Lab 6 Report: Path Planning and Following Utilizing A*, Monte Carlo Localization, and Pure Pursuit

Team 3

Thomas Buckley
Fiona Wang
Sriram Sethuraman
Ben Lammers

6.4200 RSS

April 18, 2025

1 Introduction

Ben Lammers

As we continue to develop, build, and test algorithms for our autonomous vehicle, having access to information about the robot's position will allow us to solve more complex problems.

In Lab 6, our goal was to design a path planning algorithm that uses our existing localization algorithm and a map of the surroundings to navigate from itself. For localization, our solution creates a series of guesses around initial estimate, moves those guesses around based on the robot's motion, and evaluates those guesses based on how our LiDAR data match our map. After this evaluation, we would have an estimate of the robot's position. Then, a user can provide a goal position for the robot. We then use a path planning algorithm do determine waypoints between the current position and goal, and then use Pure Pursuit to naivgate along a smooth point using waypoints from the start position to goal. We forecast success as giving a smooth and accurate estimate of the robot's position.

As our platform for development, test, and evaluation, we continue to use the same racecar for this lab as in past labs, which is a 4-wheel (front-wheel steering racecar) powered by an NVIDIA Jetson running ROS2.

2 Technical Approach

Ben Lammers, Sriram Sethuraman, Fiona Wang

The technical task was to publish the real-time position of the robot on a provided map, given an initial position guess, odometry data, and LiDAR data. Our approach relied on the Monte Carlo localization method. First, we take our initial guess of the position and orientation and sample an array of 200 particles (each with a different orientation and position) from a normal distribution centered at that guess. Next, we update those particles based on the motion of the robot using a motion model. Then, we calculate the probability of each particle representing the correct position and orientation of the robot using a sensor model that takes in LiDAR data. A particle filter then determines which particles are most likely to represent the true position, publishes a position estimate (from the average of those particles), and resamples the particles to concentrate around the most likely positions. The resampled particles are fed back into the motion model, and the process repeats.

2.1 Monte Carlo Localization

2.1.1 Motion Model

The goal of the motion model is to use odometry data to update our current particles so that they reflect the movement of the robot. The motion model takes in an array of particles, as well as odometry data, including estimated changes in position (Δx, Δy, $\Delta\theta$ in the robot frame). This array is either from the previous iteration's particle filter (described below) or sampled from a normal distribution centered at our initial guess position.

After receiving the array of particles, the motion model uses odometry data to generate updated particle positions. To account for uncertainty in the odometry readings, we add noise to the transformation equations. The noise not only captures error accumulating in the odometry data but also helps prevent premature convergence of the particles onto a single position and orientation. With a greater range of position hypotheses, the sensor model can evaluate a broader range of likely positions around the estimated state, improving overall accuracy. The final equations, with noise incorporated, are shown below.

$$x_{i+1} = x_i + (\Delta x + \varepsilon_x)\cos\theta_i - (\Delta y + \varepsilon_y)\sin\theta_i \tag{1}$$

$$y_{i+1} = y_i + (\Delta x + \varepsilon_x)\sin\theta_i + (\Delta y + \varepsilon_y)\cos\theta_i \tag{2}$$

$$\theta_{i+1} = \theta_i + \Delta\theta + \varepsilon_\theta \tag{3}$$

As we can see above, the robot's motion data are transformed by the previous estimate of θ, denoted by θ_i. We also add noise (ε) to our updates. For each particle noise is sampled from a normal distribution for x, y, and θ:

$$\varepsilon \sim \mathcal{N}(0, \sigma^2) \tag{4}$$

The standard deviation used for our normal distribution, σ, was found experimentally by testing multiple values in simulation, which is discussed further Section 3.1. The output of the motion model is a new set of particles, each representing a hypothesis for the robot's current position and orientation after accounting for the estimated motion and added noise. These updated particles are then passed to the sensor model.

2.1.2 Sensor Model

The goal of the sensor model is to determine the probability that each particle is the pose of the robot. To do this, we are given the map, the robot's most recent LiDAR scan, and the current particles poses as inputs. The output is an array containing a probability corresponding to each particle. This component is essential for our particle filter to be able to accurately localize itself without relying on pure dead reckoning, which is prone to accumulated error.

