Lab # 6 Report: Exploring Path Planning and Pure Pursuit Algorithms for Autonomous Navigation

Team 7

Pranav Arunandhi
Thomas Fisher
Brady Klein
Calvin Maddox
Gustavo Ramirez (editor)

Robotics : Science and Systems

April 27, 2023

1 Introduction - Thomas Fisher

Lab 6 brings together the robotic capabilities developed in earlier weeks to deliver joint path planner and path follower modules which interface with each other to allow our robotic car to navigate a known map efficiently. Our path planner component would need to plan a path between a start and end point as a series of straight line segments, connected end to end. Next, the path follower would be responsible for controlling the robotic car's movements as it follows the path, and for handling cases like knowing when the destination has been reached and maintaining a reasonable speed. Additionally, successful completion of Lab 6 relies on the use of a functioning particle filter localizer and a well-tested safety controller module, adding extra importance to our team's thorough understanding of previous deliverables.

To achieve this goal, we implemented an RRT* path planning algorithm which uses repeated sampling to determine a trajectory of line segments between specified points in a mapped environment. Resampling and dilation parameters allow the algorithm to be tuned as to deliver the highest performance. Alongside our path planning module, we created a driver component to navigate the car along a provided trajectory using a pure pursuit algorithm, which takes a direct path from the current location to the goal. To maximize the amount of time that our team could dedicate towards improving the integrated product, we reused logic from the Lab 3 parking controller module in constructing our pure pursuit algorithm.

2 Technical Approach

2.1 Search-Based Planning - Gustavo Ramirez

Search-based planning algorithms are a family of path planning techniques that discretize the environment into a graph or grid and search for a feasible path between the starting position and the goal location. The algorithms explore the graph by visiting nodes, expanding them, and updating their cost based on a cost function. The cost function typically takes into account the distance traveled and the presence of obstacles.

A trade-off when choosing a search-based algorithm is that they can be computationally expensive, especially in complex, high-dimensional environments. These algorithms often run in exponential time, making them less suitable for real-time applications or scenarios where the environment changes rapidly. Additionally, the discretization of the environment might result in suboptimal paths, and the performance of the algorithm can be sensitive to the choice of heuristic.

2.1.1 A* Algorithm

Out of the Search-Based family, we chose to test the A* algorithm in simulation for its optimality guarantees and efficiency. A* is a widely-used search-based algorithm that provides an optimal path given an admissible and consistent heuristic. It differs from other search-based algorithms like Dijkstra's or Breadth-First Search by incorporating a heuristic function, which estimates the cost to reach the goal from the current node. This heuristic allows A* to prioritize the exploration of nodes closer to the goal, resulting in a more efficient search.

2.1.2 Implementation

In this section, we detail the implementation of our A* path planning algorithm, focusing on the two main aspects: 1) discretizing the map into a grid, and 2) running the search algorithm on discretized representation.

Discretizing the Map into a Grid

The first step in our implementation is to discretize the given map into a grid representation. This is done by subscribing to the `/map` topic, which receives information about our obstacles in the form of an `OccupancyGrid` message. The map is represented as a 2D array, with each cell containing occupancy information. The occupancy values range from 0 to 100, with 0 representing free space, 100 representing an obstacle, and -1 for unknown values.

The `OccupancyGrid` message also contains information about our map's width, height, and resolution. This allows us to reshape our map 2D array into a (1300, 1730) shaped numpy array.

With the occupancy array in place, our `plan_path` function can access the cells to execute the search. However, we must first ensure that the indices of our grid correctly correspond to the real-world x and y coordinates. To achieve this, we implement the following transformations:

```
def world_to_grid(self, x, y, map):
max_world_x_coord = map.info.origin.position.x
min_world_y_coord = -(map.info.height * map.info.resolution
    - map.info.origin.position.y)
u = int((y - min_world_y_coord)/ map.info.resolution)
v = int(abs(x - max_world_x_coord) / map.info.resolution)
return u, v

def grid_to_world(self, u, v, map):
max_world_x_coord = map.info.origin.position.x
min_world_y_coord = -(map.info.height * map.info.resolution
    - map.info.origin.position.y)
x = max_world_x_coord - (v * map.info.resolution)
y = u * map.info.resolution + min_world_y_coord
return x, y
```

A potential limitation of using a discretized version of the map is that it may not adequately account for the physical dimensions of the vehicle. In the process of finding the shortest path, the A* algorithm tends to navigate corners as tightly as possible to minimize distance, which is not always desirable, as shown in Figure 1. To address this issue, we employ a dilation technique using `ndimage.binary_dilation` with 10 iterations, illustrated in Figure 2.

```
self.map_array = ndimage.binary_dilation(np.array(map.data,
    dtype=np.int8).reshape((map.info.height,
    map.info.width)), iterations = 10).astype(float) * 100
```

Running A* search on Discretized Grid

The primary design choice in this phase is the selection of an appropriate heuristic function. In our case, we opted for the Euclidean distance as our heuristic function due to its simplicity and effectiveness in estimating the shortest distance between two points in a continuous space. The function is defined as follows:

```
def euclidean_distance(a, b):
    return math.sqrt((a[0] - b[0]) ** 2 + (a[1] - b[1]) **
        2)
```

