Lab 6 Report: Search-based and Sample-based Path Planning

Team 15

Bilal Asmatullah
Jing Cao
Panagiotis Liampas
Megan Tseng
Michelle Wang

Robotics Science and Systems

April 24, 2025

1 Introduction

Author(s): Michelle Wang
Given a map and a functional localization method, the next step is to plan and follow a path from start to a goal. We divide this into two components: planning an efficient path and navigating along it using a custom controller.

We explored both search-based and sampling-based planning methods. Search-based methods, such as A*, guarantee optimality but can be inefficient in large spaces. Since our map is relatively small, A* performed well. For sampling-based methods, we evaluated Probabilistic Roadmaps (PRM) and Rapidly-exploring Random Trees (RRT). PRM, which behaves like a randomly down-sampled grid search, was highly sensitive to random seeds. RRT, though also stochastic, more reliably produced feasible paths. A detailed comparison appears in Section 3.2.

For path following, we implemented a pure pursuit controller and tested its performance across different speeds and curvatures.

By integrating localization, planning, and control, we successfully executed end-to-end path tracking in both simulation and real-world environments.

2 Technical Approach

2.1 Localization

2.1.1 Motion Model

Author(s): Panagiotis Liampas
A key component of a particle filter, is keeping track of a set of particles that represent the robot's current estimate of where it is, with respect to a fixed reference frame.

We are doing this by storing a set of particles for each of which we independently sample noise variables $\epsilon_x \sim \mathcal{N}(0, \sigma_x^2), \epsilon_y \sim \mathcal{N}(0, \sigma_y^2), \epsilon_\theta \sim \mathcal{N}(0, \sigma_\theta^2)$, or uniformly around 0 for each component, as shown in Figure 1c, to compute the following transformation:

$$T_{r_k,noisy}^{r_{k-1}} = \begin{bmatrix} \cos(d\theta + \epsilon_\theta) & -\sin(d\theta + \epsilon_\theta) & dx + \epsilon_\theta \\ \sin(d\theta + \epsilon_\theta) & \cos(d\theta + \epsilon_\theta) & dy + \epsilon_\theta \\ 0 & 0 & 1 \end{bmatrix}$$

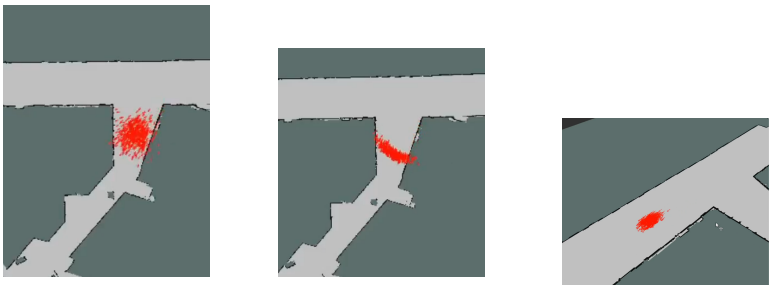
which is applied as:

$$T_{r_k}^W = T_{r_{k-1}}^W T_{r_{k,noisy}}^{r_{k-1}}, \text{ where } T_{r_i}^W = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & x_i \\ \sin(\theta_i) & \cos(\theta_i) & y_i \\ 0 & 0 & 1 \end{bmatrix}$$

and $(dx, dy, d\theta)$ are computed using the linear and angular velocities measured by an IMU or wheel/motor encoders attached to the robot, as: $(dx, dy, d\theta) = (\dot{x}dt, \dot{y}dt, \omega dt)$.

To construct the initial set of particles (before receiving any odometry measurements), we sample m particles around an estimated pose: $x_{init} \sim \mathcal{N}(0, \sigma_{x,init}^2), y_{init} \sim \mathcal{N}(0, \sigma_{y,init}^2), \theta_{init} \sim \mathcal{N}(0, \sigma_{\theta,init}^2)$.

This approach captures both the uncertainty in the initial pose estimate and the noise in the odometry data. Adjusting the noise variances results in different particle distributions over time, as shown in Figure 1. These parameters are tuned based on the robot's sensor accuracy and the properties of the environment.



(a) Gaussian noise with moderate σ_x, small σ_θ

(b) Gaussian noise with small σ_x, small σ_θ

(c) Uniform noise

Figure 1: Gaussian noise motion models concentrate the particles closer to the mean while uniform noise ones spread them out equally, achieving greater variance with fewer particles.

2.1.2 Sensor Model

Author(s): Megan Tseng

To evaluate the likelihood of each of the N particles representing the robot's true pose, we implement a sensor model that compares exteroceptive measurements to the map. For each particle, we perform ray casting to simulate LiDAR beams from the particle's hypothesized position, generating a set of expected distance measurements. This results in a stack of N simulated LiDAR messages, each containing measurements $d^{(i)}$ corresponding to particle i.

At a timestep k, the robot receives a LiDAR message containing ranges $z_k^{(i)}$. Given a particle position x_k in map m, we evaluate the probabilities of:
(1) Detecting a known obstacle,

$$p_{hit}(z_k^{(i)}|x_k,m) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_k^{(i)}-d)^2}{2\sigma^2}\right) & \text{if} \quad 0 \le z_k \le z_{max} \\ 0 & \text{otherwise} \end{cases}$$

where σ is an intrinsic property of the sensor.
(2) A measurement falling short due to an unmapped obstacle or defects in the robot's field of view,

$$p_{short}\left(z_k^{(i)}|x_k,m\right) = \frac{2}{d}\begin{cases} 1-\frac{z_k^{(i)}}{d} & \text{if} \quad 0 \le z_k^{(i)} \le d \text{ and } d \ne 0 \\ 0 & \text{otherwise} \end{cases}$$

(3) A missed measurement,

$$p_{max}(z_k^{(i)}|x_k,m) = \begin{cases} 1 & \text{if} \quad z_k^{(i)} = z_{max} \\ 0 & \text{otherwise} \end{cases}$$

(4) And a completely random measurement.

$$p_{rand}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{z_{max}} & \text{if} \quad 0 \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$$

Combined, the sensor model calculates the probabilities of each LiDAR range:

$$\begin{aligned} p(z_k^{(i)}|x_k, m) = \alpha_{hit} \cdot p_{hit}(z_k^{(i)}|x_k, m) + \alpha_{short} \cdot p_{short}(z_k^{(i)}|x_k, m) \\ + \alpha_{max} \cdot p_{max}(z_k^{(i)}|x_k, m) + \alpha_{rand} \cdot p_{rand}(z_k^{(i)}|x_k, m) \end{aligned}$$

where α_{hit}, α_{short}, α_{max}, α_{rand} are tunable parameters
A particle's likelihood is the joint probability of all LiDAR ranges given the particle's hypothesized position.

