

Lab 6 Report: Methods for Autonomous Path Planning and Following

Team 13

Xavier Sanchez (2A)
Insuh Na (16)
Inimai Subramanian (6-2)
Russell Perez (6-2)

RSS

April 24, 2025

1 Introduction - Xavier

The focus of Lab 6 was to build a dual path planning and tracking algorithm to allow the racecar to efficiently navigate between two endpoints. Both a grid search and random sampling approach were implemented and compared in simulation for robustness and efficiency. After generation, paths are fed to the tracker, a standard pure pursuit controller evaluated in simulation for a range of parameter values. Notably, pure pursuit relies on robust localization, a task complicated by the random noise introduced when operating our algorithms on real hardware. The noise has an unknown distribution, meaning a simple estimator using only odometry quickly diverges from the ground-truth car pose. We solved this problem in Lab 5 with a 'particle filter', an algorithm that uses random sampling and environmental validation to enable a distribution of particles to converge to a more stable and accurate estimate of the car's true pose. The particle filter operates using three distinct components: a motion model to update each of the car's estimated poses, a sensor model to determine the likelihood of each estimate, and a resampling scheme to shift the estimated distribution towards more likely poses. The particle filter was evaluated in simulation using real-time ground-truth data, and on the real racecar using sim-to-real checkpoints. The complete pipeline, from localization to path planning and execution, forms the core of our racecar's navigation capabilities. Looking ahead to the final challenge, this will be critical for successfully using state machine logic to interact with the environment and solve high-level tasks.

2 Technical Approach

We break the technical problem of generating and following a trajectory from start to goal pose in a map into a path planning, path following, and localization module. The path planning module generates a short path avoiding obstacles given a map and input. The path following module takes the returned path as input and uses a pure-pursuit controller to output the car’s drive commands to follow it, with the localization module providing the car’s location. These three modules work together to build the navigation stack for the robot.

2.1 Path Planning - Inimai

The path planning module generates short paths to the goal that avoid obstacles and do not cut corners. We implemented two path planning algorithms for the lab: A* and RRT. A* (a search algorithm that when implemented on a grid can provide us a path) searches through a discretized grid of points representing the map environment to find an optimal shortest path. RRT randomly samples the map environment to create a tree of potential paths until the goal is reached.

2.1.1 Map Pre-processing

To implement our A* and RRT algorithms, we first discretize the map environment into a representation to run our algorithms on. We utilize transformations in order to enforce a unit grid system so that each map pixel corresponds to 1 map point (5x5cm square in real-life corresponds to a 1x1 cell on the map). We utilize the following translation and rotation: $p' = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. Thus, we can convert our generated paths to the real-world environment.

Another common issue in path generation is corner-cutting, where paths get too close to walls or sharp corners, making them hard for the car to follow. To address this, we apply morphological operations to the map: erosion to reduce noise and dilation to enlarge obstacles, effectively padding them. Specifically, we erode using a 3x3 kernel and dilate with a 15x15 kernel. This transformation improves obstacle representation and helps the algorithm avoid unsafe paths. The transformation process is illustrated in Figure 1.

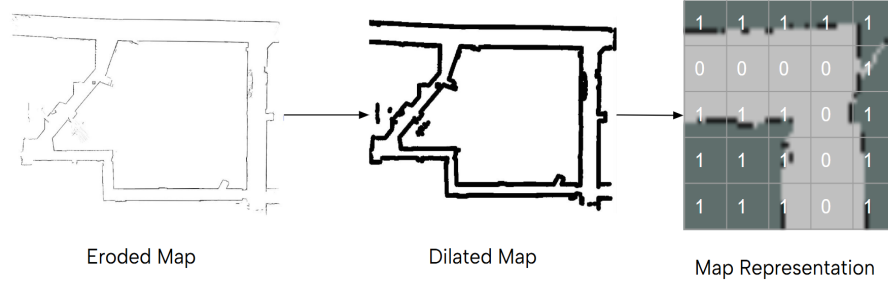


Figure 1: The map is discretized and eroded and dilated in order to account for real-world path following for the car. Obstacles are assigned a 1 and the rest of the cells are assigned 0 on the grid. The output goes through several transformations to convert to the real-world environment.

2.1.2 A* (Grid-based) Algorithm

Grid-based A* is a search algorithm that systematically searches through the discretized environment to find the optimal shortest path. It is guaranteed to produce an optimal path given a heuristic that underestimates the true cost to the goal for all scenarios. For this case, we used the euclidean distance to the goal from the current position as our heuristic because it can never exceed the true cost to the goal. The score for each path is the current calculated distance from the start pose plus the heuristic cost to the end goal.

The algorithm works as follows:

1. Define the start and goal nodes, obstacles, and heuristic.
2. Initialize an open set with the start node and an empty closed set.
3. Select the node with the lowest $f(n) = g(n) + h(n)$.
4. If it's the goal, terminate. Otherwise, move it to the closed set.
5. For each valid neighbor, update g if a shorter path is found, and record the parent.
6. Repeat steps 3–5 until the open set is empty.
7. Reconstruct the path by tracing back from the goal.

2.1.3 RRT Algorithm

Rapidly Exploring Random Tree (RRT) is a sampling-based algorithm that forms a tree by randomly sampling the space, offering faster runtimes than grid-based searches. It is probabilistically complete (can find a path with enough iterations) and converges to a sub-optimal solution. In each iteration, RRT

samples a point, checks for collisions with the nearest tree node, and connects it if collision-free; otherwise, it resamples. We use goal sampling (sampling the goal region with 0.1 probability) to guide tree growth. RRT avoids the need for a configuration space or heuristic, using random sampling to return the first valid path.

The algorithm works as follows:

1. Define start/goal poses and identify free/obstacle cells.
2. Randomly sample a free point (bias 10% toward the goal).
3. Find the nearest node in the tree.
4. Extend a new node towards the sample within a max step size.
5. If the edge to the new node intersects an obstacle, resample; otherwise, add to the tree.
6. Repeat until the goal is reached or max iterations are met.
7. Trace back from goal to start to form the path.

2.2 Path Following: Pure Pursuit - Russell

Our pure pursuit controller leverages the generated path and the particle filter model to issue steering angle commands that enable the racecar to closely follow the desired trajectory.

Accurate localization is essential for effective trajectory tracking, and we achieve this through a particle filter that estimates the racecar’s position in the world frame. This is necessary to compare the current position against path coordinates, which are also expressed in the world frame. In simulation, localization was achieved directly from the `/odom` topic. For every new odometry message, we compute a steering command. To optimize performance, we utilize vectorized computations rather than individual (x, y) coordinate comparisons.

Step 1: Finding the Closest Point on the Path

We begin by identifying the point on the path nearest to the racecar. To do this, we compute the vector from the racecar’s position to each point along the path and calculate the magnitudes of these vectors using `np.linalg.norm()`. The point with the smallest magnitude is selected as the closest point. We retain its index to initiate our lookahead point search in the next step.

Step 2: Finding the Lookahead Point

The lookahead point is defined as the point on the path that intersects with a circle of radius r (the lookahead distance), centered at the racecar’s current

position vector q . Beginning from the closest point identified in Step 1, we evaluate each path segment, defined by vector $v = p_{i+1} - p_i$, to check for intersection with the lookahead circle.