To do that we generate what each particle would see in LiDAR, and compare that with the real LiDAR data. At each particle, we multiply all the individual probabilities of each LiDAR beam matching the measured value (obtained using look-up table). We then normalize and return this array of probabilities to the particle filter.

2.1.3 Particle Filter

Our final task for our Monte Carlo localization method was to assemble our particle filter using our motion and sensor model. To do this, we implemented multiple callback functions to initialize our particles, compute probabilities, resample particles, and publish our pose. First, we select an initial pose using the 2D Post Estimate tool in RViz. Using the selected pose, we initialize our particles using a multivariate normal distribution surrounding the chosen point. Although the particles will be resampled in other callback functions based on the laser scans and odometry data, we chose to initialize with a uniform distribution to prevent the kidnapped robot problem from occurring.

After initializing, our odom_callback function determines the change in position using the twist component of the OdometryMessage. Inputting the change in position, we update the particles using the motion model explained above. Next, our laser_callback function takes a laser scan and first, downsamples the number of scans to 99. We chose to downsample from over 1000 to under 100 scans because we observed that it helped reduce how "peaked" our probabilities were and it required less ray casting. The callback function then computes the probabilities using our sensor model, raises the output by a power of $\frac{1}{3}$, and normalizes the probabilities. By raising the probabilities to a power of $\frac{1}{3}$, we were able to "squash" and spread out the probabilities, which prevented high probabilities from causing a quick and erroneous convergence.

Finally, we publish our new pose in our publish_pose function by finding the 'average' position. To do this, we find the weighted average of the x and y values based on our probabilities and find the circular mean for our θ value. To do this we used the formula:

$$\theta = \text{atan2}\left(\frac{1}{n}\sum_{j=1}^{n}\sin(\alpha_j), \frac{1}{n}\sum_{j=1}^{n}\cos(\alpha_j)\right) \tag{5}$$

Additionally, we also published our particles to RViz to help with visualization and debugging.

2.2 Path Planning

For the path planning portion of our navigation stack, we decided to implement and analyze two different algorithms—a sample based planning algorithm called RRT and a search based planning algorithm called A*. Both of these approaches had their advantages and disadvantages for our racecar's mission. To evaluate these tradeoffs, we implemented both algorithms, evaluated their performances, and chose to apply A* to our actual racecar.

2.2.1 A* Planning Algorithm

Our implementation of A* is a search-based planning algorithm that works by converting our map into a discrete graph and searching for the shortest discrete path through that graph to the goal.
To start, our algorithm discretizes and downscales the provided occupancy grid by downscaling 10x10 pixel blocks into single pixel blocks. If any of the pixels in the block are occupied, the entire downscaled pixel will be marked occupied. Next, we connect each adjancent grid pixel as a node in a graph, as we see in Fig. 1 below.

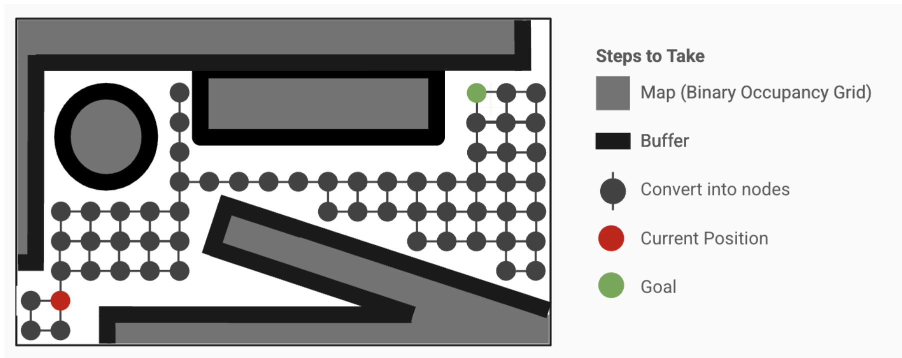


Figure 1: The map (in gray) is downscaled into a graph with connected nodes, which enables A* to search the graph.

