Path Planning and Execution on Racecar

Andy Li
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
andyli27@mit.edu

Antonio Avila
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
antonio3@mit.edu

Harmanpreet Kaur
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
hkaur@mit.edu

Mia Tian
MIT Department of Aeronautics and Astronautics
Cambridge, USA
miatian@mit.edu

Natalie Huang
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
huangn@mit.edu

1 Introduction

Harmanpreet Kaur

The purpose of this lab is to evaluate and implement path planning algorithms to determine a path to a given destination, as well as to drive along the path. This ability is majorly important to the final challenge, since we will have to navigate around a known map, going from a start pose to a goal pose autonomously, which is exactly what we are implementing in this lab.

To start, our team evaluated various path planning methods to narrow down which ones to implement. Using these methods, we planned trajectories in a known occupancy grid map from the racecar's current pose to a goal end pose. Additionally, we used our particle filter from a previous lab to localize the car in the known map to find the start pose. Finally, we used pure pursuit control to drive the car along the planned trajectories to the goal pose.

2 Technical Approach

Our team implemented a variety of methods to plan and follow paths, given a start and goal location. We optimized these methods to allow for a fast and seamless planning and following process for the racecar.

We then implemented multiple path following algorithms varying in complexity and optimality.

2.1 Path Planning Algorithms

Antonia Avila, Harmanpreet Kaur, Mia Tian

The first step to implementing path following on a racecar is to choose a path finding algorithm. Our team compared various search-based methods and sample-based methods to narrow down one method from each category to implement and test.

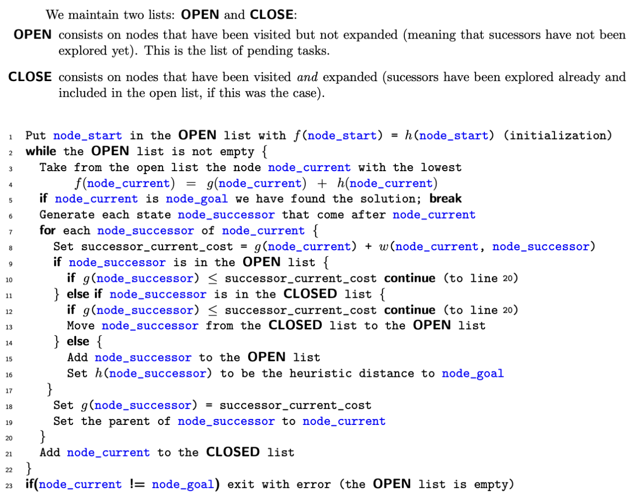
2.1.1 Search-Based Methods: A search-based method explores a known map increasingly until a path is found, or until the entire reachable map has been explored. We considered three search-based methods: Dijkstra's Algorithm, Greedy Best-First Search, and A*. Dijkstra's Algorithm considers all adjacent nodes and prioritizes paths with lower costs, always finding the path with the lowest total cost, but is computationally expensive. Greedy Best-First Search uses a heuristic function that calculates the Euclidean distance to a single goal, and prioritizes searching around nodes with the minimum distance to the goal, but is not guaranteed to provide an optimal path. A* is the best of both algorithms, by using a heuristic function that minimizes the sum of both the total cost of the path as well as the Euclidean distance to a single goal. This is comparatively computationally inexpensive, but still provides the most optimal weighted path. Since we are only driving the robot to a single goal pose, A* would be the best search-based method to implement. Fig. 1 displays the pseudocode for A*.

A* Implementation We made multiple design choices during the implementation A*. In order to discretize our map environment, we converted it into a grid. We looped through the map to find pixels with a high probability of obstacles and indicated that grid cells corresponding with those pixels were obstacles. We padded these obstacles with a radius to ensure that our car was at least a fixed distance from the wall.

The neighbor of each node included all of the cells sharing an edge with it. We also added diagonal cells as neighbors if the cells that share an edge with that diagonal cell and the original node are all empty. This better reflects vehicle dynamics because the car can move diagonally.

For the heuristic, we chose euclidean distance to the goal, which is fast to compute.

A* Optimization The testing of our A* implementation in simulation and in real-life showed that the path generated was not the most optimal for the car to follow. Clearly, the path would be more optimal if the car could move in every direction, not just the adjacent cells displayed in Fig. 2. We


We maintain two lists: **OPEN** and **CLOSE**:

OPEN consists on nodes that have been visited but not expanded (meaning that sucessors have not been explored yet). This is the list of pending tasks.

