Lab #6: Developing and Evaluating an Integrated Path-Planning and Navigation System via A* Search, Probabilistic Roadmaps, Pure Pursuit Control, and Monte-Carlo Localization

Team #7

Edward Cheng
Audrey Lee
Christopher Liem (Editor)
Drew Tufto
Richard Yeboah

Robotics: Science and Systems

April 24, 2025

1 Introduction

Yibo Cheng

In previous labs, we taught the robot how to see the world through perception and how to understand where it is on a map through localization. The final step toward full autonomy is planning a viable path and navigating it independently. In this lab, we focused on motion planning, experimenting with different ways to represent the map and exploring both search-based algorithms like A* and sampling-based approaches like Probabilistic Roadmap (PRM). Each method comes with its own trade-offs: A* guarantees an optimal path but can be slow, while PRM is faster but less reliable in certain environments. Choosing the right approach depends on the situation. With this lab, we put the final piece into place, bringing together perception, localization, and planning to achieve full autonomy.

In this lab, we were given an accurate map of the basement of the Stata Center and tasked with enabling the robot to autonomously navigate from a specified start point to a chosen goal. After evaluating both search-based and sampling-based planning methods, we decided to use the A* algorithm for path planning due to its reliability and optimality. To follow the planned path, we implemented a pure pursuit controller. Throughout this process, we also relied on the localization algorithm developed in the previous lab, since accurate positioning is crucial for staying on track. By integrating localization, planning, and control, we successfully enabled the robot to navigate the basement on its own.

2 Technical Approach

2.1 Motion Model

Drew Tufto

Our goal for the motion model is to update our particles in a manner that is consistent with the motion of our actual car. In a real deployment of our vehicle, our odometry information is subject to noise which makes a deterministic motion model infeasible, as we would quickly drift to an incorrect location. Instead, we model the possible locations the car could be at given noisy odometry and previous state, and sample from that distribution to update our particles.

2.1.1 Technical Implementation

The model takes wheel odometry as input. Specifically, the input is the incremental displacement in the local frame of the robot, given as the vector.

$$\Delta X = \begin{bmatrix} \Delta x & \Delta y & \Delta \theta \end{bmatrix} \tag{1}$$

This vector is derived by combining the vehicle's motor outputs and steering inputs through dead reckoning, and is published to the `/Odom` topic.

The heart of the model is the propagation equation that updates the position of each particle (x, y) and heading θ. We first model the noise as zero-mean Gaussians, applied independently to each dimension of the odometry. Then, we apply a simple twist matrix to the odometry position to transform it into the global frame. Combining the transformed position with the absolute change in theta, we obtain the global state vector for each particle.

Mathematically, given a particle with state $[x,\, y,\, \theta]$, the update is performed as follows:

$$\begin{aligned} dx_{\text{noisy}} &\sim \mathcal{N}(\Delta x, \sigma_x^2) \\ dy_{\text{noisy}} &\sim \mathcal{N}(\Delta y, \sigma_y^2) \\ d\theta_{\text{noisy}} &\sim \mathcal{N}(\Delta \theta, \sigma_\theta^2) \end{aligned} \tag{2}$$

$$\begin{aligned} x_{\text{new}} &= x + \cos(\theta)\, dx_{\text{noisy}} - \sin(\theta)\, dy_{\text{noisy}}, \\ y_{\text{new}} &= y + \sin(\theta)\, dx_{\text{noisy}} + \cos(\theta)\, dy_{\text{noisy}}, \\ \theta_{\text{new}} &= \theta + d\theta_{\text{noisy}}, \end{aligned}$$

where the terms dx_{noisy}, dy_{noisy}, $d\theta_{\text{noisy}}$ represent the odometry data perturbed by Gaussian noise. Note that the noise is sampled independently for each particle. Our resulting model utilized the following variance values at deployment:

$$\sigma_x^2 = 0.1(m), \sigma_y^2 = 0.1(m), \sigma_\theta^2 = \texttt{np.deg2rad}(20)$$

Finally, our motion model returns an N x 3 matrix of the updated particles, where each particle is described as

$$P_i = \begin{bmatrix} x_{new} & y_{new} & \theta_{new} \end{bmatrix} \tag{3}$$

2.2 Sensor Model

Christopher Liem

After implementing the motion model, our team's next goal was to assign likelihoods to each particle for our particle filter to use through the sensor model.

2.2.1 Likelihood Model Introduction

Let L be the number of beams in our LiDAR scan. For a LiDAR scan acquired at iteration k, $z_k^{1:L}$, define the probability that each particle $\hat{p}_k^{(j)}$ produces the scan given the particle's hypothesized position $x_k^{(j)}$ and a static map m at time step k by

$$p\left(z_k^{1:L} \mid x_k^{(j)}, m\right) \tag{4}$$

Assuming that the LiDAR measurements are independent,

$$p\left(z_k^{1:L} \mid x_k^{(j)}, m\right) = \prod_{i=1}^{L} p\left(z_k^{(i)} \mid x_k^{(j)}, m\right) \tag{5}$$

Given a particle pose estimate $x_k^{(j)}$, we can compute the ground-truth distances $d^{1:L}$ that an ideal LiDAR would observe using ray-casting on m from pose $x_k^{(j)}$. However, assigning likelihoods to each particle at every iteration is computationally expensive, so we precompute a lookup table that discretizes our continuous and complex distribution at runtime. The columns of the lookup table represent the ground truth distances which lie between 0 and $z_{\max} = 200$ and the rows correspond to the LiDAR measurements, resulting in a L by $z_{\max} + 1$ table. Discretization is discussed in section 2.2.2.

