

**Team 19: Stephanie Howe, Mary Lau, Prajwal Mahesh, Sean Crozier,
Uche Okwo**

6.141

2022 April 15

Lab 6 Report: “Go Here, Go There, Go Everywhere!” Path Planning and Pure Pursuit with Team 19

Introduction (Praj)

As we iterate on our car’s ability to autonomously navigate its environment, a crucial capability is being able to maneuver to any reachable destination. The process to achieve this goal includes two components: path planning and pure pursuit.

Path planning defines the car’s ability to determine a collision-free path from the current location to the desired destination. For this module, we elected to implement a search-based planning algorithm as opposed to a sampling-based one. Search-based planning revolves around first discretizing the environment into a graph space, and then searching the graph for the optimal path. In particular, we used the A* algorithm because of its guaranteed ability to find an optimal solution for the desired path.

Once a desired path has been determined by the A* algorithm, our car must autonomously drive along the created path. We achieved this capability by implementing a pure pursuit algorithm, which allows the car to track the path by creating a series of intermittent goal points to follow. By looking ahead to some desired point on the path, pure pursuit determines the optimal steering angle that our car must drive at to reach that intermittent goal. This process continues as new goals are formed until the path has been completed.

Integrating these two modules required more than just these two components. For one, we used this lab as an opportunity to improve upon our robot’s safety controller, which allows us to do more rigorous testing on the car moving forward. Additionally, there was a series of dependencies between past and present modules. The first requirements were localizing two points in the map space: our car’s initial position and the desired final destination. Doing so allowed path planning to then determine an optimal path given the constraints of Stata. Once this was complete, pure pursuit was able to read the desired path, and our car successfully traversed from its initial pose to the goal location.

Technical Approach

Path Planning

Search-based vs. Sampling-based (Mary) In the early stages of the project, we considered two approaches as strong candidates for our path-planner implementation, that is, search-based planning and sample-based planning. Both approaches are well-used for path-planning tasks and come with their own set of benefits and tradeoffs. Search-based planning employs graph search methods to compute trajectories over a discrete representation of the problem. Sampling-based methods form trajectories by sampling for possible points and adding them to a tree until either a solution is found or time expires. Unlike search-based planning, which demands discrete graph representation of the problem, sampling-based methods can solve for paths in continuous search space. As such, for problems with large state spaces, sampling-based planning is the more computationally feasible solution. However, a major downside of sampling-based planning is that it is not a complete solution; there is always a possibility that it may run out of time before being able to find a path. Moreover, even if it is able to find a path, it may be sub-optimal and unusual-looking. In contrast, search-based planning is complete. If the search-based planner does return a path, it will be guaranteed to be most optimal (based on the measure of optimality we coded in the algorithm), and if it does not, we know for certain that the goal is not reachable.

Having an 1300x1730 (pixel units) occupancy grid of our environment from the beginning of the project, our problem lend itself to a straightforward discrete representation as a pixel-unit grid. Moreover, our state space is very small, a mere (x,y) coordinate space. With these numbers in mind, we determined that it would be more than feasible to tackle our problem with a search-based planner. In addition, we noted that the completeness guarantees of search-based planning would be extremely helpful to achieving efficient paths for our racecar. Thus, we decided to implement a search-based planner.

Path Planning Technical Implementation (Mary)

Overall, search-based planning essentially breaks down two stages: first we need to turn the problem into a discrete graph representation; second we need to search this graph for the most optimal path.

Determining Realistic Collision-free Search Space (Mary) First step, we need to determine the search space of the problem, that is, where the robot could be without colliding with obstacles. We have prior knowledge of its environment, a 1300x1730 (pixel-unit) occupancy map of the stata basement. This occupancy map is a binary array, 1 where there is an obstacle and 0 otherwise. This occupancy map would be sufficient, if our robot was a point mass. However, in real life, our robot has dimensions (height, width, etc), which we need to take into account when constructing our collision-free search space.

To address this, we use cv2’s dilation tool to dilate the occupied portions of the occupancy map, as shown in Figure 1. We can see from the figure that dilating the occupancy map results in a realistic collision-free space for our robot. (Determining how much we needed to dilate the occupancy map was an empirical process; in other words, we eyeballed and adjusted the dilation process based on the racecar experiments we observed in simulation and in real life).



Figure 1: Occupancy Map

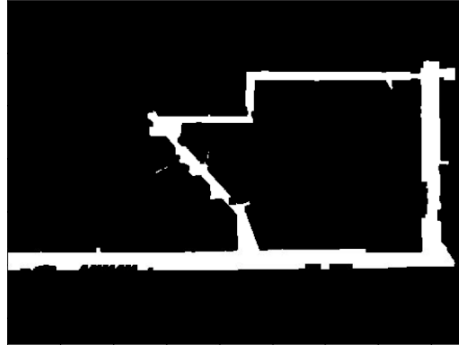


Figure 2: Occupancy Map Dilated

Discretizing Continuous Search Space into Pixel-based Grid Space (Mary) Now that we have a realistic collision-free search space, we can discretize it into a grid space based on pixel units. This is equivalent to a graph, where a “node” can be any set of (x,y) pixel coordinates from the grid. There is another part of the problem, though: our robot doesn’t operate in pixels. It operates rather, in real-world coordinates. Thus, we need to be able to convert between real-world coordinates of our robot’s map frame and the pixel-coordinates of our grid.

