Control along a path in Stata Basement using Path Planning and Pure Pursuit Algorithms

Ayden Johnson Erin Menezes Gregory Pylypovych Jose Ramirez Gloria Zhu

I. Introduction (Ayden)

Path planning is the process of connecting two points in a given space with a path, such that it's possible to move from the first point to the second by moving along the path. A very important problem is to find the path of minimum cost between a pair of points, where cost may be defined differently based on the application. The problem arises in many different forms, and in robotics it often manifests as a navigation problem: the cost of a path is simply its length, and the shortest path must be found from one location to another. Autonomous vehicles, such as self-driving cars and warehouse robots, use this type of path planning to move towards a destination while consuming as little energy (e.g. battery power or fuel) as possible, or to reach the destination within a deadline. Important aspects to consider about solutions to this problem are how close the calculated path is to a theoretical optimal path, and how much computational resources are needed to calculate this path.

The goal for this lab is to create the final module in the racecar's autonomous control, which includes methods for creating an efficient path from the racecar's position to a specific goal location, and having the racecar drive along this path as closely as possible. This module will allow to robot to navigate an environment with known obstacle locations, moving from one position to another as quickly as possible without bumping into anything. This module will play a significant role in the final challenge, specifically the objective of navigating a city setting without collisions.

II. Technical Approach

A. Path Planning (Jose, Ayden) (Ayden, Jose)

The first step of the lab was to decide which algorithm to use to design the path that the robot would take to navigate to its final destination. The algorithm would need to take in a start point, end point, and the map, and then output a path between the start and end points without leaving the map area. We decided to focus on three algorithms: BFS, A*, and RRT.

BFS (Breadth First Search) was a good starting point, as its implementation is simpler than the other algorithms, and would lay the groundwork for A*. BFS is a search-based planning algorithm; this means that it uses graph-search methods to find the best solution. In order to use those methods, the map must first be translated into nodes and edges. This was implemented by grouping the map pixels into square coordinates. The robot is given a starting coordinate and an end coordinate and then searches outward equally in all directions (Fig 1). It does so by keeping a list of coordinates that it has not checked yet. It grabs the first node, and checks if it is the endpoint. If not, it finds the neighbors of the current node and adds those to the end of the list of nodes to check. This ensures that nodes closer to the starting point are checked first, as those that are farther away are added to the end of the list, and thus wait more before being checked. The path is discovered by keeping a record of what node each node was reached from.

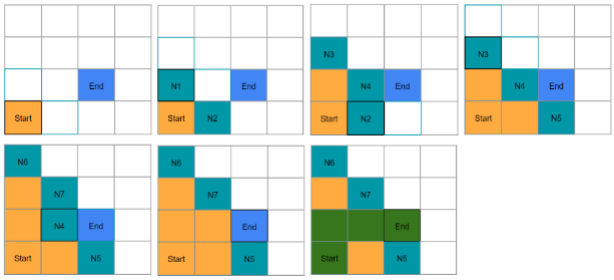

Fig. 1. Diagram of the BFS algorithm. Orange tiles have already been checked, teal tiles are going to be checked, and the green tiles are the final path. The black outline is around the tile that is currently being checked, and the numbers on the tiles indicate the order in which the tiles will be checked.

A* was the logical next step, as it could use the grid designed for BFS, but would add a heuristic that pushes the discovery of paths toward the end node (Fig 2). By keeping track of the cost required to get to a certain node, and by estimating the cost required to get from that node to the end, A* can choose nodes that supposedly lead the path closer to the desired endpoint. We chose to use the Euclidean distance as our heuristic. This is because we knew that the location that our robot would be in (the Stata basement) does not have many obstacles that completely block paths. As such, we could aim directly for our final point with the help of our heuristic.

The last algorithm considered was RRT (Rapidly-Exploring Random Trees). This algorithm has the benefit that translating the map into a grid of nodes and edges is not necessary for it to find a path. Instead, RRT iteratively builds a tree of points in the map by selecting a point at a random location, identifying the closest point currently in the tree to this random point, and adding a new point to the tree on the line segment between the closest point and the random point, if possible.

