the path, and continuously adjusts its steering so that it stays very close to the path, while avoiding collisions. The Pure Pursuit controller is a great choice for trajectory following because it is robust to edge cases (corners, obstacles, etc.), and it is straightforward to implement.

2.2.1 Subscribers and Publishers

The Pure Pursuit controller works by subscribing to our trajectory and our odometry data. The trajectory is needed because it will contain the positional values along the trajectory. The odometry data keeps track of the robot's pose, which is necessary to determine our position on the global map, and helps determine the trajectory's position in the robot's frame.
The robot will publish to three different topics: driving command, goal point position, and the positional error. The drive command is what controls the steering and speed of the robot. The goal point is published so that the user can visualize our target throughout the path. Finally, the positional error between the path and the robot is published so that our team could perform experimental analysis.

2.2.2 Determine Path Segment Closest to Robot

The trajectory that the robot Subscribes to is made up of multiple path points. The first goal of Pure Pursuit is to determine the segment of the path that it is closest to. With this information, we can start to find our target point.
Finding the distance between the robot and each segment is computationally expensive, so we used NumPy arrays to compute the distance to each segment simultaneously. Each distance was found using simple positional vectors and a distance formula. Once we found the $dist_{array}$ of all the distances, we can run np.argmin($dist_{array}$) to find the segment that the robot is closest to.

2.2.3 Determining Goal Point

The Goal Point is important because it is the point that the robot will drive towards. Fist, the algorithm took the segment and a lookahead circle into account to find their intersection points. The lookahead circle is a circle around the robot that the goal point may lay on. The line segment can either intersect the lookahead circle in two places, one place, or miss the circle entirely. If the segment intersects twice, then we determine that our goal point is one of those intersections. To narrow this down to one point, we keep the intersection that is closest to the end of the segment. This ensures that the robot will continue along the path. In the cases where the circle doesn't intersect the segment twice, the algorithm will return None, and skip to our next segment to search for a valid goal point. Searching multiple segments aids with turning stability, creating a more robust system. The algorithm checks the current segment, and the two following segments. The farthest segment to return a goal point will determine our final goal point.

2.2.4 Drive the Robot

Now the robot is going to drive towards the goal point. The steering_angle can be determined by computing $arctan(delta_y/delta_x)$, where $delta_y$ and $delta_x$ are the y and x distances between the robot and the goal point. We can find

these distances easily by finding the goal point pose in the robot's reference frame. After computing the steering angle, the angle is published to the robot's steering command.
If the absolute distance between the robot position and the final end goal, then we set the speed and steering angle of the robot to 0. This indicates that the robot has reached its final goal.

3 Experimental Evaluation

3.1 Pure Pursuit Simulation Performance

For the purpose of testing, the independent variables were the lookahead distance and the speed of the car. The dependent variable was the distance error from the trajectory. The first step in testing the pure pursuit performance was running the algorithm for an already existing path, loop2, which was provided by the starting code for the lab. After seeing the racecar being able to adhere to the trajectory well in RVIZ simulation by the naked eye, we moved on to quantitative testing methods.

In most cases, our racecar followed straight line paths with high accuracy, approaching 0 distance error, so we sought to assess our algorithm's performance when it came to turning on corners. Many of the issues dealt with during the implementation of pure pursuit were due to not having the right steering angle for our car. This meant that on turns, the racecar would often miss the trajectory, lose track of the point it was following, and loop in circles or other errant behaviors. For consistency, when trying different lookaheads and speeds, our distance error was tested while always turning the same corner, which was the first right corner in the loop2 trajectory. This turn was composed of three line segments, leading to two sharp edges in the trajectory the racecar was expected to follow.

At a lookahead distance of 1 meter, the racecar while turning was able to stay consistent within .2 meters of the trajectory for speeds between 2 and 7 m/s. This behavior is shown in Figure (1).

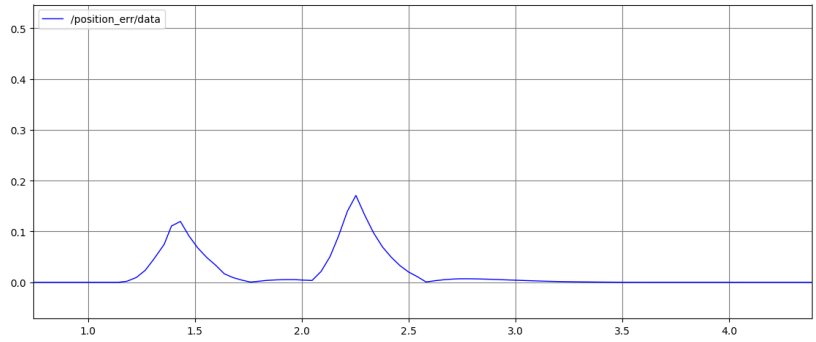


Figure 3: When running at a speed of 3 m/s and a lookahead distance of 1 meter, the car had a peak distance error around .2 meters

At 2 meters of lookahead distance, the distance error was consistently around .6 meters when turning on the two edges. This behavior is shown in Figure (2).