To convert from pixel to real world coordinates, we use the quaternion of the real world map’s origin to calculate the rotation matrix R_{rw} , which we then use

to construct the following conversion matrix $A_{pix \rightarrow rw}$.

$$A_{pix \rightarrow rw} = \begin{bmatrix} R_{rw,(0,0)} & R_{rw,(0,1)} & x_{rw} \\ R_{rw,(1,0)} & R_{rw,(1,1)} & y_{rw} \\ 0 & 0 & 1 \end{bmatrix}$$

We multiply our pixel points (x,y) with the map resolution (MR), which is the number of meters per pixel (m/pix). Then we apply the matrix $A_{pix \rightarrow rw}$ to this, which will yield us the corresponding real-world coordinates.

$$A_{pix \rightarrow rw} \left(\begin{bmatrix} x_{pix} \\ y_{pix} \\ \frac{1}{MR} \end{bmatrix} * MR \right) = \begin{bmatrix} x_{rw} \\ y_{rw} \\ 1 \end{bmatrix}$$

To convert from real-world to pixel, we invert the process: multiplying by the inverse of $A_{pix \rightarrow rw}$ and dividing by the map resolution.

$$(A_{pix \rightarrow rw})^{-1} \begin{bmatrix} x_{rw} \\ y_{rw} \\ 1 \end{bmatrix} * \frac{1}{MR} = \begin{bmatrix} x_{pix} \\ y_{pix} \\ \frac{1}{MR} \end{bmatrix}$$

Giving a high-level overview of how these conversions will be utilized, our path planner will be given a start point and end point, both of which are real-world coordinates. As our path planner's graph search algorithm operates in the pixel state space, we convert these real-world coordinates into pixel coordinates and pass them into our planner. If there is a possible path, our planner will find this path of pixel points and convert them back to real-world coordinates, feeding that real-world trajectory to the pure pursuit part of the pipeline.

Implementation of A* algorithm to Search our Grid Space (aka Graph) (Uche) Specifically, among the search-based algorithms, our team chose to use the A* algorithm due to its guarantee of optimality within a reasonable timeframe. A* is the adaptation of Dijkstra's algorithm with the insertion of a heuristic function, which estimates the cost of the path from the node under evaluation to the end node. Dijkstra's algorithm is a graph algorithm that guarantees the lowest cost path in a weighted graph from a start to end node. If implemented with a priority queue, A* is essentially Dijkstra's algorithm, except that the priority of items is the cost of the path up to the last node of the path + the heuristic from the last node of the path to the end goal.

When implementing A*, we had to choose a heuristic for the algorithm to use, where the heuristic heavily affects what kind of path and the runtime of the algorithm. The cost of the path that our code uses is the Manhattan distance of the path, as that most closely resembles what the robot will do on the pixel map, where moving diagonally costs slightly more than moving in a straight line

due to the constraints of physical distances. The optimal heuristic is the exact cost of the path to the end goal, which would guarantee minimal runtime and shortest path. However, Manhattan distances would be impossible to evaluate as a heuristic, as we would need the exact path that we would use to get to the end goal, which is what we are trying to solve for! So we must find another heuristic. An important property of the heuristic is that it never predicts a cost more than the actual cost of the exact path, in other words, the heuristic should lowerbound the actual path cost, if we want to maintain the guarantee of shortest path (even at the cost of runtime: a function that consistently reports a cost under the actual path cost will take a longer time to run than one that had the exact cost, but will return the same optimal path, rather than a function that consistently reports over the exact cost of the path, which will neither guarantee an optimal path nor a low runtime relative to the exact cost). With this in mind, the robot uses the Euclidean distance between the end node and the goal, as the distance is always less than or equal to the Manhattan distance between 2 points.

Pure Pursuit (Praj)

Once an optimal trajectory is returned by the path planning module, pure pursuit takes over to drive the robot towards the calculated waypoints. Pure pursuit is a control algorithm that steers the robot through the planned path by determining intermediate goal positions and calculating the curvature that will get the vehicle to that point. The first step is placing the robot on the map with the help of the lidar-based localization package developed in the previous lab exercise. Once the coordinates of the robot in the map frame are returned, we can use it to find the segment on the calculated path that is nearest to the robot. The path, which is represented as a series of points on the map, is checked iteratively beginning with the segment between the first and second points, followed by the segment between the second and third points, and so on. Equations (1) through (3) show how distance to each path segment between points p1 and p2 is calculated given the robot's position q.

$$p_1 = (x_1, y_1) \quad p_2 = (x_2, y_2) \quad q = (x_R, y_R)$$

$$\vec{S} = p_2 - p_1 \quad \vec{R} = q - p_1$$

$$\hat{S} = \frac{\vec{S}}{\|\vec{S}\|} \quad \vec{r} = \frac{\vec{R}}{\|\vec{S}\|}$$