CLOSE consists on nodes that have been visited *and* expanded (sucessors have been explored already and included in the open list, if this was the case).

```
1  Put node_start in the OPEN list with $f$(node_start) = $h$(node_start) (initialization)
2  while the OPEN list is not empty {
3    Take from the open list the node node_current with the lowest
4         $f$(node_current) = $g$(node_current) + $h$(node_current)
5    if node_current is node_goal we have found the solution; break
6    Generate each state node_successor that come after node_current
7    for each node_successor of node_current {
8      Set successor_current_cost = $g$(node_current) + $w$(node_current, node_successor)
9      if node_successor is in the OPEN list {
10       if $g$(node_successor) $\leq$ successor_current_cost continue (to line 20)
11     } else if node_successor is in the CLOSED list {
12       if $g$(node_successor) $\leq$ successor_current_cost continue (to line 20)
13       Move node_successor from the CLOSED list to the OPEN list
14     } else {
15       Add node_successor to the OPEN list
16       Set $h$(node_successor) to be the heuristic distance to node_goal
17    }
18     Set $g$(node_successor) = successor_current_cost
19     Set the parent of node_successor to node_current
20   }
21   Add node_current to the CLOSED list
22 }
23 if(node_current != node_goal) exit with error (the OPEN list is empty)
```

Fig. 1: Here is the pseudocode for A*. f is the total cost of a node, g is distance travelled from the start node to the current node, and h is the heuristic (which we chose to be euclidean distance from the goal).

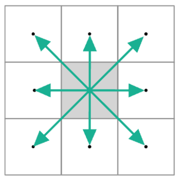


Fig. 2: This diagram shows the neighbors of a node– the cells that the car could move if they were in the center cell.

developed a way to decrease the number of turns in the path and decrease the number of points during straight line parts of the trajectory. The optimization hinges on the idea that if the car does not need to turn, it should not.

We developed a way to optimize the path after it is produced from our implementation. We loop through each point in the path; for each point, we look ahead to find the farthest point that the car could reach in a straight line from the initial point. If it exists, we delete the intermediary points. Then we move on to the next point in the loop.

This optimization gets rid of useless turns, thereby increasing the speed of the path following. It also finds diagonal paths that are not able to be produced from the grid representation. Because this optimization is conducted on the path after it is computed, the runtime is very fast.

2.1.2 Sampling-Based Methods: A sampling-based method for trajectory planning is one that relies on sampling the state space of the robot to find a path from an initial state to a goal state in state space. The sampling-based planning algorithm we chose to implement was Rapidly-exploring Random Tree (RRT), rather than Probabilistic RoadMap (PRM), simply due to its lower computation costs, since it is a single-query algorithm.

A summary of the algorithm to create the tree is provided in pseudo code below as Algorithm 1.

Algorithm 1 RRT

```
EarlyReturn $\leftarrow False$
$V \leftarrow \{v_0\}$
$E \leftarrow \emptyset$
$T = (V, E)$
for $i = 1 \ldots N_{max}$ do
    $q_{rand} \leftarrow$ Sample$(Q)$
    $v_{nearest} \leftarrow$ Nearest$(V, q_{rand})$
    $v_{new} \leftarrow$ Steer$(v_{nearest}, q_{rand})$
    if CloseToGoal$(v_{new})$ then
        $v_{new} \leftarrow v_{goal}$
        EarlyReturn $\leftarrow True$
    end if
    if NoCollision$(v_{nearest}, v_{new})$ then
        $V \leftarrow V \cup \{v_{new}\}$
        $E \leftarrow E \cup \{(v_{nearest}, v_{new})\}$
    end if
    if EarlyReturn then return
    end if
end for
```

Algorithm 1: Sample is a function mapping state space Q to a random sample q_{rand}. Nearest is a function mapping the vertices of the RRT, T, and a position in state space, q, to the closest node in T. Steer is a function mapping a node in T and a position in state space to a candidate node to be added to T. CloseToGoal is a function mapping a node in T to whether or not the state it represents is close enough to the goal position to be considered, roughly, as the goal. NoCollision is a function that tells whether or not the path between the position that two nodes in T represent is in collision with the environment.

Once the tree is created we can traverse back up the tree from the goal node, v_{goal}, to the start node, v_0, to get our path.