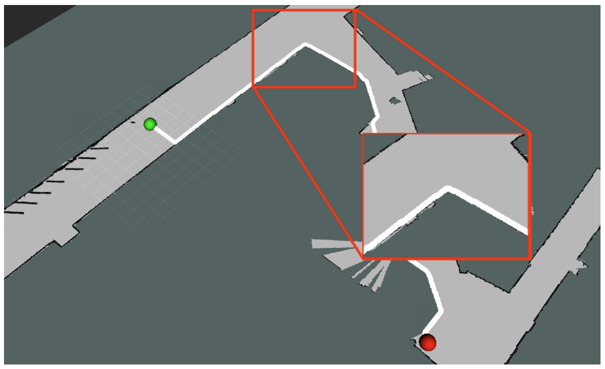

Fig. 1: **Undilated map results in a path that could lead to a collision with obstacles.** The figure illustrates the map without dilation, where the generated path (indicated by the white line) closely follows the corner of an obstacle. If the robotic car were to follow this path, it might crash into the corner due to its physical dimensions.

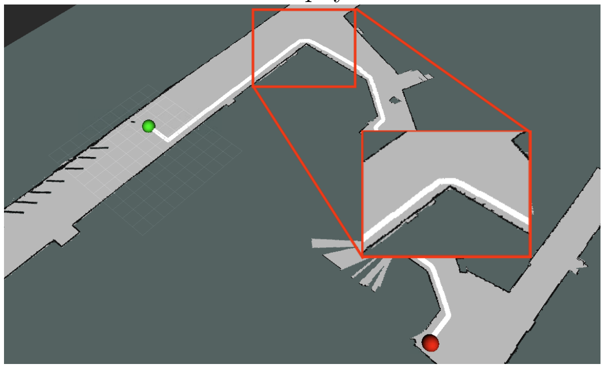

Fig. 2: **Dilation process significantly improves the path planning by accounting for the car's physical dimensions.** The figures display a comparison of the map before (top) and after (bottom) dilation. The dilation enlarges obstacles, creating a buffer zone around them, thus preventing the A* algorithm from planning paths that are too close to obstacles or involve taking corners too tightly. This enhancement contributes to a safer and more efficient navigation for the robotic car.

The actual pathfinding function, `plan_path`, follows the standard A* algorithm, with the following steps:

1. Initialize the open set, which contains the starting node, along with its cost and heuristic values. The cost is initially set to zero, and the heuristic value is calculated using the chosen heuristic function (in our case, the Euclidean distance).

```
frontier = []
heappush(frontier, (0, (start_u, start_v)))
cost_so_far = {(start_u, start_v): 0}
```

2. Check if the open set is empty. If it is, terminate the algorithm, as no path can be found.

```
while frontier:
```

3. Select the node with the lowest total cost (cost + heuristic) from the open set.

```
_, current = heappop(frontier)
```

4. If this node is the goal node, reconstruct the path by backtracking from the goal node through the parent nodes until reaching the starting node. Then, return the path.

```
if current == (end_u, end_v):
    path = reconstruct_path(came_from, current)
    break
```

5. Otherwise, remove the selected node from the open set and add it to the closed set (a set of nodes that have already been visited).

6. Generate the neighbors of the current node by considering the adjacent grid cells.

```
def get_neighbors(u, v):
    neighbors = []
    for du, dv in [(1, 0), (0, 1), (-1, 0), (0, -1)]:
        if 0 <= u + du < map.info.height and 0 <= v +
            dv < map.info.width:
            if self.map_array[-u + du, v + dv] < 50 and
                self.map_array[-u + du, v + dv] != -1:
                neighbors.append((u + du, v + dv))
    return neighbors
```

7. For each neighbor, calculate the tentative cost from the starting node to the neighbor through the current node.

```
for neighbor in get_neighbors(*current):
    new_cost = cost_so_far[current] +
        euclidean_distance(current, neighbor)
```

8. If the neighbor has not been visited before or if the new tentative cost is lower than the previous cost, update the cost for the neighbor, set the current node as the parent of the neighbor, and add the neighbor to the open set (if it is not already present).

```
if neighbor not in cost_so_far or new_cost <
    cost_so_far[neighbor]:
cost_so_far[neighbor] = new_cost
priority = new_cost + euclidean_distance(neighbor,
    (end_u, end_v))
heappush(frontier, (priority, neighbor))
came_from[neighbor] = current
```

9. Go back to step 2 and repeat the process until a path is found or the open set is empty.

After finding the path, we convert the path from grid coordinates to real-world coordinates using the `grid_to_world` function, which was defined earlier.

```
for u, v in path:
    x, y = self.grid_to_world(u, v, map)
    self.trajectory.addPoint(Point(x, y, 0))
```

Finally, we publish the trajectory as a PoseArray message and visualize it using the `publish_viz` function.

```
traj_msg = self.trajectory.toPoseArray()
self.traj_pub.publish(traj_msg)
self.trajectory.publish_viz()
```

While the chosen A* implementation is sufficient for our needs, there are other implementation choices that could have improved the algorithm's performance. For example, one could consider using a different heuristic function, implementing a bidirectional search, or employing a more efficient data structure for maintaining the open set.

