Lab 6 Report: Path Planning

RSS Team 14

Nicholas Bonaker
Juliana Chew
Alexander Koenig
Kai Maier
John Paris

April 30

Contents

1 Introduction (John)

In previous projects, feature following and recognition have been used to inform our TESSE-simulated car's navigation around physical space. However, the algorithms and implementations using these fixed feature methods can only enable a limited set of maneuvers. Using the localization techniques developed previously as well as map adjustments, we are able to plan and execute maneuvers based on global information.

Executing maneuvers based on map data and localization consists of a two-step process. First, one of 3 different path planning methods is used to chart and discover the shortest path from the given start location to the specified end location. Two of these methods, A* search and Dijkstra's, utilize search algorithms over a node network made up of separated pixel locations to find the shortest path. The other method, RRT, uses a sampling algorithm to quickly find a successful path, which may not be the shortest. Each of these methods is a compromise between runtime and path length. Using the resulting path from this first step, pure pursuit and other control techniques are then used to execute the planned trajectories.

Whether it is for a parking controller or Google Maps directions or any other feature, path planning and execution is a necessary capability for any autonomous vehicle. Fast but efficient path planning that works cohesively with the vehicles control systems opens the door to countless autonomous capabilities.

2 Technical Approach

Our approach is separated into three modules: path planning,pure pursuit, and Localization. Path planning takes the start and end poses as input, outputting a planned trajectory as a list of positions. The trajectory is then passed to pure pursuit, which sends drive commands to the car in order to follow the path as closely as possible. In the 2D simulation, we run our previously implemented Particle Filter in the background while pure pursuit is running (in the TESSE 3D simulation, we use the true pose instead). By adding localization, the believed pose is passed to pure pursuit so that appropriate drive commands are created.

2.1 Path Planning Algorithms (Juliana)

For path planning, we discretized the search space by using a given map image's pixels. Each pixel becomes a node in an adjacency graph.

To convert from real-world coordinates (denoted as (x^W, y^W)) to pixel coordinates (denoted as (x^p, y^p)), we used the map image's given orientation θ, origin (x_0^W, y_0^W), and resolution ζ meters per pixel.

$$\frac{1}{\zeta}\begin{bmatrix}\cos\theta & \sin\theta \\ -\sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}x^W - x_0^W \\ y^W - y_0^W\end{bmatrix} = \begin{bmatrix}x^P y^P\end{bmatrix} \tag{1}$$

As path planning outputs a trajectory in real-world coordinates, we also convert from pixel to real-world coordinates. The transformation is the inverse of the one above:

$$\zeta\begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}x^P \\ y^P\end{bmatrix} + \begin{bmatrix}x_0^W \\ y_0^W\end{bmatrix} = \begin{bmatrix}x^W y^W\end{bmatrix} \tag{2}$$

The following sections concerning path planning algorithms operate in pixel coordinates. For clarity, we assume all (x, y) are in pixel coordinates and do not use the symbol ^{P}.

In the following sections, we explored two main algorithms: A*, a search-based method, and RRT, a sample-based approach. These methods output a trajectory as a list of positions for the car to follow in pure pursuit.

2.2 Search-Based Algorithms vs Sample-Based Algorithms (Juliana)

We explored a search-based and sample-based algorithm specifically because of the strengths and weaknesses between the two classes. Search-based algorithms are guaranteed to find the optimal path given a cost function (if there exists a possible path), but because they search through a given (possibly large, or even infinite) space until finding a node, they often have a longer runtime and a lower

memory efficiency. On the other hand, sample-based algorithms use stochastic methods to sample from the search space, a feature that often drastically reduces runtime and increases memory efficiency. However, due to the nature of stochastic methods, these algorithms are not guaranteed to be optimal, albeit they are asymptotically complete (i.e., guaranteed to find a path, if one exists, after infinite time).

2.2.1 A* (Juliana)

A* search produces the optimal path from the start p_S to the goal p_G given a cost function by expanding in the direction of lowest cost. It accounts for both the cost to move to a given position p_2 and the cost to move from p_2 to p_G.

Given a current path P from p_S to p_1 and possible next location p_2, the cost of extending the path P from p_1 to p_2 is calculated as follows:

$$\text{Total Cost} = \text{Incurred Cost} + \text{Heuristic} + \text{Wall Cost} \tag{3}$$

Each component of the the above equation will be examined in following sections.

Incurred Cost The Incurred Cost is the actual cost of moving along a path P and extending from P's end p_1 to p_2. We define the Incurred Cost from p_1 to p_2 as follows:

$$\text{Incurred Cost} = \text{Path Length(P)} + \text{Manhattan Distance}(p_1,p_2)^{\alpha} \tag{4}$$

where $\alpha > 1$ is a constant is used to discourage turning and to incorporate dynamics. The length of P is the summed point-to-point Manhattan Distance, each to the power of α. Assuming $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, the Manhattan distance is defined as the following:

$$\text{Manhattan Distance}((x_1, y_1),(x_2, y_2)) = \|x_1 - x_2\| + \|y_1 - y_2\| \tag{5}$$

By including α, moving diagonally incurs larger costs than moving solely horizontally or vertically. This method produces straighter paths that are easier for pure pursuit to execute.

Heuristic Our Heuristic function $h(p_2)$ estimates the remaining path length from p_2 to p_G and is the Euclidean Distance from $p_2 = (x_2, y_2)$ to $p_G = (x_G, y_G)$.

$$h(p_2) = \text{Euclidean Distance}((x_2, y_2), (x_G, y_G)) \tag{6}$$

$$h(p_2) = \sqrt{(x_2 - x_G)^2 + (y_2 - y_G)^2} \tag{7}$$

By using Euclidean Distance, we ignore all map boundaries and underestimate the actual distance from p_2 to p_G. This method produces an admissible heuristic, which is essential for A^* to always produce an optimal path (given cost functions) if a path exists.

Wall Cost We applied a flat-rate wall cost C_W to locations within a distance d from a wall in the vertical or horizontal direction, where d is approximately 1/5 of the map's hallway width. With this additional cost, we discouraged A* from hugging walls.

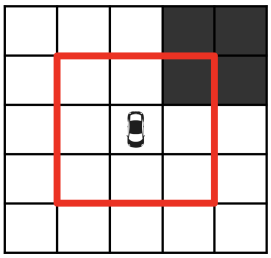

Figure 1: An example of applying wall cost, where unfilled pixels are unoccupied and dark grey ones are walls. If each pixel is 1 meter wide and $d = 1$ meter, then any wall within the red square (our boundary) is considered within d of the car. In this case, a wall pixel lies in the boundary, and the flat-rate wall cost is applied.

The wall cost function $w(p_2)$ is defined below. Assuming $p_2 = (x_2, y_2)$, all locations (x, y) within d of the car have the coordinates $x \in \{x_2 - d, x_2 + d\}$ and $y \in \{y_2 - d, y_2 + d\}$.

$$w(p_2) = \begin{cases} C_W & \text{if wall pixel within boundary} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

We tuned the wall cost C_W and chose $C_W = 40000$. Refer to section 3.2.2 for more information on tuning.