As shown in Figure 2, these four cases are characterized respectively by (1) a Gaussian distribution about the "true" distance, (2) a downward sloping line, (3) a spike at maximum range of measurement, and (4) a small uniform value.

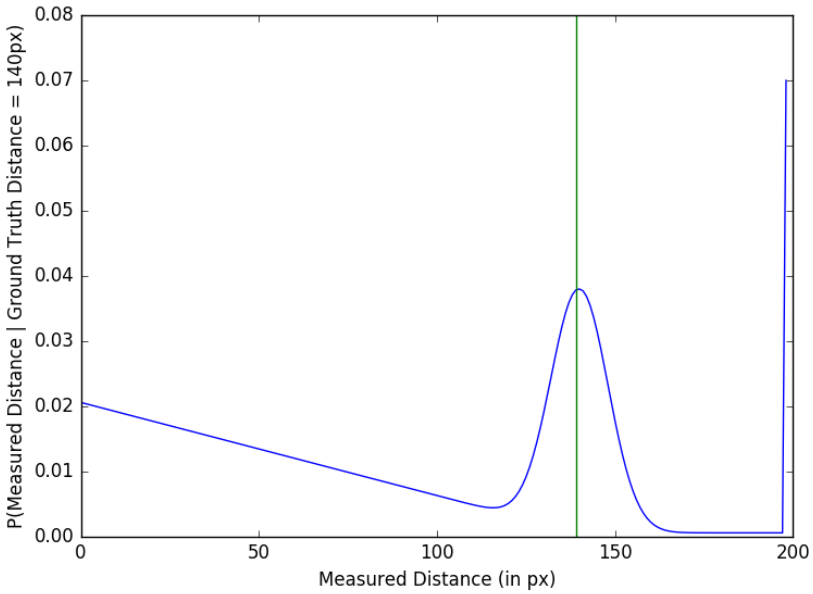


Figure 2: **Probability distribution of a LiDAR measurement.** The distance corresponding to peak probability is close to the green marker representing the ground truth distance (140 pixels) of an obstacle.

To reduce runtime during fast motion, we discretize inputs nad precompute a lookup table for the sensor model. For each casted ray d and LiDAR measurement z, we scale the distance to pixel units based on map resolution. At initialization, we populate a 201×201 table where each row corresponds to a LiDAR measurement and each column to a raycast distance. This table stores the measurement probabilities $p(z_k^{(i)}|x_k, m)$, allowing the sensor model to quickly retrieve likelihoods for any measurement up to 200 pixels, clipping longer distances as missed detections.

2.1.3 Resampling

Author(s): Michelle Wang

Particle filters can suffer from degeneracy, where a few particles dominate with high confidence while many others have negligible weight, resulting in poor pose estimates. Conversely, relying solely on high-confidence particles reduces diversity. To balance accuracy and diversity, we apply resampling strategies that maintain a spread of particles with both high likelihood and spatial variation. We evaluate the following four strategies on N particles with associated probabilities p in simulation:

1. **Random Choice Sampling**: Each particle i is sampled with replacement with probability p_i. To control the sharpness of the distribution, probabilities can be raised to a power—values less than 1 flatten the distribution (increasing diversity), while values greater than 1 sharpen it (favoring high-confidence particles).

2. **Residual Sampling**: Scaled weights are computed as $N \cdot p$. Each particle is deterministically sampled $\lfloor N \cdot p_i \rfloor$ times. The remaining particles are sampled via random choice using the residuals $N \cdot p - \lfloor N \cdot p \rfloor$, combining certainty with diversity.

3. **Stratified Sampling**: The cumulative distribution is divided into $\frac{1}{N}$-sized intervals. From each interval, a random sample is drawn, ensuring both even coverage and a higher likelihood of selecting high-probability particles.

4. **Systematic Sampling**: Similar to stratified sampling, but only a single random offset is generated. The remaining $N - 1$ samples are taken at fixed intervals of $\frac{1}{N}$, providing an efficient and low-variance approximation of the ideal distribution.

2.1.4 Particle Filter

Author(s): Jing Cao

The particle filter is the core probabilistic localization framework that integrates the motion model, sensor model, and resampling to maintain a belief distribution over the robot's pose. At a high level, the particle filter consists of five

main steps shown in Figure 3.

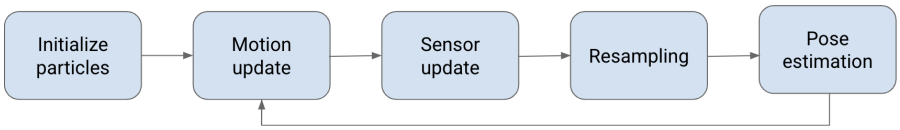


Figure 3: Overview of the particle filter algorithm.

1. Initialization: We first initialize N particles by sampling from a Gaussian distribution centered around the robot's assumed starting location $(x_{init}, y_{init}, \theta_{init})$ with specified standard deviations $(\sigma_x, \sigma_y, \sigma_\theta)$. This captures the initial uncertainty in the robot's pose estimate.

2. Motion Update: Each particle is propagated forward based on the robot's measured control inputs using the motion model described in Section 2.1 and motion equations

$$[dx, dy, d\theta] = [v_x \cdot dt, v_y \cdot dt, v_\theta \cdot dt].$$

Noise is added to account for odometry drift, causing the particle cloud to spread and reflect uncertainty in movement.

3. Sensor Update: Once particles have moved, we evaluate how likely each particle's pose is, given the latest LiDAR scan. Using our precomputed likelihoods described in Section 2.2, each particle is assigned a weight proportional to the probability of observing the LiDAR scan from that pose,resulting in a distribution where high-weight particles are more consistent with sensor data.

4. Resampling: We then resample from the particle set based on the computed weights, using random choice resampling with a power of 1. This favors high-likelihood particles while maintaining diversity.

5. Pose Estimation: To estimate the robot's current pose, we compute the weighted average of all particles. This final estimate is robust to outliers and can track the robot's true pose over time, even in the presence of non-Gaussian noise and sensor anomalies.