Additionally, adjusting the resolution of the map could be a parameter that affects the performance of the A* algorithm. However, we encountered permission issues when attempting to change the resolution parameter in the `stata_basement.yaml` file provided for this project.

2.2 Sampling-Based Planning - Pranav Arunandhi

Sampling-based path planning algorithms randomly sample points in continuous search space to generate a graph, and then use that graph to estimate a shortest path from the start to the goal. The specific algorithm we used is RRT*, a modified form of the Rapidly-exploring Random Tree algorithm that, given sufficient time, provides a more optimal result than the base algorithm.

2.2.1 RRT* Algorithm

RRT* aims to generate and iteratively improve on a tree T that contains a path from a start to a goal. It does so by repeating the following steps:

1. Randomly sample a point p_{sample} in the search space.
   (a) To improve performance and guarantee termination, we overrode the randomly sampled point and used the end point as p_{sample} instead with a probability of `p` each iteration; we talk below about how we tuned `p` in Section 2.2.4.
2. If p_{sample} is not in free space, return to step 1.
3. Find the point $p_{near} = \underset{p_{tree}}{\operatorname{argmin}} \{\texttt{dist}\,(p_{tree},\ p_{sample})\ \forall\ p_{tree} \in T\}$, where $\texttt{dist}\,(p1,\ p2)$ computes the Euclidean distance between two points).
4. Generate a point $p_{projected}$ which lies along the vector from p_{near} to p_{sample} such that
   $\texttt{dist}\,(p_{near},\ p_{projected}) = \min\{\texttt{dist}\,(p_{near},\ p_{sample})\,,\ \texttt{maxProjectionDist}\}$; we talk about how we tuned `maxProjectionDist` in Section 2.2.4.
5. If $p_{projected}$ is not in free space or the path from p_{near} to $p_{projected}$ is obstructed, return to step 1 (path obstruction is determined by the Boolean function $\texttt{pathClear}\,(p_{near},\ p_{projected})$, which is explained in Section 2.2.3).
6. Find the set of points
   $P_{near} = \{p_{tree} \mid p_{tree} \in T,\ \texttt{dist}\,(p_{tree},\ p_{projected}) \leq \texttt{maxProjectionDist}\}\,.$
7. Using the maintained mapping `costs` that maps points in the tree to the estimated cost to travel that point, find the point $p_{nearest} \in P_{near}$ and cost $c_{nearest}$ such that $p_{nearest}$ is the minimizing argument and $c_{nearest}$ is the minimum value for the function $\texttt{costs}\,[p_{nearest}] + \texttt{dist}\,(p_{nearest},\ p_{projected})\,.$
8. Add $p_{projected}$ as a point in T with $p_{nearest}$ as a parent, and add a mapping from $p_{projected}$ to $c_{nearest}$ in `costs`.
9. For each point $p_{neighbour} \in P_{near}$, if
   $c_{rewire} = \texttt{costs}\,[p_{projected}] + \texttt{dist}\,(p_{projected},\ p_{neighbour}) < \texttt{costs}\,[p_{neighbour}]$, "rewire" T such that $p_{projected}$ is now the parent for $p_{neighbour}$, and remap $p_{neighbour}$ to c_{rewire} in `costs`.

The main improvement of the RRT* algorithm over base RRT is the rewiring step, which updates points close to the new point in order to optimize paths in the tree. Another improvement comes in terms of the termination condition. In RRT, the algorithm stops sampling as soon as it has found a path to the goal; in comparison, RRT* can take advantage of the fact that rewiring can optimize paths by continuing to sample even after finding a path to the goal (we discuss how we chose how much longer to sample for in Section 2.2.4).

2.2.2 Implementation

Just like the A* implementation, the RRT* implementation subscribes to the `/map`, `/pf/pose/odom`, and `/move_base_simple/goal` topics, which provide the map, current pose, and goal pose, respectively, and publishes a PoseArray to the `/trajectory/current` topic. The PoseArray is determined by the set of points on the path in the tree from the start to the goal after sampling has ended. The input pose values are converted from the 2-dimensional $(x,\ y)$ plane in the map frame to the $(u,\ v)$ plane in the grid frame. The search is done in the grid frame, following which the points are converted back to the map frame for the PoseArray.

2.2.3 Path Obstruction

We provide more information on the Boolean `pathClear` function. We use the 2D ray-tracing algorithm proposed in Amanatides and Woo (1987), which provides the set of grid cells that a line between two points will cross. We can then check each of these cells along the path and make sure they are free - if they are, the path is clear; if not, there is an obstacle along the path, and the path is not clear.

2.2.4 Parameter Tuning

There were three parameters that needed to be tuned in the RRT* implementation.

The first is `p`, the probability of short-circuiting to the goal. If `p` is 0, due to the random sampling nature of RRT*, there is a possibility of never reaching the goal. On the other hand, if texttp is too high, such as .1 or higher, we see "bouncing" behaviour near walls such as in Figure 3. This led us to reducing the probability lower of sampling the goal to .05, which did not exhibit the same bouncing behaviour.
The second parameter is `maxProjectionDist`. Setting this value to 1 leads to RRT behaving very similarly to a randomly expanding walk in the grid, which is very slow and worse in performance that A*; we found that even a value of 5 was too high to yield results sufficiently quickly. Setting the value to 20 or higher led to very quick results, but the paths returned were very jittery, since the nodes were farther apart, making it difficult for the car follower to smooth out the path. Based on this analysis, we found the value of 10 to be optimal.