We evaluate the discriminant of the resulting quadratic:

$$b^2 - 4ac, \quad (1)$$

where the coefficients are computed as:

$$a = v \cdot v, \quad (2)$$

$$b = 2v \cdot (p_i - q), \quad (3)$$

$$c = (p_i - q) \cdot (p_i - q) - r^2. \quad (4)$$

A non-negative discriminant indicates a valid intersection. We solve for the intersection using the quadratic formula:

$$t_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad t_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \quad (5)$$

We consider only values of t between 0 and 1, which ensures the intersection lies within the bounds of the path segment. If both t_1 and t_2 are within range, we select t_1 since it lies farther along the path. If neither value satisfies this condition, we proceed to the next segment.

Step 3: Converting to the Racecar Frame and Computing Steering

Once the lookahead point is identified, we transform its coordinates from the world frame to the racecar's local frame, resulting in $x_{\text{lookahead}}^{\text{racecar}}$ and $y_{\text{lookahead}}^{\text{racecar}}$. If $x_{\text{lookahead}}^{\text{racecar}} = 0$, the lookahead point lies directly ahead, and no steering is necessary.

We then compute the curvature α using the pure pursuit model:

$$\alpha = \frac{2y_{\text{lookahead}}^{\text{racecar}}}{r^2}, \quad (6)$$

and determine the steering angle as:

$$\delta = \arctan(l_w \cdot \alpha), \quad (7)$$

where $l_w = 0.32$ m is the measured wheelbase of the racecar. This steering angle is then used to command the vehicle to follow the path.

2.3 Localization: Particle Filter

2.3.1 Motion Model - Xavier

Dead-reckoning odometry (integrating the drive commands of the robot over some time) provides an estimate of the distance and direction the robot has

traveled within that time. The motion model takes this odometry as input to update the poses of each of the particles in our distribution.

At each timestep t , we have the car's initial world frame pose r_{k-1}^W , and its instantaneous velocity v^{k-1} with respect to that pose. Integrating over the timestep Δt , this velocity becomes the relative displacement $(\Delta r)^{k-1}$, termed the **relative odometry**:

$$r_{k-1}^W = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}_{k-1}^W \quad (8)$$

$$(\Delta r)^{k-1} = \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta \theta \end{pmatrix}^{k-1} \quad (9)$$

The relative odometry is rotated to the world frame, then added to the initial pose to find the pose at the subsequent timestep $t + 1$.

$$\mathbf{C}_{k-1}^W = \begin{bmatrix} \cos(\theta_{k-1}^W) & \sin(\theta_{k-1}^W) \\ -\sin(\theta_{k-1}^W) & \cos(\theta_{k-1}^W) \end{bmatrix} \quad (10)$$

$$r_k^W = r_{k-1}^W + \mathbf{C}_{k-1}^W * (\Delta r)^{k-1} \quad (11)$$

In our motion model, we initialize many particles at $t = 0$ which together form a stochastic distribution describing possible locations for the car's true position. At each timestep, the above position update is calculated for *each particle* using the current relative odometry. To handle many particles in real-time, our code uses *NumPy* operations to perform simultaneous matrix multiplication between homogeneous transforms representing each particle and the relative odometry. Each particle is updated as follows:

$$\begin{bmatrix} \mathbf{C}_{\mathbf{k}}^W & \begin{pmatrix} x \\ y \end{pmatrix}_k^W \\ \mathbf{0}_{1 \times 2} & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{C}_{\mathbf{k}-1}^W & \begin{pmatrix} x \\ y \end{pmatrix}_{k-1}^W \\ \mathbf{0}_{1 \times 2} & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{C}_{\mathbf{k}}^{k-1} & \begin{pmatrix} x \\ y \end{pmatrix}_k^{k-1} \\ \mathbf{0}_{1 \times 2} & 1 \end{bmatrix} \quad (12)$$

2.3.2 Sensor Model - Xavier

In order to decide which particles in our distribution best represent the car's position, which will be important for updating our distribution towards convergence, we need some way to validate the particles with respect to what we know about the local environment. Using our environment map, we can calculate what each particle "sees" relative to its pose, and compare that to what the car should be seeing from its pose, given by the current LIDAR scan. Using a function provided by the TAs, a ray tracing algorithm outputs a pseudo-LIDAR scan originating at each particle.

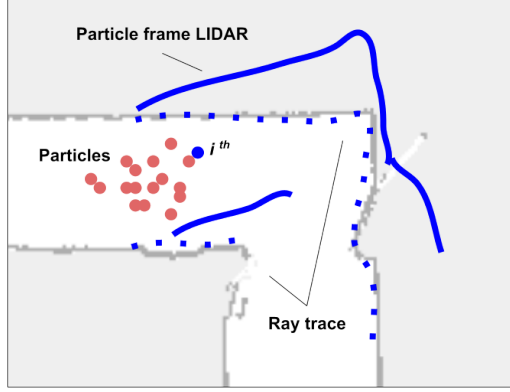


Figure 2: Each particle's ray traced environment should be compared to the LIDAR scan at its pose

We constructed a sensor model to determine how likely it is for a single ray traced scan to be a noisy version of the corresponding LIDAR scan from the car. Mathematically, the sensor model is a probability distribution constructed from a series of heuristics about the noise the LIDAR scan could introduce. For a single particle of interest, one axis represents a LIDAR measurement, and the second axis to the corresponding ray traced distance.

The four components of our sensor model

$$p_{hit} = \begin{cases} \frac{2}{d} \cdot (1 - \frac{z_k}{d}) & \text{if } 0 \leq z_k \leq z_{max} \\ 0 & \text{else} \end{cases} \quad (13)$$

assumes Gaussian noise around the true value of the LIDAR scan

$$p_{short} = \begin{cases} \frac{2}{d} \cdot (1 - \frac{z_k}{d}) & \text{if } 0 < z_k \leq d \\ 0 & \text{else} \end{cases} \quad (14)$$

assumes linearly increasing noise as scan distance approaches 0 from unexpected occlusions; increasing probability corresponds to increasing area as radius increases from a point in 2D space

$$p_{max} = \begin{cases} \frac{1}{\epsilon} & \text{if } z_k = z_{max} \\ 0 & \text{else} \end{cases} \quad (15)$$

***assumes high noise at maximal scan distance, from LIDAR not de-**

tected at the receiver due to high absorption

$$p_{rand} = \begin{cases} \frac{1}{z_{max}} & \text{if } 0 \leq z_k \leq z_{max} \\ 0 & \text{else} \end{cases} \quad (16)$$

assumes constant, low noise from random disturbances

$$p_{total} = p_{hit} + p_{short} + p_{max} + p_{rand} \quad (17)$$

The total probability is the sum of these components

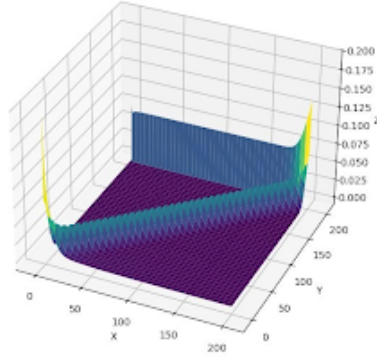


Figure 3: Full Probability Distribution of Sensor Model Noise

Particles are associated with multiple probabilities, one for each pair z_k, d of LIDAR measurements and ray traced distances. All of these independently sampled probabilities $p_{total}^{z_k, d}$, are then multiplied to find an overall likelihood for that particle. The sensor model outputs this likelihood for each particle i in our position distribution P .

$$p_i = \prod_{z_k, d} p_{total}^{z_k, d} \quad \forall p_i \in P \quad (18)$$

2.3.3 Resampling: Bring everything together - Insuh & Xavier

Now that the particle distribution has been updated with odometry (motion model), and the probability found for each particle being the ground truth pose (sensor model), we can improve the degree to which the distribution as a whole represents the ground truth pose. This is done through resampling. New particles are randomly selected from the post-motion model distribution according to the sensor model probabilities. The total number of particles does not change, and particles are sampled with replacement.