1.

$$t = \hat{S} \cdot \vec{r}$$

2.

$$c = \vec{S} \cdot \max(0, \min(t, 1)) + p_1$$

3.

$$d_{min} = \sqrt{(C - q)^2}$$

The segment with the minimum distance is chosen and all segments preceding it are disqualified from future search. A lookahead circle of a fixed radius r is drawn around the robot. Beginning with the nearest segment, the remainder of the path is checked for intersections with this lookahead circle. This is done by solving the quadratic equation in (4) on each path segment, which returns one, two, or no solutions.

$$p_1 = (x_1, y_1) \quad p_2 = (x_2, y_2) \quad q = (x_R, y_R) \quad r = r_{lookahead}$$

$$\vec{V} = p_2 - p_1$$

4.

$$t^2(\vec{V} \cdot \vec{V}) + 2t(\vec{V} \cdot (p_1 - q)) + (p_1 \cdot p_1 + q \cdot q - 2p_1 \cdot q \cdot r^2) = 0$$

5.

$$i = p_1 + t * \vec{V}$$

The furthest intersection along the path is chosen as the next goal point of the robot. If no intersection is found, the goal point is simply set on the nearest segment. This point is then transformed into the robot's coordinate frame (6) and a steering angle is calculated to drive the robot towards this goal (7). The above calculations are performed frequently—every time localization updates the robot's position on the map—such that the robot is always pursuing a goal point that is somewhere ahead of it. 6.

$$T_{goal}^{robot} = T_{world}^{robot} \cdot T_{goal}^{world}$$

$$\theta = \tan^{-1}\left(\frac{y_{goal}^{robot}}{x_{goal}^{robot}}\right)$$

$$d = \sqrt{(x_{goal}^{robot})^2 + (y_{goal}^{robot})^2}$$

7.

$$steering = \tan^{-1}\left(\frac{2 \cdot wheelbase_length \cdot \sin(\theta)}{d}\right)$$

The velocity of the robot is set proportional to its distance from the end of the path and is clipped to a constant maximum value. This ensures that the robot will drive the majority of the planned path at a constant velocity but slow down to a stop as it approaches the end. ### Safety Controller (Sean) (Edited by: Uche)

We also updated the safety controller in this lab to more closely align with our evolving requirements. Our path planning algorithm and pure pursuit controller can find shorter trajectories where the racecar is operating close to walls and corners, necessitating less sensitivity when navigating tight quarters. Our new safety controller is predictive - stopping the car when its trajectory intersects with an obstacle, and is less sensitive to nearby obstacles it won't collide with.

The updated safety controller utilizes two separate control areas, in which it checks whether any LIDAR ranges are present to decide on whether to stop or

not. The first of these ranges is a wedge - defined by a circular arc of 10 degrees originating at the LIDAR scanner, and limited to a distance from the scanner of 0.1 meters. This short arc serves to stop the racecar before a collision with unexpected obstacles, or if the other area defined by the safety controller fails to stop the racecar in time. The short arc area is highly sensitive to LIDAR ranges, requiring only a single range to stop the racecar. This acts as a measure of last resort, in which even obstacles with thin profiles or reflective surfaces that wouldn't normally return LIDAR signals can be stopped for, in addition to obstacles moving through the path of the racecar without warning.

The second of these ranges is a rectangular representation of the racecar, with equal width and length. The steering angle and speed from the drive command sent to the racecar are used to approximate the racecar's position in 0.5 seconds. This prediction assumes the steering angle and speed will be constant for the elapsed time and uses a bicycle approximation to quickly calculate the racecars trajectory. The bicycle approximation simplifies the motion of the racecar to that of two wheels aligned along the center - with these two wheels representing two points along the chord of a circle, and assuming the front wheel is tangential to the arc of the circle, the radius of the circle can be found using the cosine rule, and equals

$$R = L / (2 * \sin(\alpha_s))$$

Where L is the length of the racecar.

From this radius, the change in pose can be calculated by finding the rotation around the circle,

$$\theta = \frac{v * t_e}{R}$$

And thus

$$\Delta = [\cos(\theta) * R, \sin(\theta) * R]$$

The elapsed time must be chosen carefully here. If the time is too short, the racecar won't be able to stop in time. However, if the elapsed time is set to be too large, a gap will develop between the small arc area and the rectangular area that the racecar is blind to. A sudden change in steering angle at high speeds could result in the safety controller only checking points beyond the wall it is set to collide with.

Because the rectangular area is larger and further from the racecar than the short arc area, it's also more likely to encompass false LIDAR ranges. To compensate for this, the stopping condition of the predictive area is that two or more LIDAR ranges are within the forward area.

Experimental Evaluation

Path Planner Evaluation (Mary)

Testing Path Planner Robustness to Various Paths (Mary) The first thing we did to evaluate our path planner was run it through various test cases and observe if it was able to find a path if the goal was reachable, or otherwise, return a “Path Not Found” message. Here, we included images the paths generated from the test cases.