The final parameter is how long to sample for after the algorithm has found the goal, which we encoded as a percentage of nodes sampled thus far. Again, we had to balance the runtime against the result. Leaving it at 0% meant that we were not seeing much benefit from using RRT* over RRT, since the paths generated were very suboptimal, such as in Figure xx. On the other hand, setting it to 25% led to the algorithm taking a very long time, and timing out on

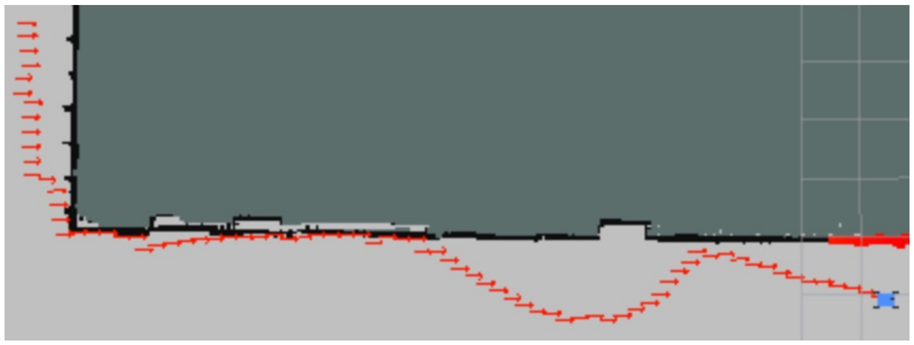

Fig. 3: **The "bouncing" behavior observed in RRT* when the goal-sampling probability parameter is set too high.** In this example, the path starts from the bottom right and moves towards the goal in the top left. However, upon reaching the wall, the path "bounces" away instead of continuing towards the goal. This behavior results from an excessively high goal-sampling probability, which can be mitigated by adjusting the parameter for optimal performance.

the autograder. We found that setting this to 10% led to both straightening out the paths and finishing relatively quickly.

2.3 Pure Pursuit - Brady Klein

While the problem of finding an efficient path between locations is critical to any autonomous robotic system, we must be able to efficiently traverse this path in order for the robot's motion to have any actual value. Using the pure pursuit tracking algorithm, our vehicle identifies a lookahead point on the calculated trajectory and executes the necessary steering commands to arrive at this point.

Figure 4 below illustrates the flow of our pure pursuit tracking algorithm. We start with the odometry data derived from our robot's particle filter localization. Using this estimated position and the calculated trajectory, we can find the closest waypoint on the trajectory. Next, we take the coordinates of this waypoint and the next index on our path to determine a line segment of interest. From here we can use relatively simple geometry in order to calculate a goal point along the line segment of interest. Once identified, we can calculate an appropriate steering angle and proceed toward that point. This process is iterative and will repeat until the trajectory goal is reached.
We will break down these steps in greater detail below.

2.3.1 Line Segment Identification

To begin following our trajectory, we must first identify the nearest line segment to our car's current localized position. We illustrate this problem below in Figure

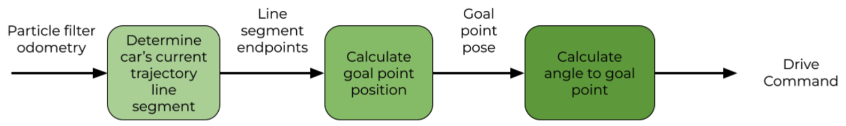
Particle filter odometry
Determine car's current trajectory line segment
Line segment endpoints
Calculate goal point position
Goal point pose
Calculate angle to goal point
Drive Command

Fig. 4: **Flowchart illustrating our pure pursuit controller algorithm.**

5.

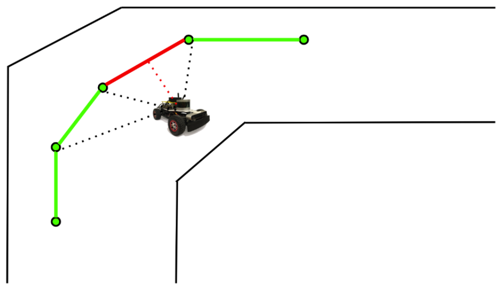

Fig. 5: **Our goal is to identify the line segment closest to our car, shown in red**.

To solve this problem, we can setup the following equation for each line segment:

$$t = \frac{(c_x - p_{1x}) * (p_{2x} - p_{1x}) + (c_y - p_{1y}) * (p_{2y} - p_{1y})}{L} \tag{1}$$

Here, L = the length of the line segment. We then use t to solve for the shortest distance, d, between our car's position and the line segment. From here, we utilize NumPy arrays to solve for the line segment which minimizes this distance and return its endpoints.

2.3.2 Goal Point Identification

Once we have a line segment of interest, we can construct a circle around our car with radius equal to a set lookahead distance and calculate the two points where this line intersects the circle. This setup for this problem is illustrated in Figure 6.
To solve for these intersection points, we can parameterize our system equation which gives us the following:

$$t^2(v * v) + 2t(v * (p_1 - q)) + (p_1 * p_1 + q * q - 2p_1) = 0 \tag{2}$$

We can easily solve this quadratic equation to find two values t which then allows us to solve for the two points on the line segment that intersect our circle.