2.2 Pre-processing for planning

Author(s): Jing Cao

To prevent paths from being planned too close to obstacles, we applied dilation to expand obstacle boundaries. Figure 4 shows a comparison between the original map (right) and the dilated map (left). We found that overly aggressive dilation blocked feasible paths in narrow corridors, while insufficient dilation led

to collisions with nearby obstacles. Choosing an appropriate dilation size was therefore critical for balancing safety and accessibility.

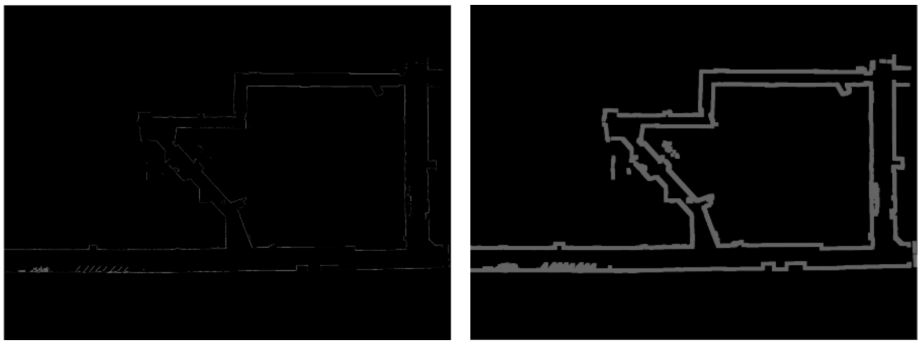


Figure 4: The orignal map is on the right and the processed map with dilation is on the left.

2.3 Search-based planning using A*

Author(s): Jing Cao

We used A* search to compute the most optimal path from a specified starting point to a goal location. The algorithm begins by initializing a priority queue with the start node and iteratively expands the node with the lowest estimated total cost, along with its neighbors. This process continues until the goal is reached or all nodes have been explored. A* selects nodes based on a cost function $f(n) = g(n)+h(n)$, where $g(n)$ is the true cost from the start to the current node and $h(n)$ is a heuristic estimate of the remaining cost to the goal. In our implementation, we use Euclidean distance as the heuristic function. Figure 5 shows the optimal path generated by the A* algorithm.

Figure 5: Generated path from search-based planning using A* with downsampling factor of 5. Green point is the starting point, red point is the goal point.

To improve computational efficiency, we downsampled the map to reduce the search space. If we let n be the downsampling factor, each grid cell in the downsampled map was the result of a maxpool over $n \times n$ pixels in the original map. Maxpooling corresponding regions in the original map preserves obstacle boundaries to avoid collisions. Figure 6 shows the difference between the original and downsampled maps. This approach significantly decreases the number of nodes A* needs to evaluate. Figure 7 shows the path generated using A* with a downsampling factor of 5, where the green point marks the start and the red point indicates the goal.

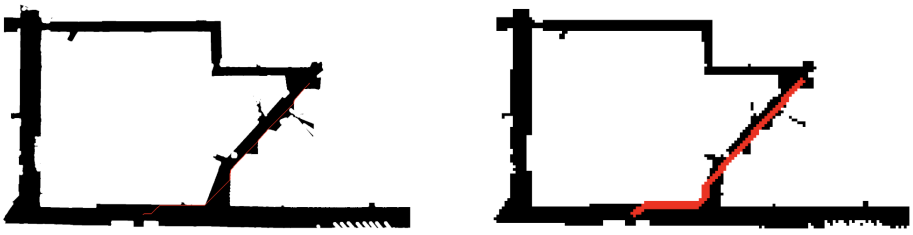


Figure 6: The left map has not undergone downsampling. The right map has undergone downsampling with a factor of 10.

Figure 7: Generated path from search-based planning using A* with downsampling factor of 5. Green point is the starting point, red point is the goal point.

2.4 Probabilistic Roadmap Method (PRM)

Author(s): Michelle

PRM is a sampling-based method that builds a path by randomly sampling points in free space and connecting them to form a roadmap. An overview is shown in Figure 8. During graph construction, sampled points are connected if the edges between them are collision-free. Increasing the number of samples creates a denser roadmap but increases computational cost. Similarly, increasing the maximum edge length allows broader coverage but makes it harder to reach tight spaces.

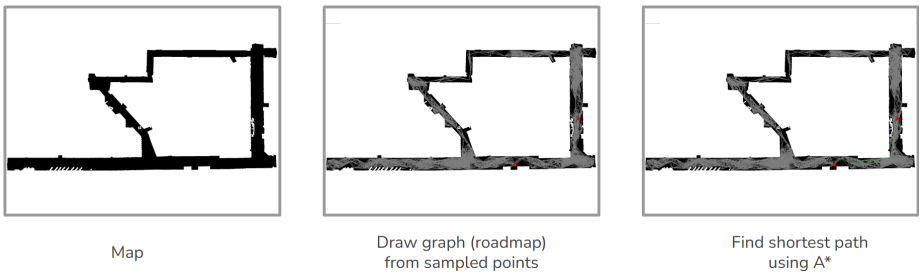


Figure 8: The stages of PRM. The rightmost figure depicts a generated shortest path

After generating the roadmap, we add the start and goal points and connect them to the nearest nodes. We then run A* with Euclidean distance as the heuristic to find the shortest path. In practice, PRM was highly sensitive to the random seed and often failed to find consistent paths for the same start and goal. This issue was more pronounced with fewer samples, resulting in incomplete roadmaps (Figure 9). Conversely, using a large number of samples made PRM resemble grid-based search, reducing its practical advantage.

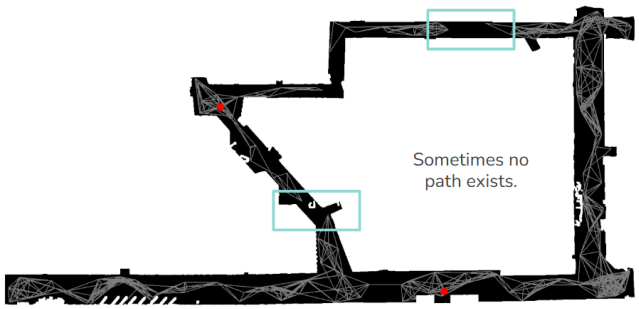


Figure 9: PRM fails when not enough nodes are sampled in the configuration space, leading to a discontinuous roadmap

2.5 Randomly-exploring Random Trees (RRT/RRT*)

Author(s): Panagiotis Liampas

Another random sampling-based method we tried for path planning is Randomly-exploring Random Trees, which iteratively constructs a tree T, the root of which is the starting pose X_{start}, and which includes all reachable poses (x, y, θ) that have been found.

