Lab 6 Report: Path Planning

Team 13: Five Fellas and a Ferrari

Allen Baranov
James Morin
Jian Ming Chen (Editor)
Remeyn Mechura
Ryan Xiao

6.4200/16.405, Robotics: Science and Systems
April 26, 2024

1 Introduction

[*Remeyn Mechura*]
A major milestone in any autonomous vehicle design is path planning. This is a large part of what makes a programmed vehicle "autonomous". In the previous lab, we took map information and implemented a localization algorithm to allow our car to know its relative position in a known region. In this lab, we again take map data, including fixed obstructions, walls, etc. and design a path planning algorithm that connects a number of points on the map into a path, or a trajectory. This trajectory optimizes distance travelled, as such it finds the shortest route the car can take between the set of points to reach its goal pose. Once we built a path, we need to be able to follow it. To this end, we decided to go with the tried and true pure pursuit method. This simple method allows our car to find a point on the path and follow it, ultimately converging with it. The steps we followed and will outline in this report to accomplish our goals are:

1. Test various path finding algorithms and select one that meets our needs.
2. Build and fine tune a pure pursuit algorithm that is responsive and smooth.
3. Combine the two and analyze their performance in simulation.
4. Finally, get everything onto the car and running.

2 Technical Approach

There were two distinct parts to implement for this lab: path planning and trajectory following. There exist multiple methods for the robot to achieve these tasks. In this section, we will explore these different methods and the trade-offs associated with each. We will also explain our reasons for choosing specific design choices in our implementation.

2.1 Path Planning

2.1.1 Processing the Map

[*Jian Ming Chen*]
For our path planning algorithms, we received an OccupancyGrid input, which represents a map discretized into a high-resolution grid of cells. Each cell is assigned a number between 0-100 equal to the probability that an obstacle is present in that cell. We simplified the representation to a binary case of either a open cell or blocked cell by updating any cell with a non-zero value to 100 (blocked) and keeping any cell with a zero value as 0 (open).

Another concern with path planning is that the most optimal path can tend to hug walls and closely cut corners. As our real life robot is not a point mass as assumed by the path planning algorithms, we used binary dilation on the map to enlarge the obstacle boundaries beyond their original sizes. This provided additional leeway for the robot in case the algorithm generated paths too close to the walls.

2.1.2 Path-finding Algorithms

Search Based - A* Graph Search

[*Allen Baranov*]
A* is a graph algorithm that finds the best path between any two nodes of a graph efficiently. This approach is achieved in a BFS-like way, judging a specific path by how close it gets to the goal. It is implemented by adding a cost function representing the distance already travelled and a heuristic function estimating the remaining distance to the goal. This is captured in the equation below:

$$c(n) = f(n) + g(n)$$

Where $c(n)$ is the total cost function, $f(n)$ is the distance already travelled on the path, and $g(n)$ is the estimate for remaining distance to the goal. The path with the lowest total cost is chosen as the next path to look at.

We use an example of A* to illustrate its speedier performance versus BFS. Consider the coordinate grid as a graph. We start at the origin and want to arrive at (5,0). Suppose as well that the heuristic is Euclidean distance to the goal. A* begins by checking all the neighbors of $(0, 0)$, choosing to prioritize $(1, 0)$ because its cost function is 5, which is lower than the cost function in all other directions. Then, A* checks the neighbors of $(1, 0)$, choosing to prioritize $(2, 0)$ using the same logic. This goes on until we arrive at $(5, 0)$ with checking only 5 nodes. BFS would have checked many more nodes, since it would have checked not only the neighbors of $(1, 0)$ but also $(0, 1), (0, -1)$ and $(-1, 0)$ which are all not in the direction of the goal.

Fig. 1. The robot is able to navigate between any start (green point) and end (red point). Given a start and end, a path is calculated using A* and plotted using a white marker.

We used A* to successfully generate short paths between two points in the

Stata basement. In this lab, many heuristics were available. The only criteria for a heuristic to be admissible (meaning it guarantees the shortest path is found) is that the heuristic may never overestimate the path's remaining distance from the goal. We found that changing the heuristic does not give us a better path and only affects runtime. Therefore in our case, we opted to use Euclidean distance as a heuristic. We ran the algorithm in simulation and tried many combinations of starting and ending points (see figure 1). For any combination of points, the algorithm successfully found a path. Therefore, we concluded that we had successfully created a path-finder.