One upside of this approach is that its computational complexity can scale well to higher dimensions. For our lab, we are working in only two dimensions, so no real speed up to path planning via search based methods was observed. In fact, RRT would sometimes take longer than A* to find a path. This is because the obstacles in our environment are dense, and we have one very narrow pathway in the map, leading RRT to need many samples to find a feasible path through the environment. Another downside of RRT is that it is not optimal, meaning the paths that it returns have no guarantees about minimum time, minimum distance, etc.. Overall, RRT was useless for this lab as state space search in two dimensions is a relatively quick and easy problem that can be solved to optimality.

2.1.3 Comparison of Path Planning Methods: Search-based methods and sampling-based methods compare and contrast in various ways; we will specifically compare the two methods our team has chosen to implement, A* and RRT. Both algorithms are single-query and dynamic, so they re-build their graphs every time the starting or ending point is changed, and calculate a new single path from start to end every time. However, A* is generally more complex than RRT, especially with higher dimensions and higher resolution graphs. Additionally, while RRT can explore large, empty spaces quicker, A* always provides an optimal path with minimum distance. For this lab, we are only dealing in a two dimensional space and would strongly prefer an optimal path, we speculated that A* would meet our standard better than RRT would, but nonetheless, we were able to implement both to experimentally compare the results in real life (as described ahead in Section 3.1).

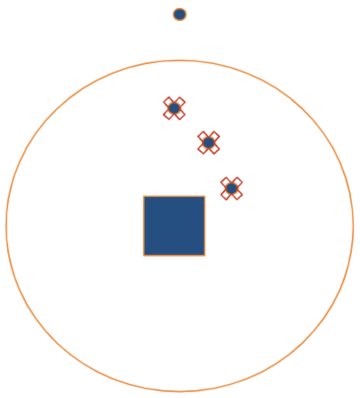


Fig. 3: A visualization of the Closest Point approach. The circle has a radius of the lookahead distance, and the points inside the circle are marked as visited. The point just outside the circle is the target point.

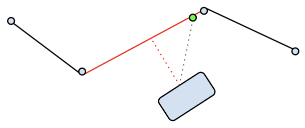


Fig. 4: A visualization of the linear interpolation approach. We locate the closest line segment to the current position of the car, shown in solid red. Then, we find the point on that line segment that is the lookahead distance away, shown in green, which is the target point.

2.2 *Path Following Algorithm*

Natalie Huang

Our general approach was to utilize pure pursuit to accurately follow the planned trajectory. A main challenge we encountered was how to efficiently find a target point for the pure pursuit steering. To do this, we tried three different approaches.

2.2.1 Closest Point: In the Closest Point method, we defined the target point as the closest point in the path to the car that is greater than the lookahead distance distance. To find this, we sequentially looked at points in the path that were within the lookahead distance of the current position of the car and marked those as visited, then took first point that was greater than the lookahead distance away and set that as the target.

2.2.2 Linear Interpolation: In the Linear Interpolation method, we created linearly interpolated line segments between each point in the path. Then, at each point in time, we found the closest line segment to the current car's position. To do this, we found the projection of the current position p onto the line segment vw, where v and w are consecutive points in the path. The equation for this projection is $\frac{(p-v)\cdot(w-v)}{|w-v|^2}$. We clamp this quantity to between 0 and 1 in the cases that the closest point on the line segment is an endpoint. Then, we return the distance between point p and the projection to find the distance from the line segment. Next, we found the point along that line segment that was the lookahead distance away from the car, and set that as the target point. Figure 4 shows an example path and the target point given the car's current pose.

2.2.3 Splines: Andy Li

An approach to procedurally generate a path that can generally be followed more closely by a physical system is to use smooth *splines*, piecewise polynomial functions. For ease of implementation, we chose to use standard Cubic Hermite splines, which are defined parametrically by the initial and final positions and derivatives for each piecewise function of the spline according to four polynomials:

$$\vec{s}(t) = (2t^3 - 3t^2 + 1)\vec{p_0} + (t^3 - 2t^2 + t)\vec{m_0} + (-2t^3 + 3t^2)\vec{p_1} + (t^3 - t^2)\vec{m_1}, 0 \le t \le 1 \quad (1)$$

This creates a smooth curve between the two points, satisfying $\vec{s}(0) = \vec{p_0}, \vec{s}(1) = \vec{p_1}, \vec{s}(0)' = \vec{m_0}, \vec{s}(1) = \vec{m_1}$. Given a sequence of points from our path planning algorithm, we can interpolate them into a spline by creating a function for each pair of adjacent points. Given the points $\{\vec{p_0}, \vec{p_1}, ..., \vec{p_n}\}$, we can create a spline with n functions $\{\vec{s_1}(t), \vec{s_2}(t), ..., \vec{s_n}(t)\}$:

$$\vec{s_i}(0) = \vec{p_{i-1}}, \vec{s_i}(1) = \vec{p_i}, \ \forall\ 1 \le i \le n$$

