Lab 6 Report: Analysis of Graph of Convex Sets, RRT, and Pure Pursuit Algorithms for Optimal Path Planning and Path Following on RACECAR

Team 7

Julianne Miana
Binh Pham
Liane Xu (Editor)
Michael Zeng

Robotics: Science and Systems

April 27, 2024

1 Introduction - Michael Zeng

In this lab, our overall objective was to get our mobile robot from a start position to a target position in minimum time, while avoiding collisions. We solved this problem in two parts: optimal path planning and path following. The objective of optimal path planning is to find a "path"–an ordered set of (x, y) coordinates–for the mobile robot to follow to get from its current position to a user-defined target (x, y) position, while avoiding collisions in a predefined map, and while minimizing some user-defined cost function. The objective of path following is to compute control inputs for the robot in order to track the path. In this lab, we developed algorithms that were able to achieve both objectives successfully, in simulation, and on hardware, allowing our robot to devise complex paths and reach areas with dense obstacles at high driving speeds.

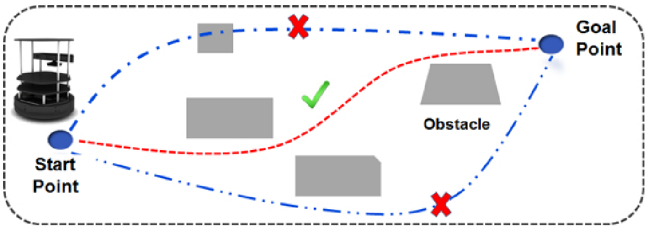


Figure 1: The problem path planning and path following: the robot must find the collision-free red path and follow it to reach the goal point.

Previously, we've worked with algorithms that utilize sensors to react to real-world events and objects; for example, using LiDAR (Light Detection and Ranging) detections and odometry data to derive the robot's position or to follow a wall. In this lab, we move from *reactive* algorithms to algorithms that solve for global optimality ahead of time. We experimented with multiple new concepts such as randomly exploring tree search, graph search, and numerical optimization in the RRT (Rapidly-exploring Random Tree), GCS (Graph of Convex Sets), and pure pursuit algorithms, and saw a new level of speed and navigational abilities unlocked on our robot.

2 Technical Approach

2.1 Overview - Binh Pham, Michael Zeng

To achieve optimal path planning, we selected two distinct methods to implement and evaluate. The first–RRT–builds paths by sampling positions in space and expanding a tree. RRT is popular for its ability to explore challenging and dense environments. The second–GCS–relies on a graph search over convex sets. GCS is effective for its formulation of globally-optimal obstacle-free path planning as a convex numerical optimization problem.

To test our path planning algorithms, we developed a closed-loop Pure Pursuit controller that allows the robot to track the planned path. We chose Pure Pursuit for its ability to smooth jagged paths produced by either RRT or GCS and correct for error.

Combining our algorithms for path planning and path following, we can get our robot to navigate from any start position to any target position within the predefined map. Figure 2 provides a simplified overview of our system architecture.

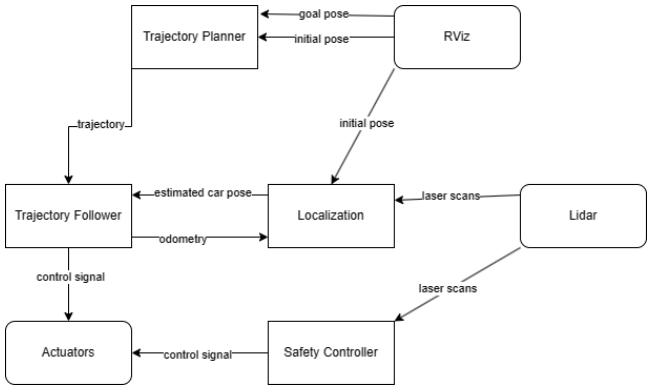


Figure 2: A diagram of our system architecture. The Trajectory Planner uses data from RViz to plan paths. The Trajectory Follower uses the trajectory it receives from the Trajectory Planner and the car's estimated pose from Localization to control the car's actuators to follow the path. Localization uses odometry and laser scans to estimate the car's pose. And the Safety Controller uses laser scans and takes control of the car's actuators to stop the car if necessary.

2.2 Rapidly Exploring Random Tree (RRT) - Julianne Miana

The RRT algorithm, known for its rapid exploration of space, uses random sampling in a map to grow a tree of nodes from a starting node until a defined goal node is reached. This tree of nodes is a set of vertices and edges that connect them. The nodes added to this tree have a parent node and an incurred path length from the start node.

RRT uses this growing tree of nodes to explore regions and find a collision-free path from a starting point to an end point. A collision-free path in this instance means a path that does not intersect any known obstacle in our map. This planner is different from a search-based planner, which constructs and utilizes a configuration space that it searches through for an optimal path between two points, usually through the use of a heuristic function to determine optimality. RRT avoids the need for a configuration space and heuristic altogether by using an exploration tree and random sampling of space to return the first found path from start to end. At a high level, the algorithm works in the following way:

1. Define a start node, a goal node, obstacles, and unoccupied spaces. These are supplied by the interactive map on RViz.

2. Randomly sample a point in unoccupied spaces.

   (a) To optimize performance, define a frequency at which the goal point is used as the sample point to help bias the growth of the search tree

towards the goal.

3. Find the nearest node to this sample point.

Figure 3: Step 1 of the RRT algorithm defines our starting and endpoints along with our obstacles and unoccupied space (lefttmost image). Step 2 randomly samples a point (middle image) and Step 3 finds the nearest node to this point (rightmost image).

4. Create a new node in the direction of this sample point at a specified max step size.
5. Check if this new node is collision-free.
   (a) If so, add this node to our tree with the nearest node as its parent.