After the graph is created, we run an A* search starting from the current position and search for the goal. The A* algorithm explores nodes in the order that they minimize the objective function below:

$$f(n) = g(n) + h(n) \tag{6}$$

In this equation, $g(n)$ represents the accumulated cost (number of nodes that were explored to get to this node), and $h(n)$ is a heuristic, which in this case is the Euclidean distance to the goal:

$$h(n) = \sqrt{(x_n - x_{goal})^2 + (y_n - y_{goal})^2} \tag{7}$$

$h(n)$ is always less than the true cost to the goal, because the true cost must be at a minimum the Manhattan distance to the goal, which is greater than or equal to the Euclidean distance to the goal. Thus $h(n)$ is an admissible heuristic, and A* will return the most optimal path to the goal (though this is affecting by the resolution of our graph). This is the primary advantage of A*. The primary disadvantage of A* is runtime $O(b^d)$, where b is the branching factor 3, and d is the length of the solution (in number of edges). Given the high number of edges in the graph, this grows quite fast.

2.2.2 RRT Planning Algorithm

The overall strategy of RRT is to grow a tree by randomly sampling our space and connecting the feasible paths to our tree until we reach our goal location. To tackle this task, we start from our given start position and randomly generate new nodes using a uniform distribution. To help our tree grow towards the goal position, we implemented a goal bias that returns the goal position instead of a random node 10% of the time, helping push our tree in the correct direction. After generating a random node, we find the nearest pre-existing node in our tree and create a node along the edge between the random and current node that is a set ”step” distance from the current node. This “step” distance is a parameter that we tuned while testing our algorithm, and it controls the resolution of our path; a smaller step size creates a bumpier path while larger step size utilizes longer, straighter lines. This process can be visualized in Fig. 2 below, showing how random and new nodes are generated.

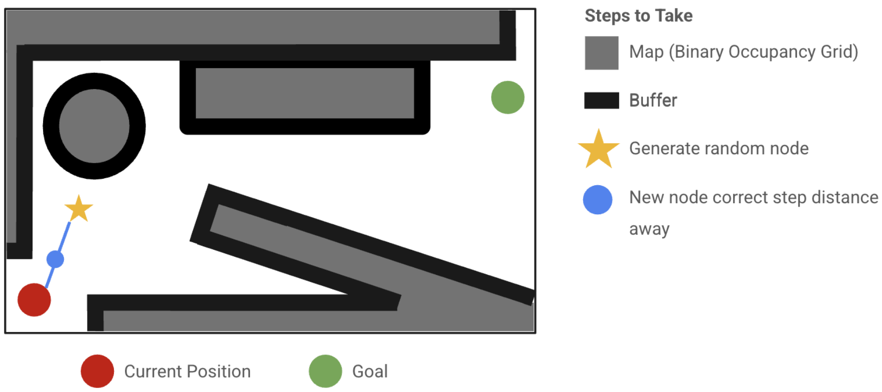


Figure 2: Random nodes are generated and new nodes are added to a tree, a given step distance away from the nearest node.

If our new node does not hit the wall, we add it to our tree. We continue to generate these random nodes until we reach the goal where we backtrack to generate our final path. Our final tree will look something like Fig. 3:

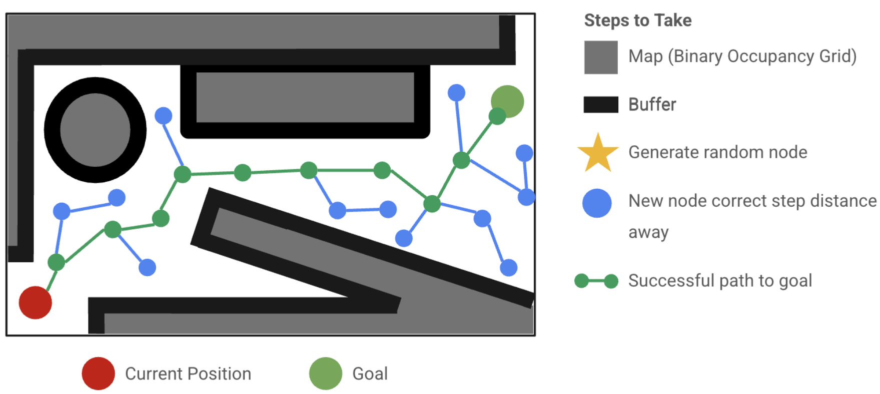


Figure 3: Our tree continues to sample the space until it reaches the goal and backtracks to create our path.