To ensure smoothness, the ending derivative of each function must be equal to the starting derivative of the following function, as this creates a continuous derivative over the entire spline. More precisely, we must ensure that the following holds true:

$$\vec{s_i}(1)' = \vec{s_{i+1}}(0)', \ \forall\ 1 \le i \le n-1$$

During testing, we found that setting the derivative equal to the normalized vector between the previous and following point led to qualitatively satisfactory splines and found no reason to change this heuristic.

$$\vec{s_i}(1)' = \vec{s_{i+1}}(0)' = norm(\vec{p_{i+1}} - \vec{p_{i-1}}), 1 \le i \le n-1$$

The beginning and ending derivatives of the entire spline can be set arbitrarily. To ensure smooth integration with Algorithm 5 for path following, we set them equal to the normalized vector difference between the terminal point and its adjacent point.

$$\vec{s_1}(0)' = norm(\vec{p_1} - \vec{p_0})$$

$$\vec{s_n}(1)' = norm(\vec{p_n} - \vec{p_{n-1}})$$

An example spline, with derivatives visualized at the endpoints of each function, is shown below in Figure 5.

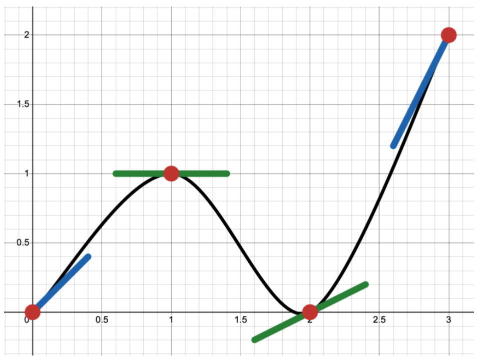


Fig. 5: Visualization of a Cubic Hermite spline with control points (0,0), (1,1), (2,0), (3,2). The derivatives at each point are set according to the rules described above.

To utilize splines in a path following algorithm, we first use the Newton-Raphson method to find the closest point to the racecar position $\vec{p_{car}}$ on the spline, outlined in Algorithm 2. Since the method conditionally converges to a local zero of the given function, we run the method on the first derivative of the squared distance from the point on the spline defined by the input t and the racecar position $\vec{p_{car}}$. This finds a zero of the first derivative, which corresponds to an extrema of the squared distance itself.

$$sqDist(t) = norm(\vec{s}(t) - \vec{p_{car}})$$

Algorithm 2 Closest Point on Spline

```
$curBest \leftarrow \infty$
$bestT \leftarrow 0$
**for** $s = 0 \dots N_{starts} - 1$ **do**
    $curT \leftarrow s/(N_{starts} - 1)$
    **for** $i = 1 \dots N_{iters}$ **do**
        $curT \leftarrow curT - \frac{sqDistDeriv(curT)}{sqDistSecondDeriv(curT)}$
    **end for**
    **if** $sqDist(curT) < curBest$ **then**
        $curBest \leftarrow sqDist(curT)$
        $bestT \leftarrow curT$
    **end if**
**end for**
**return** $bestT$
```

Given the smoothness of the spline, running the algorithm from several initial t values along each function in the spline and finding the best result nearly always guarantees convergence to the true closest point on the spline in practice. We found that starting the algorithm along 5 equidistant points on each function and running 5 iterations led to more than enough precision within reasonable runtimes.

Once we have found the closest point on the spline to the racecar pose, we must find a point along the spline that is the correct lookahead distance away from the racecar, allowing the racecar to continuously follow the spline as Pure Pursuit is run. To do this, we can use a simple iterative algorithm that assumes the derivative of the spline is constant at the point given by the current t value and adjusts the t value accordingly to reach a point a given distance away. Because this assumption becomes more accurate as the change in distance desired becomes smaller, iterating this process many times allows us to reach a point the correct distance away, even given an arbitrary function. A good initial guess for t is the desired length (length to closest point plus lookahead) divided by the total length of the spline. Such an algorithm is outlined in Algorithm 3.

This also requires a way of quickly calculating the length of a spline up to an arbitary t value. For this, we utilize Gaussian quadrature, a well-known algorithm to estimate the length of functions that can be well approximated with polynomials with extremely high accuracy. This is outlined in Algorithm 4.