Occasionally, instead of a random location, the algorithm will select the goal location and add a point to the tree that extends the tree towards the goal. Eventually, the tree will

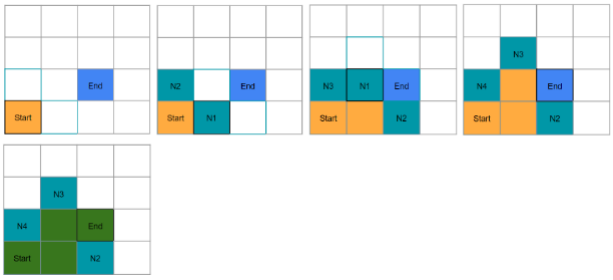

Fig. 2. Diagram of the A* algorithm. Orange tiles have already been checked, teal tiles are going to be checked, and the green tiles are the final path. The black outline is around the tile that is currently being checked, and the numbers on the tiles indicate the order in which the tiles will be checked.

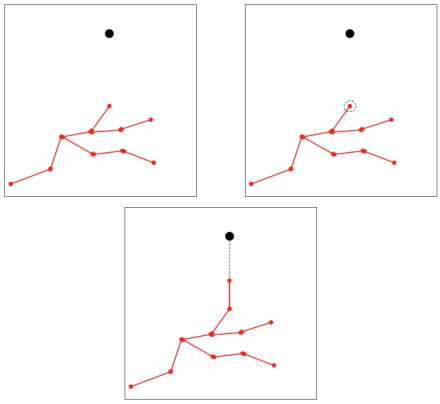

Fig. 3. The process of extending the tree (in red) towards a random point (in black). The tree is extended from the closest point in the tree to the random point, circled in the second image.

reach the goal location, and the algorithm will build a path from the edges of the tree between the start and goal locations.

Because the tree extends towards any location from the closest point in the tree, the paths that RRT generates are usually reasonably efficient, and they can be generated relatively quickly, although the paths may not be completely optimal.

B. Pure Pursuit (Gloria, Erin)

Given a planned trajectory in the form of a list of points, our pure pursuit algorithm serves to actually drive our vehicle and accurately follow the path. The advantage of pure pursuit is that it is both intuitive and straightforward to implement, but it suffers in that it is unable to exactly follow every trajectory, depending on how parameters are tuned.

The pure pursuit algorithm consists of two steps:

1) Find the next target point (gx, gy) for the car
2) Determine the corresponding steering angle to arrive at that point

Both steps are continuously updated at every time step, ensuring that the vehicle is responsive to changes in the path.

The first step of the Pure Pursuit algorithm is to find the immediate target point of the vehicle. This point is determined by looking ahead a specific distance L around the car and finding the intersection of this radius with the trajectory (Fig. 4).

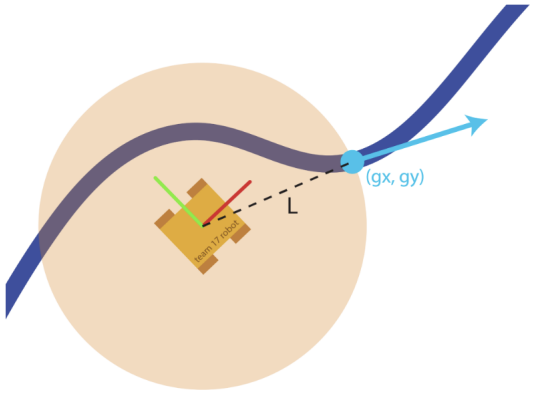

Fig. 4. The point that the vehicle uses to calculate its steering angle towards is always a point on the path that is L meters from the car, where L is the pre-defined look ahead distance. If there are multiple points that fit this description, then the point that is in front of the car is chosen to be the target point.

At a more granular level, we accomplish this first by finding the segment of the trajectory that is closest to the robot. Because the trajectory is composed of a series of straight lines, we simply iterate over each segment and find its perpendicular distance from the car. Once we have the closest segment, we check if the look ahead radius intersects with the segment. If not, we move down the trajectory until we find a segment that does intersect. The resulting point is then our target (gx, gy).