Implementing RRT helped demonstrate its advantages and disadvantages. The main disadvantage we noticed was that the generated path is not optimal. RRT is not asymptomatically optimal because as the number of samples increases, our solution does not necessarily converge towards an optimal path because it is greedily exploring our entire space. Furthermore, RRT is a single-query planner because it evaluates a single path between the start and goal position, but we saw that a single-query is sufficient for our racecar. The time complexity of a basic RRT algorithm is $O(n \log n)$ where n depends on the number of nodes in the tree. This produced relatively fast results, with runtimes recorded below in the evaluation section. Additionally, our RRT algorithm required very little post-processing besides backtracking to obtain our path. Overall, RRT produced speedy results but sub-optimal paths that were not always smooth enough for our racecar applications.

2.3 Pure Pursuit Path Following

The goal of the pure pursuit path following module is to accurately follow the path generated by the path planning algorithm. The inputs to the pure pursuit are the estimated pose of the robot from the particle filter algorithm and a list of way points from the path planning algorithm.

The path planning module outputs the path as a list of way points in x-y coordinates relative to the map frame, which the pure pursuit module listens for and stores internally. The pure pursuit module also listens for real-time pose updates from the particle filter, and when it receives one, it does the following calculations:

First, the closest line segment connecting two consecutive way points is found using matrix algebra. This function returns the index of the first way point in that segment.
Next, we compute the point of intersection of a circle of radius L_{ah}, centered at the race car's position, with the closest line segment we just found. If that line segment and circle do not intersect, we try the subsequent segments on the path until we find one that does. This intersection point is the orange "look ahead" point, as shown in Figure 3.

Given the position of the look ahead point and the position of the robot, we calculate deltax and deltay, which is the robot's relative position from the look ahead point.

$$\Delta x = x_{\text{lookahead}} - x_{\text{robot}}, \tag{8}$$

$$\Delta y = y_{\text{lookahead}} - y_{\text{robot}}. \tag{9}$$

Additionally, we know the orientation of the robot (theta) with respect to the map frame from the pose given by the particle filter. Putting all this together, we can calculate eta, the heading of the robot relative to the tangent line to the path at the look ahead point as follows:

$$\eta = \arctan\left(\frac{\Delta y}{\Delta x}\right) - \theta. \tag{10}$$

Now we have all the information that the Pure Pursuit control law needs to calculate the steering angle δ. This includes the length of the robot L, the relative angle η, and the lookahead distance L_{ah}. Using this information, δ is computed as follows:

$$\delta = \arctan\left(\frac{2\,L\,\sin(\eta)}{L_{\text{ah}}}\right). \tag{11}$$

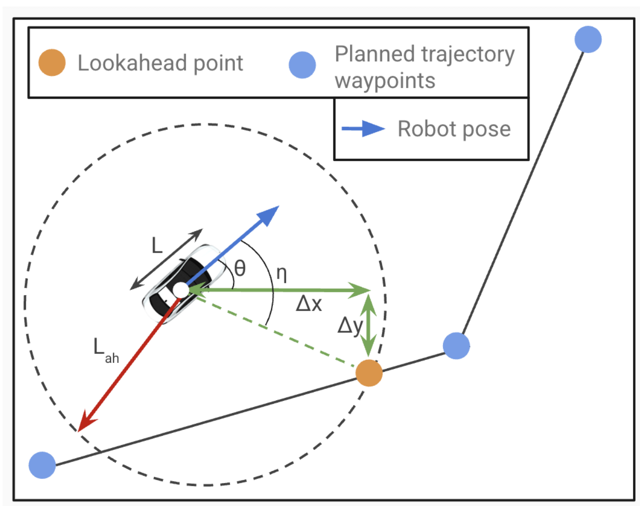


Figure 4: Pure pursuit geometry diagram. We can see all the dimensions needed to solve the pure pursuit control law.