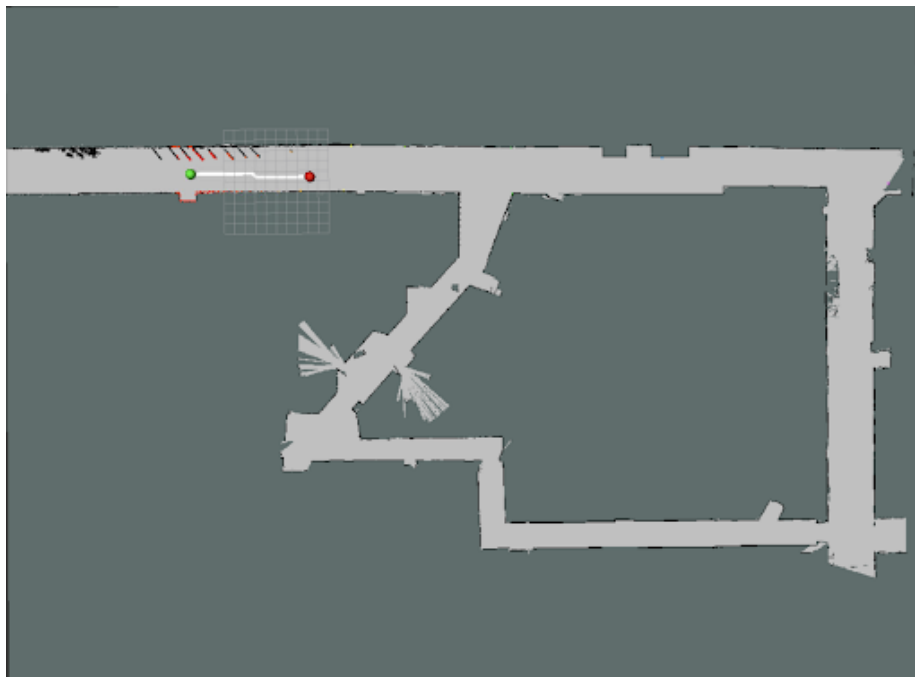


Figure 3: TC0: Short straight path

We also had a test case for impossible paths, where we placed the goal point in an unreachable area (in the occupied part of the map). Our planner always returned correctly that a path could not be found (which due to the completeness guarantee of search-based planning, we know for certain should be true).

For these test cases, we also noted the amount of time our code took to run the A* algorithm. The planner always took less than 1 second to carry out A star search, even for the case where a path was not reachable (Test 4: leftmost to rightmost corner of the map). As such, it seems there is not much computational tradeoff for having a complete path planning solution.

Testing Path Planner vs. Human (Mary) For the second part of our path planner evaluation, we decided to compare planner-generated paths with