Note that in most cases, the look ahead radius actually intersects with the trajectory in two points, so we must ensure that we are targeting the point in front of the vehicle rather than behind it. If the car is too far away from the trajectory, it will not be able to see it at all, and will remain stationary.

Once the target point is determined, the next step is to determine the corresponding steering angle of the car. The steering angle is determined by how far the car is from the point, which in this algorithm will always be the look ahead distance, and by the relative location of the goal point in the car's coordinate frame (Fig. 5).

A simple formula, in Equation 1 was used to find this angle, which was directly inputted into the vehicle's drive publisher.

$$steering\_angle = \frac{k * 2 * \mid gy \mid}{look\_ahead^2} \tag{1}$$

where k is an experimentally determined constant.

Once we have our angle, we publish it to the car along with a constant drive velocity v. This value is updated every time step to ensure that the car is always updating itself to track the trajectory.

For our implementation of pure pursuit, we are able to tune four parameters:

1) L (the look ahead distance)
2) k (the steering angle tuning constant)
3) v (the constant speed of the racecar)

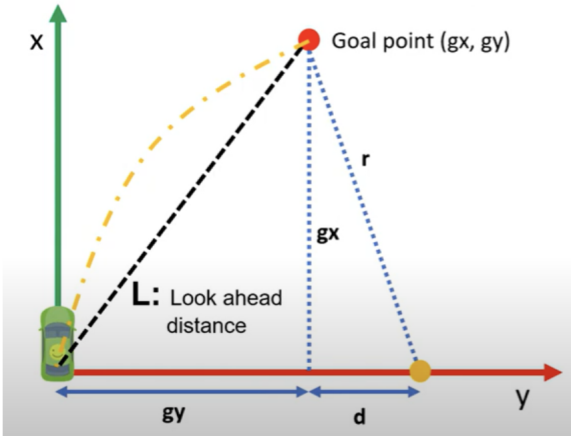

Fig. 5. The location of the goal point and the car in the world frame was used to determine the location of the goal point relative to the car. This point is (gx, gy). Using a predefined look ahead distance, L, the steering angle of the car to get to the goal point was defined as in Equation 1.

4) f (the frequency at which the racecar's steering angle is updated)

By adjusting these values, we are able to control the amount of oscillation in the vehicle, its ability to handle non-smooth paths, and the level of fidelity the racecar can maintain to the true trajectory. In the next section, we discuss experimental methods we took to optimize L and v. For k and f, however, we found that a constant value of k = 1.0 and f = 20Hz worked well for our applications.

As a note on the frequency, the complexity of our pure pursuit algorithm varies depending on the number of segments in the trajectory. For highly complex trajectories, it suffers in that it needs to compute perpendicular distances from every single segment to the car at every time step. A possible optimization in this case would be to keep track of the nearest segment from the previous time step and search outwards from that index at the next step. However, as we were mainly dealing with sparse trajectories (¡50 segments), this was not an issue. Thus, we are able to increase the frequency of publication up to about 30Hz. However, we found that the limiting factor for our system's responsiveness was actually the rate at which the racecar's odometry updates, which actually tells us where the racecar is located in the world. Because this bottleneck value updates at about 10Hz, increasing our pure pursuit frequency further would not affect the performance of our system, so we chose to keep it at around 20Hz for all testing purposes.

III. Experimental Evaluation (Erin)

The Path Planning and Pure Pursuit algorithms were evaluated using various metrics to demonstrate their effectiveness and/or point out areas to be optimized in a future iteration.

A. Path Planning (Jose, Ayden)

The path planning algorithms were initially compared via their asymptotic runtimes. The BFS algorithm has a runtime in $O(N + E)$, where N and E are the nodes and edges of the graphical representation of the map, respectively. The A* algorithm has a runtime in $O(E{*}\log(N))$, and the runtime of RRT is in $O(P{*}\log(P))$ for the number of points in the tree P. Based on this, the A* algorithm was selected as the best algorithm for the path planning module, because of its low asymptotic runtime and its accuracy in producing optimal paths.