Procedure Using the equations above, the overall steps for A* are as follows:

Algorithm 1: Main A* procedure

Input: p_S, p_G
Result: Planned path from p_S to p_G
parentPointers $\leftarrow \{p_S : \text{None}\}$
costAtStart $\leftarrow h(p_S)$
queue $\leftarrow [(\text{costAtStart}, p_S)]$
bestCost $\leftarrow \{p_S : \text{costAtStart}\}$
while *queue nonempty* **do**
  cost, $p_1 \leftarrow$ queue.pop()
  if $p_1 = p_G$ **then**
    break
  end
  for *(Neighbor* p_2*, CostToMove* c_2*) in GetNeighbors(*p_1*)* **do**
    totalCost $\leftarrow \text{cost} + c_2 + h(p_2) + w(p_2)$
    if p_2 *not in bestCost or totalCost* $<$ *bestCost[*p_2*]* **then**
      parentPointers$[p_2] = p_1$
      bestCost$[p_2] =$ totalCost
    end
  end
end
return reconstructPath(parentPointers)

To find the neighbors of p_1, we used a jump size to define the maximum distance the neighbors can be in the vertical and horizontal directions. As search-based algorithms such as A* typically have long computation times and high memory usage, including a jump size not only reduces the number of positions in the trajectory but also decreases runtime.

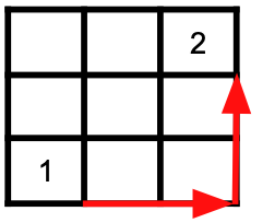

Figure 2: An example of jump size from location 1 to location 2. If each pixel is 1 meter wide, then the jump size is 2 meters.

Increasing the jump size decreases the path resolution, thereby decreasing computational time. With the reduced resolution, however, the resulting path length increases as the path becomes more "block-like" and less direct.

We chose our jump size to be approximately 2 meters, or 6 pixels. For more information on tuning this parameter, refer to section 3.2.1.

The algorithm for finding neighbors is outlined as follows:

Algorithm 2: GetNeighbors

Input: p_1, jumpSize, map,α
Result: The set of neighbors and their corresponding costToMove
if *EuclideanDistance(p_1,p_G) < jumpSize* **then**
    jumpSize = 1
end
x,y = p_1
neighbors = set()
for *dx in {-jumpSize,0,jumpSize}* **do**
    for *dy in {-jumpSize,0,jumpSize}* **do**
        if $dx \neq 0$ `or` $dy \neq 0$ **then**
            p_2 = (x+dx,y+dy)
            if p_2 *in map* **then**
                costToMove = ManhattanDistance$(p_1, p_2)^\alpha$
                neighbors.add((p_2,costToMove))
            end
        end
    end
end
return neighbors

The algorithm for `reconstructPath` reconstructs the path backwards using the parent pointers. It is outlined as follows:

Algorithm 3: reconstructPath

Input: parentPointers
Result: Planned path from p_S to p_G
path $\leftarrow$ [p_G]
parent $\leftarrow$ parentPointers[p_G]
while *parent* **do**
    path.insert(0,parent)
    parent $\leftarrow$ parentPointers[parent]
end
path.insert(0,p_S)
return path

A Note on Dijkstra's Algorithm To compare our path planning algorithms with the optimal path length, we implemented the traditional Dijkstra's algorithm (which runs in $O(V + E \log V)$), which is also search-based and asymptotically optimal. Similar to A*, it expands in the direction of lowest cost until it finds the goal. Unlike A*, however, it does not take any heuristic into account, expanding outwards in all directions instead of prioritizing expansion in the direction of the goal like A*.

Because Dijkstra expands in all directions, it explores more locations than A*, leading to a much longer computation time on the order of minutes instead of seconds. Due to this constraint, this algorithm on its own is generally not as useful for self-driving vehicles, where planned paths often need to be generated quickly. However, as it does not consider the heuristic nor wall cost, it is guaranteed to generate the optimal path as it considers the actual incurred cost when traversing a path.

The implementation for Dijkstra is nearly identical to A*, where the only difference is that Dijkstra sets $h(p_2) = 0 \ \forall p_2$. As we aimed to use a traditional "vanilla" version of the algorithm as the baseline optimal path length, we set our tunable parameters to the following values to run with maximum resolution:

$$jumpSize = 1 \tag{9}$$

$$\alpha = 1 \tag{10}$$

$$C_W = 0 \tag{11}$$

2.2.2 RRT (Alex)

The next algorithm we pursued relies on a random-sampling-based approach. Compared to search-based path planning algorithms, sampling-based algorithms are generally able to achieve faster computational runtimes at the expense of lack of optimality and repeatability (i.e., the path is longer than the shortest possible path, and is slightly random each time). Solely within the context of this lab, vehicular performance does not rely on fast runtime because the path is generated before the vehicle begins to drive. In the broader context of developing general robotic capabilities, however, sampling-based algorithms are of interest for vehicles that need to plan or modify paths on-the-fly — for example, if the vehicle is performing simultaneous localization and mapping (SLAM), it will need to create new paths as it explores new areas of the map.

We down-selected to the rapidly-exploring random tree (RRT) approach for the simplicity of its implementation. In RRT, trees are collections of nodes which define potential vertices in the generated path, and connections between those nodes which define the set of nodes that constitutes a path. Two trees, one starting at the origin and one at the goal, are procedurally generated by forming nodes at random. These nodes connect to the closest existing node on each tree if no obstacles block the path. The trees connect (and therefore a path

is found) when a single node can be added to both trees simultaneously. This implementation of RRT does not take into account vehicular dynamics.

Some discussion should be given to RRT*, a modified version of RRT not implemented here. RRT* generates trees identically to RRT, but once the trees connect, it restructures the tree connections so that it finds shorter paths within the existing tree — essentially, for each node, if a node with lesser total path length is found nearby, it replaces the existing node. While RRT* generates shorter, straighter paths than RRT, its implementation is more involved and it has a longer runtime, so our team chose not to implement it.

The procedural steps of RRT are as follows:

Algorithm 4: RRT Main Procedure

```
Input: Origin, Goal, Map
Result: Planned path from Origin to Goal
TreeI = Tree.init(Origin)
TreeF = Tree.init(Goal)
while not ConnectionFound do
    RandomNode = generateRandomNode()
    CloserNodeI = RandomNode.moveCloserToTreeI()
    if noCollisions(TreeI, CloserNodeI) then
        TreeI.extendTo(CloserNodeI)
    end
    if noCollisions(TreeF, CloserNodeI) then
        TreeF.extendTo(CloserNodeI)
    end
    if AddedToBothTrees then
        ConnectionFound = True
    else
        CloserNodeF = RandomNode.moveCloserToTreeF()
        if noCollisions(TreeF, CloserNodeF) then
            TreeF.extendTo(CloserNodeF)
        end
    end
end
Path = ReturnConnectedNodes(TreeI,TreeF)
```

These functions are generally straightforward, but contain some subtle implementation details:

- Checking for obstructions between two nodes tends to be the most computationally expensive portion of RRT. The collision checker finds the map pixels comprising the line connecting the nodes via axis-wise linear interpolation, then references the map to determine whether there are obstructions on those map pixels.