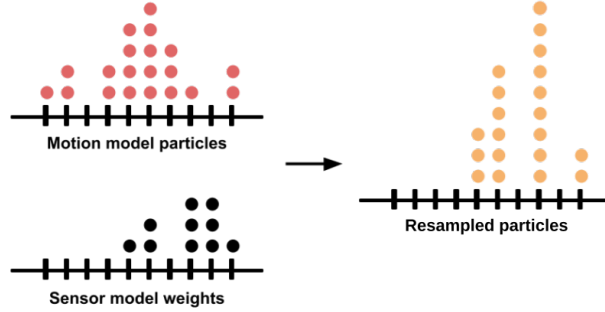


Figure 4: Simplified 1D depiction of particle resampling

After resampling, the particle filter publishes its current estimate for the racecar’s pose, taken as an ‘average’ over the pose distribution at that time step. Our ‘averaging’ procedure is a simple average over X and Y, and a simple ‘circular’ average over θ .

$$x_{\text{avg}} = \frac{1}{N} \sum_i x_i, \quad y_{\text{avg}} = \frac{1}{N} \sum_i y_i, \quad \theta_{\text{avg}} = \tan^{-1} \left(\frac{\sum_i \sin \theta_i}{\sum_i \cos \theta_i} \right) \quad (19)$$

We chose this definition for effectiveness, interpretability, and simplicity. For unimodal distributions, higher particle density in a given location gives more weight to the pose estimate, as expected. When dealing with a multimodal distribution, in most cases a simple average would translate the uncertainty of the particle filter to the final pose estimate. This makes it obvious when the algorithm is unable to handle an environment.

3 Experimental Evaluation

3.1 Path Planning - Inimai

We decided to evaluate the A* and RRT algorithms quantitatively and qualitatively in order to choose the best one to use for the car.

3.1.1 Quantitative Evaluation

We ran 20 trials in simulation for A* and RRT on three test scenarios of varying path lengths and difficulties that are representative of the Stata basement map.

Figure 5 shows the three cases.



Figure 5: Three test cases provided a representative evaluation of the algorithm on Stata basement. From left to right, the scenarios represent (1) short straight path, (2) medium corner path, (3) long complex path. Start poses are in green, end poses are in red, and the white traces designate the generated paths.

We evaluated the algorithms based on a comparison of computation time for the paths and path length. As our grid-based A* algorithm is deterministic, it found the same path length over each trial, while the RRT paths varied greatly in distance. Overall, we found that grid-based A* outputs shorter paths than RRT in less time. However, both algorithms found collision-free paths 100% of the time. Full numerical results are shown in Table 1.




| Scenario | | A* | RRT |
|---|---------------------|-------|-------|
|  | Path length (m) | 21.0 | 22.0 |
| | Compute time (s) | 0.036 | 0.431 |
| | Collision-free path | 100% | 100% |
|  | Path length (m) | 30.14 | 31.82 |
| | Compute time (s) | 0.069 | 0.72 |
| | Collision-free path | 100% | 100% |
|  | Path length (m) | 70.7 | 85.2 |
| | Compute time (s) | 0.342 | 1.02 |
| | Collision-free path | 100% | 100% |

Table 1: Numerical comparison over 20 trials shows that A* finds more optimal paths in less time for all three scenarios. Both algorithms find collision free paths for all trials.

3.1.2 Qualitative Evaluation

We noticed that RRT produces more variable paths with inconsistent compute times for the same start and goal poses. The final RRT paths differed greatly within consecutive runs, sometimes being close-to-optimal in length, and other times being much greater in length (such as following the opposite side of the Stata basement loop). Figure 6 shows an example of two such paths.

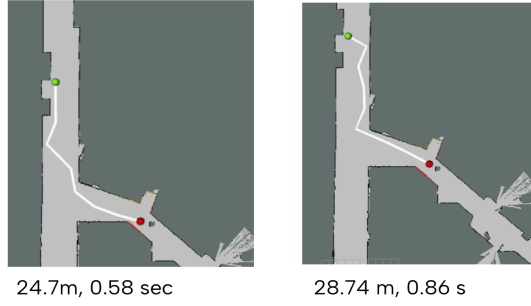


Figure 6: RRT has varied compute time and path length output for the same start and goal poses. The two paths shown demonstrate the variance between generated trajectories with different compute times.

With our heuristic of Euclidean distance, grid-based A* provides deterministically optimal paths for consistent compute times. Thus, due to our qualitative and quantitative evaluations, we chose A* to use for the real car.

3.2 Pure Pursuit - Xavier

Upon initial implementation, the pure pursuit controller performed reasonably well on a typical path generated by the planner in the Stata basement. To improve performance, we tested the relationship between racecar velocity and lookahead distance. A standard test path was used, and the car was driven at three speeds (1 m/s, 1.5 m/s, and 2 m/s) across a range of lookahead distances. The time-integrated distance between the car and the target path served as a measure of net divergence, and a graph of distance over time visualized the path segments where the car experienced the most and least difficulty.

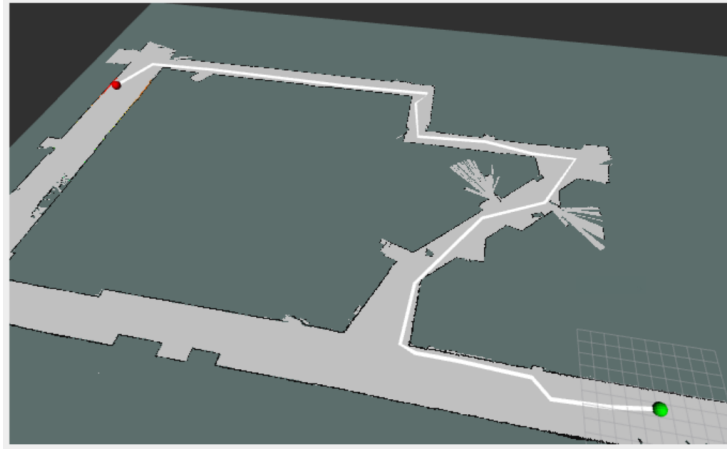


Figure 7: Standard test path for sim testing of pure pursuit look-ahead vs. speed

| Look-ahead Distance (m) | Integrated Error (m*s) |
|-------------------------|------------------------|
| 1 | 2.2 |
| 1.5 | 5.08 |
| 2 | 7.54 |
| 2.5 | 11.22 |