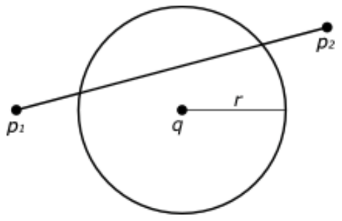

Fig. 6: **Our car's location is represented by point q, with a circle surrounding it having a radius, r, equal to our lookahead distance.** The objective is to determine the intersection point(s) between the line segment connecting points p1 and p2, and the circle. These intersection points represent the goal point(s) for the pure pursuit algorithm, guiding the car's movement along the planned trajectory.

We then set whichever point is closer to the next trajectory way point as our goal point.

An important edge case that we also handle is when the endpoints of the line segment fall within our circle. In this case, we will then look at the next line segment along the path to determine our goal point.

2.3.3 Steering Angle Calculation

Now that we have calculated the goal point, we must first transform these coordinates in the map frame to the robot frame in order to find a steering angle to publish. We use quaternions to calculate a rotation matrix and then use matrix multiplication to find the coordinates of our goal point in the robot frame. From here, it is simple trigonometry to determine a steering angle which we then publish.

3 Experimental Evaluation

3.1 Path Planning Evaluation - Calvin Maddox

3.1.1 A* and RRT* Comparison

In evaluating our path planning algorithm, our main metric of evaluation concerned the runtime of the algorithm. While we considered both A* and RRT* as viable options, ultimately the faster speed that RRT* was able to achieve in finding a path led us to choose it over A*. In order to test these two algorithms, we looked at four different types of paths, which are shown in Figure 7, where a start position is marked by a green circle, and the end goal is marked by a red circle. These four different paths tested the ability of the algorithms to find a path on both a direct and indirect route, as well as on a longer or shorter distance to the goal, with the most difficult task being roughly equivalent to the real-life track the car was to drive on. After running our algorithms on each of these paths, we found the runtimes shown in Figure 8, which clearly indicates that in each scenario, RRT* is able to outperform A* by finding a path in a shorter amount of time.

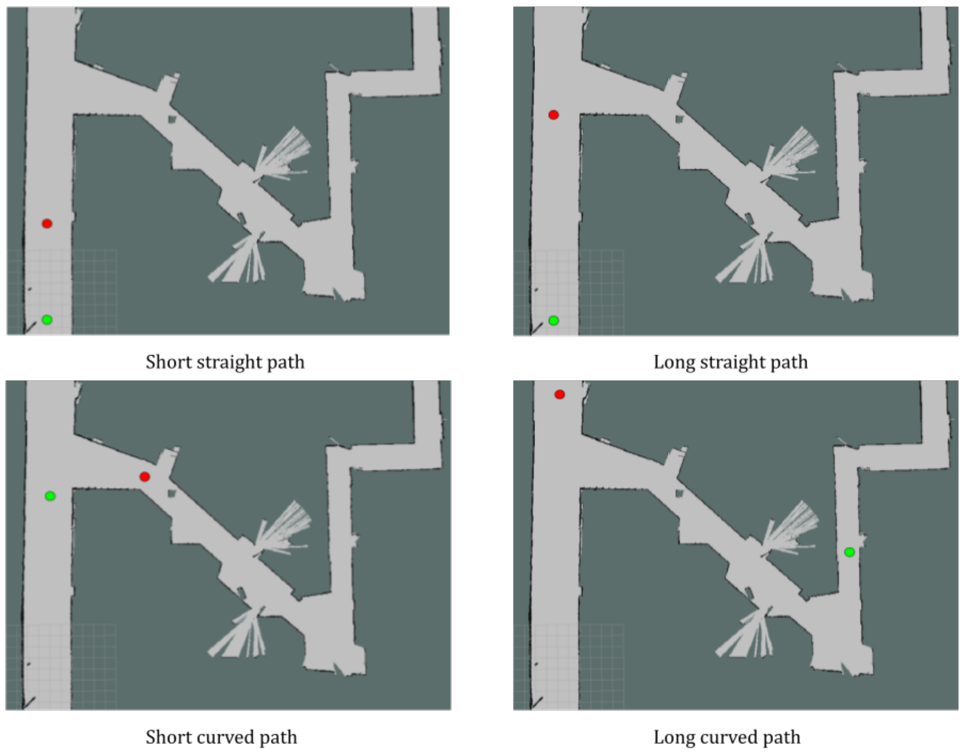

Fig. 7: **Four distinct path types for evaluating path planning algorithms.** These paths challenge the algorithms on direct and indirect routes, as well as varying distances to the goal. The green circle indicates the starting position, while the red circle denotes the end goal. Our evaluation revealed that RRT* consistently outperforms A* in terms of runtime efficiency across all path types.