- A basic approach to generate random nodes is to randomly choose points within the map boundaries, but this approach creates biases. Uniform random selection within the map biases the trees away from map edges, resulting in poor runtime where paths must closely follow the map edges. Instead, it is desirable not for the node positions to be uniformly random, but for their directions relative to each tree to be uniformly random. To approximate uniform directional randomness, we expand the map dimensions by 3x when generating points. Moving the random node closer to each tree usually brings it within the boundaries of the map; where it does not, the point is rejected. The exact expansion factor is non-critical — the likelihood any random point generates within the original map boundaries (where the effect of the bias is more present) is $(\frac{1}{\text{expansion factor}})^2$, which is less than 11.1% for any factor over 3x.

- The function moving the random node closer to the tree is defined below:

$$\vec{x}_{\text{new random node}} = \vec{x}_{\text{closest node}} + (\vec{x}_{\text{random node}} - \vec{x}_{\text{closest node}})/\gamma, \quad (12)$$

  where γ is the distance division factor, i.e., how much closer the point becomes. Note that there is a coupling between γ and the map expansion factor for node generation; when the map dimensions are multiplied by three, the most effective value of γ is three times greater. Section 3.2.4 of this report discusses the tuning of γ.

2.2.3 Pruning Nodes: Search after Construction (Alex)

For A* and especially for RRT, the path does not tend to take the shortest route possible even in regions where no obstacles are present; node pruning is performed to alleviate this issue. Once A* or RRT generates the path, we remove all nodes that are unnecessary to avoid obstacles. First, we check for obstacles between the start node and all the successive nodes; if there are none, the intermediate nodes are removed. We then progress to the next extant node and repeat the same process. We repeat this process until we arrive at the goal node.

Figure 3: An example of pruning nodes. The pruned trajectory is slightly shorter as a result of taking a more direct path. With fewer turns, the pruned path is also easier for pure pursuit to follow.

2.3 Pure Pursuit (Kai)

Using the trajectory generated by one of the two path planning algorithms, the car steers along the path using a Pure Pursuit model. This step requires 3 main inputs:

- Car Location: Gathered through odometry
- Lookahead Distance: Tunable and parameterized
- Trajectory: Array of poses from path planning

Lookahead Point We first established a goal point that the car will steer towards. This "lookahead" point is a transitive goal and allows the car to continuously pursue a point on the desired trajectory. We determined this point by using the intersection of a line segment (derived from the trajectory) and a circle derived from the lookahead distance and the car's location.

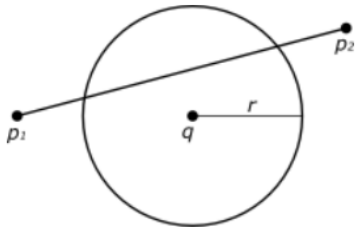

Figure 4: Geometric representation of intersection between a line segment and circle

Using the figure above, we define the following values in the world reference frame as:

q= Car Location
r= Lookahead Distance
p_1= Closest Node
p_2= Next Node

In order to find the intersection between a circle and a line segment, we first

find the following values:

$$V = p_2 - p_1 \tag{13}$$
$$a = V \cdot V \tag{14}$$
$$b = 2(V \cdot (p_1 - q)) \tag{15}$$
$$c = p_1 \cdot p_1 + q \cdot q - 2p_1 \cdot q - r^2 \tag{16}$$
$$discrim = b^2 - 4ac \tag{17}$$
$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{18}$$
$$intersection = Vt + p_1 \tag{19}$$

With these values, we can calculate the value of any intersection point if they exist. However, we must consider the following cases:

- $discrim < 0$: The line segment does not intersect the circle
- $0 \leq t \leq 1$: The intersection point does not lie within the bounds of the line segment

If a single point passes both of these conditions, then it is set as the lookahead point. If both intersection points pass the above conditions, then we set the point closer to p_2 as the lookahead point.

Edge Cases There are a number of edge cases to consider when choosing a lookahead point. These edge cases must be considered before the lookahead point is passed to pure pursuit since false goal points will cause the car to steer off course. The main edge cases to consider are:

1. The current line segment does not lie within the lookahead distance.
2. The intersection points found by the geometric model does not fit within the limits set by the line segment.
3. The closest point is the last node.

For each of these cases, the following solutions were implemented:

1. If there is no intersection using the current line segment, then continuously select trajectory poses further in the array.
2. Set the last node as the lookahead point.

Steering Angle Now that the goal point has been found, the pure pursuit driving model can be applied. However, in order for pure pursuit to work, the lookahead point must first be transformed into the reference frame of the robot.

To do this, we used a rigid body transformation using the robots localized pose and the lookahead point:

$$pose_{robot}^{world} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} p_{robot}^{world} \\ \theta \end{bmatrix} \tag{20}$$

$$pose_{lookahead}^{world} = \begin{bmatrix} p_{lookahead}^{world} \end{bmatrix} \tag{21}$$

$$R = \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix} \tag{22}$$

$$p_{lookahead}^{robot} = R \cdot (p_{lookahead}^{world} - p_{robot}^{world}) \tag{23}$$

Following the rigid body transformation, we can establish the lookahead point as a transient goal for the pure pursuit model as shown below.

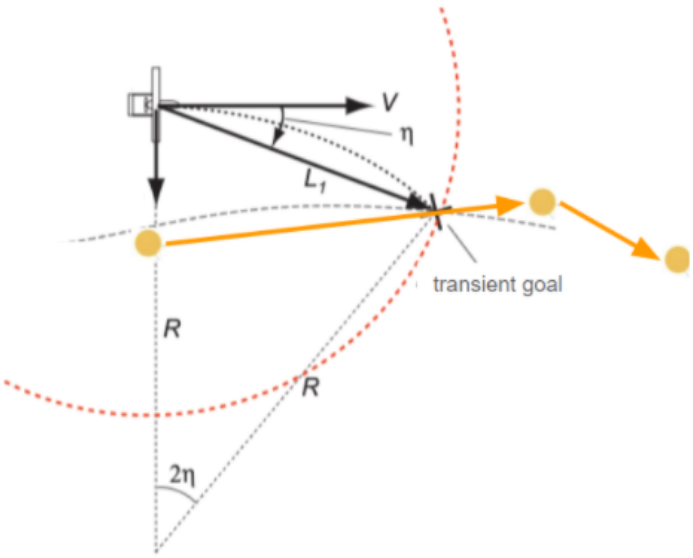

Figure 5: Visual representation of Pure Pursuit model

Using the transformed values for the lookahead point, we can find calculate the

steering angle:

$$L_1 = \text{Lookahead Distance} \tag{24}$$

$$pose_{lookahead}^{robot} = \begin{bmatrix} x \\ y \end{bmatrix} \tag{25}$$

$$\eta = tan^{-1}(\frac{x}{y}) \tag{26}$$

$$R = \frac{L_1}{2 * sin(\eta)} \tag{27}$$

$$\delta = \text{Steering Angle} = tan^{-1}(\frac{\text{Wheelbase}}{R}) \tag{28}$$

$$\tag{29}$$

3 Experimental Evaluation

3.1 Simulation Method (Alex)

In the analyses below, three unique representative start and goal location pairs were selected from the Stata basement map, each of which result in paths with different overall lengths and required turns. The paths shown below were generated with Dijkstra's path planning algorithm to ensure we found the shortest possible path to use for comparison. The Dijkstra runtimes and distances for each path are contained in the table below.