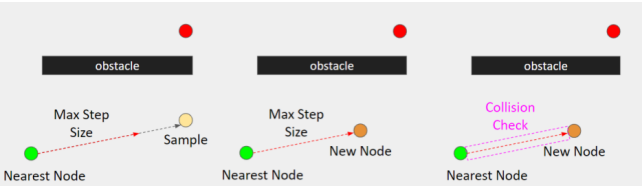


Figure 4: Step 4 creates a new node at a max step size in the direction of the sample (leftmost and middle images). Step 5 checks if the edge to the new node is collision-free (rightmost image).

6. Repeat steps 2-5 for N samples or until the goal node is added as a node to the tree.
   (a) This termination condition can be modified based on variables such as time constraints and map size.
7. Retrace steps from goal node to start node and return path.

In analyzing the algorithm's performance, four benefits of RRT make it a good candidate for path planning:

1. the algorithm is **probabilistically complete**: the algorithm will return a solution (a path from start to goal), if it finds one.

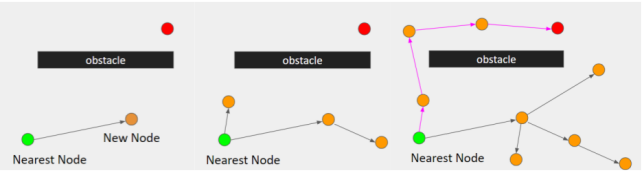


Figure 5: Step 6 extends the tree until a path is found between our start and end nodes (leftmost and middle images). Step 7 retraces the path from goal to start and returns it (rightmost image).

2. the algorithm performs only **one round of query**: as soon as the algorithm finds a path from start to goal, it immediately outputs this found path and does not need to perform another round of search.

3. **dynamics can be implemented**: physical limitations, such as sharp corners or turns, can be avoided by modifying how the tree extends an edge to a new node. A physics-based equation for position, for example, can also be used to generate the next point or vertex to add to the exploration tree.

4. sampling **eliminates the need to construct a search space**: there is no need to build a representation of the map (e.g. a grid of squares, Voronoi diagrams, etc) to search through for a path. This allows the algorithm to quickly explore regions and reduce computation time and memory space.

Given these characteristics, the algorithm is a good choice for an efficient path planner for considerably sized spaces/maps. However, this is not a perfect algorithm and has drawbacks that can be improved upon. Some important considerations are

1. **map size**: as a map increases in size, the algorithm will need more sample points to find a path from our starting position to our goal position, especially if they are far apart. Without techniques like biasing tree growth, this increase in sampling could mean increased computation time.

2. **time constraints**: depending on how the termination condition is configured, RRT can be allowed to run as long as it takes (even for an infinite time horizon) in order to find a path, which is suboptimal for computation time and in determining if a path even exists between two points.

3. **physical limitations** to path: given the Stochastic nature of how new nodes are generated and added, the paths RRT produces can be jagged or "squiggly", which is not optimal in terms of path length and ease of navigation for our car.

4. **variability**: RRT's random sampling means the same path is not guaranteed to be returned every time, which means some returned paths may be less optimal than others.

Given these drawbacks, the following modifications were made in our implementation to optimize our planner's performance in terms of path length and computation time:

1. **bias sampling** towards goal at specified frequency: we used the goal pose as our sample point 20% of the time. This helped extend the exploring tree towards the goal node faster since this gave a general direction towards which the tree should grow.

2. **sample within known unoccupied space**: we avoided sampling in out-of-bounds or occupied spaces of the map, which helped filter out unusable sample points. This made our algorithm more efficient since it only considered viable sample points in the map.

3. **increase step size**: we could extend the map faster (and therefore reach the goal faster) by increasing the distance between nodes.

4. **implement RRT***: we implemented this variant of RRT to find the minimum-length path between nodes to return an asymptotically optimal path from start to goal positions.

RRT* is a variant of RRT that returns an asymptotically optimal solution (a minimum-length path) of the path-planning problem. The key difference between this variant and RRT is that for every new node RRT* considers, the tree is rewired to keep the minimum-length path between nodes. This allows the algorithm to return an optimal path once it finds one between our starting and goal poses. Although our team implemented RRT* to further optimize RRT, we did not have enough time to evaluate its performance against our other planners. Therefore, the later section on Experimental Evaluation will be limited to RRT and our search-based planner only.

2.3 Graph of Convex Sets (GCS) - Michael Zeng

The GCS algorithm differs greatly from RRT in that it is determinstc (always returns the same path for a pair of starting and ending points) and search-based. The main principle of GCS is to formulate obstacle-free path planning as a convex optimization. This gives the algorithm the characteristics that it can always be solved quickly and to global optimality using numerical solvers. To do this, the algorithm relies on a few fundamental mathematical principles:

The first is the discretization of collision-free space into convex sets. Let's imagine an optimization problem over decision variables v (in the case of path planning, v might be a vector containing an (x, y) coordinate, for example). Then, linear constraints of the form $Av \geq b$, where the capital letter A denotes

a matrix, the lowercase letters v and b denote vectors, and v is our decision variables, are known to maintain convexity of the optimization problem. We can think of a convex set as a set of $Av \geq b$ constraints; in two dimensions, $Av \geq b$ would simply be an edge of a polygonal convex set. Requiring that a point on our path be inside of a user-defined convex set, then, is a convex constraint on the optimization.

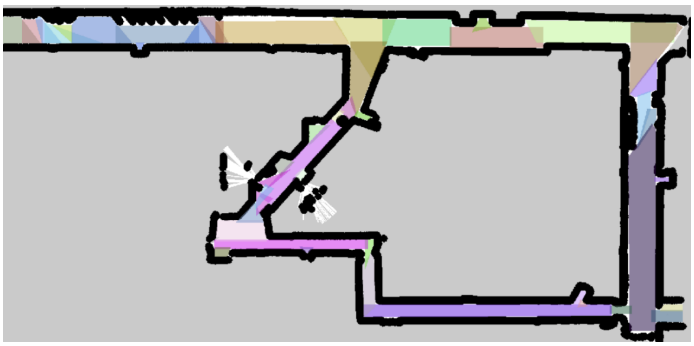


Figure 6: The discretizaton of collision-free space in Stata Basement map into convex sets. Each convex set is a different color.