Algorithm 3 Get Lookahead T

```
$effDist \leftarrow lookaheadDist - dst(\vec{p_{car}}, closestP)$
**if** $effDist <= 0$ **then return** $closestT$
**end if**
$desiredLength \leftarrow gauss(closestT) + effDist$
$t \leftarrow desiredLength/gauss(1)$
**for** $i = 1 \dots N_{iters}$ **do**
    $derivMag \leftarrow norm(getSplineDeriv(t))$
    **if** $derivMag > 0$ **then**
        $t \leftarrow t - (gauss(t) - desiredLength)/derivMag$
        $t \leftarrow min(1, max(t, 0))$
    **end if**
**end for**
**return** $t$
```

Algorithm 4 Gaussian Quadrature

```
$coefPairs \leftarrow getGaussianCoefs()$
$half \leftarrow (end - start)/2$
$avg \leftarrow (start + end)/2$
$length \leftarrow 0$
**for** $cf_0, cf_1$ IN $coefPairs$ **do**
    $length \leftarrow length + norm(getSplineDeriv(avg + half * cf_1) * cf_0$
**end for**
**return** $length$
```

One detail in implementation that was important for correct function was the end behavior. The spline is only well-behaved up to $t = 1$, or the ending pose, so we clamped the Algorithm 3's output to $[0, 1]$. As the racecar approaches the end point, this means that the target point will remain the end point, but running Pure Pursuit with a tiny lookahead will make the racecar's behavior highly unstable and sensitive to error as it gets close to the end point. To fix this, we effectively extend the spline past the end point along the tangent line at the end point. This small modification can be implemented as follows:

Algorithm 5 End Condition

```
$distToEnd \leftarrow dst(\vec{p_{car}}, closestP) + gauss(closestT, 1)$
**if** $distToEnd < lookaheadDist$ **then**
    $target \leftarrow norm(getSplineDeriv(1))$
    $target \leftarrow target \cdot (lookaheadDist - distToEnd)$
    $target \leftarrow target + getSpline(1)$
**end if**
```

Once a suitable lookahead point has been obtained, it can be fed into the Pure Pursuit controller as the target point. Given suitable parameters, the racecar will converge towards accurately following the spline until the end goal has been reached. Examples of Pure Pursuit path following running with a Cubic Hermite spline path are shown in Figures 6 and 7.

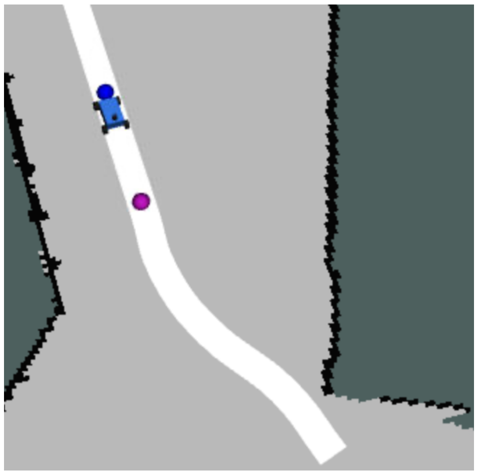


Fig. 6: Screenshot from RViz visualization of the racecar following a Cubic Hermite spline path interpolated from points generated using A*. The blue point is the visualization of the closest point to the racecar on the spline, and the pink point is the visualization of the target point that the racecar is following, forward of the blue point by exactly the lookahead distance. The slight difference between the racecar position and the blue point is not error, but rather latency caused by the algorithm not running in real-time.

2.2.4 Pure Pursuit Steering: Andy Li, Natalie Huang Once we have a target point, we used standard Ackermann dynamics with our car's physical measurements to determine the correct steering angle needed to approach that point. As the racecar drives along the path, the target point updates forward accordingly, until the entire path has been successfully traversed. A simple end condition across path following algorithms is to check whether the racecar position is within a certain distance of the final point in the path, in our case set to 0.1 meters.

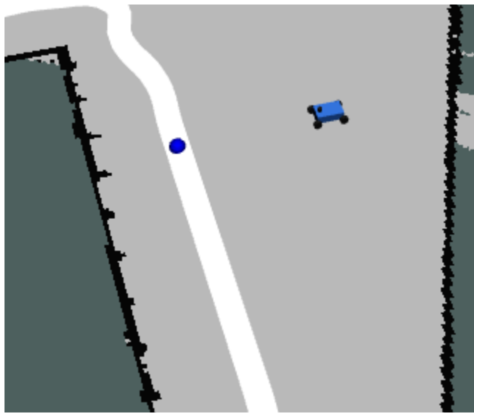


Fig. 7: Screenshot from RViz visualization of the racecar following the same path, but starting much farther from the path. The effectiveness of the closest point algorithm can be seen here. The target point is overlaid under the closest point because they are coincident when the racecar is more than the lookahead distance away from the spline.