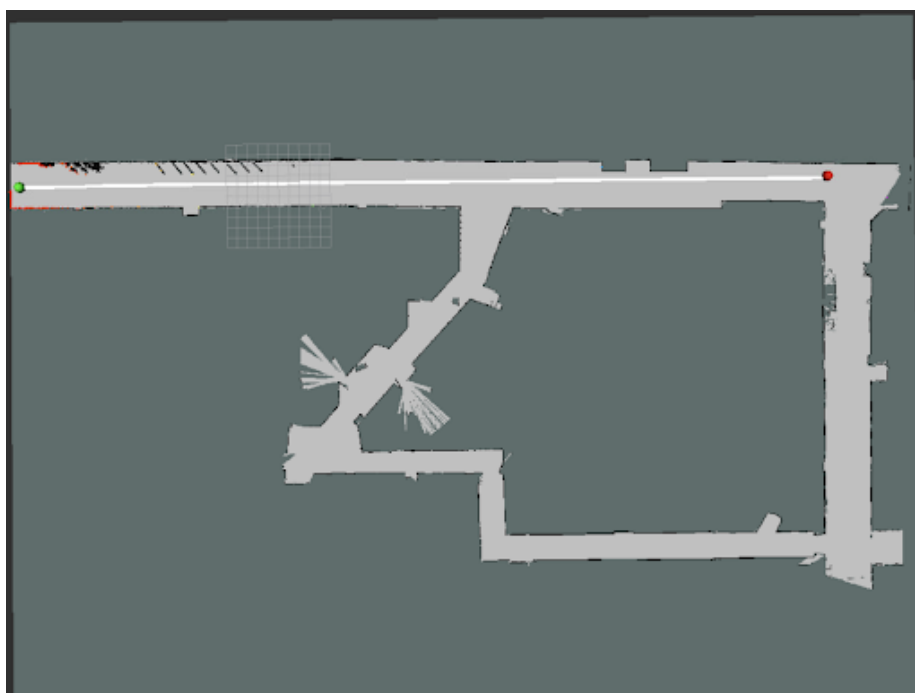


Figure 4: TC1: Long straight path



Figure 5: TC2: Simple corner turn



Figure 6: TC3: Complex corner turn

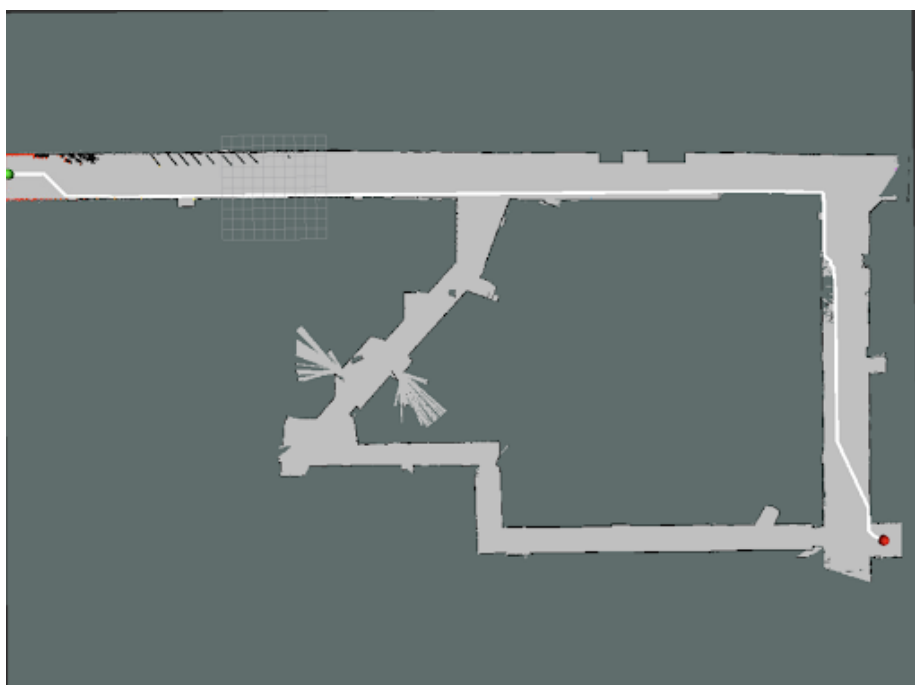
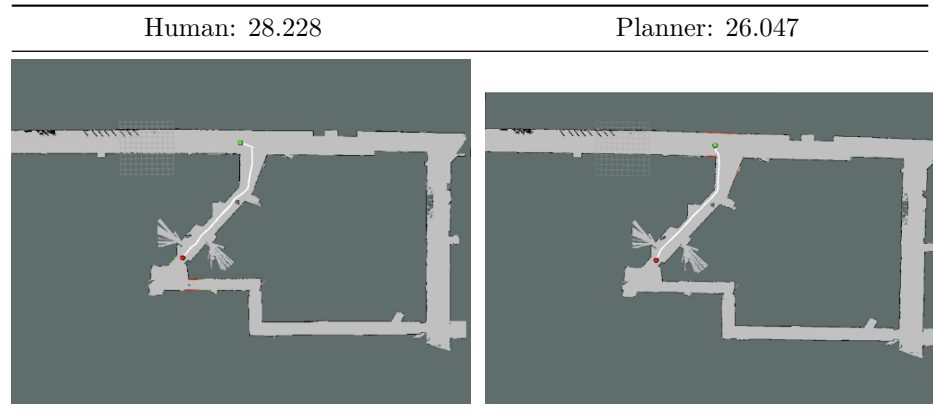


Figure 7: TC4: Leftmost to rightmost corner of the map



human-drawn paths. We asked a member of our team, who did not work on the planner, upon being given a start and end point, to draw the likely path on the map that they would naturally plan for themselves to walk. We have included pictures of the human-drawn path (left) and the planner-generated path (right), as well as each path's total distance.

We can see from the test cases below that the planner consistently outperforms the human, returning a path that requires less distance for the racecar to travel. The main reason for this out-performance is the fact that the planner is able to move in ways that a human would not intuitively move. For instance, the planner tends to hug the walls and only moves across the hall when it absolutely has to (i.e. when it has to make a turn in the opposite direction or when it is nearing the goal point, which is closer to the opposite wall). In contrast, when walking along a path, humans usually prefer to move along the center and maximize their distance from the walls. The planner has no such preferences and always seeks to move in whatever way that gets the racecar to its destination most efficiently.


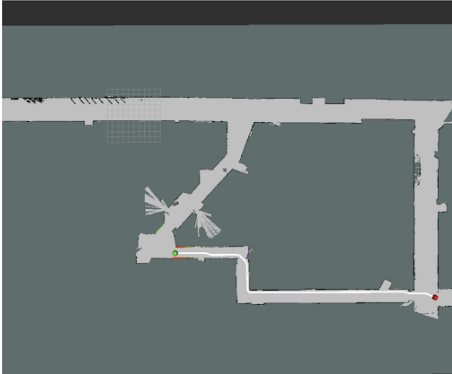
Test case 0



Test case 1

Human: 73.721	Planner: 70.823
	

Test case 2

Human: 55.542	Planner: 54.652
	

For these test cases, we also made sure to create situations where the robot/human would have more than one viable path to their destination. Of course, the human had no problem ascertaining which path would be most convenient, and we wanted to ensure that the robot would also have such sensibilities. As we can see from these test cases, the robot always picks the most optimal choice.