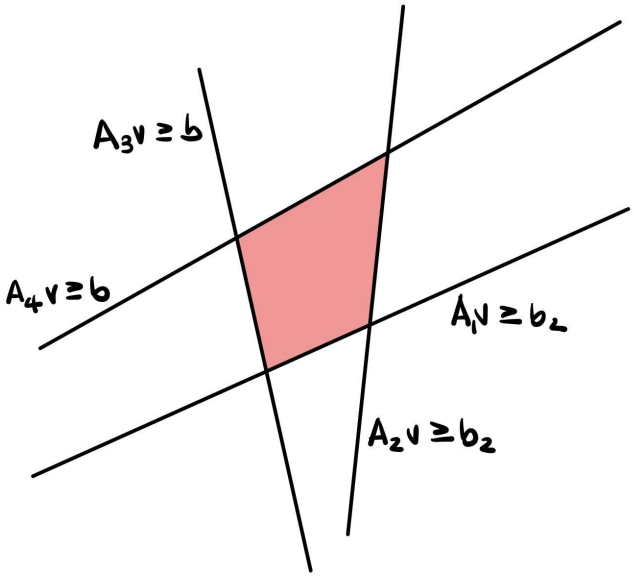


Figure 7: Representation of a 2D convex set as a set of $Av \geq b$ convex constraints.

The second mathematical principle underlying GCS is the "convex hull" property of Bezier splines. The "convex hull" property states that if all control points of a Bezier spline are inside of a convex set, then the entire Bezier spline lies within that convex set. Extending the example optimization problem described above, we can imagine that, instead of optimizing points directly on the path, we can optimize the control points of a Bezier spline. Then, if we parameterize our path as that Bezier spline, enforcing non-collision for the entire path can be done using convex constraints on a few control points. This Bezier spline parameterization of the path is beneficial because it ensuress smooth paths and reduces the number of decision variables, and therefore the solve time, of the optimization.

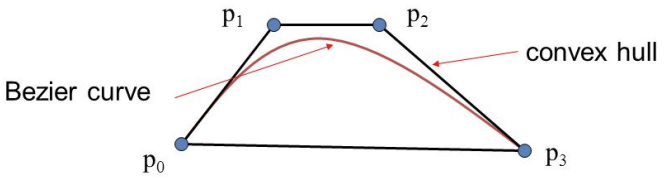


Figure 8: "Convex Hull" property of a cubic Bezier spline illustrated. If the Bezier spline were the path being optimized, then, so long as the control points of the spline are inside a collision-free convex set, the path would remain collision-free.

Now, we can see how the GCS optimization constrains a path to be collision free. But GCS must still figure out how to find a path that connects the start and target positions. To do this, GCS uses principles from graph search. Each convex set is "given" a Bezier spline. Each spline becomes a node in a graph. The start and target positions are also added as nodes. Then, edges are drawn between nodes whose convex sets overlap. The edges encode the convex set constraints, geometric Bezier-spline constraints, and equality constraints on the last and first control points of neighboring Bezier splines to enforce continuity of the path.

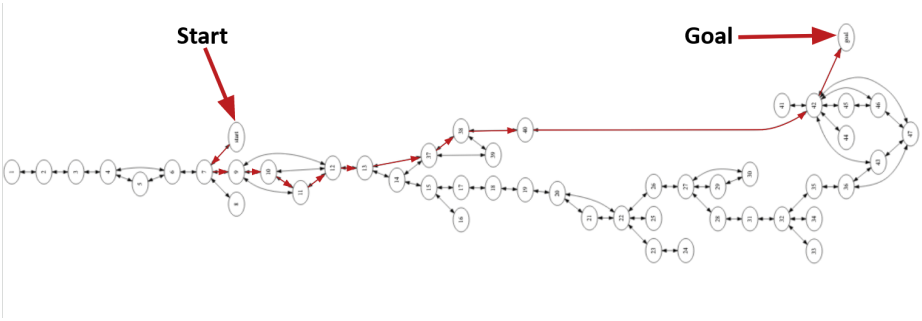


Figure 9: Example of a GCS graph, showing the connectivity between convex sets, as well as a potential path through convex sets from the start to goal positions.

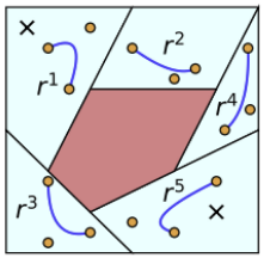


Figure 10: Each convex set is "given" a Bezier Spline, where the control points of the spline are constrained to be within the convex set.

To ensure the path is smooth when transitioning between Bezier splines, we also constrain the derivatives of Bezier splines in two adjacent convex sets at the shared control point to be equal.

Finally, to finish the formulation of the optimization problem, we add a simple cost function: path length. Since the car is not capable of going fast enough to warrant concern about losing traction, shorter paths that include wall-hugging turns are safe to follow.

The optimization is then solved using commercial numerical solvers, giving the control points for a globally-optimal (with regard to path length), collision-free path from the start to the target position.