This is done by randomly sampling unoccupied points X_{sample} within the map limits, and computing the Dubin's path from $X_{nearest}$ to X_{sample}, where $X_{nearest} \in T$ is the node closest to X_{sample}, and we find the pose X_{new} that's a predefined distance d from the node or right before it collides with an obstacle. That pose is then added to the tree, with an edge from $X_{nearest}$ to X_{new}. If X_{new} can reach the goal without colliding, we backtrace from the goal to construct the final path. That process is demonstrated in Figure 10.
To make the tree construction more efficient in finding a path towards the goal pose X_{goal}, we sample X_{goal} with a probability p (goal bias), while we also increase the sampling space, centered around the starting pose, each iteration. Also, each time we find a new reachable pose, we check if the paths towards its neighbors in T can be improved using X_{new}, or if the path towards X_{new}

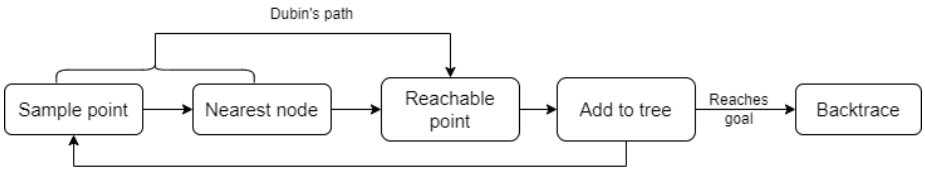


Figure 10: Overview of the RRT path-planning algorithm.

can be improved using one of those neighbors, by keeping track of the path length towards each node. That results in much more straightforward paths but requires a lot of additional computations, so we only use it after finding a path to the goal. That process is illustrated in Figure 11.

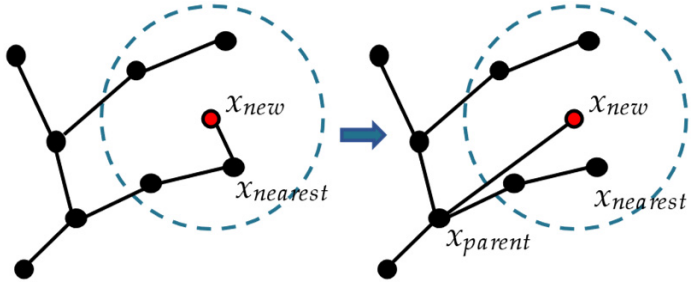


Figure 11: The rewiring step of RRT*.

In order to further optimize the paths created, we randomly sample pairs of points (u, v) in the path and remove the points in between if there is an unobstructed Dubin's path from u to v. The final result of the algorithm (with the tree constructed and the path being published) is illustrated in Figure 12.

Figure 12: The tree constructed using RRT*, and the resulting path after shortcutting.

2.6 Controller

Author(s):Megan

To enact the planned trajectory, we used a Pure Pursuit controller similar to the one designed for line following. Since our trajectory is represented by a series of segments, pure pursuit allows us to apply and tune a lookahead distance to determine steering angle.

To accomplish this, we apply a series of steps to identify a new lookahead point at every pose estimate received:

1. Sort trajectory segments by proximity.
   For each trajectory segment defined by points $\mathbf{s_i}, \mathbf{s_{i+1}}$, and vector $\mathbf{v}$, we use our estimated robot pose $\mathbf{p}$ from localization to calculate minimum distance to segment $\mathbf{d_i}$.

   $$d_i = \|\mathbf{p} - (\mathbf{s}_i + t \cdot \mathbf{v})\|$$

   where $\mathbf{t}$ is a projection parameter defined by

   $$t = \frac{(\mathbf{p} - \mathbf{s}_i) \cdot \mathbf{v}}{\|\mathbf{v}\|^2}$$

and clipped to [0,1].

2. Check segments for intersections defined by lookahead distance.
   For a lookahead distance $\mathbf{r}$, we define a circle with radius $\mathbf{r}$ and find the intersection to each segment via

$$\|\mathbf{s}_i + t \cdot \mathbf{v} - \mathbf{p}\|^2 = r^2$$

   where $\mathbf{t}$ (clipped [0,1]) gives parametrized location(s) of lookahead points.

3. Choose appropriate lookahead point.
   We choose the point furthest along the trajectory and extract the point in world coordinates via $\mathbf{s}_i + t \cdot \mathbf{v}$.

As shown in Figure 13, with the relative (x, y) location of the lookahead point, we calculate a turning radius $\mathbf{R}$ via

$$\mathbf{R} = \frac{r}{2 \sin(\tan^{-1} \frac{y}{x})}$$

Using wheelbase $\mathbf{w}$, we calculate steering angle to be

$$\tan^{-1} \frac{w}{R}$$

and send this via AckermannDrive commands with a constant speed.

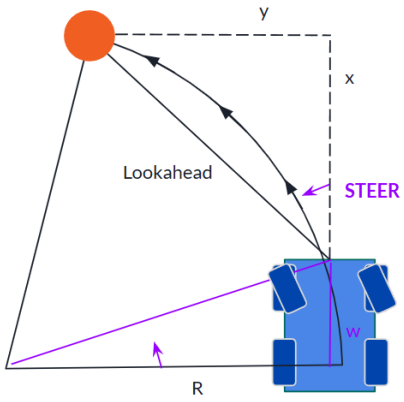


Figure 13: Schematic of geometry illustrates commanded steering angle derived from robot wheelbase and position of chosen lookahead point.

3 Experimental Evaluation

3.1 Localization evaluation

3.1.1 Particle Filter Working in Simulation

Author(s): Jing Cao
Initially, we tested the particle filter's performance in simulation. We collected ground truth pose data from the simulator and compared it with the estimated robot pose over time, computing the mean absolute error in the x and y positions and the orientation θ.

In the noise-free simulation, odometry and LiDAR measurements are ideal, allowing the particle filter to maintain a tightly clustered and highly confident particle distribution. As shown in Figure 14, the pose estimation error remains consistently low across all three dimensions:

- x error: 0.2642 m
- y error: 0.0522 m
- θ error: 0.0127 rad

This confirms that the particle filter performs well when sensor readings and motion data are accurate, and the filter converges quickly and stably to the true pose.