Table 2: Velocity = 1m/s — path divergence with look-ahead distance

| Look-ahead Distance (m) | Integrated Error (m*s) |
|-------------------------|------------------------|
| 1 | 1.23 |
| 1.5 | 2.99 |
| 2 | 3.5 |
| 2.5 | 6.17 |

Table 3: Velocity = 1.5m/s — path divergence with look-ahead distance

| Look-ahead Distance (m) | Integrated Error (m*s) |
|-------------------------|------------------------|
| 1 | 1.38 |
| 1.5 | 2.34 |
| 2 | 2.69 |
| 2.5 | 4.61 |

Table 4: Velocity = 2m/s — path divergence with look-ahead distance

From sim-testing it is clear that minimizing path divergence reduces the path

divergence metric. Not included are results for a 0.5 m look-ahead, at which point the car often loses the path entirely. This suggests the smaller look-ahead increases the car's agility (oscillatory behavior) at the cost of robust tracking. Simultaneously, faster velocities reduce the integrated error when comparing tests at the same look-ahead, because a given degree of oscillation can be more quickly corrected.

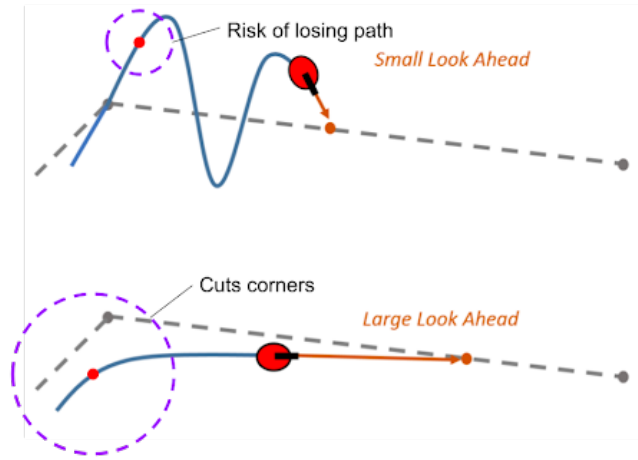


Figure 8: Comparing small and large look ahead distance on performance
Credit: MathWorks

Additionally, qualitative inspection of the error graphs for the best performing look-ahead distances at each velocity shows two large peaks corresponding to two sharp curves (circled in red) in the test path from Fig 7. Near zero error exists for long straight segments, such as after the second large peak (boxed green).

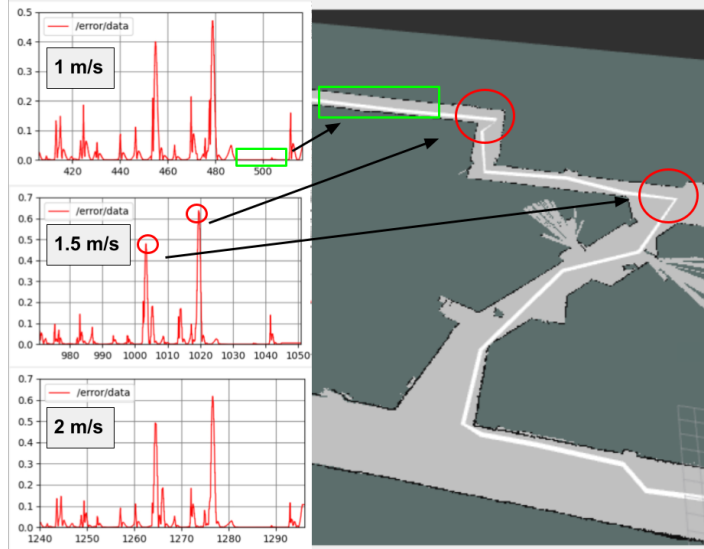


Figure 9: Connecting error-time plots to path characteristics

3.3 Particle Filter

3.3.1 Simulation - Russell

Our initial evaluation involved testing the motion and sensor models with unit tests, all of which passed. After developing the particle filter, we validated its performance in software using various testing procedures.

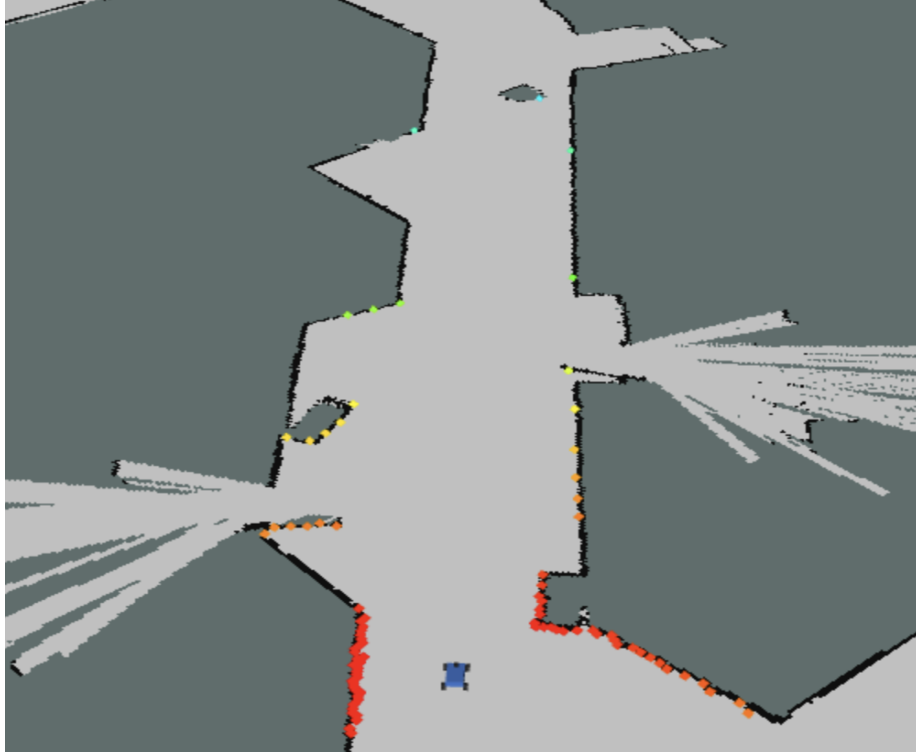


Figure 10: Feature-rich hallway right outside the lab

To evaluate performance under varying noise levels, we recorded a ROSbag of the car driving through the feature-rich hallway outside the lab using our wall-following implementation (Figure 10). This environment, rich in visual features, was ideal for localization. The ROSbag enabled repeatable and controlled tests. We varied only the noise in dx and dy , kept dt fixed at $\frac{\pi}{15}$, and used equal noise values for dx and dy .

To evaluate performance, we defined two key metrics:

$$\text{distance error} = \sqrt{(x_{\text{ground truth}} - x_{\text{average pose}})^2 + (y_{\text{ground truth}} - y_{\text{average pose}})^2} \quad (20)$$

$$\text{angle error} = ((t_{\text{ground truth}} - t_{\text{average pose}} + \pi) \bmod 2\pi) - \pi \quad (21)$$

Here, t represents orientation relative to the world frame. Both errors were published in real time and visualized using rqt. We also computed the overall average of these metrics across the run. These final average values served as key performance indicators.