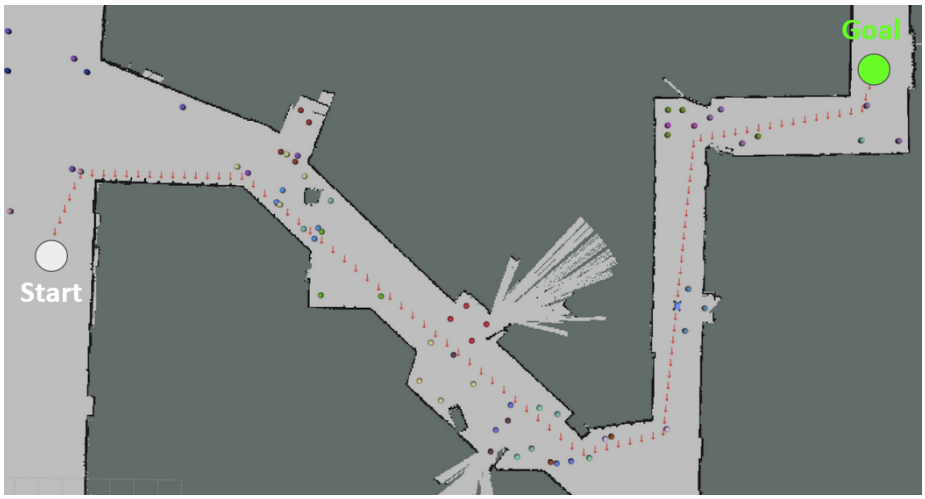


Figure 11: Result of the GCS algorithm on a given start and end position in the Stata basement map. The red arrows denote points sampled along the path.

2.4 Path-Following: Pure Pursuit - Binh Pham

Our path follower is a Pure Pursuit controller: The car follows a trajectory by continuously driving towards, or "pursuing", points on that trajectory at a pre-determined lookahead distance away.

The main advantage of using a pure pursuit controller over other controllers, such as PID, it is very easy and intuitive to tune. Despite being simple, it also works very well, even being able to smoothly follow rough trajectories.

2.4.1 Finding the Pursuit Point

The first step of pure pursuit controlling is finding which line segment contains the point the car should pursue at any given moment. Because the provided trajectory is made of line segments, we cannot just find the intersection of the lookahead circle with the trajectory right away.

Instead, we find the closest line segment to the car, excluding points whose endpoints are within the lookahead distance. This means that we must find the minimum distance to all line segments. We compute this distance by finding the projection of the car onto each line segment and finding the car's distance to that projection.

We will walk through an example for one line segment. All coordinates are in the map frame. We define u as the car's point, v as the segment's start point, and w as the segment's endpoint. First, we calculate the length of the projection

of $u - v$ onto $w - v$:

$$l := \frac{(u - v) \cdot (w - v)}{\|w - v\|}. \tag{1}$$

We then compute the proportion of this projection's length to $w - v$'s length:

$$t := \frac{l}{\|w - v\|}. \tag{2}$$

We then clamp t to be between 0 and 1 so that we find the minimum distance to a point on the line segment, not outside of it:

$$t := \max\{0, \min\{1, t\}\}. \tag{3}$$

Finally, we get the projected point and find the distance:

$$\text{minimum\_distance} := \|v + t(w - v)\|. \tag{4}$$

We exclude line segments whose endpoints are within the lookahead distance, i.e. $\|w - u\| <=$ lookahead, from being chosen as the "closest" line segment. We exclude these line segments by adding infinity to their minimum distances. We exclude these line segments because, as illustrated by Figure 12, if we are near the end of the line segment, we want to move onto the next line segment in the path.

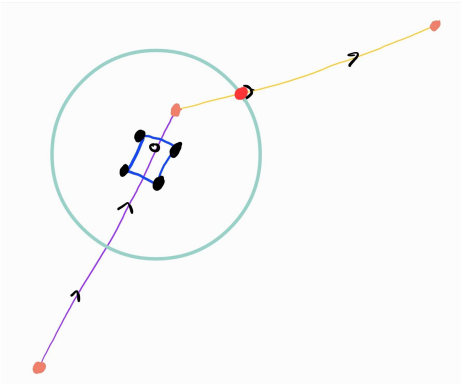


Figure 12: A small example where the closest line segment's endpoint is within the lookahead circle. The closest line segment is the purple line. However, the car is near the end of the purple line and should start pursuing points on the orange line instead. To achieve this, we exclude the purple line from being considered as the closest line segment since its endpoint is within the lookahead circle.

Finally, the line segment with the minimum minimum_distance is chosen as the closest line segment.

Once we find our line segment of interest, we need to find its intersection with the lookahead circle. That intersection will be the point our pure pursuit controller will pursue. Once again, we accomplish this with vectors. We know that the point is on the line, and its distance to the car is the lookahead distance. Let ℓ be the lookahead distance. Thus, we have the equation:

$$\ell = \|v + t(v - w) - u\| \tag{5}$$

We need to solve for t. We can manipulate this equation into a quadratic equation where the coefficients are:

$$a = \|w - v\|^2 \tag{6}$$
$$b = 2((w - v) \cdot (v - u)) \tag{7}$$
$$c = \|v\|^2 + \|u\|^2 - 2v \cdot u - \ell^2. \tag{8}$$

Now we can use the quadratic equation to solve for t:

$$t = \frac{-b + \sqrt{b^2 - 4ac}}{2a}. \tag{9}$$

We use the larger solution for t because we want the point that is further along the path. If we used the smaller solution, our car would be going backwards in the path. Finally, we use t to get the point of intersection:

$$\text{target\_point} := v + t(v - w). \tag{10}$$

Note that is we cannot solve for a real value of t, this means that there is no intersection. In this case, we choose the target point to be the endpoint of the line segment.