3.1.2 Real-World Performance and Wall Dilation

While in simulation we were always able to successfully route a path without collision, in the real world producing a path requires more care for the physical size of the robot as well as other possible obstructions. In evaluating our real-

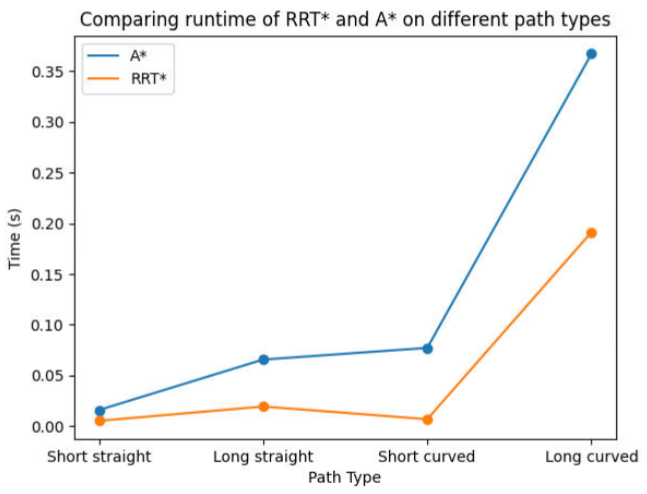

Fig. 8: **Runtime comparison between A* and RRT* algorithms across various path types.** The results demonstrate RRT*'s superior efficiency in finding a path, consistently outperforming A* in all tested scenarios.

world performance, we found that many of the paths produced by RRT* would either bring the robot too close to the wall, not leave room for error where the map was not completely accurate, or would collide with objects up against the wall. While in the last case we may not want to assume there will be objects against the wall, we wanted to safeguard our robot as much as possible, and so we experimented with a dilation factor which would, for the purposes of the algorithm, treat the walls as being larger than they were. In evaluating this metric, we looked at the tradeoff between the additional time it could take to add this factor, and how well it performed in preventing the robot from colliding with objects on its path. We evaluated a range of values from 1.1 to 2.0, and eventually found that a dilation factor of 1.2 was optimal for our implementation. Lower factors would fail to adequately prevent collisions by not leaving enough room, while larger ones would either take a noticeably longer amount of time to find a path, or would fail to find one at all due to the degree the walls were dilated to.

3.2 Pure Pursuit Evaluation - Thomas Fisher

Our pure pursuit module was tested to determine how accurately the car was able to follow a provided trajectory both in simulation and in the real world. Additionally, we show a data-supported foundation for the speed and lookahead parameters used by our algorithm by comparing the performance of pure pursuit across a number of different values.

3.2.1 Error in Path Navigation

To assess the performance of our pure pursuit implementation in simulation, we measured the euclidean distance between the car's ground truth pose and the nearest point to that pose out of the line segments composing the trajectory (a metric henceforth referred to as path error). Using ground truth pose instead of the car's estimated pose which is inferred from localization has the benefit of avoiding the noise spikes which are intrinsic to a sampling algorithm like the one used for the particle filter. Note that access to the car's ground truth pose is only possible for simulated runs, leading to one measurement difference between results shown in this section for simulated vs. real world runs.

Figure 9 below shows the path error during a simulated run of the car in the Stata basement map. The graph contains little noise due to the measurement of ground truth position and results in a overall low average path error for route containing many turns.

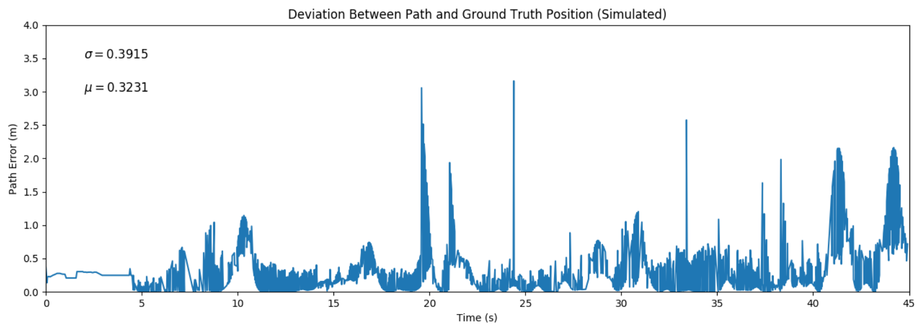

Fig. 9: **Simulated path errors are low in magnitude and consistent**. Slight peaks which correspond to switching line segments are an expected artifact of our algorithm's implementation. Mean and standard deviation are shown.

A similar graph of path error is shown below in Figure 10 which compares localized position against the trajectory which the car is following during a real world run. As expected for localization-based positioning, the run includes sudden spikes in error which we attribute to the random nature of our particle filter implementation occasionally seriously misrepresenting the car's location for a brief period of time. Also of note on both Figures 9 and 10 is the expected pattern of error spikes to reach the lookahead distance before returning back down to a near-average value at a more linear rate. This is due to the pure pursuit algorithm's "jump" from one line segment to the next when a trajectory waypoint falls inside the lookahead circle. Despite the appearance of large spikes in error through the car's navigation, this pattern confirmed an expected outcome of our implementation and did not significantly impact the overall average error.