In the performance evaluation section, the two primary metrics we analyzed were runtime, the computational time it takes for a given algorithm to compute a path, and distance optimality, the generated path distance as a fraction of the shortest path distance.

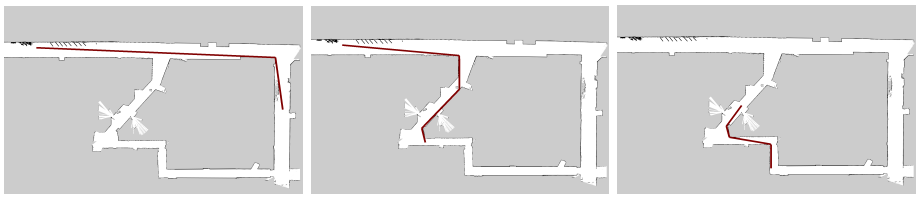

Figure 6: The three representative paths used for performance analysis. The first (left) path is long but mostly straight, the second (middle) path is shorter but more curvy, and the third (right) path is the shortest and most curvy.

Table 2: Dijkstra performance on the three representative paths.

Path	Runtime [s]	Path length [m]
Path 1	625.1	83.5
Path 2	145.0	62.7
Path 3	20.13	29.0

3.2 Tuning Path Planning Parameters

3.2.1 A*: Jump Size (Nick)

The jump size controls the resolution of the map when constructing the graph. A larger jump size means fewer pixels or nodes in the graph, which leads to faster computation but lower resolution. When the jump size is too large, the path becomes blocky and less direct. We chose a jump size of 0.3 meters to minimize computation time while keeping the path short.

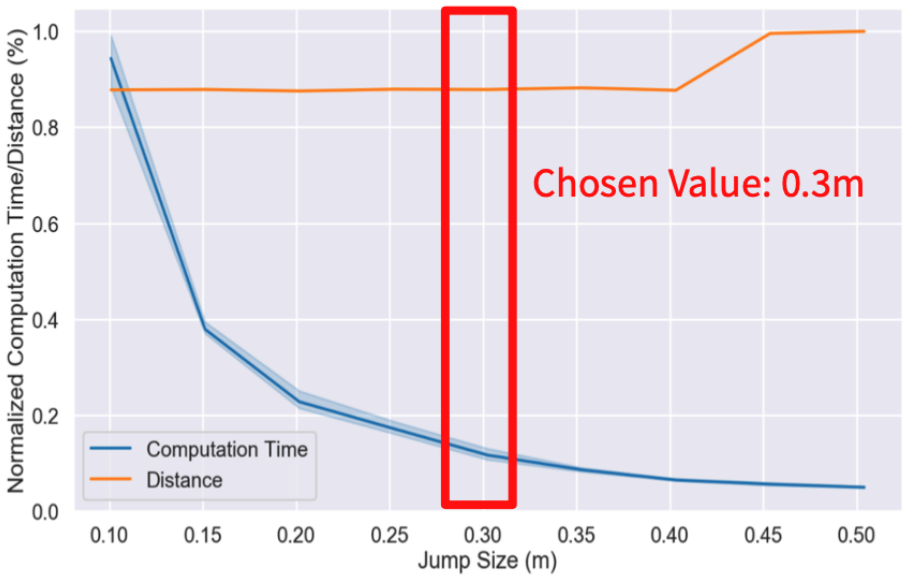

Figure 7: The path length and run time of A* for varying jump sizes on a singular path. We chose our jump size to be 0.3m to have a lower computation time and lower path length. We chose a smaller jump size than necessary to not be close to the area where path length increases. Depending on the map and start and end points, the exact jump size where path length increases can vary, so being further from that boundary gives buffer for this variability.

3.2.2 A*: Wall Cost (Nick)

We tuned the wall cost parameter of the A* heuristic in a similar manner. The wall cost parameter is added to the heuristic to penalize paths for being too close to a wall. However, this goal is often in opposition to the main goal of finding the shortest path. This leads to a "struggle" between these goals, with the heuristic winning out at some nodes, and the shortest path goal at others. Increasing the wall cost gives the heuristic more influence over the search.

This "struggle" is illustrated in Figure 8 as we tuned the wall cost parameter. Samples of the paths found by A* search are above three values of increasing wall cost(red to green). The red path (when the wall cost is small) hugged the wall leading to a sub-optimal, longer path. Looking at the orange path (made with a moderate wall cost), we see the wall cost heuristic began to win out as the dip by the orange arrow became a straight line. Finally, the green path (with a large wall cost) shows the wall cost win more and more as it moves the path by the green arrow. In fact, the search found an 11% shorter path when the wall cost was greater than 30,000. We chose a value of 40,000 to be conservative.

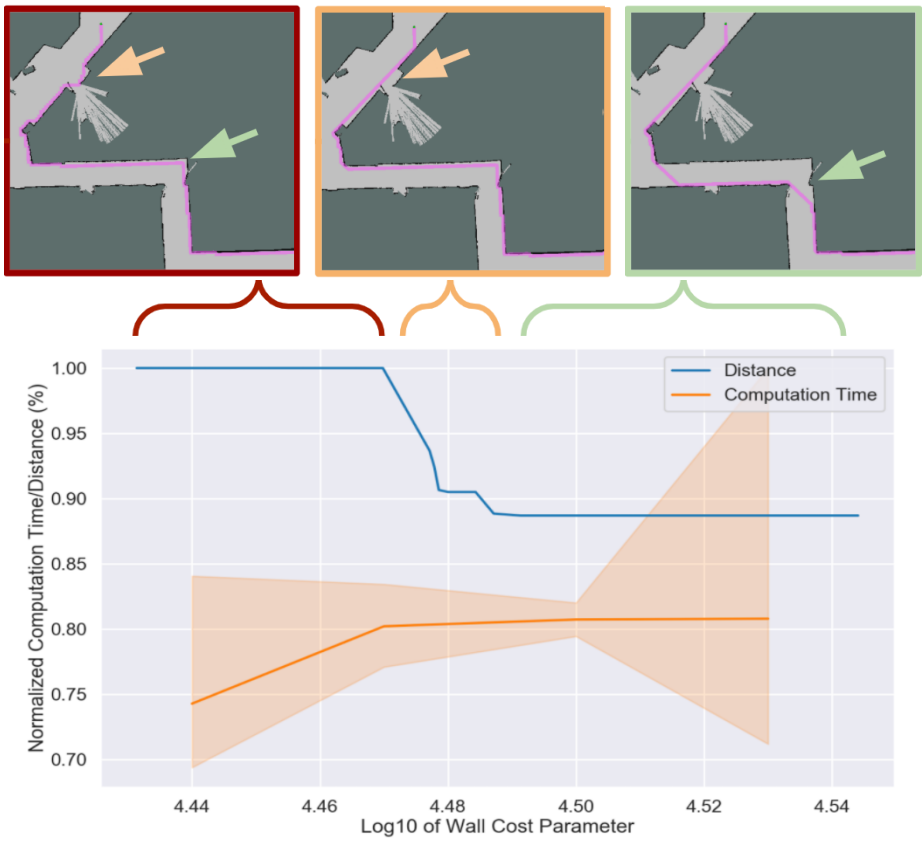

Figure 8: The path length and run time of A* for varying values of the wall cost parameter on a singular path. The wall cost parameter penalizes the search for being too close to a wall – larger values tend towards more direct paths, but slightly longer computation times. The red range in Figure 8 shows when the wall cost is small and the search found longer paths that hugged the wall. In the orange and green ranges, the wall cost started to gain influence on the heuristic and move the path away from the walls. When the cost is large enough in the green range (above 30,000), we gain an 11% increase in path efficiency.