2.4.2 Controlling Steering and Speed

All coordinates are in the car's frame for this section. To control the steering angle, we use the generalized instantaneous pure pursuit steering law presented in lecture 5:

$$\delta := \arctan \frac{L \sin \theta_{\text{target}}}{\ell/2 + L \cos \theta_{\text{target}}} \tag{11}$$

where δ is the steering angle, L is the wheelbase length, ℓ is the lookahead distance, and θ_{target} is the angle to the target point from the car. L is 0.3m. We use the target's coordinates in the car's frame to get θ_{target}:

$$\theta_{\text{target}} = \arctan \frac{y_{\text{target}}}{x_{\text{target}}}. \tag{12}$$

We vary ℓ, the lookahead distance based on θ_{target}. Specifically, these two values are inversely proportional:

$$\ell = \ell_{\max} - \frac{|\theta_{\text{target}}|}{\theta_{\max}} (\ell_{\max} - \ell_{\min}) \tag{13}$$

where $\ell_{\max}$ is the maximum possible lookahead distance, $\ell_{\min}$ is the minimum possible lookahead distance, and $\theta_{\max}$ is the angle to the target at which the lookahead distance will be at its minimum. |target| is also clamped to be between 0 and $\theta_{\max}$ in equation 13. Finally, to control the speed, which we will call v, we make it proportional to the lookahead:

$$v := k\ell \tag{14}$$

where k is the ratio between speed and lookahead. $\ell_{\max}$, $\ell_{\min}$, $\theta_{\max}$, and k are values we manually tune. On the real car, we set

$$\ell_{\max} := 2.0 \tag{15}$$
$$\ell_{\min} := 1.0 \tag{16}$$
$$\theta_{\max} := \frac{\pi}{2} \tag{17}$$
$$k := 1.25. \tag{18}$$

In simulation, we set k to be 2, since we can actually go up to 4m/s in simulation. On the real car, the speed limit is somewhere above 2m/s and below 2.5m/s.

We vary lookahead inversely with the angle to the target because when we make a turn, a high lookahead leads to corner cutting, as illustrated in Figure 13. Additionally, we vary speed directly with lookahead, which is the same as varying speed inversely with the angle to the target because if we are going straight we want to go fast. And if we are turning, we want to slow down to follow the path more accurately.

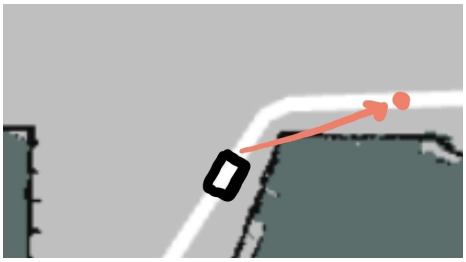


Figure 13: An illustration of the car making a turn with high lookahead. The white line is the followed trajectory. The red arrow represents the lookahead and the red point is the target point. We can see that the car will cut the corner with a high lookahead and hit the wall. To fix this, we make the lookahead shorter by making it inversely proportional to the angle to the target point.

2.5 Localization Tuning for the Real Car - Binh Pham

In the previous lab, we had localization issues on the real car. First, our estimation in position was slightly off. Second, sometimes the particles would converge outside the map. Since we were testing on higher speeds for this lab, these issues became worse. To fix these issues, we changed the way we added noise to the resampled particles. For each resampled particle's x-coordinate, we drew the added noise from $\mathcal{N}\left(0.015\, v_{\text{car}}\, \cos\left(\theta_{\text{car}}\right), \left(0.05\, v_{\text{car}}\right)^2\right)$. For each resampled particle's y-coordinate, we drew the added noise from $\mathcal{N}\left(0.015\, v_{\text{car}}\, \sin\left(\theta_{\text{car}}\right), \left(0.05\, v_{\text{car}}\right)^2\right)$. And for each resample particle's angle, we drew the added noise from $\mathcal{N}\left(0, \left(\frac{\pi}{30\, v_{\text{car}}}\right)^2\right)$. v_{car} is the speed of the car.

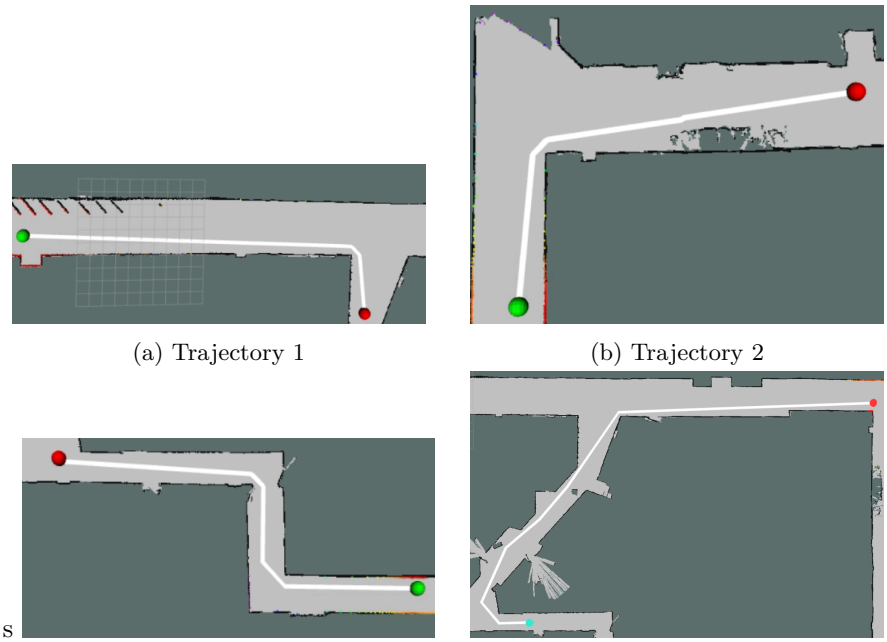
Because we made the mean non-zero for both x and y, we moved each particle in the direction of the car every time we added noise. This fixed the underestimation of distance. Previously, we only had the mean for the noise added to x be non-zero, which was geometrically incorrect because the coordinates are in the map's frame, not the car's frame.

Additionally, we made the standard deviation of the noise added to the angle be inversely proportional to the speed of the car. The reasoning behind this is that when the car is moving fast, it is likely going straight.