One shortcoming of our algorithm is that it produces slightly sub-optimal paths, but the algorithm still works for pure pursuit controllers. In figure 2, we see many locations where a sub-optimal choice of path is taken. Producing sub-optimal paths is a signal that the algorithm is implemented incorrectly, since A* always produces the shortest path. The error likely arises because diagonal and side moves are treated as having the same length. The result is a robot that takes zig-zag paths more freely as these are equivalent to straight lines under that regime.

Fig. 2. In some cases, we do not produce the optimal path. In the figure above, circles indicate places where the path is sub-optimal, which is due to many reasons. We see that sometimes, the path takes the two legs of a right triangle rather than the hypotenuse, which is never shorter.

We found that despite the shortcoming, the algorithm proved sufficiently robust for applications involving pure pursuit controllers where the path-finding

demands are relatively tolerant of sub-optimal route calculations. Pure pursuit is known to cut corners because the look-ahead distance is further ahead, so jagged edges in the path smoothen. Thus, while our implementation may not have the full theoretical efficiency of A*, it's good enough for the specific purpose of planning a path for our car from point to point.

Sample Based - Probabilistic Road Mapping (PRM)

[*Ryan Xiao*]
PRM is one of the more widely used sample based path-finding algorithms. Initially, a set of points are sampled at random from the map. These points must not collide with any obstacles in the map. The start and goal points are also included in this set of points. An undirected weighted graph can then be constructed by iterating through every pair of points that have a distance between each other that is lower than a certain threshold. We can add an edge between the two points to our graph if the edge does not go through any obstacles. The weight of the edge will be the euclidean distance between the two points in the edge. Once the graph has been constructed, Dijkstra's algorithm can be used to find the shortest weighted path between the start and goal points.

There are two parameters of PRM that one can adjust. The first is the number of points to be initially sampled to construct the graph. More points can lead to more optimal paths, but it can also lead to greater computational costs. Less points can lead to greater efficiency, but it may lead to some more interestingly shaped paths being generated. The other tuneable parameter is the maximum distance at which points can be to form an edge. A higher maximum distance means that more pairs of points will be in consideration for possible graph edges. This will lead more possible edges that are able to be traversed, which will increasing the possible algorithm run time but yield more more optimal paths.

2.1.3 Comparison of Path-finding Algorithms

[*Ryan Xiao*]
Similar to a BFS, A* is an exhaustive search algorithm that considers every possible point in the map to construct a graph. As a result, the algorithm will theoretically iterate through all possible paths in order to obtain the most optimal path between the start and goal points. We can measure the

optimality of a path by total length of the path. Generally, a shorter path between the start and goal points is more desirable. Although finding the most optimal path is favorable, this algorithm requires much more computational intensity to execute. Most notably, the number of points in consideration grows proportionally to the size of the map. So for maps cover entire areas like neighborhoods or even cities, this type of search would be inefficient and not be able to complete within a viable time frame, given the hardware present on current autonomous cars.

Sample based path finding algorithms have been developed in order to mitigate this issue. Unlike search based algorithms, sample based algorithms are not guaranteed to generate the most optimal solution. This is because they only consider a subset of all possible points in the graph, which means a fewer number of points will be considered when constructing the weighted graph. This method can substantially reduce the computational cost, making it possible for autonomous cars to plan paths with lower energy hardware. However, using this algorithm does sacrifice some optimality. Because points are randomly sampled, there may be some areas of the map that will not be accessed due no points being sampled there. Further, paths around corners may have weird shapes or may be deemed impossible to round. These kinks in the path can lead to an increased total distance, which may increase the time it takes for the robot to traverse from the start to the goal.

For this lab, we decided to implement the A* algorithm because the map of the Stata basement is relatively small, consisting of 10,000 total points (100x100 Occupancy Grid). This means that the computational cost is not as big of a problem, so a higher level of path optimality is preferred. Using this algorithm, our robot will plan the shortest possible path between the start and goal, allowing it to traverse most efficiently between points on the map.