3.2.3 A*: Performance (Alex)

The path generated by A* is deterministic, i.e., consistent across every run, but the runtime can vary from run to run. Therefore for the three representative paths described in Section 3.1, we calculated the distance normalized with respect to the Dijkstra-generated shortest path distance, and the mean runtime over 200 runs normalized with respect to the Dijkstra runtime than for longer, straighter paths.

Table 1:A* performance on the three representative paths

Path	Runtime ÷ Dijkstra runtime	Path length ÷ shortest path
Path 1	$7.003 * 10^{-3}$	1.0030
Path 2	$1.075 * 10^{-2}$	1.0069
Path 3	$1.586 * 10^{-2}$	1.0727

From these values and figures, we can see that A* is generally on the order of 100 times as fast as Dijkstra while producing paths that are only approximately 0.3% to 7% longer. For shorter, curvier paths, A* encounters a performance drop relative to Dijkstra, as its runtime increases and its generated path is lengthened compared to the Dijkstra path. If a faster but near-optimal algorithm is needed, A* appears to be an appropriate choice.

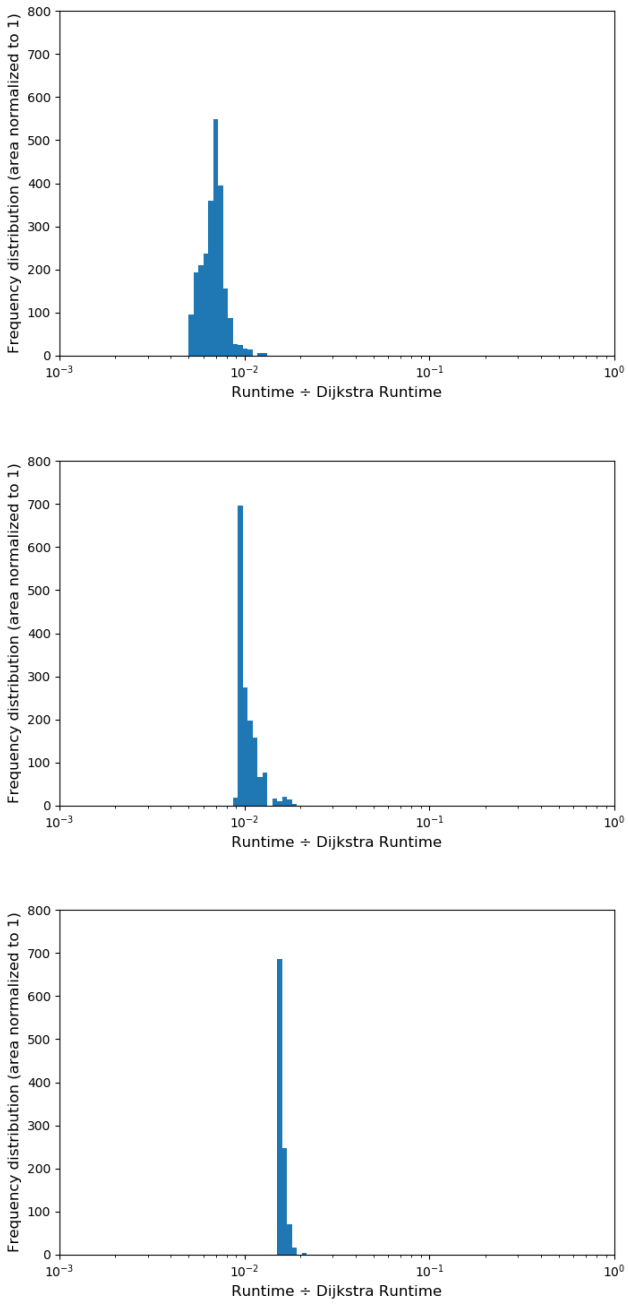

Figure 9: A* runtime performance on the three representative paths, path 1 (top), path 2 (middle), and path 3 (bottom). For shorter, curvier paths (2,3), the A* runtime is a larger fraction of the Dijkstra runtime, but generally A* is roughly 100 times faster than Dijkstra.

3.2.4 RRT: Distance Division Factor (Alex)

As discussed in Section 2.1.2, the distance division factor γ moves the randomly-generated nodes of RRT closer to the existing tree. This parameter can be tuned to improve performance. In this section, we look at the performance on reference path 1 exclusively; however, in our qualitative testing experience, the relationship shown in the graphs below held for other paths as well.

Based on the graphs below, we chose a distance division factor of 10 to balance runtime with distance optimality. Lower factors decrease path length meagerly for a significantly increased runtime (e.g., the median $\gamma = 2$ run took 6 times as long as the median $\gamma = 10$ run but its paths were only 3% shorter), whereas higher factors were worse in both runtime and distance optimality.

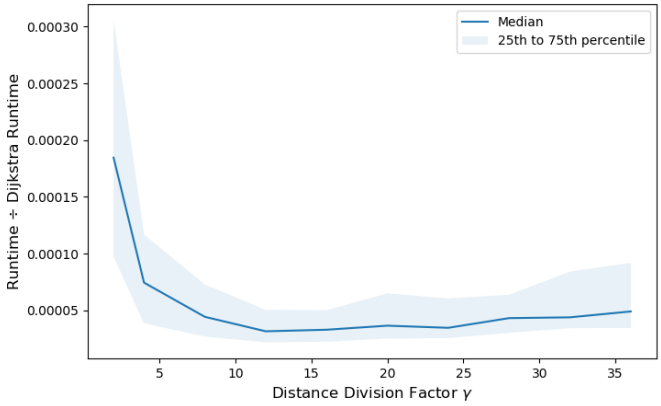

Figure 10: The effect of the distance division factor on RRT runtime on reference path 1. 400 runs were performed for each factor. Generally, low (2-5) factors increase runtime significantly compared to the best runtime, and high (20+) factors increase runtime marginally compared to the best runtime.

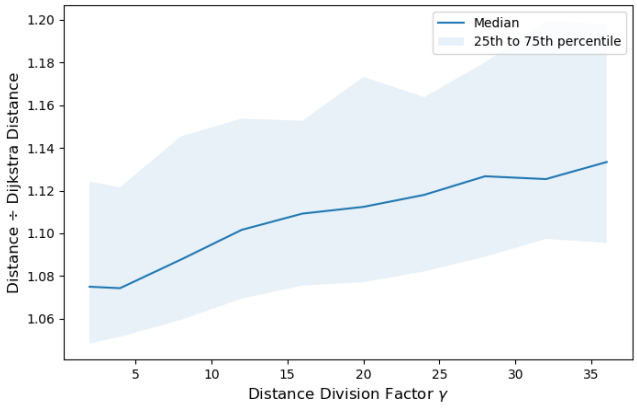

Figure 11: The effect of the distance division factor on RRT path length on reference path 1. 400 runs were performed for each factor. The general trend is that lower distance division factors result in shorter paths.