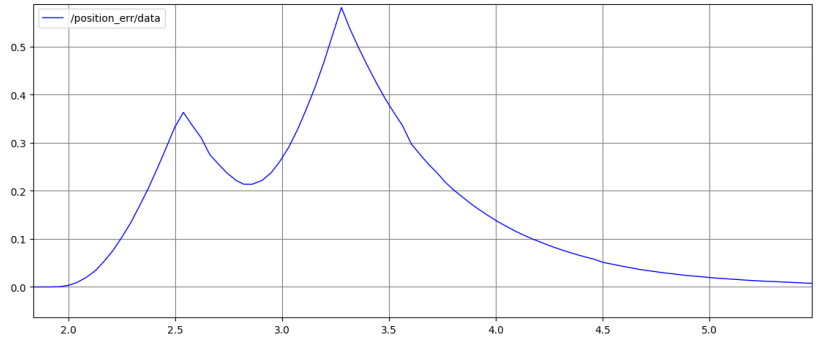


Figure 4: When running at a speed of 3 m/s and a lookahead distance of 2 meter, the car had a peak distance error around .6 meters

At 3 meters, the distance error hovered over 1 meter. At a rather large lookahead distance of 5 meters, the racecar would be over 2 meters off the trajectory, meaning that it most likely was turning too early once it saw the curve.
The general trend was that as the lookahead distances increased, the distance error increased accordingly and proportionally. This relationship (shown in Figure (3)), based on simulation testing, seemed to be a linear relationship when plotted for a car speed of 3 m/s, which is a realistic speed for the racecar to be driving at.

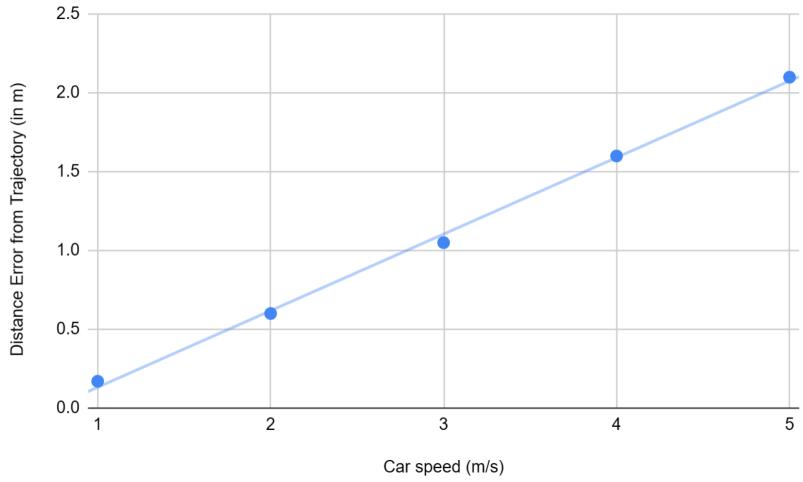


Figure 5: Testing different lookahead distances at the same speed led to a linear relationship between the lookahead distances and distance error

Based on these results, we concluded that a lookahead distance of around 1 meter would be ideal for deployment on the racecar, due to its consistent performance across differing speeds. Since increasing the lookahead distance led to worse performance in terms of distance error, testing distances below 1 meter to find an optimal value will be a new goal going forward.

3.2 Plans for Deployment

Due to some difficulties in implementing the algorithms for path planning and tracking, we were unable to fully deploy our code suite on the racecar by this report's due date. As we move from simulation to racecar, the aim is to use the error publisher once more to determine appropriate speeds and look ahead distances for the racecar based on our simulation conclusions. In the simulator, some speeds both higher and lower than the maximum car speed were tested to ensure that the pure pursuit algorithm was robust and able to handle the car moving slowly and speedily. For the racecar, the aim is to set the driving speed as close to the maximum speed while not jeopardizing safety and performance. Based on the lab's performance on the racecar, our team might elect to make edits to our parking controller. The plan for these edits was for the safety controller to edit the steering angle while attempting to minimize the distance error from the trajectory. Currently, the safety controller looks at a range of LaserScan values within a certain field of view. If enough points are within a stopping distance, the racecar slows down to a halt. Adding the steering functionality would ensure that if the racecar has the possibility of avoiding an

obstacle, it will do so and utilize coming to a stop as a last resort.