During testing, the 2D pose estimate was manually initialized in RViz to start the particle filter.

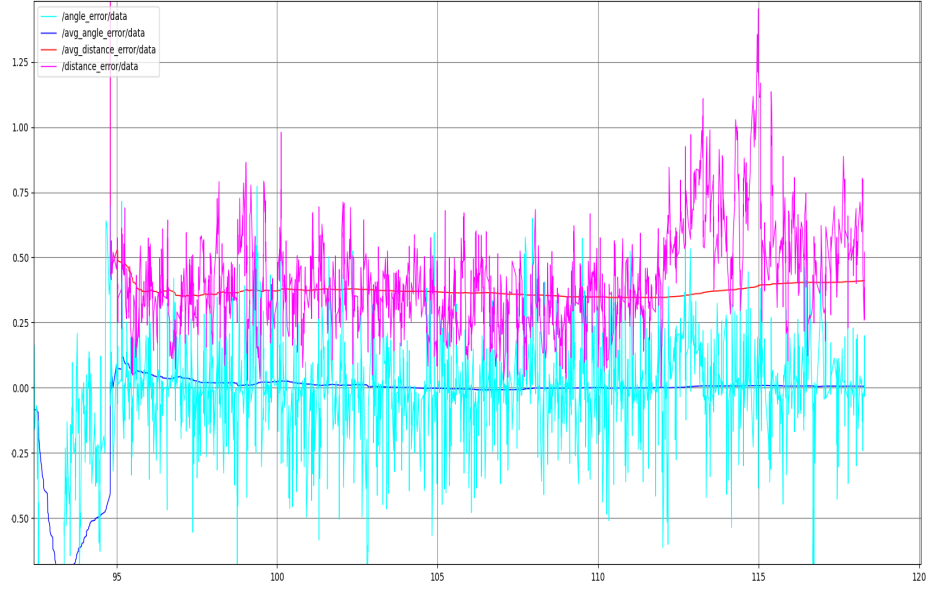


Figure 11: Simulation run in feature-rich hallway using noise levels $(0.1, 0.1, \frac{\pi}{15})$

With noise levels of $(0.1, 0.1, \frac{\pi}{15})$ in the x, y, and theta directions respectively, the filter showed consistent error trends. As seen in Figure 11, the average angle error converged toward zero, while the distance error stabilized around 0.375 meters. We repeated this with other noise levels:

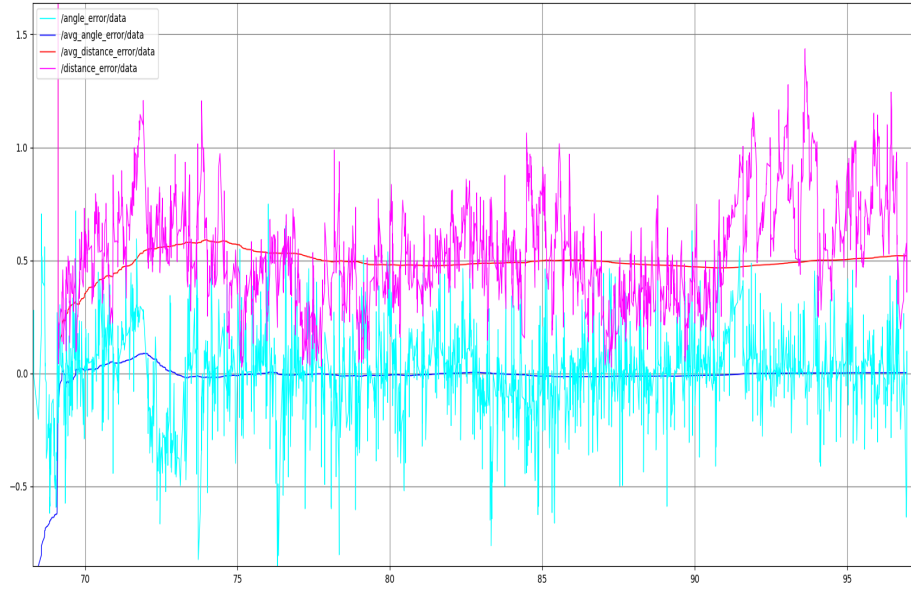


Figure 12: Simulation run in feature-rich hallway using noise levels $(0.2, 0.2, \frac{\pi}{15})$

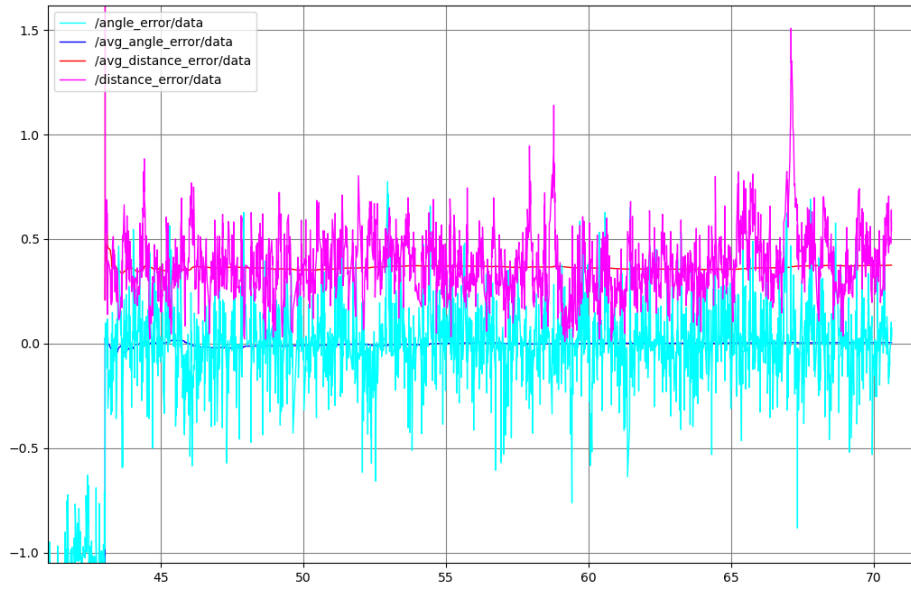


Figure 13: Simulation run in feature-rich hallway using noise levels $(0.4, 0.4, \frac{\pi}{15})$