3.2.5 RRT: Single-Query vs. Multi-Query Performance (Alex)

Since RRT relies on random sampling to generate paths, any particular path instance has a chance of being either better or worse than the mean generated path length. This trait is generally not ideal because it means that some generated paths will take much longer routes than others. However, it can also be used to our advantage — by taking the minimum-length path from a number of "meta-samples" of randomly-generated paths, the overall runtime is increased, but the final generated path is likely to be shorter than the mean. This approach is typically called "multi-querying". The graphs below illustrate this tradeoff for various sample sizes, ranging from 1 sample (normal RRT path-generation) to 20 samples, analyzed on each of the three reference paths discussed in Section 3.1. 1,000 sets of each query size were randomly drawn with replacement from a pre-computed set of 10,000 RRT runs.

Statistical averages on Gaussians suggest that the runtime increases linearly with the number of samples whereas the minimum path length falls as the square root of the logarithm of the number of samples. Specifically, the following equation describes the bounds of the expected minimum path length value, assuming a Gaussian distribution:[1]

$$\frac{1}{\sqrt{\pi \log 2}} \sigma \sqrt{\log n} \leq E[Y] \leq \sqrt{2} \sigma \sqrt{\log n} \tag{30}$$

[1] Kamath,Gautam. "Bounds on the Expectation of the Maximum of Samples from a Gaussian".http://www.gautamkamath.com/writings/gaussian_max.pdf

where σ is the standard deviation, n is the sample size, and $E[Y]$ is the expected value of the difference between the mean path length and the minimum path length within the sample.

The actual distribution of path lengths is not Gaussian; it is heavily skewed right (i.e., the mean is greater than the median), and has a minimum-possible path length of 1 (compared to the shortest possible path). Therefore, the above expression should not be expected to hold strongly in this application, which is confirmed by the trend in the graphs. Since we know these bounds are under-estimates, wee only plotted the upper bound Gaussian-derived estimate of the expected minimum path length within each sample size.

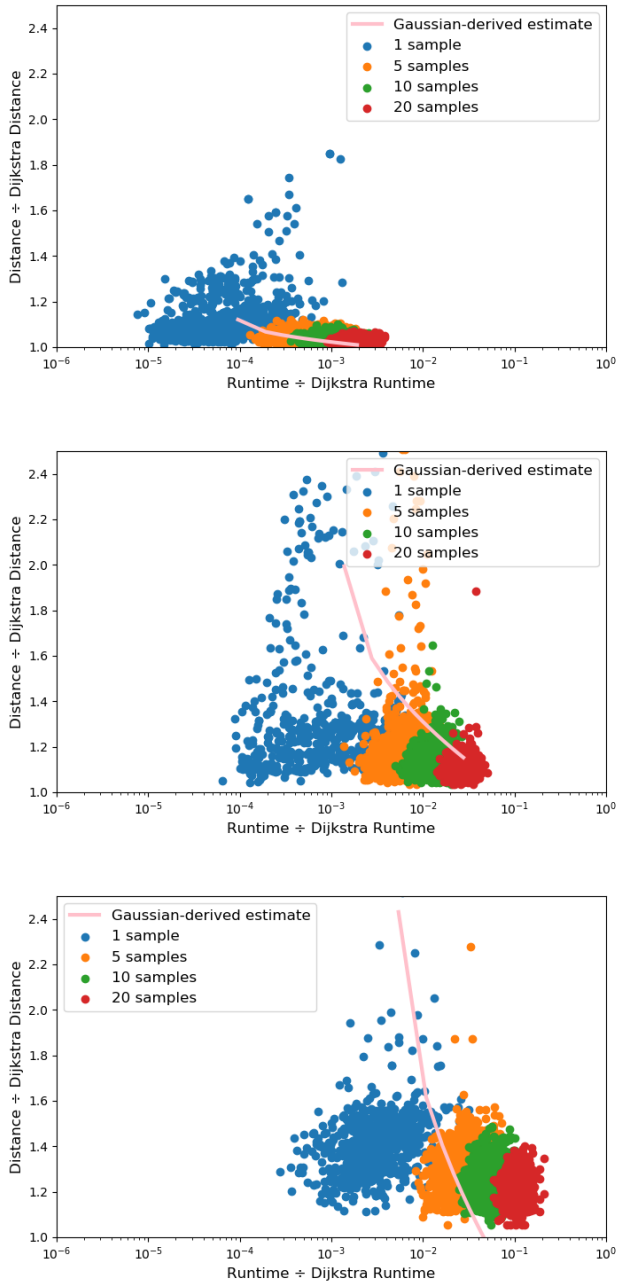

Figure 12: The effect of various RRT meta-sample sizes on the three representative paths: path 1 (top), path 2 (middle), path 3 (bottom). Using the minimum-length path from a greater number of generated paths increases the runtime, but is more likely to return a shorter path.

3.3 The Pareto Frontier of Runtime vs. Distance Optimality (Alex)

A Pareto frontier is the space of solutions where no performance specification can be improved without worsening a different performance specification. This frontier is of interest because it rigorously defines the best existing algorithms when tradeoffs exist between different performance metrics. For our path planning algorithms, the relevant Pareto frontier is the space of solutions which achieve the best distance optimality for various runtimes. We used Dijkstra's algorithm as a reference path planning algorithm as it is able to find the shortest path if a possible path exists. Dijkstra's algorithm, not shown in the graphs below, also represents a Pareto-optimal solution because it always achieves the shortest path, despite having long runtimes. As in the analyses above, we utilize three representative reference paths to determine this frontier. For RRT, we averaged over 1000 runs for each sampling number; for A*, we averaged over 200 runs.

One use of this frontier is to determine what the best algorithm is for a given runtime requirement. If we have a robot that needs to generate paths within, say, 10^{-2} seconds, we can generate the Pareto frontier for relevant paths the robot may need to generate to determine whether to use Dijkstra, A*, or RRT. Alternatively, if we have a certain distance optimality requirement, say 5% longer than the shortest possible path, we can determine which algorithm meets that goal with the shortest runtime.