2.2 Path Following - Pure Pursuit

[*Ryan Xiao*]
Once a path has been generated, the robot needs to follow that path. This can be done using a pure pursuit controller. In brief, controller works by selecting a point on the calculated trajectory and outputting a steering angle at which the robot would reach the selected point if it were to continue at its current velocity.

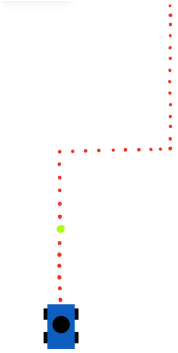

Fig. 3. The robot selects a pursuit point (green point) on the discretized trajectory (red points) based on a specified look-ahead distance.

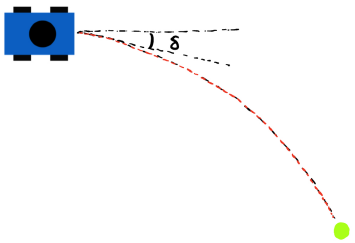

Fig. 4. The robot, traveling at a constant velocity, receives a steering angle command (δ), directing it on an arced path (dotted red line) that intersects with the pursuit point (green point).

The calculated trajectory from the path planner is represented by a list of coordinates in reference to the world frame. At each time step, the robot begins by calculating the point in the trajectory that is closest to its current position. This is done using numpy array operations, which allows the us to efficiently calculate the distance euclidean distance from the robot's current position to each point in the trajectory. The index of the point closest to the the robot's position can then be obtained using the numpy argmin function. From this point, we search for a point on the trajectory that is a specified look-ahead distance away. This can be done by iterating over all points in the trajectory, starting from the index of the closest point to the robot on the trajectory. If the point checked is a distance greater than or equal to the specified look-ahead distance, it becomes the robot's pursuit point, or point that it is trying to "pursue", shown in Figure 3. A stopping condition for the algorithm can be generated in the case when the robot's position is closest to the goal point on the trajectory. Under this condition, we will know that the robot has traversed to the end of the path and can therefore stop running the path follower until a new trajectory is received.

The simplest way for the robot to reach the pursuit point is to hold a steering angle and travel at a constant velocity. Using these drive commands, the car will travel in an arced trajectory that intersects with the pursuit point as shown in Figure 4. To reduce the complexity of our controller, we keep the velocity of the car constant, the controller only outputs the desired steering

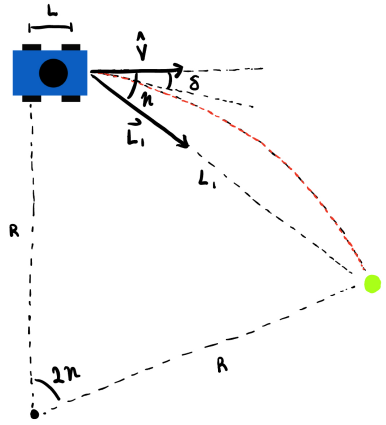

Fig. 5. This is an example of a robot taking in a steering angle command (δ) in order to follow the desired arced path (dotted red line) to reach the pursuit point (green point). The wheelbase length (L), distance between the robot and pursuit point (L_1), unit velocity vector of the robot ($\hat{V}$), vector between the robot and pursuit point ($\vec{L_1}$), and η are labeled above. These values are present in the formula to calculate the magnitude of the steering angle command (δ).

angle. This offers a possible optimization opportunity in the final project where we can adjust both speed and steering angle when navigating an area. This can help the robot traverse the trajectory at a greater speed, and get from the start to the goal more efficiently.