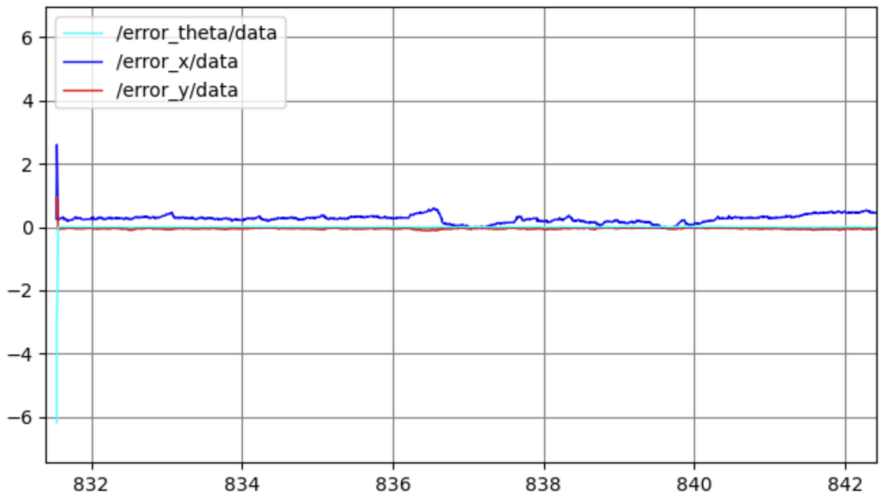


Figure 14: Particle filter working in simulation with no motion or sensor noise. Errors in pose remain very low and stable throughout the trajectory.

3.1.2 Noisy Odometry/Motion Model

Author(s): Bilal
Although our algorithm performed well in simulation, real-world odometry includes noise absent in idealized environments. In simulation, access to ground-truth poses allows us to measure errors in x, y, and θ. To test robustness, we injected artificial noise into odometry data and evaluated how pose estimation accuracy degraded, providing insight into expected real-world performance.

To ensure consistency across different noise levels, we recorded a single ROS bag of the robot's motion in simulation. We replayed this bag four times, each time injecting different noise profiles into the odometry:

- **No Noise (Baseline)**
- **Gaussian Noise (1 m/s, 0.5 rad/s)**
- **Gaussian Noise (2 m/s, 1 rad/s)**
- **Gaussian + Uniform Noise (1 m/s, 0.5 rad/s + additional uniform noise 1 m/s, 0.5 rad/s)**

We recorded the differences between the ground-truth pose (available from the simulator) and our estimated pose for each condition as shown in Figure 15.

- **From Baseline to 1 m/s Gaussian Noise**
  - x-error change: 0%
  - y-error change: $\approx 4.3\%$
  - θ-error change: $\approx 16.7\%$
- **From Baseline to 2 m/s Gaussian Noise**
  - x-error change: $\approx 14.3\%$
  - y-error change: $\approx 4.3\%$
  - θ-error change: $\approx 33.3\%$
- **From Baseline to Gaussian + Uniform Noise (1 m/s, 0.5 rad/s)**
  - x-error change: $\approx 23.8\%$
  - y-error change: $\approx 17.4\%$
  - θ-error change: $\approx 83.3\%$

Although these percentages may look large in some cases (especially for θ), the absolute differences in x and y remain modest. The overall error increases provide a reasonable worst-case estimate of how our method might behave on physical hardware, giving us confidence that our algorithm is robust to moderate levels of odometry noise.

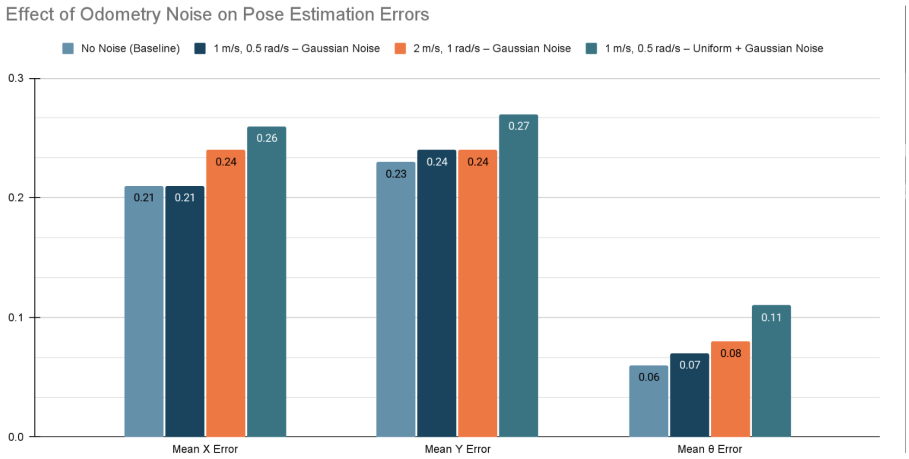


Figure 15: Figure summarizes the mean errors in x, y, and θ. As expected, the baseline (no added noise) yields the smallest errors. When noise is added, errors do increase, but generally remain within acceptable bounds.

3.1.3 Resampling Study

Author(s): Bilal

To test resampling robustness under extreme conditions, we added Gaussian noise with $\sigma = 1.5$ m in x and y, and 0.4 rad in θ at each motion model update. We evaluated six resampling methods using mean absolute errors in x, y, and θ. The `random.choice` method (with probabilities raised to 1) consistently yielded the lowest errors and was selected for our final implementation. Systematic resampling performed poorly and failed to track the robot. Full results are shown in Table 1.

Method	Mean X Error	Mean Y Error	Mean θ Error
random.choice, power = 1/3	0.70	0.21	0.05
random.choice, power = 1	**0.64**	**0.21**	**0.04**
random.choice, power = 2	0.68	0.21	0.05
Stratified	0.91	0.22	0.05
Residual	0.90	0.22	0.05
Systematic (lost robot)	14.0	0.98	0.13

Table 1: Mean error performance of resampling strategies under high noise

3.1.4 LiDAR-Map Alignment for Real-World Evaluation

Author(s): Panagiotis Liampas

To visually inspect our particle filter's performance in real-world conditions, we transform the LiDAR scan by our average particle pose and evaluate its alignment with the map, as shown in Figure 16. A way to quantify this is by calculating the average negative log probability of the LiDAR rays detecting an object in the correct position based on the map:

$$\frac{-\log(prob_{total})}{\text{num beams}} = \frac{-\log(\prod prob_{beam})}{\text{num beams}} = \frac{\sum -\log(prob_{beam})}{\text{num beams}} = \overline{-\log(prob_{beam})}.$$

where a lower value indicates that our system works well in the real world.