3 Experimental Evaluation

3.1 Path Following - Binh Pham

To evaluate our pure pursuit trajectory follower, we used four pre-built trajectories as shown in Figure 14. We plotted the car's distance to the trajectory as the car followed each trajectory and recorded the average distance over the entire trajectory. We calculated this distance by finding the distance of each line segment in the trajectory as described in Section 2.4.1 and taking the minimum distance. Of course, we conducted these tests entirely in simulation as real life testing would have additional error from localization. Table 1 summarizes our results.



(a) Trajectory 1

(b) Trajectory 2

s

(c) Trajectory 3

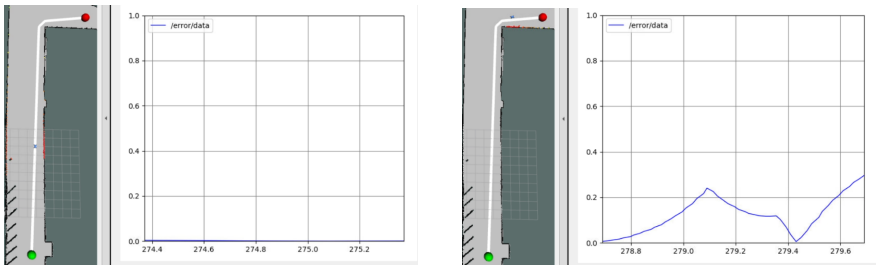
(d) Trajectory 4

Figure 14: The four pre-built trajectories we will test on the pure pursuit controller. The white line is the trajectory, with the green point marking the starting position and the red point making the ending position. The staff provided trajectories 1-3, and we built trajectory 4 to cover the Stata track.

Table 1: Distance error values in meters (rounded to the nearest thousandth) for 4 different trajectories.

Trial	**Trajectory 1**	**Trajectory 2**	**Trajectory 3**	**Trajectory 4**
1	0.051	0.074	0.123	0.059
2	0.053	0.079	0.122	0.059
3	0.055	0.079	0.124	0.059
Average	0.053	0.077	0.123	0.059

The car performed best on trajectory 1, with an average distance error of 0.053m, and worst on trajectory 3, with an average distance error or 0.123. Examining the trajectories and comparing it to their average distance errors, we can qualitatively see that the follower performs best when the trajectory and straight and worse when the trajectory has turns. Trajectory 1 is mostly a straight line with a single turn at the end, while trajectory 3 has consecutive turns. We can also examine the plots to quantitatively see that the car performs worse during turns, as shown in Figure 15.



(a) Straight Segment (b) Turn Segment

Figure 15: The distance error plots of two different segments of trajectory 1. The plot for the straight segment shows a distance error near 0. The plot during the turn segment shows a much higher error.

We can further tune our pure pursuit parameters to decrease the error during turns. However, we determined that this is not necessary as it follows turns well enough. Tuning the car to follow turns too closely can actually be a negative. For example, if a trajectory is really jagged, we do not want to our follower to "overfit" it by following it too closely: we want the car to smoothly follow it. Overall, our pure pursuit controller follows paths well.

3.2 Path Planning - Liane Xu, Michael Zeng

To evaluate our GCS and RRT implementations, we determined three scenarios with different starting and ending points:

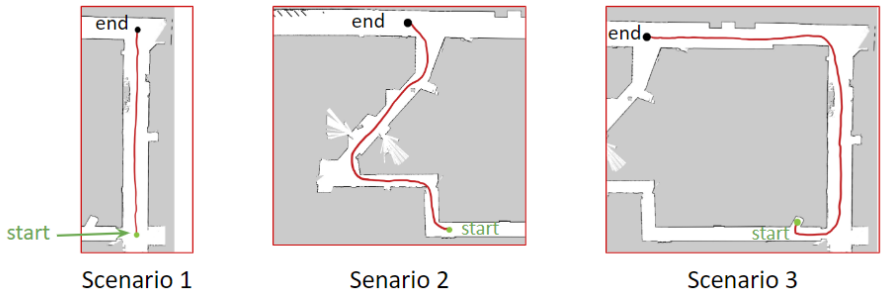


Figure 16: Scenario 1 (no turns between start and end), Scenario 2 (multiple tight turns between start and end), Scenario 3 (multiple turns). The green dots are the starting positions and the black dots are the end positions. The red line represents a possible path from start to end.

To plan the paths, we ran GCS and RRT for each scenario. Since GCS is deterministic, it returns the same path every time it runs, so we only ran it once per scenario. On the other hand, RRT is a randomly-exploring algorithm, so

we ran three trials for each scenario. Then, we ran the pure pursuit controller on a subset of the resulting paths.

To quantitatively compare RRT and GCS, we devised four metrics: computation time to generate the path, path length, time to completion for the robot to follow the path from start to finish using pure pursuit, and the collision rate (if the path ever causes the car to collide with an obstacle). The results are recorded in 2.

Table 2: Quantitative metrics compared between RRT and GCS. Values for RRT are averaged across three trials.

	Scenario 1 GCS	Scenario 1 RRT	Scenario 2 GCS	Scenario 2 RRT	Scenario 3 GCS	Scenario 3 RRT
Computation Time (s)	4.61	8.08	4.79	91.50	4.44	45.10
Path length (m)	33.68	34.46	52.18	81.35	73.61	83.82
Time to Completion (s)	8.50	8.60	12.59	20.20	18.20	21.30
Collision-free Completion Rate (%)	100%	100%	100%	100%	100%	100%

3.2.1 Scenario 1 results

Visually, RRT (Figure 17) and GCS (Figure 18) differ in their straightness. Whereas RRT produced paths that were more squiggly due to its exploratory nature, GCS is optimized for path length so it produces a straight line from the start to the end. The RRT path tended to straighten out toward the end because our new-node sampling was biased toward the end node.

Therefore, it makes sense that the path lengths and time for the car to complete the path were similar (Table 2).