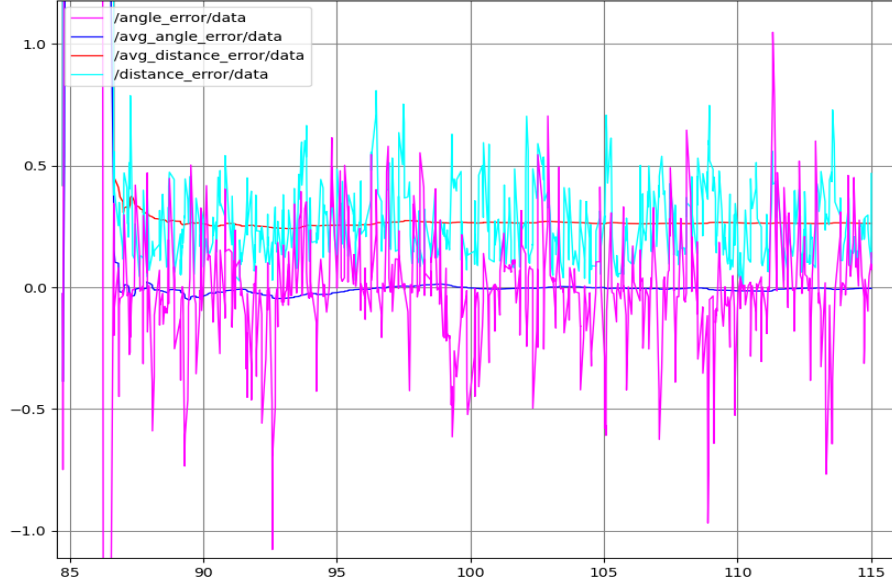


Figure 14: Simulation run in feature-rich hallway using noise levels $(0.8, 0.8, \frac{\pi}{15})$

| Noise Level | Average Distance Error (meters) | Average Angle Error (radians) |
|------------------------------|------------------------------------|----------------------------------|
| $(0.1, 0.1, \frac{\pi}{15})$ | 0.41 | 0.0056 |
| $(0.2, 0.2, \frac{\pi}{15})$ | 0.52 | 0.0026 |
| $(0.4, 0.4, \frac{\pi}{15})$ | 0.38 | 0.0051 |
| $(0.8, 0.8, \frac{\pi}{15})$ | 0.26 | 0.0043 |

Table 5: Resulting average error values from differing noise levels on feature-rich hallway

Across all runs in the feature-rich hallway, average distance errors stayed within a narrow range, averaging 0.39 meters. Average angle error was 0.0044 radians (0.25°), indicating consistent orientation accuracy.

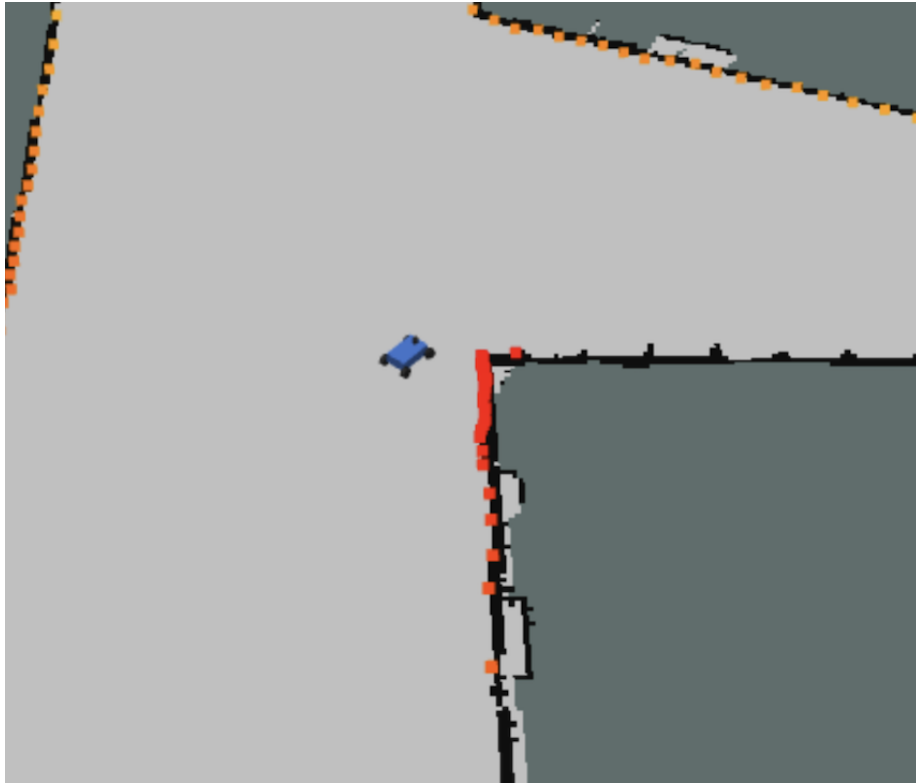


Figure 15: Turn from a feature-less hallway

We repeated the same tests on a different ROSbag. Again, only noise levels were changed. This time, we wanted to see how well the particle filter could handle a turn in a feature-less hallway as in Figure 15.

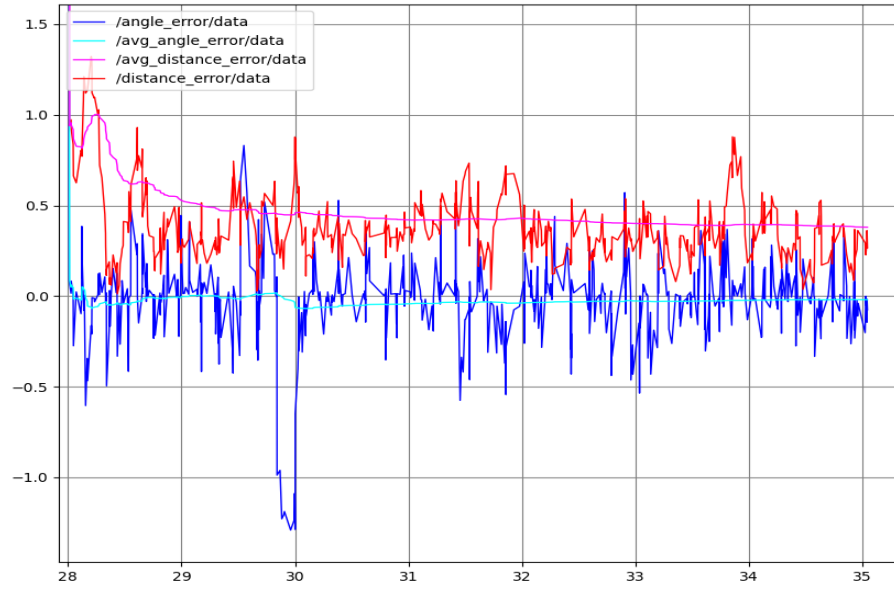


Figure 16: Simulation run in turn from feature-less hallway using noise levels $(0.1, 0.1, \frac{\pi}{15})$

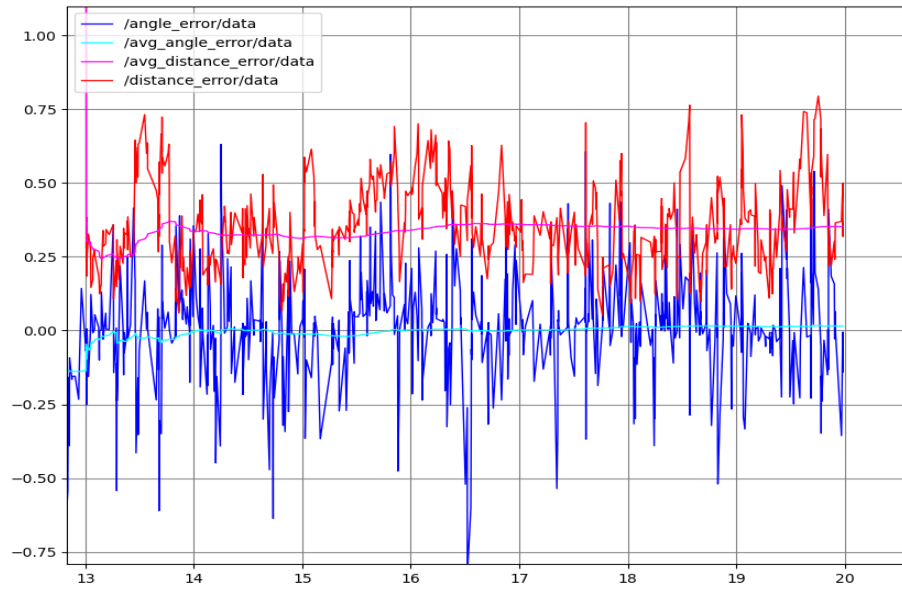


Figure 17: Simulation run in turn from feature-less hallway using noise levels $(0.2, 0.2, \frac{\pi}{15})$