3 Experimental Evaluation

Fiona Wang, Thomas Buckley

Before we could evaluate our entire navigation stack, there were many modules and unit tests we needed to pass first. Specifically, we outlined our evaluation to be as follows:cb

1. **Monte Carlo Localization Evaluation in Simulation and Real World.** Our performance metrics were based on the average distance error between the estimated and ground-truth position.
2. **Path Planning Evaluation in Simulation.** We evaluated by comparing the runtime and path length of each generated trajectory
3. **Full Navigation Stack Evaluation in the Real World.** We evaluated by using cross track error.

3.1 Monte Carlo Localization Evaluation

In order to evaluate our Monte Carlo Localization, we wanted to evaluate each of our models and the particle filter in both simulation and the real world. Before testing the particle filter, we first utilized the provided unit tests to ensure that our motion and sensor models were performing without any major issues.

3.1.1 Simulation Evaluation

To evaluate our particle filter, we first tested in the simulation. Our most initial tests focused on using the 2D Pose Estimate tool to check that our localization was accurately estimating the robots position. After debugging initial issues, we used our wall follower to qualitatively and quantitatively evaluate our localization in simulation. To do this, we focused on testing two paths: Stata Path A, a path with multiple turns and indents, and Stata Path B, a straight wall. By recording Rosbags, we were able to replay the robot's path and visually inspect if our particle cloud and odometry arrows were accurately updating and following the racecar's path. We graphed the results of our tests, overlaying the estimated path with the actual path (the ground truth position) and determining the error between them. These graphs are displayed in Fig. 5 below, and it can be seen that the estimated path matches the actual path relatively well.
Following these initial plots, we aimed to quantify the impact of introducing noise into our motion model. We incorporated Gaussian noise into the robot's motion to reflect uncertainty during movement. We observed that larger noise values caused the particles to spread out more broadly, allowing the particle filter to recover effectively from divergent paths. Conversely, very low noise levels led particles to converge tightly, which increased the risk of the robot's true position falling outside this narrow particle distribution.

To systematically evaluate the effect of noise, we tested various values of the noise standard deviation in simulations and measured the average distance error between the particle filter's estimated position and the ground-truth position. As summarized in Table 1, the mean position error slightly increased with higher noise levels on Stata Path A. This result aligns with our expectations, as introducing more noise naturally contributes additional uncertainty to the estimate.

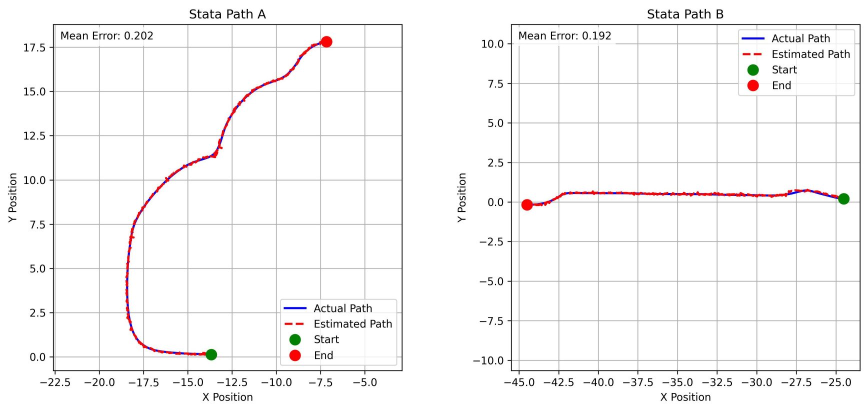


Figure 5: **In Simulation Trials.** Our localization was evaluated in simulation by running the wall follower on two paths (path a on the left and b on the right) and comparing the estimated path with the actual path (ground truth position).