3 Experimental Evaluation

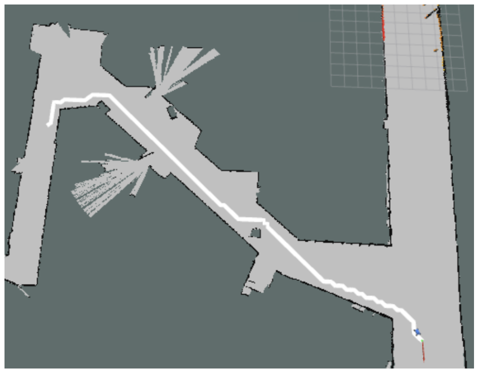


Fig. 8: Screenshot from RViz visualization of the path in simulation we used to evaluate the runtime of each path following approach. This roughly matches the staff-marked path in Stata that we tested our real car on.

3.1 Path Planning

Mia Tian

This subsection explores the simulation results of A* and RRT. It uses qualitative and quantitative metrics to compare the paths of the A* implementation before and after the optimization as well as between A* and RRT.

3.1.1 Performance of A:* Testing A* via simulation showed that a higher resolution of the grid increased complexity, but a lower resolution sometimes did not produce solutions because grid cells with obstacles may be blocking the path. Larger obstacle padding allowed for safer driving but also produced the same issue of solvability.

For testing on the car, we used a grid resolution of 16 cells per square meter and .25m padding. Real-life testing proved that .25m padding was not enough. The images in this section were produced with a grid resolution of 64 cells per square meter and .37m padding.

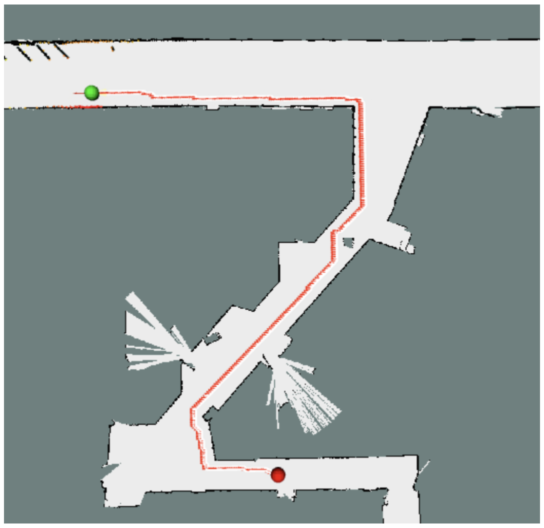


Fig. 9: This is a trajectory produced by the A* implementation before optimization. The red dashes show points on the trajectory; there are 382 points.

A* Before Optimization The choice of map representation (grid) and heuristic (euclidean distance) were largely in favor of faster runtime, but they did not account for vehicle dynamics well. This is shown through the trajectory in Fig. 3.1.1. There are frequent and large changes in direction as well as the shear number of points slow down the path following of the car. A* is optimal given the graph representation, but it is not the most optimal trajectory for the car to follow. This qualitative analysis demonstrated a need for an improvement to our path planning implementation.

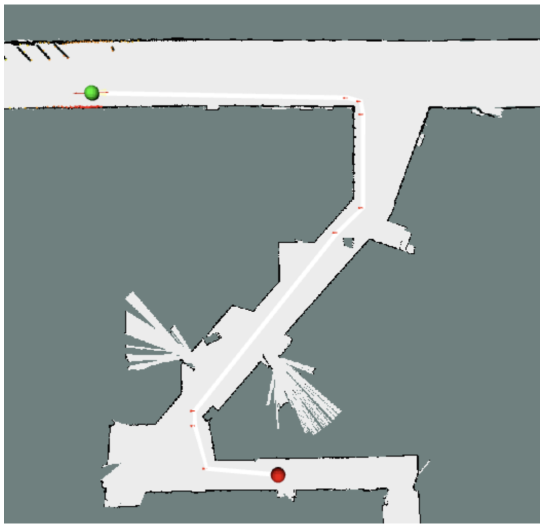


Fig. 10: This is a trajectory produced by the A* implementation after optimization.