B. Pure Pursuit (Erin, Gloria)

The Pure Pursuit algorithm was evaluated by how well it controlled the vehicle along a given path. This error value was calculated by finding the distance between the vehicle's position and the closest point on the path to the vehicle. The error was measured across two parameters: the vehicle's speed and the look ahead distance in our algorithm. The speed was varied in increments of 0.5m/s between 0.5m/s and 4.0m/s and the look ahead distance was varied in increments of 0.25m between 0.25m and 1.0m.

Thirty-two data sets were collected, one for every speed and look ahead distance combination across an extensive path in the Stata basement (Fig. 6). The path included straight and curvy segments, as well as tight turns that would test the algorithm's ability to control the vehicle and keep it on top of the path.

Fig. 6. The path through the Stata basement that was used for testing the Pure Pursuit algorithm. The path includes various turns and corners.

Tighter turns and curves were expected to create errors since the vehicle is veering off the path and then correcting for its overshoot. Since the error was strictly a distance from the path and thus always positive, a sinusoidal settling function would look like strictly positive bumps rather than a normal sine wave centered around an error of zero. As in Figure 7, the error spikes at the first of the two tight turns, but interestingly is much lower on the second tight turn. Additionally, the car is quite steady on straight paths, and just slightly goes off path on the curve.

It was observed that slower speeds tended to produce less error on average along the entire path (Fig. 8 left). This was as expected, since the algorithm was publishing a new steering angle at a specific frequency, which meant that a

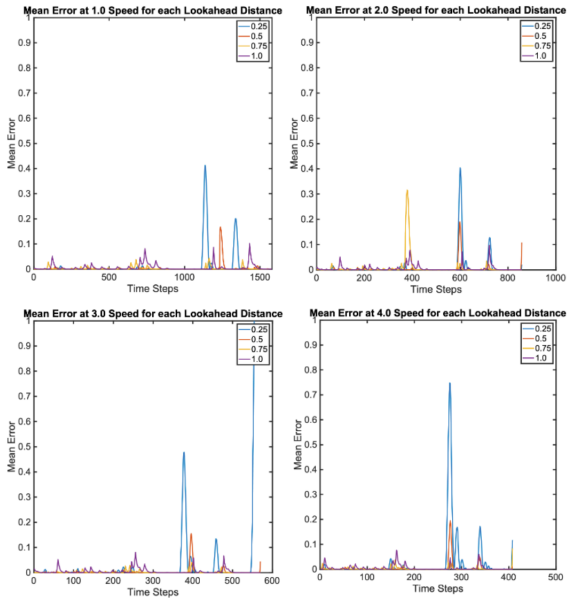

Fig. 7. Error plots over time for various speeds and look ahead distances. The peaks occur at the tight corners, where the vehicle is veering off the path.

faster speed resulted in fewer published speeds across the same path. Thus, slower speeds allowed the algorithm to adjust the vehicle's steering angle more times throughout the drive, which naturally led to smoother and more accurate control and path following.

Unlike the simplicity of the speed to error relationship, the look ahead distance's relationship with the mean error was not as simple to predict. The data, though limited on data points, seems to follow a quadratic best fit line, where the minimum sits between 0.5 and 0.75m of look ahead distance (Fig. 8 right).

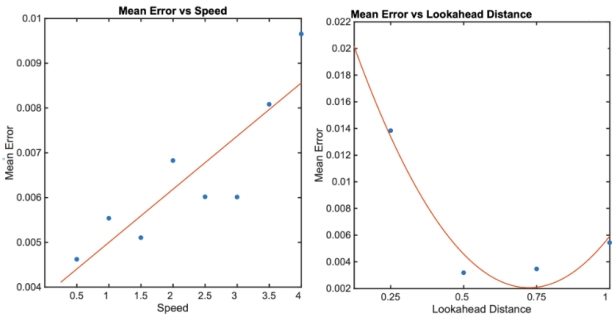

Fig. 8. Left: Slower speeds lead to a smaller distance error from the path. The data shows a fairly positive trend. Right: A quadratic trend is found between the look ahead distance and the mean distance error between the vehicle and the path. The minimum error lies on a look ahead distance close to 0.75m.