Gradescope Autograder (Uche)

In terms of the Gradescope autograder, our path planner does an excellent job, scoring 2.99 out of 3.0 for the planning portion.

Combined, the path planner and pure pursuit controller scored 3.98 out of 4.0. Also, the pure pursuit drove 98.96% of the path within the allotted 500 seconds.

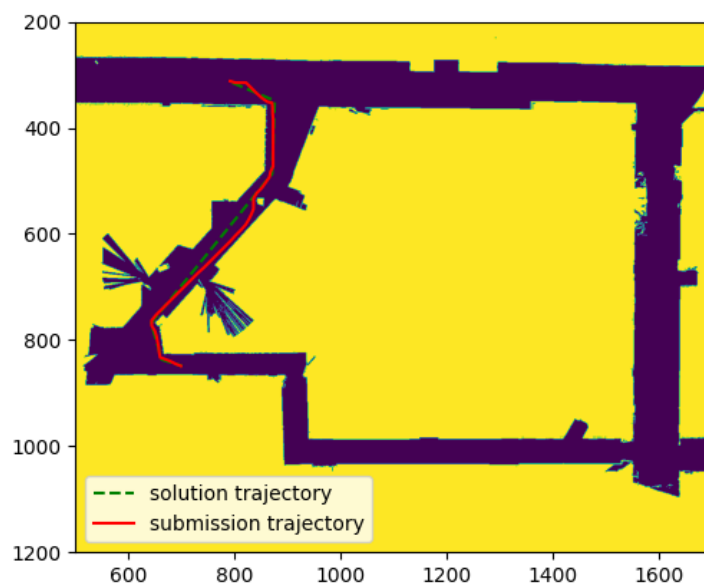


Figure 8: Gradescope Path Planning Graph showing the staff solution and the path planner's path

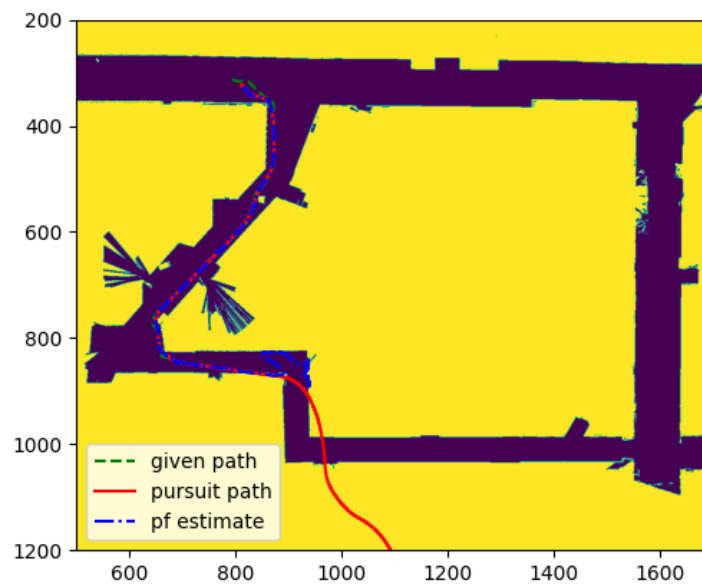


Figure 9: Integrated graph showing staff solution, path planner path, and pure pursuit path

Pure Pursuit Evaluation (Stephanie)

Pure pursuit testing took place in a few stages. First, we needed to test the ability of the controller in isolation. To do so, we constructed very simple trajectories in simulation to run the algorithm against. During this stage, we used our own judgment to determine whether or not we considered pure pursuit operational enough to move forward to the next step of testing.

The next phase of testing included integration of the path planning and pure pursuit modules in simulation. We also had to lean on the previous localization module to help localize the robot on the map. Once the initial position of the robot was set, we could run our path planner to plan a path in simulation, as well as pure pursuit in order to send drive commands to the car. For this iteration of testing, we again used judgment to determine if the car was following the paths outputted by path planning reasonably. We also wanted to understand how well the car was performing at a quantitative level. In order to do this, we computed a distance error over time, which reflects how close the car is to the path at a given point in time.

The figures below show the results of path planning and pure pursuit in simulation. The associated plot shows how closely the car is tracking the path over time, with the y-axis being distance from the path in meters. Some notable points about the graph are its consistency while the car is driving straight (from 42 seconds to 50 seconds). The error when following a mostly straight path is consistently under .025m, and even hits zero at many points. This oscillation of the distance error reflects the slight oscillations in the car as it corrects its steering angle. After these small oscillations, the larger spike in the graph can be attributed to the sharp right turn the car makes. Depending on the lookahead distance, the car will “cut” corners, hence the spike in distance, which is still quite low in this case. Ultimately, our pure pursuit controller demonstrates impressively low distance error in simulation.