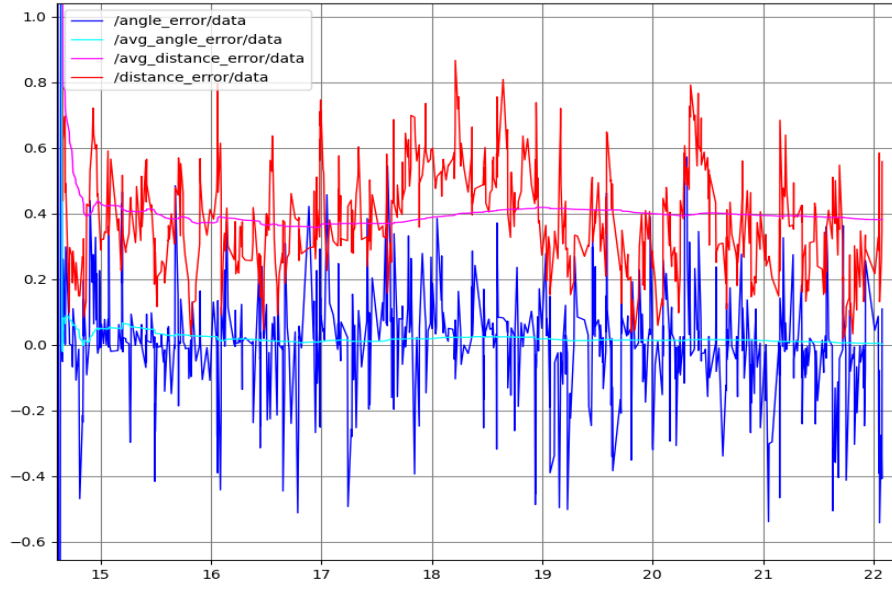


Figure 18: Simulation run in turn from feature-less hallway using noise levels $(0.4, 0.4, \frac{\pi}{15})$

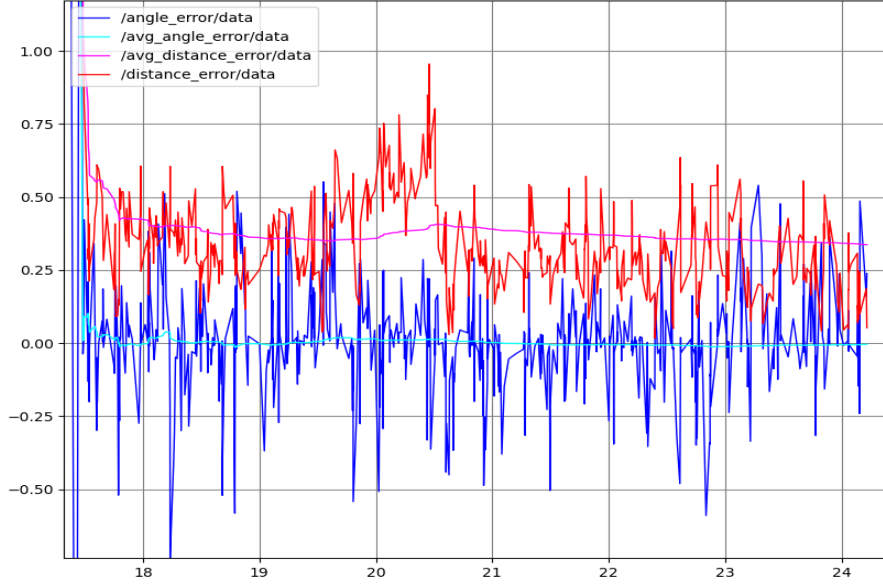


Figure 19: Simulation run in turn from feature-less hallway using noise levels $(0.8, 0.8, \frac{\pi}{15})$

| Noise Level | Average Distance Error (meters) | Average Angle Error (radians) |
|------------------------------|---------------------------------|-------------------------------|
| $(0.1, 0.1, \frac{\pi}{15})$ | 0.38 | 0.02 |
| $(0.2, 0.2, \frac{\pi}{15})$ | 0.35 | 0.015 |
| $(0.4, 0.4, \frac{\pi}{15})$ | 0.38 | 0.0012 |
| $(0.8, 0.8, \frac{\pi}{15})$ | 0.34 | 0.004 |

Table 6: Resulting average error values from differing noise levels on turn from feature-less hallway

Despite fewer landmarks and a turn, the filter remained stable. The average distance error was 0.36 meters, and average angle error was 0.01 radians (0.57°). These experiments demonstrate the particle filter’s reliability across noise levels and environments. Errors stayed low and consistent, indicating strong localization performance even under degraded sensing conditions.

3.3.2 Racecar - Inimai & Insuh

Testing the system implementation on the racecar involved evaluating the localization algorithm both statically and dynamically (with teleop maneuvers).

Quantitative Evaluation

Since ground-truth odometry wasn't available, we used static localization for benchmarking. We selected three checkpoints across different feature environments and recorded their positions in RViz using the Measure tool.

Checkpoint 1 was in a feature-rich hallway, Checkpoint 2 at a T-junction, and Checkpoint 3 in a sparse hallway. They are representative of unique scenarios in our map. We measure their real-life coordinates to compare with our simulation.

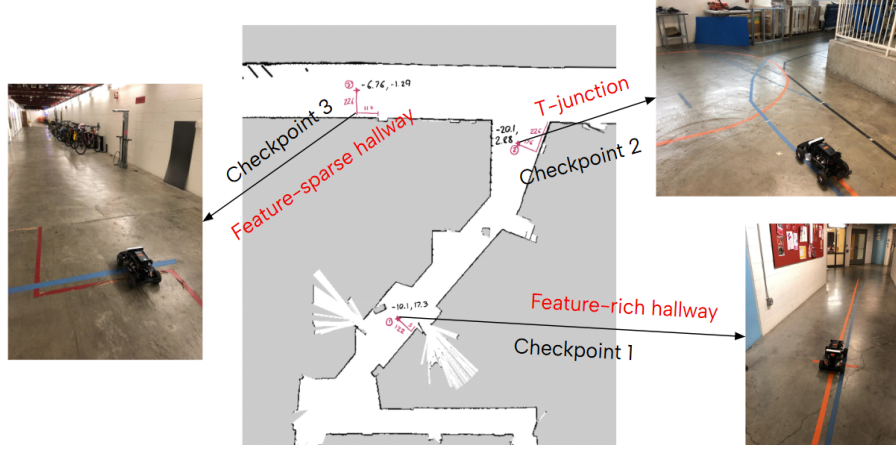


Figure 20: The three checkpoints used for static localization evaluation placed on the map and shown in real-life.