3.2.2 Scenario 2 results

For Scenario 2, RRT (figure 19) created paths that went in two different directions. Whereas RRT Trial 1 went to the left, Trials 2 and 3 went to the right. Trial 1 took the longest to compute (Table 2) because it is expanding a tree in a hallway that has many turns, where there is low probability for nodes to be randomly selected in a way that grows the tree. Overall, GCS was able to compute the path in 4.79 seconds, while it took an average of 91.50 seconds for RRT to compute its paths.

3.2.3 Scenario 3 results

For Scenario 3, both RRT and GCS produced paths that went counterclockwise. It should be noted though that RRT could produce a clockwise path such as the one from Scenario 2, but it is probabalistically unfavorable, so it did not appear in the three trials that were run. Again, GCS had a faster compute time.

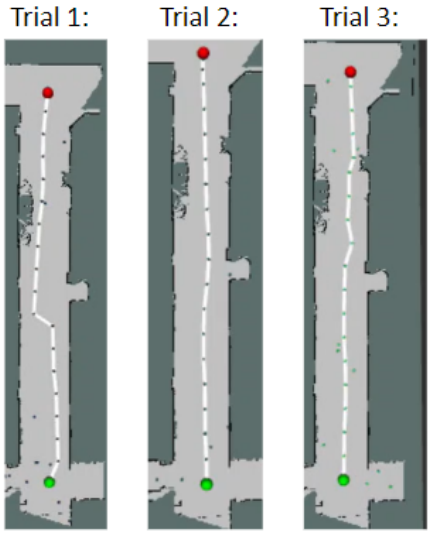


Figure 17: Three paths created by three separate runs of RRT with scenario 1 start and end points.

Figure 18: Path created by GCS with scenario 1 start and end points.

Figure 19: Three paths created by three separate runs of RRT with scenario 2 start and end points.

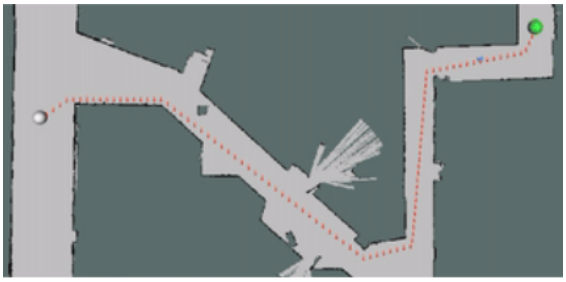


Figure 20: Path created by GCS with scenario 2 start and end points.

Figure 21: Three paths created by three separate runs of RRT with scenario 3 start and end points.

Figure 22: Path created by GCS with scenario 3 start and end points.

3.3 Integrated Testing on Real Car - Binh Pham, Liane Xu

For testing on the real car, we ran the car on the marked track in the Stata basement, illustrated by Figure 23. This would put our path planning, path following, and localization to the test.

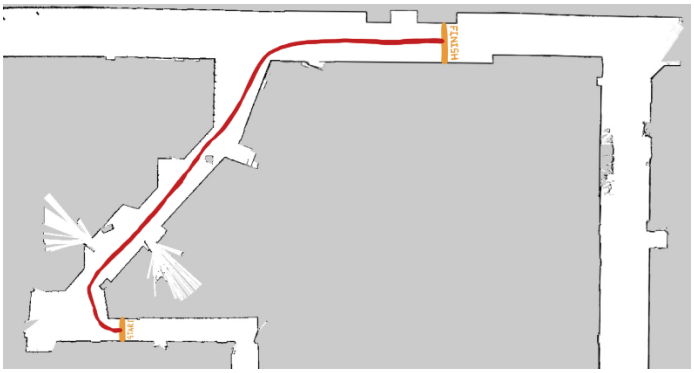
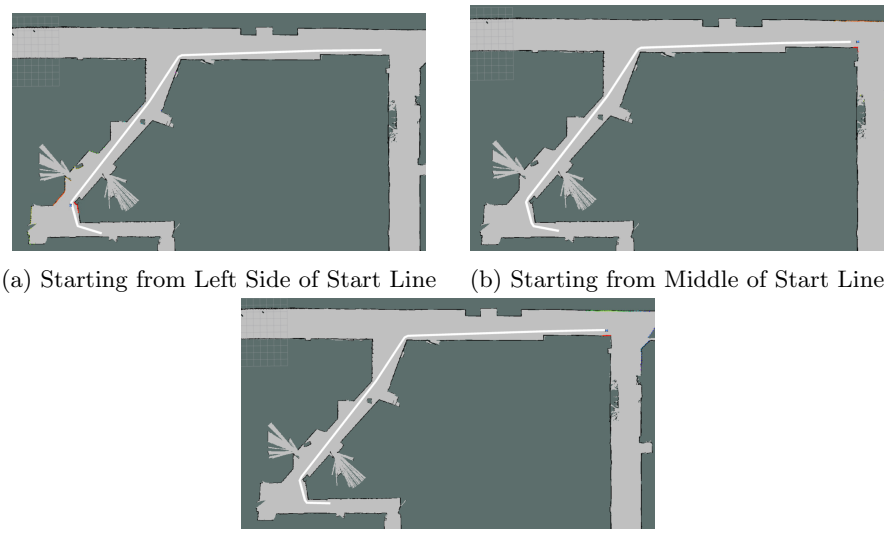


Figure 23: A rough illustration of the marked track in the Stata basement. The start line is near the bottom of the figure and the finish line is near the top of the figure.

We created three different paths using GCS that start from different points on the starting line, as shown in Figure 24. We placed the car's front wheels of each of the 3 starting positions: left, middle, and right. Then we had the car follow the corresponding path. We ran the right starting position twice because our first run had localization issues that led to the car understeering the first turn. Timing started when the car moved and ending when one of the wheels crossed the finish line. We timed the runs by overlaying a video stopwatch and syncing the start of the timer with the start of the car's movement. We then found the time by going frame-by-frame to find when the car's wheel crossed the

finish line. These videos can be found here: `https://youtu.be/9iEJiYGdTYw?si=qS7dvnY6KL6tNMef`. The times are recorded in Table 3.