Table 1: Different Values of Noise std in Motion Model on Stata Path A

x, y noise	theta noise	Error (Mean $\pm$ Std Error)
0.05	0.1	0.1273 $\pm$ 0.0539
0.10	0.1	0.1980 $\pm$ 0.0559
0.15	0.1	0.2124 $\pm$ 0.0670
0.20	0.1	0.2124 $\pm$ 0.0670

3.1.2 Real World Evaluation

When moving from simulation to the real world, our particle filter worked but had a major problem. The 200 particles in our filter would very quickly converge to the same pose. We discovered that this rapid convergence occurred because our initial probability distribution contained only a few poses with disproportionately high probabilities. Consequently, the resampling step repeatedly selected these high-probability particles, causing premature convergence and a loss of diversity within the particle set. This made the particle filter ineffective in real-world testing, as it lacked the necessary robustness to recover from early inaccuracies and failed to adequately represent uncertainty, severely limiting its localization accuracy and reliability.

We made three major changes to fix this. First, we increased the standard deviation of the Gaussian noise in the motion model from 0.05 to 0.10, as this showed good experimental results. We kept the standard deviation of theta constant throughout these experiments, as our goal was to increase the spatial distance between particles for increased robustness. Next, we added a time delay for when we resampled particles that we could adjust. We noticed that this resulted in "pulsating" clusters of particles, which would gradually grow, then converge once we resampled. However, this performed much better than reasmpling on every update of our sensor model. Finally, we "squashed" the probability distribution by putting it to the power of $\frac{1}{3}$ as mentioned in the technical approach section. This further increased the diversity of sampled points and maintining good accuracy.

To assess real world evaluation, we conducted two experiments where we recorded the path of the robot while recording a Rosbag. We then compared the recorded video to the estimated pose from our particle filter qualitatively. Our particle filter was highly accurate over long durations (multiple minutes) and for complex paths in the Stata basement.

3.2 Path Planning Evaluation

Next, to evaluate our path planning algorithms, we tested our RRT and A* approaches in simulation and compared their performance qualitatively as well as quantitatively through runtime and path length comparisons. Specifically, we tested using 3 different trajectories—each with varying lengths and direction—and overlaid the path generated by our RRT and A* algorithms as shown in Fig. 6. By plotting the graphs together, it is clear that A* created paths that were much smoother than RRT. This is an important result for our applications because smoother paths will be easier for our pure pursuit to follow. In addition to the difference in smoothness, we also observed that RRT would occasionally take unnecessary turns and S-shapes, reducing its overall optimality.

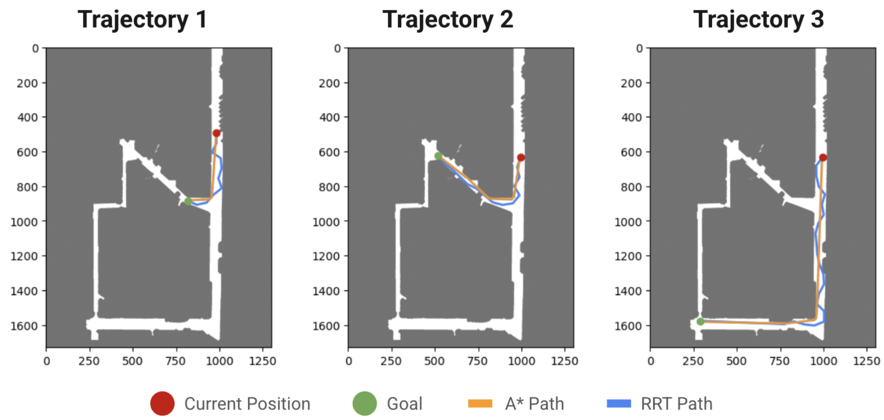


Figure 6: **Comparison of RRT and A* graphs for 3 trajectories.** Our path planning algorithms were evaluated qualitatively by running 3 unique trajectories and overlaying the results produced by RRT and A*. These graphs show how A* produced smoother, more optimized paths.