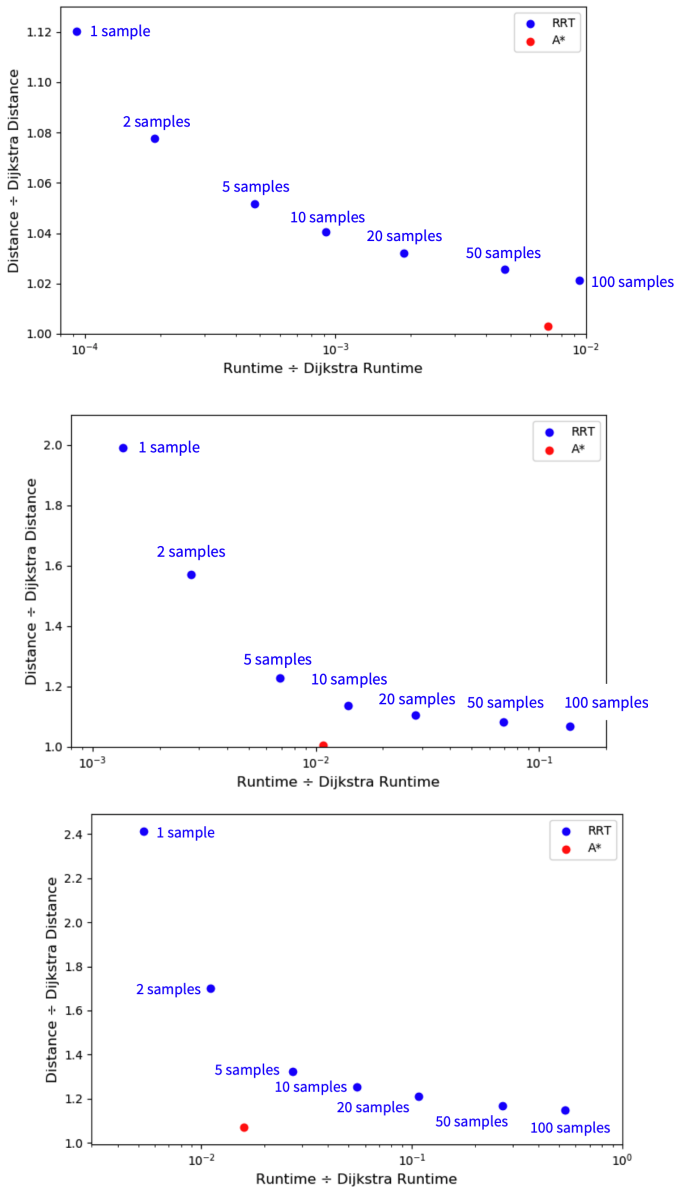

Figure 13: The performance-space of our path planning algorithms, evaluated on reference path 1 (top), path 2 (middle), and path 3 (bottom). Any solutions above and to the right of a given solution are not Pareto-optimal and are not in the Pareto frontier, since strictly better algorithms exist. Pareto-optimal RRT solutions generally use low numbers of samples (1-50 samples for the straighter path 1, and 1-2 samples for the curvier path 3) to generate the path.

From these graphs, A* has a lessened performance drop compared to RRT when utilized on shorter, curvier paths versus longer, straighter paths because the relative position of A* is further down (shorter path generation) and to the left (faster runtimes) compared to the bulk of RRT. On path 1 (which is relatively long and straight), 1-sample RRT is 12% longer than A* and 10^2 times faster, whereas on path 3 (which is relatively short and curvy), 1-sample RRT is 130% longer than A* and only 3 times faster. Therefore, A* tends to comprise a greater portion of the Pareto frontier on curvier paths. This result is expected because A* runtime is generally heavily dependent on path length but not on obstacles, whereas RRT can connect nodes arbitrarily far apart essentially instantaneously where there are no obstacles, but must generate random nodes to connect points which have obstacles between them.

Notably, on no path do any of the three algorithms (Dijkstra, A*, or RRT) strictly outperform the entirety of another algorithm — therefore none of the algorithms are strictly better than the other, but rather they each have their own niche in the path planning solution-space.

3.4 Tuning and Evaluating the Pure Pursuit Controller in 2D Simulation (Nick)

We tuned our pure pursuit controller by evaluating its performance on five predetermined trajectories. We measured performance as the average deviation (over 20 runs) in meters from these trajectories. We performed these simulations while varying the look-ahead and push parameters to determine values that minimize deviation from the path.

Look-Ahead Distance Parameter The look-ahead distance specifies how far in front of the car the controller should look to find a target point on the trajectory line. If this distance is too small, it can lead to instabilities and oscillations; too large, and it can can cause car to cut corners and stray from the path. As evident in Figure 14, we found a look-ahead distance of $1.0m$ provided the most stable following of the trajectory while reducing error on corners.

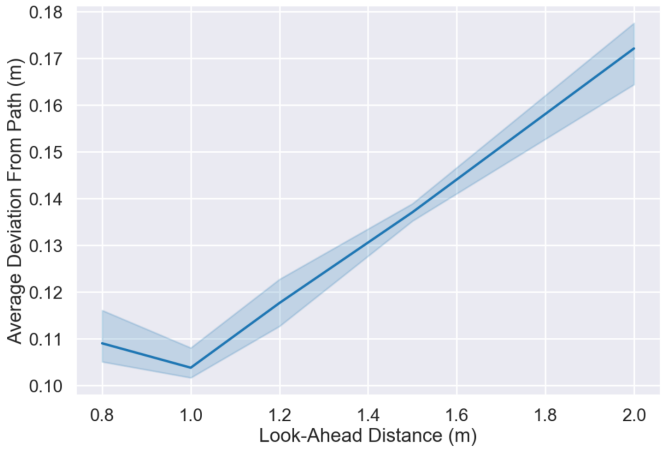

Figure 14: Tuning the look-ahead distance of the pure pursuit controller. We found a look-ahead distance of $1.0m$ provided the most stable following of the trajectory while reducing error on corners.

Push Parameter As the name suggests, the push parameter pushes the controller's perceived location of the car forward: it causes the controller to think the car is farther ahead than it actually is. This parameter can add stability and resistance to oscillations by allowing the controller to act earlier when the trajectory changes up ahead. Figure 15 shows the effects of this parameter on the performance of the pure pursuit controller. We found a value of $0.1m$ increased stability and lead to higher performance.

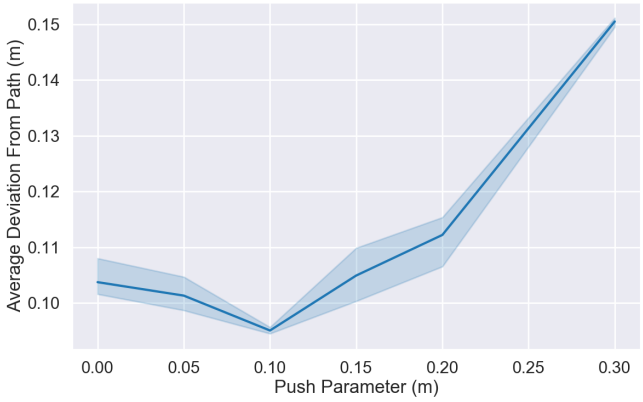

Figure 15: Tuning the push parameter of the pure pursuit controller. A value of $0.1m$ provided additional stability to the car and decreased trajectory deviations.

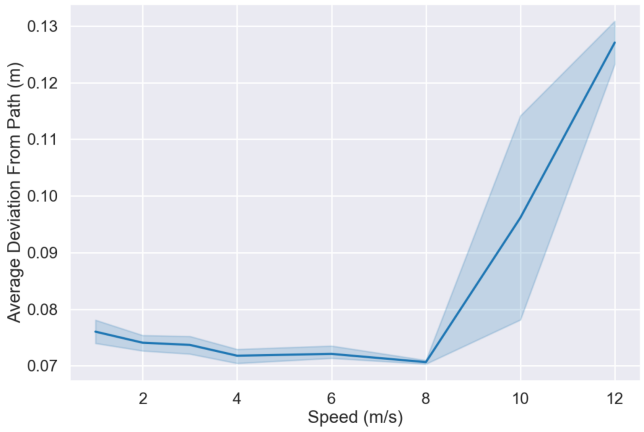

Figure 16: Performance of our pure pursuit controller in the 2D Simulation. Error was measured as average deviation from the planned path in meters. Our implementation had a peak accuracy of $0.072m$ at a speed of $4m/s$, and was stable up to speeds of $8m/s$.