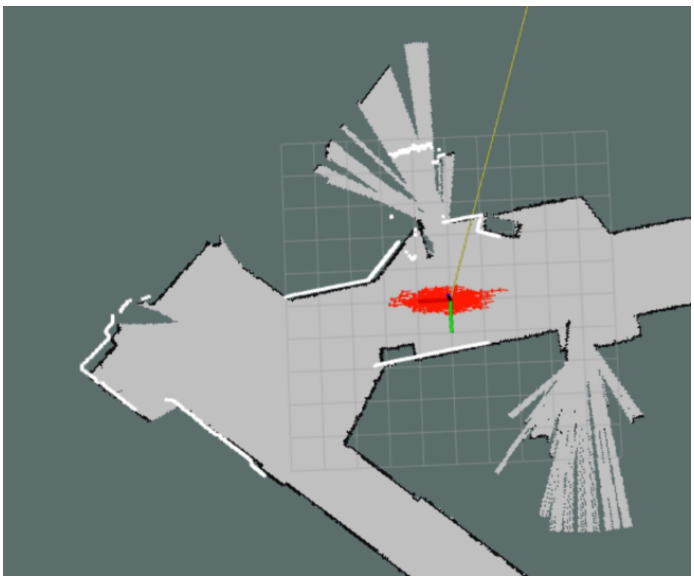
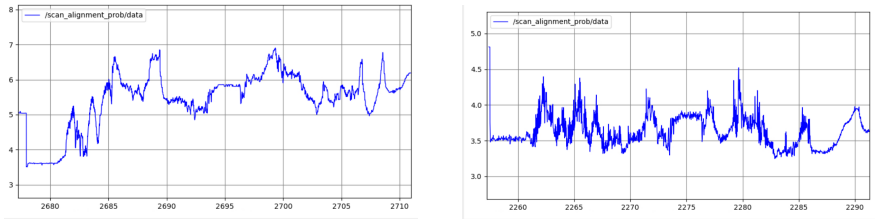


Figure 16: The LiDAR scan is seemingly well aligned with the map walls, indicating that the position estimate of the robot is relatively accurate

This is closely related to the average distance between the detected position and the actual position of a point intersected by the LiDAR beam, since the sensor model is really similar to a Gaussian distribution.

Figure 17 shows that, quantitatively, our system works better in the real world, especially in terms of handling the set of particles and updating it, using both odometry and LiDAR, to align the 2D scan with the walls that are on the map, getting closer to an accurate estimate of its position.



(a) Deterministic model using odometry (b) Particle Filter

Figure 17: A comparison of the LiDAR-map alignment between the deterministic motion model and the noisy one, in the real world, using the metric described above.

3.2 Comparing path-planning algorithms

Author(s): Jing Cao

We compared three path planning algorithms—A*, PRM, and RRT*—based on their ability to generate feasible and efficient paths in a 2D map environment.

As shown in Figure 18, A* produced the most direct and optimal path, cleanly navigating narrow passages. On the original map, A* generated a 40.2-meter path in 4.83 seconds, and with downsampling, reduced planning time to 0.467 seconds while maintaining path quality.

PRM failed to produce a valid path due to poor graph connectivity, as seen in Figure 18. Its performance was inconsistent and offered no advantage in small, static environments.

RRT* successfully generated a smooth 43.4-meter path in 2.06 seconds, though it was less direct than A* and introduced curvature that could challenge the controller. The path, shown in the right panel, was natural but less precise.

Overall, A* with downsampling offered the best balance of speed and optimality. PRM proved unreliable, while RRT* remains useful for larger or more dynamic environments.

3.3 Controller evaluation

Author(s): Megan

We first evaluated our controller in simulation using an A*-planned path, chosen for its optimality. Performance was measured by the mean absolute distance from the path. As shown in Figure 19, error spikes occur at both shallow and sharp turns. A spike at the start reflects the mismatch between the car's initial orientation and the planned path.

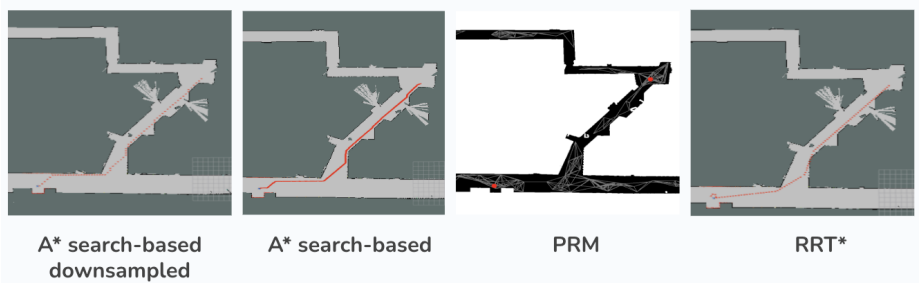


Figure 18: Comparison of A* search-based planning, PRM, RRT* on generating a path from the same start and goal position.

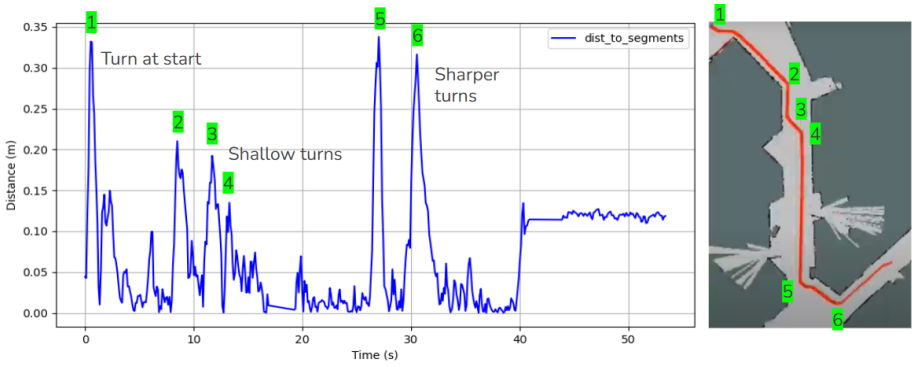
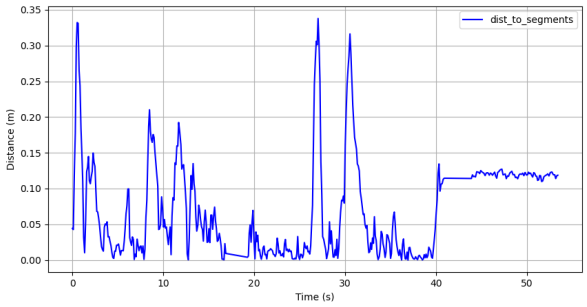


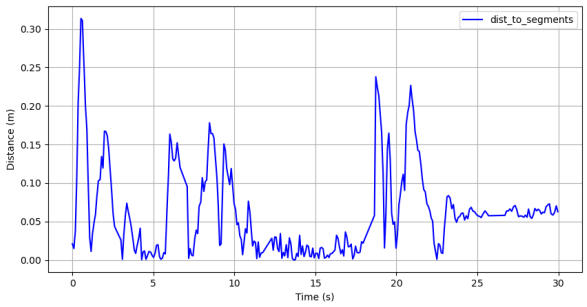
Figure 19: Absolute distance from trajectory for the duration of the path shown on the right. The green numbers mark turns in the path.