In addition, it seems as though the shorter look ahead distances seem to not see enough of the path, especially on tighter turns (Fig. 9). For example, in Figure **??**), where the blue set of data corresponds to the 0.25m look ahead distance, the error while going around the tight corners is much higher than that for larger look ahead distances.

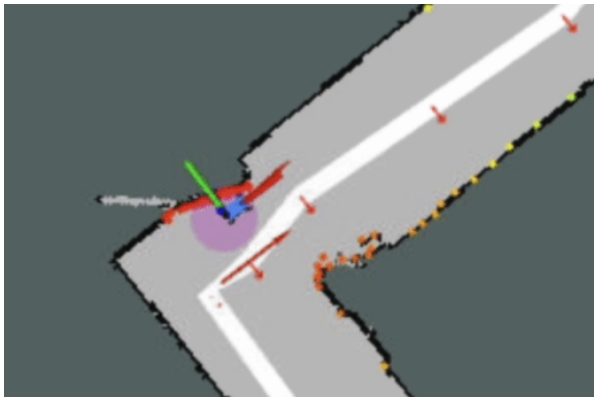

Fig. 9. The image demonstrates a large overshoot from the path as the vehicle attempts to make a right turn at 4.0m/s. The look ahead distance does not see any path, so the vehicle runs the previous drive command until told to change when the path is in view.

Furthermore, larger look ahead distances tend to look too far ahead of the car such that on turns where the path intersects with the look ahead distance circle multiple times, the vehicle's next way point is already on the next rotated segment even though the vehicle has not started turning (Fig. 10). This is a problem because it creates a lot of distance error since the vehicle is able to follow curves by not being fully on top of the path.

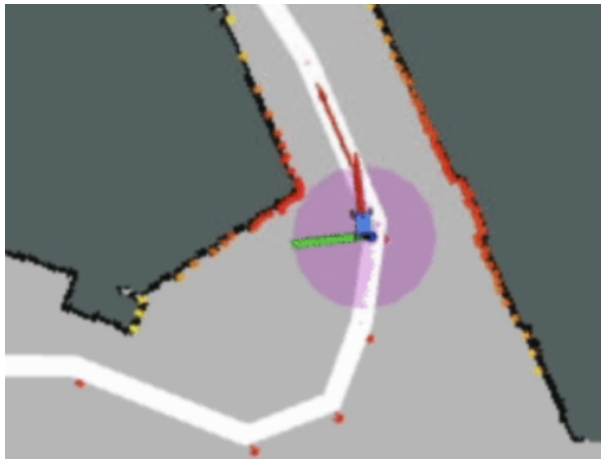

Fig. 10. The look ahead distance (purple circle) is too large such that the car is consistently offset from the path. The look ahead distance is identifying a "go-to" point that is farther along the curve, and therefore the car makes a smaller turn than expected.

Finally, it is expected that the lowest speed plus the 0.75m look ahead distance would have the smallest mean error. This, however, was not fully evident in the data, as the data showed that a speed of 4.0m/s with a look ahead distance of 0.75m had the smallest error (Fig. 11). It is expected that with more data points, a more accurate estimation of the most ideal speed and look ahead distance could be obtained. It is also predicted that a more robust optimization could be made to identify the "best" parameters to choose, based on the maximum time the

follower should take, the distance of the path to walls, the curviness of the path, etc.

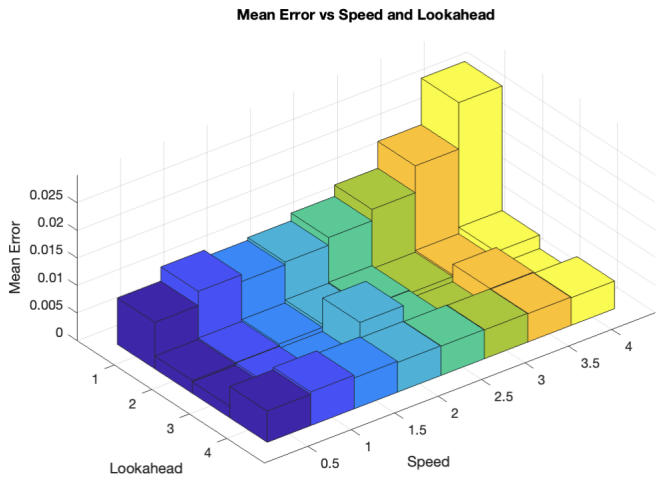

Fig. 11. A 3D bar graph showing the mean distance error from all thirty-two datasets across speed and look ahead distance.