Final Performance After tuning the controller parameters, we used the same simulations to evaluate its performance at varying speeds that pushed the limits of its capabilities. Our implementation was quite accurate across a range of speeds, with a peak accuracy of $0.072m$ at $4m/s$ as shown in Figure 16. This

accuracy held up to speeds as fast as $8m/s$ before the controller became unstable. Interestingly, speeds below $4m/s$ performed less accurately, perhaps due to an increased susceptibility to oscillations along the straightaways.

3.5 Performance on TESSE-Specific Maps (John)

The success of the RRT and A* path planning from RViz Simulation translated well to the TESSE Simulation environment, particularly the city roads map. However, despite the tuning of wall buffering parameters, the path's still tended towards hugging the walls tightly. While this did not create issues in the city roads map, the extraneous obstacles in the standard city map frequently created collisions when the path hugged the building walls tightly.

Due to the inherent randomness of the RRT path finding method, the car's performance varied from run to run even with the same start and end position. For example, in the following situation, two successive runs of RRT between the same set of points produced two separate viable paths. This is significant as the car had a fatal collision on the second path rather than the first.

Figure 17: Successive RRT path plans on the city roads map for the TESSE Simulator with the same start and end points.

Using paths created by the A* or RRT algorithms on the city roads map, the car was able to successfully follow paths with straights and corners. However, the car did oscillate even with tuning of the lookahead distance, speed and push factor. The car operated best at a larger lookahead distance of 5m or greater with a push factor 1.2m. Occasionally, sharp corners and street side objects did become issues. In the future, a combination of path planning execution and wall or object detection algorithms could be used to help avoid collisions.

When using the city map rather than the city roads map, the path planning algorithms tended to find routes that were plotted through uncharted objects on the map. Occasionally, this resulted in shorter but unrealistic paths. In the

following situation the path plotted by RRT on the city roads map plotted a less direct, yet feasible path through empty roads only.

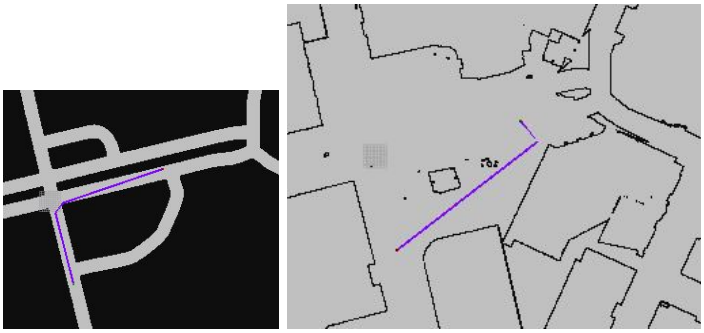

Figure 18: RRT path planning comprising the plotting on the standard map as opposed to plotting on city roads.

In comparison to RRT, A* took significantly longer in TESSE, but produced paths that were always in the most direct yet viable direction. However, because of this, the path frequently hugged the corners, infrequently causing issues when rounding corners with pure pursuit.

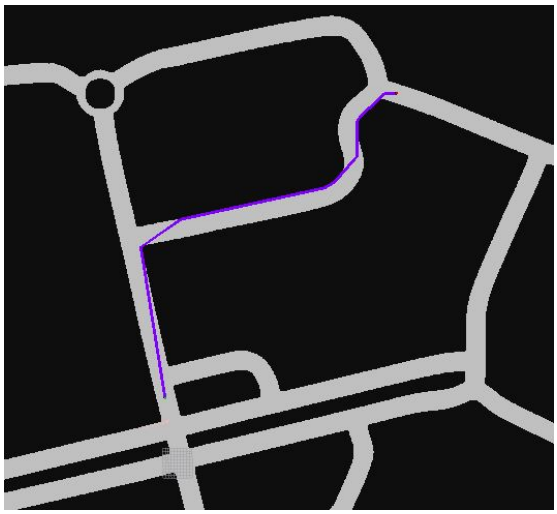

Figure 19: Path planned by the A* algorithm, comparable to the path starting and end point seen in the first photos.

4 Conclusion (Juliana)

Overall, we implemented path planning in both 2D and 3D simulations, exploring three path algorithms (Dijkstra, A*, and RRT) and implementing pure pursuit to follow the planned trajectories as closely as possible. Our pure pursuit model has a high performance as, given a trajectory, it is within 0.072 meters from the trajectory on average and can reliably follow the path when traveling up to 8m/s.

From our analysis, we found that our three path planning algorithms are unique compromises between runtime and path length. While each algorithm has strengths, none of the three are strictly better than another for all situations. For instance, Dijkstra's algorithm produces the optimal path but has the longest runtime, while RRT (by its stochastic nature) is nonoptimal but is by far the fastest. A*, though, lies in the middle of these two extremes, having a more optimal path length than RRT and the second fastest runtime. Depending on the model and the circumstance, it may be appropriate to choose one algorithm over the other.

This lab can certainly be expanded and improved upon. For instance, we can further improve on pure pursuit in TESSE because the simulated car is unstable when following paths. In addition, we can use path planning for obstacle avoidance, and use a variable velocity and lookahead distance to better follow curve trajectories. With these improvements, we can use path planning for obstacle avoidance, as roads often have light posts, traffic stops, and other objects.

With path planning underway, we look forward to strategically following paths, a feature that will be useful in the upcoming final race!

5 Lessons Learned

Nick I learned that tuning can have massive influences on the effectiveness of a controller, and that developing a consistent evaluation method is crucial to this process. I also learned to start early in this process, because simulations take a WHILE.

Juliana I learned that it often takes some creativity in implementing path finding algorithms. Although there are set algorithms out there for A* and RRT, we had to tailor the procedures to meet our expectations. For instance, we had to raise the costToMove and add a wall cost to discourage path kinks and wall clips in A*. Thinking "out of the box" was definitely useful when we were solving problems with path planning behavior as it produced unique and simple solutions!

Alex In this lab I learned the power of random sampling both to creatively generate solutions to problems in robotics as well as to analyze the performance of those solutions. I also found it highly useful to abstract away the path planning problem from its specific python/ROS implementation — most of my work was done with either a pen and paper or on a small test map I created with a single wall.

Kai This lab has taught me the power and importance of edge cases. When coding the pure pursuit model and establishing the lookahead point it was very important to constantly monitor the edge cases and thoroughly think through exactly how our model needs to respond to each case.

John I learned that in debugging things are often not what they appear. For example, I thought we had issues with our global to map transforms in the TESSE implementation however looking more closely at where and when our code was failing gave me insight as to where the edge case failures were. Had I not spent the time looking into where the failures came from and instead try to blindly fix things, I would have been spinning in circles with transforms. I also learned how to implement the search algorthims we learned in classes like 6.006 into real life with applicable nodes!