The car followed the planned path at 1.0 m/s with a 1.0 m lookahead, staying within the 0.33 m error caused by the initial turn. This indicates that the controller accurately tracks complex paths and behaves as expected around turns.

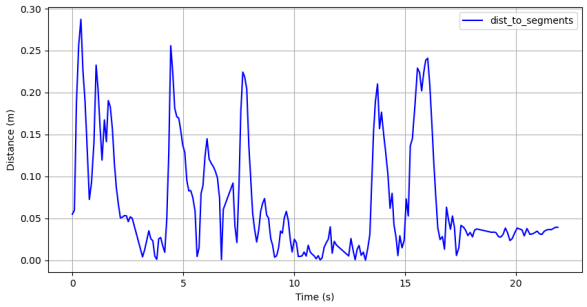
We then tested the controller across speeds from 1.0 to 2.0 m/s and lookahead distances from 1.0 to 1.2 m. Figure 20 shows the resulting trajectory errors.



(a) Speed 1.0 m/s, Lookahead 1.0 m. MAE: 0.071 m



(b) Speed 1.5 m/s, Lookahead 1.0 m. MAE: 0.063 m



(c) Speed 2.0 m/s, Lookahead 1.2 m. MAE: 0.072 m

Figure 20: Absolute distance from trajectory showing consistent controller reactions to turns at various speeds. The car is executing the same complex path as in Figure 19.

Across the three tested speeds, the mean absolute error varied by only 0.009m, indicating minimal sensitivity to speed in average path deviation. The plots

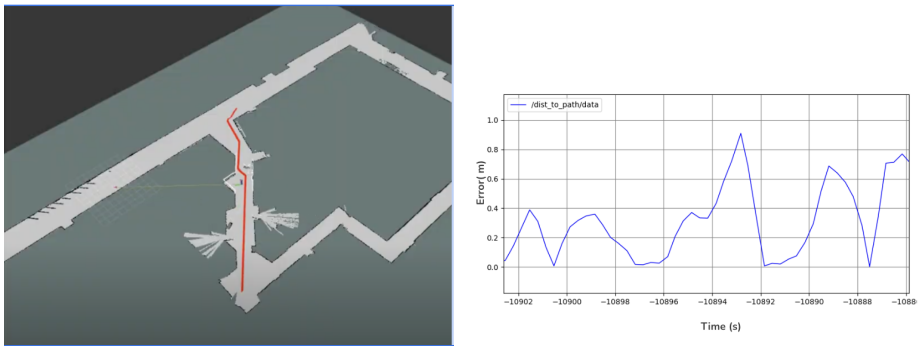
show consistent controller behavior across settings and fast convergence to low error on straight segments.

3.4 Running planning and the controller in real life

Author(s): Bilal

Real-life deployment of our system introduced the issues of odometry noise and LiDAR misalignment, causing the controller to deviate from the planned path due to localization errors. Testing in environments with distinctive features mitigated those issues, but it is not a viable long-term solution.

A* produced a path of length 36.03 m, consisting of straight segments connected by sharp turns, as shown in Figure 21. The controller followed straight segments with MAE of 0.1 m, slightly higher than simulation (0.07 m), due to real-world odometry noise and localization drift. The large lookahead caused shortcutting of sharp turns and deviation from the path, resulting in peaks in the error plot.

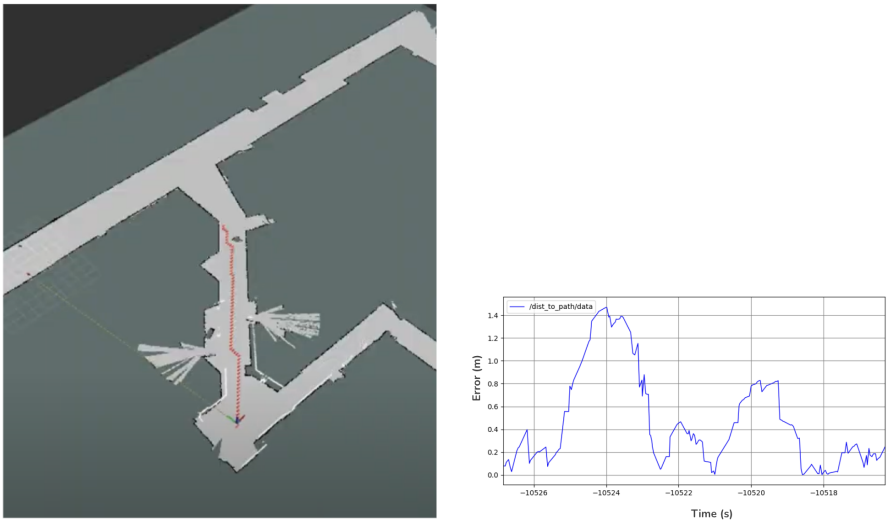


(a) Real-world path execution using A* (length: 36.03 m).

(b) Deviation from path over time for A* path.

Figure 21: Controller performance on an A* path in real-world conditions, showing a mean error of 0.1 m on straight segments.

Next, we evaluated a path generated by downsampled A* (path length: 22.48 m), shown in Figure 22. While the path itself was feasible, localization drift mid-run led to errors as large as 1.4 m, even on straight segments. This suggests that the controller itself remained functional, but the robot's belief of its location became significantly misaligned with reality during execution.



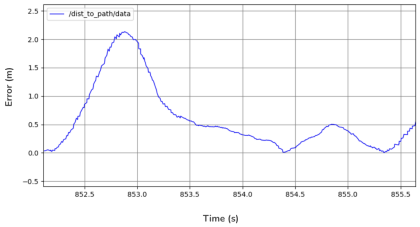
(a) Downsampled A* path (length: 22.48 m).

(b) Path deviation due to localization drift.

Figure 22: Downsampled A* execution reveals challenges with localization.