4 Conclusion/Lessons Learned

4.1 Conclusion

At this point in our autonomous process, the robot can predict its position through localization, plan and follow an optimal path from a start to end state, and can perform object detection. The ability to plan an optimal path efficiently is crucial for autonomous capabilities. In real world applications, these types of algorithms are what direction apps use to find an optimal route between two locations. Moving forward, our team will determine which of our path finding algorithms will return the best trajectory for different maps.
Using a Pure Pursuit controller gave the robot a respectable method of following a path. This is crucial because an excellent trajectory will not matter if the robot cannot follow it well. This would be similar to an autonomous car driving in the wrong lane or taking erroneous turns. While the current controller performs well, there is certainly room for improvement. The algorithm could possibly implement a variable lookahead distance, depending on how striaght the path is. Also, the drift of the physical robot will adjust the performance.
The current path planning and pure pursuit combination work well for the scope of this lab, but there are adjustments that can be made in the future to increase the overall robustness for new maps.

4.2 Lessons Learned

4.2.1 Erina

The hardest part for this lab was the map to pixel and pixel to map conversions. While all of the labs, especially the last one, provided me with the confidence and knowledge on how to apply conversions, when visualizing poses, they would not be aligned to the right positions. Debugging by creating markers was most helpful to visually see how the conversions lined up to the map frame. However, it was frustrating to debug some of the conversions because at first glance, it seemed as if the points were being converted correctly. Specifically, points like the starting pose was identified correctly. However, when looking at the generated random nodes, for example, I would see that it would take points that were outside of the bounds. In the beginning, I thought RRT was running exceptionally long because of how I computed functions like finding the nearest node. Next time, I should take these labs in smaller steps, confirming that the conversions are done correctly before implementing any of the path planning algorithms.

4.2.2 Cole

I wrote and implemented the Pure Pursuit portion of the lab, and I had a very difficult time determining the steering angle of the robot. At first, I was finding my driving angle using only global coordinates. I found the global x and y distances from the goal point to the robot, and drove at a steering angle of $arctan(x/y)$. I thought that this was reasonable, but issues arose when the robot was turning. This is because the robot's heading affects the steering angle, as well. To solve this, we used quaternions and transforms to determine the goal point's position in the robot's frame. Now, the heading is already accounted for, and we can simply extract the new x and y differences. Using generalized coordinates wherever possible can make the system more robust, so I will surely be implementing this more often in labs.

4.2.3 Waly

Personally, I found this lab very technically challenging and often found myself confused about the coding components. It often was a bit difficult figuring out quite where to start or what I could currently be helping with. In retrospect, reading more resources and gaining a better background understanding of pure pursuit as an algorithm would have helped me better contribute to the implementation of the path following. As we divide up work going into the final challenge, I aim to review the content for my parts more thoroughly and reach out during Office Hours more frequently to better help my team in regards to coding.

4.2.4 Preston

Every lab so far has been more and more difficult to debug. I expect this is because we are designing more complex functionality, and more complexity means more issues. Thankfully this is the second-to-last lab. I didn't expect to encounter as much difficulty as we did with path following, but I take it as a lesson that more simultaneously interacting elements will generate more unforeseen issues that require much more time to sort out. Going forward to the final assessment, I hope to spend more time working together in person with my team and trying to integrate with the car early so we can sort out integration bugs while we develop.

5 Addendum

In order to not clutter the report, we listed out by team member which sections they primarily authored. The listing is recorded below.

5.1 Waly

- 1.1) Overview

- 1.2) Goals and Implementations
- 3.1) Pure Pursuit Simulation Performance
- 3.2) Plans for Deployment
- 4.2.3) Waly

5.2 Cole

-

5.3 Erina

- 2) Technical Approach
- 2.1) Path Planning Search Algorithm
- 2.1.1.1) Modifying the Map
- 2.1.1.2)Real World and Pixel Conversions
- 2.1.4) Implementing RRT
- 4.2.1) Erina

5.4 Preston

- 2.1.2 Implementing A*
- 2.1.3 A* Practical Considerations
- 4.2.4 Preston