(a) Starting from Left Side of Start Line (b) Starting from Middle of Start Line

(c) Starting from Right Side of Start Line

Figure 24: The three different paths planned by GCS to complete the Stata track. Each path starts from a different starting position on the start line.

Table 3: Stata track times from different starting positions.

Starting Position	**Time (s)**
Left	23.49
Middle	23.18
Right	23.59
Right (Retry)	23.34
Average	23.40

Overall, the times are very consistent, no matter the starting point. With the exception of the first run starting from the right side of the start line, the car followed the planned trajectory very closely. We are satisfied with these times as the path is fully optimized and the path following has no noticeable flaws. The only impactful improvement we can think of would be to increase the speed limit on the car. Other than the car's consistency on the track, it is hard to draw other conclusions about the performance of our car quantitatively, without a comparison to a different team's track times. We hope that our team will win the best track time, which would quantitatively show that our localization, path planning, and path following work very well together on the real car.

4 Conclusion - Julianne Miana

In this lab, our team successfully implemented two path planners and a path follower to get our car to follow a predetermined path in the Stata basement. The two path planners we implemented and tested are RRT (Rapidly-exploring Random Tree) and GCS (Graph of Convex Sets). RRT, a sample-based approach to path planning, utilizes random sampling of points in space to grow a tree of nodes and find a collision-free path between user-defined starting and end points. GCS, on the other hand, is a search-based and deterministic approach to path planning that formulates collision-free path planning as a convex optimization problem. This enables quick solve time and guarantees a globally optimal solution (a smooth minimum-length path between our starting and end points). While both planners are time-efficient, each makes a tradeoff between optimal returns, and memory and simple computation: GCS takes on heavier computation to guarantee optimal returns while RRT gives up optimal solutions for faster exploration and simple computation.

Further tuning on our localizer and Pure Pursuit controller was also done to improve our path-follower. On our localizer, we modified the way we added noise to our sampled particles' position and angle to fix an underestimation issue and to improve performance at high speeds. For our Pure Pursuit controller, we modified our lookahead distance calculations to avoid cutting corners and improve the accuracy of our path following. Our improved path-following program was then combined with our planners to assess our path-planning performance.

We tested our planners both in simulation and on our physical car. In simulation, we used three pairs of starting and ending points to evaluate performance in different sections of the Stata basement. As metrics, path-generation computation time, path length, path-following completion time, and collision rate were utilized. Our results indicate that GCS is the more optimal path planner in terms of computation time and path length. It also produced paths that led to faster path-following by our car, but both planners were successful in producing collision-free paths. Given these results, GCS was the planner implemented into our physical car, along with our localizer and controller.

Experimental evaluation on our real car showed consistent and accurate path-following. Using the provided Stata track with defined starting and finishing lines, we ran GCS, which produced three paths. We then tested our car's path-following capabilities on these three paths and observed consistent track completion times for an overall average of 23.40 seconds. Overall, we are satisfied with this consistent tracking as it is indicative of how well our localizer, controller, and path-planner work together on our car.

As we look ahead to the final challenge, we would like to implement modifications to further optimize path-planning and path-following performance. One modification is to increase the step size in RRT to reduce computation time and

produce less jagged paths. We would also like to test our RRT* implementation and determine how it compares to GCS. For GCS, we also aim to reduce its computation time below 4 seconds. An additional capability that we aim to implement is dynamic or real-time planning to account for moving obstacles.

5 Lessons Learned

5.1 Julianne Miana

Lab 6 was technically challenging but ultimately rewarding. I thought that tuning our working RRT planner and implementing RRT* was fun and taught me a lot about RViz visualization tools and implementing a well-known planning algorithm. Because I ran into some issues with implementing RRT, getting our RRT* algorithm to work was delayed and we did not have enough time to test RRT* with GCS and RRT. I therefore hope to work on this for our next lab as we optimize our path planner. In terms of collaboration, working on separate modules that we could later integrate helped us divide our work and make progress outside of lab sessions. I think that this also allowed each of us to focus on and improve our skills/knowledge in a specific section of the lab, which we could then share with the others once we were ready to integrate our modules.

5.2 Binh Pham

Lab 6 was very fun. I worked on the pure pursuit controller, which was relatively easy compared to path planning. I worked on the physical car mostly after finishing pure pursuit, tuning localization and the safety controller. It was very satisfying to see the car complete the full track on its own in real life. I do wish I worked more on the path planning though since it seemed more technically challenging, and I hope to get that opportunity in the final challenge as our current algorithms seem to have suboptimal computation times.

5.3 Liane Xu

I found Lab 6 to be fun because there was a well-defined problem (path planning) but there were a lot of ways to approach it. It was rewarding to implement RRT from scratch. There were a lot of components, including creating a data structure to manage the growth of the tree, collision detection, map dilation, visualization, adding new nodes, and overall optimizing the algorithm. I'm glad that our pure pursuit controller can handle wiggly paths.

5.4 Michael Zeng

Lab 6 definitely exemplified the importance of accurate localization. Even when we had highly functional path planning and path following algorithms that worked perfectly in simulation with ground truth odometry, this fell apart

quickly where even small localization errors in real life would compound. Also, computation time was shown to be both very important and difficult to optimize. We had trouble getting both our RRT and GCS to run quickly even though both should be relatively fast algorithms. We will continue improving runtime into the final challenge. Regarding collaboration, this lab definitely showed the power of ROS, where we could work on separate modules that integrate seamlessly with almost no effort, as long as we pre-define the publishers and subscribers.

References

[1] T. Marcucci, J. Umenberger, P. A. Parrilo, and R. Tedrake, 'Shortest Paths in Graphs of Convex Sets', *arXiv [cs.DM]*. 2023.