Lastly, we tested the controller using a path generated by RRT*, shown in Figure 23. The resulting trajectory was smoother and more curved compared to A*, but still feasible. We observed an initial error peak exceeding 2.0 m, caused by the vehicle skipping over a sharp turn due to the large lookahead distance. However, once the robot entered a more linear portion of the path, the error dropped rapidly and stabilized.

(a) RRT* path execution with curved trajectory.



(b) Controller error over time on RRT* path.

Figure 23: Controller following an RRT* path: early deviation due to turn curvature, followed by error recovery.

Overall, these experiments demonstrate that while our controller performs well on feasible paths, real-world performance is limited by the quality of localization. Sharp turns and localization drift remain the main sources of path-following error.

4 Conclusion

Author(s): Bilal

In this lab, we implemented and evaluated a complete path planning and control pipeline for autonomous navigation. Building on our prior localization framework, we integrated both search-based (A*) and sampling-based (PRM, RRT*) planning algorithms with a Pure Pursuit controller.

A* consistently produced optimal paths and performed best when combined with downsampling to reduce computation time. PRM exhibited inconsistent behavior due to its reliance on random sampling, while RRT* reliably produced smooth, feasible paths—particularly in environments where A* struggled. This suggests that A* is well-suited for structured indoor maps, whereas RRT* offers greater robustness in more complex or dynamic settings.

The Pure Pursuit controller showed reliable performance across varying speeds and lookahead distances in simulation. In real-world tests, it tracked straight paths well, but localization drift during sharp turns or in ambiguous areas introduced noticeable deviation. Our results indicate that localization errors were the primary source of large tracking deviations.

Our results validate the feasibility of deploying the full navigation stack on hardware. However, they also highlight key challenges, including the sensitivity to localization accuracy and the trade-offs posed by path curvature and map resolution.

Future work includes improving the localization pipeline, adaptively tuning look-ahead distance, and applying post-planning processing to reduce curvature-induced errors.

5 Lessons Learned

Michelle Wang: I had never worked on path planning problems before, and it was honestly a shock when shortest path algorithms I learned about in Intro Algorithms popped up. Similar to the particle filter, I had a strange moment in this lab where I actually understood everything on a theoretical level (not just an application level like usual) for the first time. I also had not worked so closely with probabilistic algorithms before (like PRM), and it was surprising to see how reliant everything was on the random seed. In the future, this is a great thing to keep in mind as I work with more complex methods. Finally, systems integration was a bit more involved than I would have liked and we encountered many weird edge cases not in sim. This taught me that it might be valuable to do hardware testing in parallel with development in the simulation. Especially in the final challenge when there are so many pieces to put together, it will be good to have hardware verification of each individual component.

For teamwork, I think 3 labs under our belt has helped us understand each other's working styles. For example, I personally cannot be productive when it is 3 AM, so I front load a lot of work, contribute significantly to the planning of the lab, and might have a semi-functional implementation before others have started writing code. Before, this might be seen as pressuring others, but now, I think we have a good understanding of how everyone works and solid trust that the work will get done (but it might get done at varying times). I really appreciate that we have such a working relationship that can only be built with time. I also think our familiarity has led to more straightforward communication and honesty. This can be a double-edged sword as some comments might unintentionally seem aggressive or carry blame, but I believe the misunderstandings can be talked out, and it is far more valuable for everyone to feel comfortable speaking their mind.

Megan Tseng:

I learned a lot about how different path planning algorithms worked while doing this lab. Although I worked mostly on developing the controller, I was able to see where different algorithms failed and succeeded. In addition, understanding how the controller interacted with the planned path was interesting; for example, we had to change the dilation used by the path planning code to accomodate how closely the car was able to follow a narrow turn. Different algorithms represented paths differently (with more or less dense points), which also affected the performance of the controller and allowed me to see what is necessary for smooth trajectory following. Of course, I also further solidified my understanding of the math behind pure pursuit and efficiently finding a point to follow.

In terms of communication, I worked very closely with Bilal on the controller and learned how to efficiently divide labor without sacrificing either of our understandings of how the controller worked. We traded off on debugging certain issues so we could each get fresh eyes on it, and having someone to discuss with made it a lot easier to come up with new ideas.

Jing Cao:
This lab gave us hands-on experience with search-based and sampling-based path planning, helping us understand the strengths and limitations of A*, PRM, and RRT*. A* produced the most optimal paths, especially when combined with downsampling, though excessive downsampling impacted path quality. RRT* was faster and generated smoother paths but was less precise, while PRM was unreliable in our environment due to its sensitivity to random sampling. Implementing the pure pursuit controller also highlighted the importance of path feasibility for execution. Overall, the lab reinforced the need to choose planning methods based on map complexity, performance goals, and robot constraints.

Bilal Asmatullah:
In this lab I worked primarily on the controller, so I feel like I missed out a bit on the interesting path planning stuff. Nevertheless, it showwed how we can work together as a team on separate modules, and then integrate them together while understanding modules developed by others at an abstract level. This really makes for efficient team work.
For the controller modules, I worked closely with Megan. Starting off her initial code for the controller really prevented me from tedious math and stack overflow posts, and helped me focus on the controller logic crucial to our lab. Debugging the controller took a long time, and I even failed completely while trying to debug it for hours consecutively. However, coming at the problem again later that day, with a hypothesis of possible sources of errors and systematic debugging, helped fix bottleneck bugs of the controller. This debugging was unexpected and therefore, even though our team had agreed on a timeline to get our independent components done, this caused a slight delay. However,

everyone was really accommodating and meanwhile, they worked on fixing their own algorithms. This taught me accommodating for unforeseen circumstances and having empathy for teammates. Moreover, during debugging Megan and Panos gave very crucial insights on their implementation and having that support was extremely valuable.

Panagiotis Liampas: In this lab, I learned a lot about Randomly-exploring Random Trees and various optimizations that can be applied to make them usable in real life. I explored the logic behind sampling random controls and creating a tree of reachable poses, and implementing the respective code from scratch allowed me to get a deep understanding of the topic, which will be really useful in my future robotics endeavors.
I also realized the importance of rigorous testing both in simulation and in real life, since there are many unexpected issues that may come up in real life when integrating multiple components with one another, where everything needs to work as intended. That became apparent when we integrated localization, planning, and the controller on the racecar, which took much longer than expected to get to a state where we could reliably collect data and ensure that our system was working well.
This is the reason we should start integrating early and completely debug every one of the components we are going to be using beforehand, so that integration is smoother and we have more time for data collection.