After obtaining coordinates for the true and localized positions, we calculated the straight-line distance between them. Table 7 shows the raw data and error per trial. Multiple trials were conducted to ensure consistency.

| Checkpoint | Checkpoint Type | Real (x,y) coordinates (m) | Localized (x,y) coordinates (m) | Distance Error (m) |
|------------|------------------------|----------------------------|---------------------------------|--------------------|
| 1 | Feature-rich Hallway | (0.05, 1.22) | (0, 1.35) | 0.14 |
| 2 | T-junction | (2.26, 1.26) | (2.09, 1.05) | 0.27 |
| 3 | Feature-sparse Hallway | (1.10, 2.26) | (2.75, 2.25) | 1.65 |

Table 7: Resulting distance error values for each checkpoint. Distance error increases with decreasing features available.

The localization algorithm performed well in areas with sufficient features (Checkpoints 1 and 2), showing minimal distance error. At Checkpoint 3, the lack of

features led to an x-direction error of 1.65m, though the y-coordinate remained accurate.

Qualitative Evaluation

–Particle Filter–

We evaluated the localization algorithm by placing and driving the car under various conditions in the Stata basement.

Static Localization The car was placed in front of a column in the feature-rich hallway, in the sparse hallway (bike storage), and at the corner between them. Localization was accurate in rich areas but consistently off by about 1 meter in the x-direction in sparse areas due to a lack of distinctive features.

Dynamic Localization The car was driven through routes including the T-junction, corner, and sparse hallway while monitoring the particle filter and average pose in RViz. The pose tracked well in rich areas but drifted in sparse ones. The algorithm remained responsive during zig-zag paths and high-speed motion, demonstrating robustness.

–Path Planning–

We tested path planning, localization, and pure pursuit by visually confirming that the planned RViz path matched the car’s trajectory in real-world tests.

Grid Search using A* Planning with A* on a grid-based map reliably produced optimal paths from start to goal in all test cases.

Pure Pursuit The controller followed the generated path in RViz, though the actual path was misaligned with the map due to localization issues.

Localization Accurate localization was key to reliable path tracking. Performance issues stemmed from:

- A rightward steering bias from a misaligned servo.
- A slow localization refresh rate, which delayed pose updates and caused the car to follow outdated paths.

Since hardware changes weren’t possible, we increased the localization update rate to 20 Hz. This allowed the system to track real-time deviations more accurately, improving alignment between the planned and actual paths.

4 Conclusion - Russell

In this design phase, our team successfully developed a robust autonomous navigation pipeline consisting of a Monte Carlo localization algorithm, dual path

planning approaches using A^* and RRT, and a pure pursuit controller for path following and steering command generation.

Our particle filter implementation integrates a motion model - used to update the poses of particles based on control input - and a sensor model, which evaluates the likelihood of each particle representing the racecar's true position. Together, these models enable our particle filter to continually refine the estimated pose through resampling based on particle weights, resulting in reliable localization.

For path planning, we experimented with both A^* and RRT algorithms, each operating on a discretized map of the environment. While A^* systematically explores grid-based paths to determine the shortest path to the goal, RRT constructs a rapidly-expanding sampling tree that explores the state space. Although RRT provides flexibility through random sampling, the speed and reliability of A^* made it a better fit for our application, and it was ultimately chosen for integration.

The pure pursuit controller uses the output of the localization and path planning modules to compute steering angles that guide the racecar along the path. By leveraging a lookahead point—computed based on a configurable distance parameter—and translating it into curvature-based steering commands, the racecar is able to follow the trajectory with precision and stability.

While we have achieved strong performance across all components, there is still room for refinement. In particular, we plan to enhance the particle filter by optimizing its runtime efficiency, improving initialization in feature-sparse regions, and evaluating its robustness across varied map environments. These improvements will be crucial in ensuring generalization and reliability as we transition into more complex navigation scenarios in the next phase.

Overall, this phase has established a strong foundation for autonomous navigation, and we are excited to continue iterating and expanding the capabilities of our system moving forward, particularly in the final challenge.

5 Lessons Learned

5.1 Xavier

From a technical perspective, I learned that thinking through an algorithm is one thing, but that programming something that works is another thing entirely, not to mention writing efficient code. From this perspective, it is very important to leverage the strengths of your teammates *and of yourself*. It is important to understand how things are working, and to learn how to do something for yourself, but there are diminishing returns on struggling through something that

a teammate could do in their sleep instead of using your skills on another branch of the project. Communication is an integral part of this, because teammates need be willing to understand each other's strengths, and how to structure the project so that each person's skills are fully leveraged. After implementation, a reflection on *how* something was done and *why* an initial attempt didn't work still allows for valuable learning. I also learned that while documenting quantitative tests can seem intimidating, it is very rewarding and surprisingly straightforward if you take things one step at a time. I actually appreciate evaluation much more now as a path to intentionally improve a system. Lastly, I learned that sometimes just being there to support your team is important, even when there is not much you are physically assigned to do. Being there to offer a helping hand, bounce ideas off one-another, and support the team's morale towards reaching a goal is an important part of what it means to be a team.

5.2 Russell

Technical: Through this lab, I gained a solid understanding of the pure pursuit controller and how it differs from traditional PID control. Previously, we had relied on PID controllers for path following, but fine-tuning it's multiple parameters proved challenging and time-consuming in practice. In contrast, the pure pursuit controller leverages geometric reasoning to generate steering commands and requires only a single tunable parameter—the lookahead distance. This made it much easier to adjust and test experimentally. Learning this approach gave me a deeper appreciation for simpler, more intuitive control strategies that can often outperform more complex alternatives in real-world settings.

Communication and Collaboration: Early in the project, I communicated to my teammates that I would only be available to design the pure pursuit controller and would not be able to participate in experimentation due to scheduling constraints. I realized the importance of setting expectations early and being transparent about availability. This allowed the team to plan accordingly and distribute tasks more effectively. I'm glad I communicated clearly and will aim to be more available and involved in future stages of the project, particularly during testing and final integration.

5.3 Insuh

Through this lab, I was able to appreciate the importance of making sure every subsystem of the car was robust. Even though we had a great path planning algorithm, we weren't able demonstrate the capability of it on the actual car until we made sure that the other systems on the car, namely the localization, was able to compensate for the mechanical bias that the hardware had.

5.4 Inimai

Technical: From this lab, I deepened my understanding of path planning, particularly through the implementation of A* and RRT. I learned how to design and tune these algorithms to account for real-world obstacles. Working with numpy and optimizing array-based operations taught me the importance of computational efficiency in real-time systems. Overall, the lab reinforced how algorithmic design, optimization, and validation come together to build efficient trajectory generators.

CI: This lab emphasized the importance of clear, precise communication within a technical team. Collaborating with teammates on dividing tasks path planning and following required consistent updates, shared documentation, and frequent check-ins to ensure alignment and integration of our modules. Presenting our findings to the staff helped me practice distilling complex algorithms into accessible explanations.