For each of the three trajectories, we also recorded the runtime and path length for RRT and A*, producing Table 2:

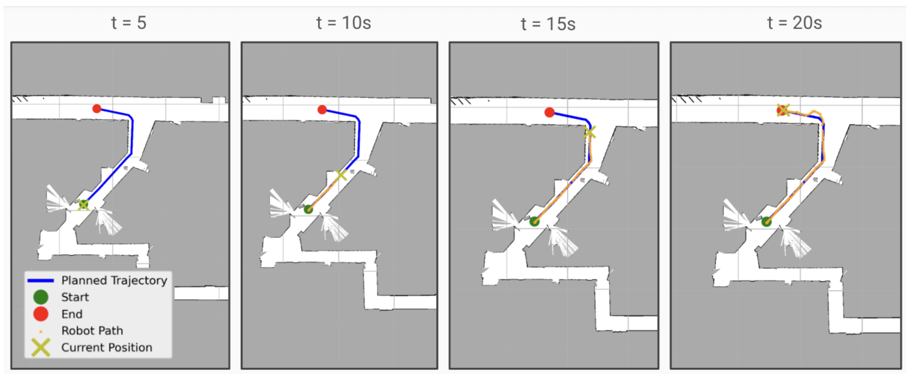
Trajectory	**Runtime (sec)**		**Length (pixels)**	
	RRT	**A***	**RRT**	**A***
1	0.094	0.431	604.341	517.157
2	0.103	0.423	840.647	771.454
3	0.153	0.454	1746.320	1607.450

Table 2: **Comparison of runtime and path length between RRT and A* for three trajectories.** These results show that RRT had shorter runtimes but A* produced shorter length paths.

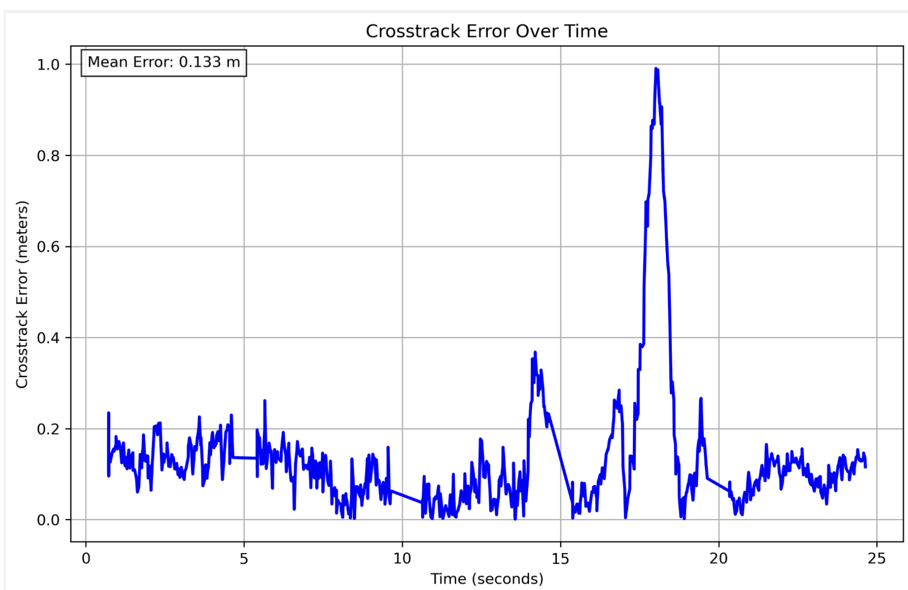
From all three trajectories, it is clear that RRT consistently had a shorter runtime, but A* produced shorter trajectories. From the data, we decided to implement A* on our racecar, choosing to tradeoff average runtime for smoother, shorter trajectories.

3.3 Full Navigation Stack Evaluation

To evaluate the complete navigation stack, we had the robot follow the trajectory shown in 7a. We then compared the planned trajectory to the actual trajectory (as estimated by our particle filter) every five seconds. We also computed the crosstrack error, in 7b. As shown in the figures, the robot closely follows the path we chose. However, we found that the robot would struggle to take hard turns. This can be seen on the panel at 15 seconds, where the robot veers off course substantially at the turn and immediately corrects itself. This is also what causes the large peak in the crosstrack error around 15 seconds.