The magnitude of the steering angle command (δ) is calculated using the formula,

$$\delta = tan^{-1}(\frac{2Lsin(\eta)}{L_1})$$

where L is the wheelbase length, L_1 is the distance between the robot and the pursuit point, and η is the angle between the robot's unit velocity vector, $\hat{V}$, and the vector going from the robot to the pursuit point, $\vec{L_1}$. $\hat{V}$ is given by $< cos(\theta), sin(\theta) >$, where θ is the orientation term of the robot's pose in reference to the world frame. L_1 can be obtained by subtracting the robot's current position coordinates from the coordinates of the pursuit point. These values can all be visualized in an example in Figure 5. From these vectors, there is enough information to calculate η through dot product:

$$\hat{V} \cdot \vec{L_1} = ||\hat{V}||||\vec{L_1}||cos(\eta) \Rightarrow \eta = cos^{-1}(\frac{\hat{V} \cdot \vec{L_1}}{||\hat{V}||||\vec{L_1}||})$$

However, dot products do not denote any directionality or sign associated with the calculated η, so a final step in determining what direction the robot needs to turn is crucial in order for the controller to function properly. The direction can be obtained by applying a rotation to $\vec{L_1}$ using a 1D rotation matrix, with the robot's current orientation, θ, as the angle at which the vector should be rotated. Once the new vector has been calculated, it can then be observed that the y value of the rotated $\vec{L_1}$ vector denotes the direction in which the robot needs to turn. If the y value of the vector is greater than 0, the robot needs to turn to the left, and the final δ will be positive. If the y value of the vector is less than 0, like the example shown in Figure 3, the robot needs to turn to the right, and the final δ will be negative. If the y value equals 0, the robot is actually heading straight for the pursuit point, so δ would be 0.

3 Evaluation

3.1 Path Finding Algorithm

[*James Morin*]
As discussed in Section 2.1.1 The first part of our path planning was the discretization of the map into individual points that act as "nodes" in a graph. Nearby nodes connect to each other, and running our A* algorithm on our graph representation of the map enables us to find a path to the destination.

To make sure we don't send the racecar to any point too close to a wall, we dilute the map with a square kernel before converting the map into a searchable graph. The size of this kernel was the first parameter that had a large impact on our path finding performance. In this case, there was a clear answer to how large the kernel size should be. The kernel size had to be no greater than 27 pixels for all open portions of the map to be traversable, and the larger the kernel, the less likely the car was to veer into a wall on sharp turns, so our kernel size stayed at 27.

The heavy lifter in our path planning was our implementation of A*. Originally we had used a python package with A* built in, but since we cannot easily install packages on the racecar, we had to implement A* from scratch.

This worked really well, but we found that it didn't actually find the optimal path. The benefit of A* is that with an appropriate heuristic, we should often find an optimal path faster than a more generalized search like BFS. However, The results we were getting had some small areas along the path where it was clear that a slightly more optimal solution could be found. Figure 2 shows points where a sub-optimal path is taken. We were pretty sure that this was due to the fact that we'd forgotten to weigh diagonal edges greater than straight edges, but even after this fix, something was still off and we had similar results with similar sub-optimality. However, despite not finding THE optimal solution, we always found a very good solution to path between any two points, so we decided to be satisfied with this implementation. Our pure pursuit controller would smooth out some of the extraneous corners in the path as well.

3.2 Path Following Algorithm

[*James Morin*]
We tested a couple of combinations of speed and look-ahead distance for our path follower that we described in Section 2.2. For each setup described, we ran a simulated path as shown in Figure 6.

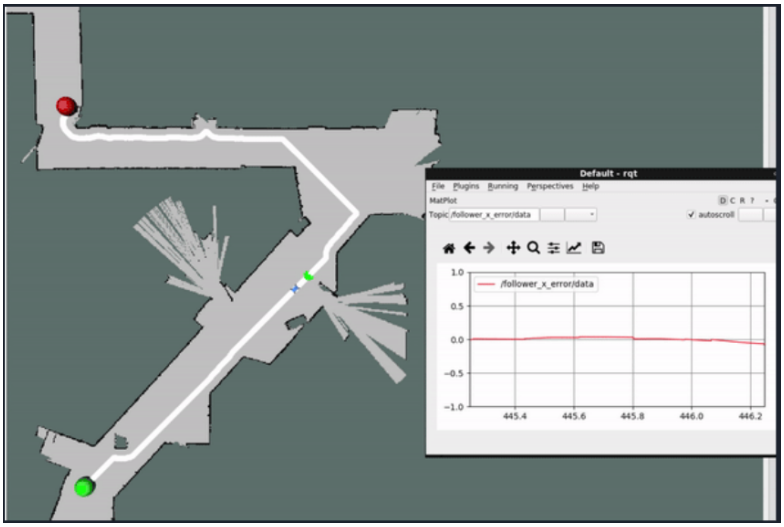

Fig. 6. Each test had the same start and end point, and each path was the same. We observed the lateral deviation from the path at each time step in a live graph.