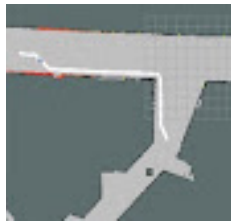


Figure 10: Pure Pursuit Path 0

Once we were confident in simulation, we moved forward to running pure pursuit at the car level, and tested it in person with a variety of paths. Some notable outcomes are that pure pursuit was successful in navigating through Stata basement when turns were wider. However, in the case of tight corners, the car was unable to clear the corner. This is due to our static lookahead distance. In order to improve this, we should implement a variable lookahead distance to be

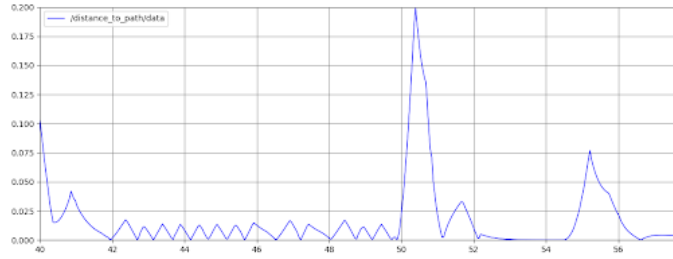


Figure 11: Pure Pursuit Graph 0

smaller when the path has high curvature, and larger when the path is straighter. The images below demonstrate the cars ability to use pure pursuit navigate along straight lines and wider turns according to a path created by our path planner. Again you can see that the distance error of pure pursuit is extremely low. The car manages to stay within 0.15m from the planned path, and the spikes in this error can once again be attributed to areas of higher curvature of the path.

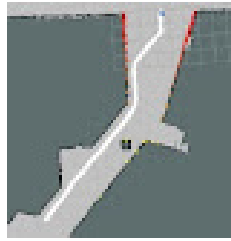


Figure 12: Pure Pursuit Path 1

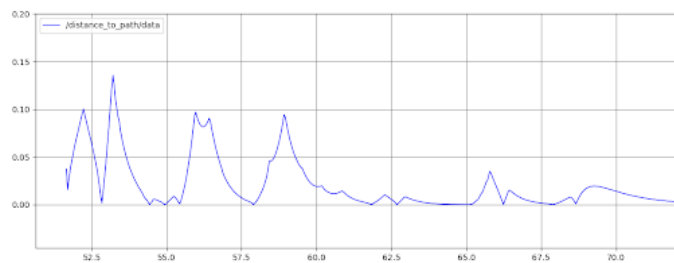


Figure 13: Pure Pursuit Graph 1

Updated Safety Controller Evaluation (Sean)

To evaluate the safety controller, we ran both the original and predictive safety controllers through a series of tests in simulation. Starting parallel to a wall at a

distance of 0.1 meters, the racecar was given a command of constant speed and steering angle such that its trajectory was a circle. For the racecar to return to its original position, the safety controller can't trigger as it approaches the wall at a close distance. The steering angle and speed of the racecar were modulated to simulate various scenarios.

Speed (m/s)	Steering Angle	Original SC	Predictive SC
0.5	$\frac{\pi}{48}$	Pass	Pass
0.5	$\frac{\pi}{24}$	Pass	Pass
0.5	$\frac{\pi}{12}$	Pass	Pass
1	$\frac{\pi}{48}$	Pass	Pass
1	$\frac{\pi}{24}$	Fail	Pass
1	$\frac{\pi}{12}$	Fail	Pass
2	$\frac{\pi}{48}$	Fail	Pass
2	$\frac{\pi}{24}$	Fail	Pass
2	$\frac{\pi}{12}$	Fail	Pass

The predictive safety controller continues to perform at high speeds and sharp angles, while the original safety controller is over cautious and prevents the car from returning to its original position at speeds as low as 1 meter per second.

Conclusion (Sean)

Both our path planning algorithm and our pure pursuit controller are effective in navigating the racecar to its destination. The path planning algorithm, through intelligent use of map discretization and management of nodes in the location queue, is capable of finding paths on time spans described in milliseconds. These paths meet the requirements of the Gradescope testing, scoring over 2.99/3, beat human intuition in path construction as shown in our evaluation, and are path exhaustive, not compromising thoroughness for its efficiency. The nearly instant pathfinding times on a grid with over at most 2 million pixels validates our choice to choose a search based rather than sampling based approach. The pure pursuit controller, built utilizing the previously designed localization node, follows the trajectory the path planning algorithm designs to an error of less than 0.2 meters. This error can be further reduced using a variable lookahead distance, in which the racecar will target segments along the designed trajectory at an adjusted distance to the racecar in order to more closely adhere to trajectories of various shapes.