(a) Path Planning Evaluation on Stata Path C. We compared our planned trajectory (in blue) to the estimated trajectory (in orange, particle filter) for a single path in the Stata basement (snapshot every 5 seconds).



(b) Stata Path Following Crosstrack Error. We computed the crosstrack error by computing the minimum Euclidean distance to the path at each point.

Figure 7: Trajectory evaluation and crosstrack error on Stata paths.

4 Conclusion

Sriram Sethuraman

We were successfully able to integrate Monte Carlo localization, path planning, and pure pursuit path following to enable our race car to autonomously navigate through the Stata basement. The combination of our odometry-based motion model and the LiDAR-based sensor model allowed our particle filter to effectively update an accurate estimate of the car's pose in real time. Our A* based path planning algorithm was efficiently able to plan an optimal trajectory for the robot to follow. Using the data from the particle filter and path planner, the pure pursuit algorithm calculated the best steering angle to follow that trajectory accurately.

Qualitative analyses conducted in both simulation and the real world demonstrated our algorithm's ability to closely follow a planned trajectory. These results showed close alignment with estimated positions and planned trajectories. Quantitative results further supported our findings, showing low cross-track error between ground truth planned trajectories and estimated poses, both in simulation and real-world testing.

Future improvements will focus on tuning the system to be more effective at higher speeds, as we noticed slight oscillations after taking corners at increased speeds. This will involve tuning parameters for the localization and pure pursuit. Additionally, we plan to optimize the runtime of our path planning algorithms even further. We learned valuable technical and teamwork related lessons that we will also carry with us into the next lab.

5 Lessons Learned

Fiona Wang

This lab taught me many lessons both technical and collaborative. From a technical perspective, I learned how to implement RRT and how to optimize the paths it produced by altering the step size. I thought it was really fascinating to be able to visualize the trajectories I was creating using RViz, which not only looked cool, but was incredibly helpful for debugging my algorithm. Getting the path planning to work on simulation was super important and made transferring to the actual robot a smooth and efficient process. Although we did not end up implementing RRT on the racecar, it was still a very rewarding project! From a collaborative standpoint, I thought that our team organization was super effective this week. We were able to split up the map callback, two path planning algorithms, and pure pursuit between the four of us and convened to implement everything onto the robot. Our timelines matched up super well and everyone used their time effectively. This was the first lab that we finished early, giving us more time to perfect our briefing slides.

Ben Lammers
I learned real-world applications of my prior Bayesian probability coursework. I learned how math can model uncertainty in real-world data, such as odometry and sensor readings, but still produce precise results. Implementing these algorithms on hardware helped solidify my understanding of these concepts, making them feel more practical and tangible. From a communication perspective, I learned the importance of proper documentation and how to collaborate effectively. Multiple times, my teammates and I found ourselves rewriting each other's code simply because we did not understand how it worked or that it was already tested and complete. In the future, I plan to ensure that all code is well-documented with clear explanations of the logic behind each function and method. I want to prioritize communication among team members, ensuring that we discuss our approaches and progress regularly to avoid doing the same work.

Thomas Buckley
During this lab, I gained insight into the value of having a standardized testing methodology and clearly defined evaluation pipelines. As our system continues to grow in complexity, consistently and accurately assessing performance becomes increasingly challenging. For example, we needed to run many systems at the same time (localization, simulator, wall_follower, the evaluation pipeline), as well as have a standardized starting and ending position for the car. Additionally, as system complexity grows, we needed to be more creative in our evaluation metrics (especially in the real-world where we can't easily obtain a ground truth for robot position). Building out a scalable way to do this evaluation was difficult, but helped us rapidly evaluate robot parameters which will save time later on.

Sriram Sethuraman
A robust understanding of the math behind the algorithms we are implementing makes the programming and debugging much more efficient. I also learned that communication of progress and code comments are very important so we know what work has been done and we don't do any redundant work. And lastly, when we worked in parallel we were able to get a lot of things done very efficiently.