Table 1: Results from simulation tests

velocity (m/s)	look-ahead distance (m)	max error (m)
1.0	1.0	$\pm$ 0.3
2.0	1.0	$\pm$ 0.5
4.0	1.0	N/A
4.0	2.0	$\pm$ 1.2

As would make sense, the faster the car moves, the larger the deviation for the path will be. Each time step the car moves a larger distance, which means that less corrections are made for a given path.

We also observed that a car that moves faster requires a longer look-ahead distance. It takes a certain amount of time for the racecar to calculate and update the steering angle of the car to follow the path. With larger deviation away from the path, a larger steering angle may be required to turn to converge with the path, and so the curvature of the car's trajectory is more drastic. In the same amount of time, the position and orientation of the car can change much more if we are moving faster than slower. Trying to converge with a point in the path that is too close means that we do not smoothly converge with the path and instability arises. In our simulation with the car moving at 4m/s and a look-ahead distance of only 1m, the car crashes into walls in wild swings and loses its localization as a result, hence no quantifiable data on the error of that trial. Increasing the look-ahead distance to 2m led to a stable and capable path follower once again, even at the higher speed.

3.3 Racecar Performance

[*Jian Ming Chen*]
The real world evaluation mainly comprised of testing the robustness of the pathing planning and pure pursuit code on the physical racecar. The path planning algorithm was tested extensively in simulation, but we also tested it on the physical racecar by setting different ending points and confirming that optimal trajectories were being created when possible. After confirming path planning was working as intended, we used the race track in the Stata basement to test our pure pursuit code by varying look-ahead distances and

speeds of the racecar. We analyzed the physical behavior of the car and compared it to the perceived location of the racecar in Rviz on the computer.

Table 2: Results from real life racetrack

velocity (m/s)	look-ahead distance (m)	time (s)
1.0	1.0	62.7
2.0	2.0	27.3
2.5	2.2	25.0

A trend we expected was a decrease in the time needed to complete the racetrack as we increased the speed of the racecar, which is verified by our data in Table 2. After asking a TA, we realized that the physical racecar has an artificial speed limit of 2.5m/s, so we could only test up until that cap.

In addition to the expected trend, we noticed while testing that small look-ahead distances resulted in increased oscillations on the robot while following the trajectory but had better tracking. This behavior was apparent when the robot followed curves in a trajectory as it was able to track the turn well but the larger steering angles issued because of the smaller look-ahead distance made the turn inefficient and would sometimes lead to oscillations. In some cases, a small look-ahead distance with a fast velocity would result in the robot moving ahead of the pursuit point, resulting in a complete loss of tracking.

On the other hand, larger look-ahead distances would reduce oscillations but had worse tracking. When turning around corners, a larger look-ahead distance would result in smaller steering angles that are issued earlier, which would have less oscillations. However, these earlier steering commands could cause the robot to turn too quickly and crash into a corner it was trying to avoid.

In our testing, we used the lessons learned from simulation to increase the look-ahead distance as we increased the speed of the robot. We found that at a speed of 1.0 m/s, a look-ahead distance of 1.0 m had the best performance. And at speeds of 2.0 m/s and 2.5 m/s, look-ahead distances of 2.0 m and 2.2 m respectively were optimal. Our testing comprised mainly of observing how closely the robot followed the planned trajectory without oscillating and turning corners too tightly. After finding a possibly optimal look-ahead distance, we would repeat the trial multiple times to confirm that the robot

does not deviate substantially from the trajectory. At the artificial max speed of 2.5 m/s and a look-ahead distance of 2.2 m, our robot was able to finish the Stata racetrack in 25.0 seconds as shown in Table 2.