A* After Optimization The optimized trajectory visibly has a lot less turns and points. It is able to produce paths that are not only in the grid directions.

We used a numeric metric to compare the trajectories– the sum of the euclidean distance between each point and the next point on the trajectory. Assuming that a path follower successfully traverses the points, this metric serves as a minimum distance to follow the path. The below table compares the trajectories before and after optimization (Fig.) and Fig.).

	Min Distance	# of Points
Before Optimization	53.21m	382
After Optimization	51.58m	10

TABLE I: Comparison of A* paths before and after optimization, shown in Figs. 3.1.1 and 3.1.1.

The optimization results in a smaller minimum distance to follow the path. There are also significantly less points, meaning the path follower will run quicker. We noticed a negligible increase in runtime of the path follower after adding the optimization.

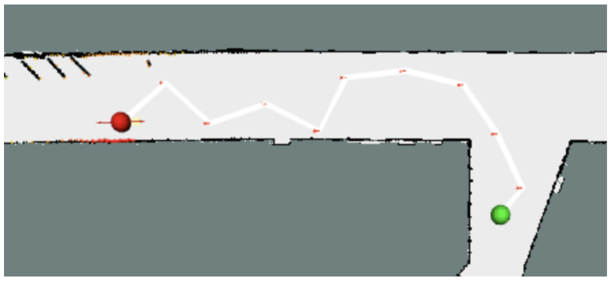


Fig. 11: This screenshot shows a trajectory outputted by RRT.

3.1.2 Performance of RRT: Fig 11 shows a trajectory outputted by RRT. The algorithm is able to find a successful path to the destination, but it is visibly not optimal. Quantitative comparison between RRT and A* will be performed in the following section.

3.1.3 Comparison of Path Planning Performance:

	Trajectory 1	Trajectory 2
A*	30.83m	22.27m
RRT	35.58m	28.75m

TABLE II: Comparison of RRT and A* paths found in two separate trajectories, shown in Figs. 12 and 13. Evidently, A* returns paths with shorter lengths than RRT does.

Figures 12 and 13 and Table II show a compare and contrast of our simulation performance of RRT and A*. We used the same numeric metric– the sum of the euclidean distance between each point and the next point on the trajectory. It is clear from these results that the A* algorithm outputs a more distance-efficient route, as shown in Table II. We chose to use A* for path planning.

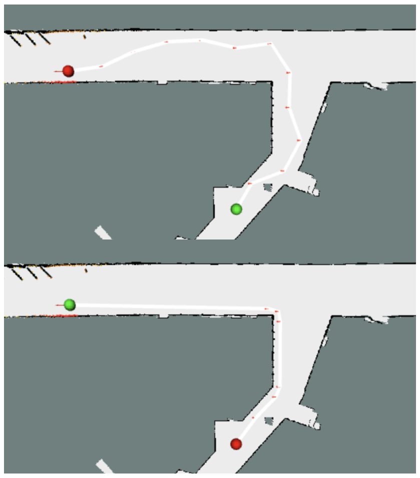


Fig. 12: (Trajectory 1) The first image shows a trajectory planned by RRT and the second shows a trajectory planned by A* given the same initial and final conditions. The red dashes represent the points in the trajectory.

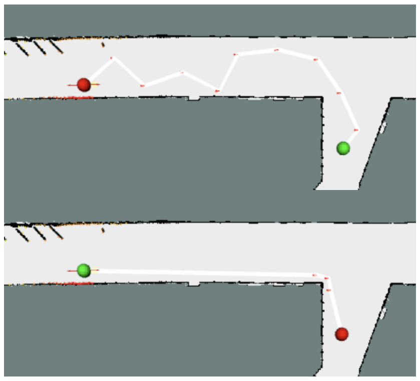


Fig. 13: (Trajectory 2) The first image shows a trajectory planned by RRT and the second shows a trajectory planned by A* given the same initial and final conditions. The RRT trajectory yields 11 points, and A* trajectory yields 5 points.

3.2 Path Following

Natalie Huang

	Closest	Linear	Spline
1	24.2583	16.838	17.37
2	24.158	16.286	16.89
3	24.455	16.584	16.56

TABLE III: The runtimes, in seconds, of the three methods for determining the pure pursuit target point. We found that the two interpolation methods had similar runtimes and were both faster than the closest point method.

We used the both qualitative and quantitative metrics to evaluate path following in sim. In RViz, we visualized the planned path and ensured that the car followed it closely. We also used runtime in simulation as a quantitative metric to compare the performance of each target point finding method. We ran each follower on the same path, which roughly corresponded with the path laid out by the staff in Stata. The speed of the follower was set at 2.0, with a lookahead of 0.8. The path we used is shown in 8.