IV. Conclusion (Gloria)

In this lab, we implemented both path planning and path following algorithms, evaluated how they responded to various system parameters and driving conditions, and deployed our software onto the physical racecar. For path planning, we found that the A* algorithm using a square grid worked well for our applications, and for path following, we found that a pure pursuit algorithm was both simple to implement and intuitive to tune. Finally, we found that slower vehicle speeds and a look ahead distance between 0.5 - 0.75m were the most ideal combination within our test data to minimize the vehicle's distance error from our proposed path.

A. Future Work (Gloria)

To improve our path planning further in the coming weeks, we will optimize our A* algorithm by implementing a more efficient system of checking new points to add to the trajectory. We also plan on planning in pixel space to speed up our search. Additionally, we hardcode in high-level plans depending on which points we're planning between to prune our search space even more. To expand upon our path following techniques, we will implement a variable lookahead distance so that the car can reorient itself on the trajectory if it moves too far out of the way, and we will experiment further with system parameters in order to achieve the smoothest ride. Additionally, we will improve upon our stopping condition so that the car is guaranteed to stop within one meter of the target end of the trajectory. Within the broader world of racecar autonomy, path planning and following are extremely important as they allow the car to efficiently and accurately navigate around a known map. Combined with the odometry module from our previous lab, we are able to orient our car in Stata basement and instruct it to move to any location autonomously. As we implement higher level search and logic procedures in future labs, we will be able to easily reuse the modules we implemented in this lab.

Direct shooting with MPC is a highly effective method that allows planning to make many of our large scale cybernetic systems more robust. By experimenting with this strategy for smaller systems and making it work in progressively more realistic environments, we can potentially discover solutions that could be applicable to robotics problems today.

B. Lessons Learned (Ayden)

For this lab I was able to apply some prior knowledge on path planning algorithms in a new context (planning over a map that isn't already a graph, implementing algorithms for an autonomous vehicle, etc.), so there was much to learn about this. I also learned the pros and cons of sample-based path planning algorithms like RRT.

C. Lessons Learned (Jose)

The lab was a good refresher on A*, as I had learned it a while back, but had forgotten it. It was also an interesting introduction to sampling-based planners and pure pursuit. Also had fun making the graphs again for my section, its fun to try to turn words into images that convey the same meaning.

D. Lessons Learned (Erin)

This lab was one of the first ones where the "putting on the robot" went way smoother than I imagined, which either means this lab was set up that way, or we are just getting smarter. Anyways, the pure pursuit algorithm was fun to write, and I learned about various graph searching algorithms. I also got practice making a lab briefing and report.

E. Lessons Learned (Gloria)

It was so cool seeing the odometry from the previous lab come into play this time! Also things worked surprisingly well once we put them onto the robot. I think the biggest takeaway I got from this lab is that RViz is kind of annoying to work with and not as high-bandwidth as I thought. I think this lab also had some interesting higher level logic where the robot had to wait for trajectories to be published, odometry updates, pose directions, etc. in order to make the next move, and I think handling robot state and decision making in the more complex final challenge will be particularly interesting.

F. Lessons Learned (Greg)

This lab gave me a lot more confidence in our racecar control pipeline's ability to deploy in the real world, with our system behaving as expected in controlled conditions. It makes me excited to now be able to move on to focusing on speeding up its performance now that it can execute high-level commands reliably at low speed. I learned about what the pure pursuit algorithm is. I was surprised how effective a particle filter localization scheme can be as input to planning algorithms for robot movements.