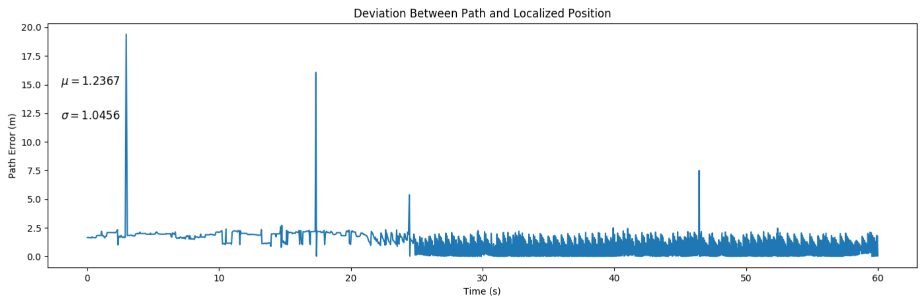

Fig. 10: **Real world path errors are low in magnitude and consistent**. Slight peaks which correspond to switching line segments are an expected artifact of our algorithm's implementation. Noise spikes are also an expected result of using particle filter localization. Mean and standard deviation are shown.

Additionally, we also recorded the change in path error over time for the same run to collect a measurement of how steady the car's route was to its goal point. For a direct route with few oscillatory turns, we expected that the change in path error from one cycle to the next would remain nearly constant. However, our results (shown in Figure 11 below) demonstrated that the change in error was much more oscillatory than we had anticipated, leading us to believe that the car's localization made it difficult to pin down a consistent value by which the error changed per cycle.

3.2.2 Parameter Tuning

Our pure pursuit algorithm contains a number of parameters which dictate the performance of navigation. The speed of the car and the lookahead distance are the two primary parameters which respond to tuning. To use the best values of these parameters possible in our implementation, we tested pure pursuit on the staff provided trajectory for a range of different values, using the metrics in Section 3.2.1 to determine best performance. Table 1 below demonstrates a collection of data over 4 runs in simulation which supports our team's choice of velocity value.
While increased speed is observed to lead to greater deviation from the supplied trajectory, our team chose to prioritize speed and will look to tune other parts of the pure pursuit algorithm in the future to lower the associated path error. We performed similar analysis on the value of lookahead to arrive at the value used in our final implementation.

Shown below in Figure 12 is a visual representation of a similar comparison

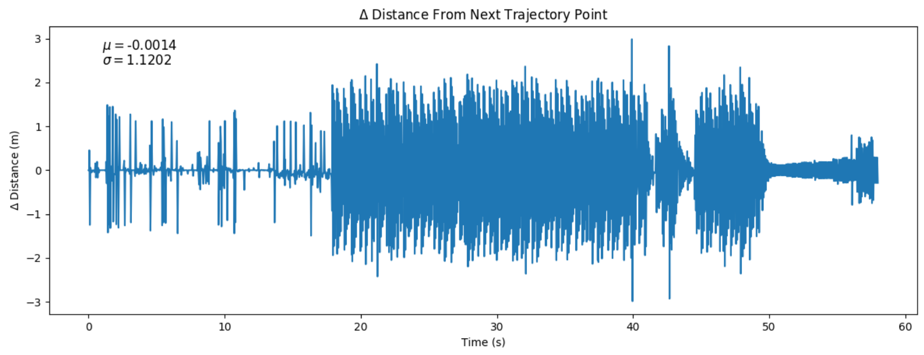

Fig. 11: **Oscillatory change in real world change in path error over time suggests our initial assumptions regarding this metric were incorrect.** Values observed were nearly constant in magnitude but varied rapidly between positive and negative values. Mean and standard deviation are shown.

Speed (m/s)	μ (m)	σ (m)	Time (s)
0.5	0.3298	0.2027	297
1.0	0.2627	0.1742	149
1.5	0.9142	0.4872	122
2.0	1.0324	0.5804	104

Table 1: **Experimental data shows a trade-off between accuracy and time.** Comparing the performance of pure pursuit across a number of speed values allowed our team to select the value which best affected the outcome of the metric we weighted most heavily.

used by our team in deciding upon a performant value of lookahead distance. By examining the error data (specifically average path error) of runs over identical trajectories and speeds with varied lookahead values, we were able to settle upon a value of 1.5 m for our final implementation. Note that our team did not test speeds greater than 2 m/s due to the high risk of damage to the racecar. Tests at and exceeding that speed will be required for the final challenge, however.