3.3 *Testing on Real Racecar*

When we ran our planner and follower in real life, we discovered differences in performance compared to simulation.

For our A* implementation, we used a grid resolution of 16 cells per square meter and .25m padding. We deemed that .25m padding was not enough due to the car's proximity to the wall. Thus we iterated on our code to increase the resolution so that we could increase the padding and thus safety of the car.

The path-following time was significantly longer in real life. For example, the closest point approach in simulation had an average runtime of around 24 seconds, whereas the same test in real life averaged around 34 seconds. We found that though the more complex interpolation approaches worked better in simulation, the closest point approach worked best in real life and resulted in smoother driving. We believe this is due to differences in runtime, where more complex approaches do not run fast enough on the racecar to provide drive commands at a sufficiently high frequency.

Also, this testing occurred before we decided to optimize the A* path. We found that the car drove jerkily, with a lot of little turns to drive in a wide hallway. We believe this is because our A* implementation outputted a trajectory with frequent (but unnecessary) turns and many points. This testing inspired the optimization of our A* implementation by pruning points and useless turns.

A video of our racecar navigating the path successfully can be found here.

4 Conclusion

Harmanpreet Kaur

In this design phase, our team was able to successfully implement path planning and path following in our racecar autonomously. Our implementation of both A* and RRT allowed for a direct quantitative comparison of both methods in simulation. For this situation of a two dimensional space with preference for an optimal path, A* easily turned out to be the preferred algorithm, and then it was optimized to reduce unnecessary points and changes in direction. Monte Carlo Localization with a particle filter was used to localize our racecar to find the start pose. Pure pursuit control was then used on the racecar to follow the trajectories given by our path planning methods.

Before the final challenge, our team will continue to test and optimize both our optimized A* and our interpolation methods for the physical racecar, since they both successfully worked in simulation and allow for smoother path following, which will aid us in the final challenge. But regardless, our team successfully seamlessly integrated the aforementioned modules to produce our autonomous path following racecar.

5 Lessons Learned

5.1 *Andy Li*

I really enjoyed being able to apply my previous knowledge of path planning and path following, specifically splines and associated algorithms, to this class. Although I was unable to successfully implement spline path creation on the actual racecar, I was able to validate the effectiveness of my spline-based interpolation in simulation and more rigorously write out the math behind algorithms I have used in the past.

As for communication, I enjoyed working with all of my teammates. I felt that we each had our own strengths and collaborated extremely well to obtain a functional, cohesive end product technically, with everyone contributing aspects that they could be proud of and added to our overall result.

5.2 *Antonio Avila*

The technical aspect of the lab was interesting. I worked on writing RRT. I had learned about it in another class before this one, so I was eager to implement it. It turned out to be a bad way of path planning, however there are improvements to be made in the algorithm.

Communication wise I felt I communicated enough, but maybe could have been more helpful when testing. I was only able to test on the car for a few hours. Communication in the group, I thought, was good, and most people got a chance to help with the lab.

5.3 *Harmanpreet Kaur*

In this lab, it was very interesting to be able to compare path planning algorithms in real life, since most of my work before now has been purely theoretical. I feel like we did well overall incorporating the modules together as well as having a successful end result of an autonomous car that can race around the Stata basement with just the goal pose hard-coded in. It's an awfully like our capabilities as humans to be able to get to a known goal in a known map, so it was very cool to be able to implement something like this in a robot.

5.4 *Mia Tian*

I enjoyed learning and implementing A*. In the beginning, I was under the impression that the my original implementation would produce optimal results for the car. Testing showed that it was only optimal given the graph representation. I had to problem solve and think of ways to improve our design. This turned out to be fruitful because I am very happy with our final path planning implementation. Communication wise, I think the team worked well to split up individual tasks whiling stay communicative about our progress. I realized how much more difficult it is to explain on paper the technical process and results.

5.5 *Natalie Huang*

I enjoyed getting to test different combinations of path planners and followers in the lab, and feel like I learned about the implementations as well as theoretical strengths and weaknesses of different methods. I enjoyed getting to test the car in real life and it was very rewarding to see the car actually be able to navigate around the basement using our previous localization work. Communication-wise, I think the team worked well on each taking charge of different parts of the lab and combining them into a cohesive system. I think we could work on coordinating the methods of evaluation for each of our approaches earlier so we can all collect the necessary data in the process.