Lessons Learned (Lesson in tech and CI from each person)

Uche

Technical Real talk: my original implementation of the A* algorithm was slow. *Incredibly* slow compared to what our end result was. The reason behind that was a relatively simple change: using pointers to nodes. Originally, I stored the path as a list of nodes: I stuck every single node that a path went through

into a list and called it a day. To calculate the cost of any given path, I would iterate over all the nodes of the path, which I had since I stored them. The only problem with this method is that it was slow, really slow. I didn't take advantage of the fact that the problem wasn't to find path costs, but to find a *single optimal* path cost between 2 nodes. Using this fact later, the robot could take the path information, and reduce it to 1 node, and instead, keep track of the optimal cost of the path to a node and which node that path came from in a separate structure outside of the priority queue. Once the robot's algorithm hits the end node, it simply backtracks using this structure to produce the optimal path. This heavily cut down our robot's A* runtime, from taking on the order of seconds to milliseconds.

CI Managing static websites manually is not a lot of fun. Things get out of sync, sometimes the structure of a page is unclear, so you have to do formatting work, and copy-paste only takes you so far, where you have to do it at multiple different locations just to make a simple change. I decided that I had had enough of that tedium, and moved the site to a SSG (static site generator). Once I got everything integrated and working, I could heavily reduce the amount of boilerplate I had to change everytime I added a new page (adding new pages is now simply as easy as creating a new file and adding some metadata to that file), and now I have some confidence to be able to slightly move off of the default template we were given. (For more concrete details, I moved everything to the SSG Zola because it is a relatively freeform SSG that also had a Github Action to generate the site, so that I wouldn't have to manually do anything. You can check out the website code [here](#). I could also simply port my report generating code to follow the directory format of Zola, that being that static files go in static/ and the resulting site kind of Just Works [some nitty gritty details of browser URLs, and how the site is generated meant that I did have to tug of war with the CSS a little bit, but overall, it was a smooth experience].) I was kind of surprised that I could do it in one afternoon...

Stephanie

Technical The technical portion of this lab was both interesting and rewarding. I was particularly working on the development of pure pursuit and then in integrating the modules onto the racecar. I found that looking back through previous resources from lecture and recitation slides was the best way to refresh on the previously learned algorithm. We also learned while testing the importance of having the pure pursuit lookahead distance be variable. For example, the car couldn't clear tight corners so we determined that it was important to vary the lookahead distance to be smaller where there was more curvature and higher corners, and then larger for straighter areas to reduce oscillation.

Communication While working on the technical portion of this lab, I found it useful to have more than one person assigned to each module. Moreover, as we refreshed ourselves on pure pursuit and iterated through the code, it was

extremely useful to have another point person to talk out loud with and work through the necessary math of the algorithm. I also think we had our best briefing yet on this lab. In the past we have been a little shaky or run over time, but our coordination was really nice this time. We all learned how to say more with fewer words and try to communicate only the most important pieces to be concise.

Sean

Technical When testing nodes on the robot, it's important to routinely verify that commands are being received. In-person hardware time is a valuable resource requiring coordination between multiple team members, and the more time spent evaluating algorithms and progress the more concrete our next steps can be. Additionally, it's easy to misstate the objective of a sub-project such that it doesn't align with the objective of the project as a whole. When designing our original safety controller, our objective was to prevent any hardware damage. Though this seems to be a safe goal, following this objective to its furthest extent prevents the racecar from being able to complete its actual objective: safe driving and navigation. Because of that misstatement, the safety controller needed to be redesigned with the actual objective in mind.

Communication Being separated from the team for a week prevented me from easily collaborating on the aspects of this project that required in-person work. However, this separation and inability to collaborate enabled me to wrap up another aspect of our project that needed to be improved. Though having all members of the team working towards the same immediate goal is the default, it precludes us from devoting the time necessary to integrate our past work with our current requirements.

Prajwal

Technical I found it was particularly helpful to create little test sketches to ensure the operation of various segments of code before integrating them into the main program. Particularly with pure pursuit—which had a number of operations that needed to happen one after the other—piecewise testing allowed us to confirm that every individual segment in the chain of operations was working correctly and debug efficiently if it was not. #### Communication Having another pair of eyes look at existing code or simply explaining it to someone else proved to be very useful for debugging. This not only lets someone with a fresh perspective weigh in on the problem, but more importantly, the act of explaining a solution forces us to think more critically about it and makes it easy to spot errors.

Mary

Technical I think going to office hours as much as possible was critical to helping me finish the path planner. At the beginning, I was blocked on things

that I didn't think would be the hardest part of the path planner, like pixel-to-real world conversion or visualization of the trajectories. The solutions for these issues weren't very hard, but it was so important to get them fixed as soon as possible, because they were essential to debugging and verifying our path planning algorithm. As such, TA help was invaluable, because they already had experience with most of these bugs. Going to office hours saved me from a lot of time and unnecessary effort.

Communication We did encounter an unprecedented difficulty in this lab—one of our teammates got COVID. Of course, they couldn't come on-campus, so we reassigned parts and figured out a timeline that would allow them to maximize their contribution to the project, despite them not being able to be with us in-person. Also, I found it useful to work with a second teammate on the path-planning module, because we could walk through the code together and bounce ideas off of each other. Our teamwork led to us achieving a super speedy and accurate implementation of the A star graph search algorithm.