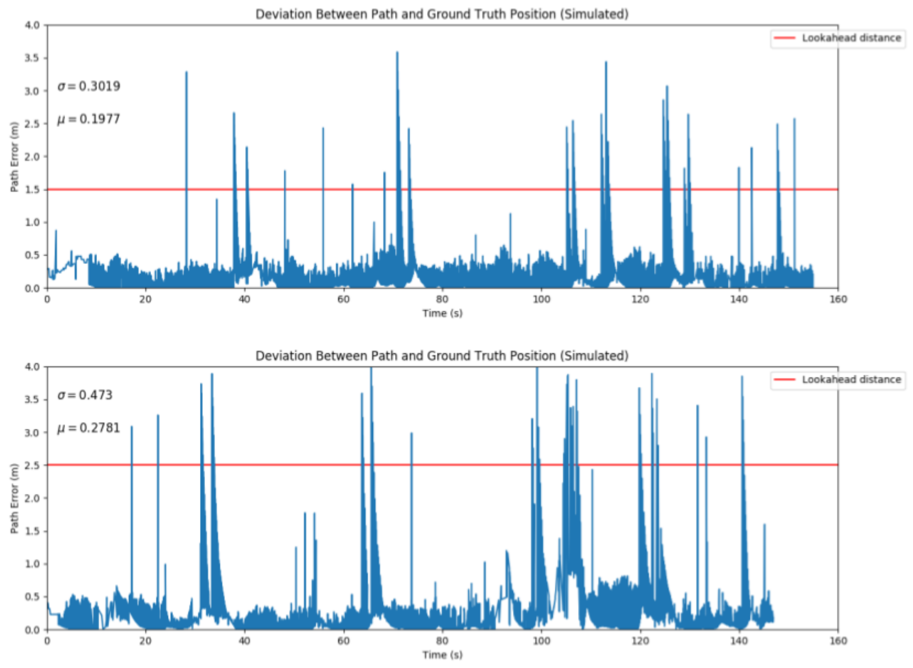

Fig. 12: **Visual comparison of error allowed our team to select an accurate lookahead value.** Examining the mean and stadard deviation (shown) for different lookahead values (red lines) revealed how adjusting this model parameter affected the performance of pure pursuit.

4 Conclusion - Calvin Maddox

In this lab, our team implemented a two-part approach to autonomous driving, the first part planning the path that the car takes using a sampling-based algorithm, and the second following that trajectory using a pure pursuit controller. Through our testing we have shown both that each part on its own is capable of performing its designated task, as well as the ability of these two parts to work in tandem to produce a robust strategy for autonomous navigation by our robot. Additionally, we have shown that based on our evaluation of various parameters involved in each algorithm, from the dilation factor of the map used in the path planning, to the speed and lookahead distance used in pure pursuit, we were able to improve the performance of our overall implementation.

As with most labs however, there are still places where we could spend more time. In this case, while we have satisfactorily tested in simulation, and were able to visually confirm the efficacy of our algorithm in the real world, there is still room for additional testing in this environment. In particular, spending additional time attempting to minimize the error in distance from the path of our robot while maximizing the speed it is able to follow the trajectory it is given could result in even better performance at this task.

While there is still room for growth, we have been able to achieve a robust solution to the task of autonomous navigation, and are now ready to bring this solution and the knowledge we've gained while implementing it to the final challenge.

References

Amanatides, J. and Woo, A. (1987). A fast voxel traversal algorithm for ray tracing. *Proc. Eurographics '87*, pages 1–10.

5 Lessons Learned

5.1 Pranav

Building directly on top of work we had done for previous labs by integrating the path planner with our localization module emphasized the importance of understanding the full stack of our system, as it introduced an entirely new suite of challenges that were not present in other labs. We've also managed to hone our skills at delivering effective briefings on our results - I think it reflected well on our abilities as a team as, although at the time of the briefing, our results weren't yet what we had hoped for, we were able to explain our future plans concretely and follow through on that in time for this report. Personally, I have been able to improve on my debugging skills even further in this lab - the parallelization of tasks led to needing to understand different coding styles when integrating the path planning and pure pursuit aspects of the lab in order to address any issues that arose.

5.2 Thomas

Lab 6 presented our team with unique challenges due to the complexity of the software involved. By this point in the semester, the module our team was developing for Lab 6 depended on the correctness of several other components developed over the course of the class. This interconnectedness meant that debugging issues was often a difficult process that involved examining failure points from a high level and investigative testing was a must. Our team made great progress in coordinating code changes shared among members, relying on each other's familiarity with previously-implemented components, and dividing into further "subteams" to better leverage the benefits of parallelization.

5.3 Calvin

Like in lab 5, this lab really tested our skills in integrating work from across our team, which presented challenges both with performing this integration, and with testing the overall performance, as there were multiple ways we could attack the task of maximizing the performance of our implementation. Additionally, accounting for the differences between navigation in the simulation and navigation in the real world, which required considering a number of factors such as the physical size of the car as well as possible inaccuracies in the map, was an interesting challenge to face. I think we've been able to find a good groove for division of work as well as working our communication channels through this and previous labs, which puts us in a great place for the final challenge.

5.4 Brady Klein

Something we have learned and that I think that we did well this lab was letting each other really play to our strengths. Because of the magnitude of this lab, it was imperative to split up our work. We found that certain members of the team had knowledge well suited to particular sections of the lab. By allowing these members to handle the parts they felt most comfortable with and filling in the gaps, we were able to work through certain parts of the lab extremely efficiently. Like the previous lab integration was very important because of how we had to split up the work. We had to integrate not only the overall path planning and following system but also within the systems themselves. Because of this, communication was of the upmost importance.

5.5 Gustavo Ramirez

Throughout the technical aspect of this project, I realized the crucial role of visualization tools in identifying issues and validating my work. Incorporating markers and other visualization aids in RVIZ proved to be invaluable in quickly pinpointing problems and better understanding the behavior of the A* algorithm. Additionally, I noticed that a considerable amount of time spent on debugging was due to understanding the intricacies of various packages, libraries, and the code we had previously built. This experience highlighted the importance of keeping track of required dependencies and ensuring that the code I write is compatible with them. On a collaborative level, our teamwork and communication style have significantly improved. We have become more adept at modularizing given tasks and working in parallel, which allowed us to tackle complex problems more effectively. This continuous growth in our team dynamics has contributed to the success of this project and prepared us to face the final challenge.