3.4 Torn out Wires...

[*James Morin*]
As we tested our racecar in the Stata basement to see how quickly we could finish the race, our car became unable to steer. We tried re-building our packages and restarting teleop, but nothing made the steering work again. We soon discovered that the wire to the servo that controlled steering had been completely torn out. The driveshaft caught the wire and as it spun it tore it from both ends. Subsequently we had no steering and so our testing had come to an end.

Thankfully, this was after we had our briefing and after we were able to test our racecar on the basement race route. The racecar is pending repair, but we got almost everything we wanted done for Lab 6 before it occurred and we are in a position to work on the final challenge conceptually until the racear is ready to hit the track again.

4 Conclusion

[*Remeyn Mechura*]
After the dust has settled, we have designed, built, and implemented, both in a simulator and on our real car, a strong, efficient path finding algorithm and a robust, smooth pure pursuit algorithm. Accomplishing this lab was a significant milestone in autonomy, and it relied heavily upon the knowledge and experience gained in all of the previous labs combined.

For our technical implementation, we were able to work separately and simultaneously on both the path planning and the trajectory following for troubleshooting, debugging, and team member workload purposes. Once both were independently successful, we combined them into a single package that we believe to be ideal. In designing our path finding algorithm, we considered many of the more common and popular methods and chose based on

our expected operational environment, optimizing for distance travelled and reducing computational complexity. For our trajectory following implementation, the most important thing beyond making a fast, geometrically accurate and stable program, was hand selecting the look-ahead distance. This parameter can make or break (literally) our car, so tailoring it specifically and precisely was a primary factor in the completion of the path following portion of the lab.

5 Lessons Learned

5.1 Allen Baranov

On the theory side, this lab has taught me about how there are different ways to approach a problem. I did not expect there to be multiple viable strategies for path finding that all approach the problem in drastically different ways. It goes to show how it is good to explore many avenues while problem-solving to increase chances of finding one of the multiple ways to solve it. On the practical side, I learned how it is very necessary to know the algorithm I am trying to implement when debugging. If I can't work through step-by-step what the code is doing, I'm not able to find where a potential bug is. It is more efficient to go back into the lectures and rehash my understanding than try to work on the lab without the requisite understanding.

5.2 James Morin

During this lab our racecar became severely damaged when the steering servo's wire became torn out by the driveshaft. This was very unexpected, since this was not even on our radar as something that could go wrong! We didn't budget any time for such a setback, and thankfully it occurred after we'd completed everything for the briefing and begun some successful tests of the race course. I learned that setbacks can always occur, even seemingly out of nowhere, so the earlier I work on something the more time I'd have to adapt to changes like this, which I'm not immune to.

5.3 Jian Ming Chen

In terms of technical lessons, this was another lab that was relatively theory-intensive for me as I had to implement an algorithm that I didn't know before. This lab was a chance for me to review technical resources about path planning and include them in my implementation. Furthermore, the fact that there were multiple path planning algorithms to consider taught me how important it is to plan and analyze tradeoffs between different possibilities before implementation, a critical skill for problem-solving. As for communication lessons, this lab had many different components to collect data for, which gave me a chance to consider more deeply on what data to collect to highlight the points I wanted to explain.

5.4 Remeyn Mechura

I learned a lot in this lab, mostly the theory behind the pure pursuit algorithm, which I think was the coolest part. Unfortunately I am still behind the power curve when it comes to actually implementing the code, running the robot, etc. but in the end as long as progress is being made, I am satisfied. I also learned a lot during the collaboration sessions we had, getting to watch these guys do this stuff was like magic.

5.5 Ryan Xiao

This lab really taught me the importance of understanding the details of each algorithm. When analyzing the various path finding algorithms, it was important for me to think critically about each one and ask why they work, why they produce optimal/suboptimal solutions, what data structures they use, and what its run time complexity was. Asking these questions both helped me to gain a deeper understanding of how the algorithms worked, and it allowed me to see the various tradeoffs that the algorithms had over each other. Further, when implementing the pure pursuit controller for path following, it was important to understand the underlying geometry associated with the formulas given. This proved to be crucial in debugging the controller where a hidden sign error tucked away in a dot product calculation veered the robot off the path and into the abyss.