2.2.2 Likelihood Model Discretization

As mentioned previously, we discretized our sensor model distribution to enhance efficiency. Let d be the ground truth range of a particular measurement. We model the likelihood of each range measurement as a mixture of likelihoods of four different events:

1. A known obstacle in the map is detected. The event can be modeled as a Gaussian distribution centered around the ground truth distance between our predicted pose and the nearest known obstacle on the map

$$p_{\text{hit}}\left(z_k^{(i)} \,|\, x_k,\, m\right) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\left(z_k^{(i)}-d\right)^2}{2\sigma^2}\right) & \text{if } 0 \le z_k \le z_{\max} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

   where σ is the standard deviation of the Gaussian. For our implementation, we chose $\sigma = 8.0$.

2. A short measurement is detected, possibly due to unknown obstacles and internal LiDAR reflections. The distribution for this event is a downward sloping line.

$$p_{\text{short}}\left(z_k^{(i)} \,|\, x_k,\, m\right) = \frac{2}{d} \begin{cases} 1 - \frac{z_k^{(i)}}{d} & \text{if } 0 \le z_k^{(i)} \le d \text{ and } d \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

3. A large measurement is detected, possibly because LiDAR beams did not bounce back to the sensor. This is a uniform distribution.

$$p_{\max}\left(z_k^{(i)} \,|\, x_k,\, m\right) = \begin{cases} \frac{1}{\epsilon} & \text{if } z_{\max} - \epsilon \le z_k^{(i)} \le z_{\max} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

4. A completely random measurement is detected. This is modeled as

$$p_{\text{rand}}\left(z_k^{(i)} \,|\, x_k,\, m\right) = \begin{cases} \frac{1}{z_{\max}} & \text{if } 0 \le z_k^{(i)} \le z_{\max} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

The law of total probability allows us to write the likelihood of range measurement i as

$$\begin{aligned} p\left(z_k^{(i)} \,|\, x_k,\, m\right) &= \alpha_{\text{hit}} p_{\text{hit}}\left(z_k^{(i)} \,|\, x_k,\, m\right) \\ &+ \alpha_{\text{short}} p_{\text{short}}\left(z_k^{(i)} \,|\, x_k,\, m\right) \\ &+ \alpha_{\max} p_{\max}\left(z_k^{(i)} \,|\, x_k,\, m\right) \\ &+ \alpha_{\text{rand}} p_{\text{rand}}\left(z_k^{(i)} \,|\, x_k,\, m\right) \end{aligned} \tag{10}$$

where α_{hit}, α_{short}, $\alpha_{\max}$, and α_{rand} are the priors, or our previously estimated distributions, of acquiring a hit, short, max, or random measurement, respectively. In order for this to be a proper probability distribution,

$$\alpha_{\text{hit}} + \alpha_{\text{short}} + \alpha_{\max} + \alpha_{\text{rand}} = 1 \tag{11}$$

We chose $\alpha_{\text{hit}} = 0.74$, $\alpha_{\text{short}} = \alpha_{\max} = 0.07$, and $\alpha_{\text{rand}} = 0.12$.

2.3 Particle Filter

Christopher Liem

In this section, we discuss how to utilize the motion and sensor models to implement Monte Carlo Localization on our robot in simulation and the real world.

2.3.1 Particle Initialization

We begin by initializing the particles. We subscribe to a ROS topic, which retrieves the robot's pose estimate from when the algorithm began. This yields a `PoseWithCovariance` message with information about position, orientation, and covariance. The robot's x and y coordinates are extracted from the positional information in Cartesian form. The rotation around the z-axis, θ, can be extracted by converting the orientation, which is a quaternion, to yaw using the formula

$$\theta = \arctan\left(2 \cdot (q_w q_z + q_x q_y),\, 1 - 2 \cdot (q_y^2 + q_z^2)\right) \tag{12}$$

The covariance matrix of the message takes the form

$$\sum = \begin{bmatrix} \sigma_x^2 & \cdots & \cdots & \cdots & \cdots \\ \cdots & \sigma_y^2 & \cdots & \cdots & \cdots \\ \cdots & \cdots & \sigma_z^2 & \cdots & \cdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \cdots & \cdots & \cdots & \cdots & \sigma_\theta^2 \end{bmatrix} \tag{13}$$

Each entry along the diagonal σ_k^2 represents the variance of k and it follows that σ_k is the standard deviation of k. Each particle $\hat{p_i}$ can be initialized in the form

$$\hat{p_i} = \begin{bmatrix} X_i & Y_i & \Theta_i \end{bmatrix} \tag{14}$$

where X_i, Y_i, and Θ_i follow the Gaussian distributions

$$\begin{aligned} X_i &\sim \mathcal{N}(x, \sigma_x) \\ Y_i &\sim \mathcal{N}(y, \sigma_y) \\ \Theta_i &\sim \mathcal{N}(\theta, \sigma_\theta) \end{aligned} \tag{15}$$

Sampling each particle from a Gaussian distribution effectively adds noise to our particles, accounting for any uncertainty within our odometry estimates. Furthermore, this process guarantees that our particles are initialized within the scope of the robot's initial pose. We could've initialized the particles randomly across the Stata basement, but doing so may have required numerous particle updates to converge to the robot's actual pose, which is inefficient. Although randomly initializing particles randomly around Stata basement has its advantages such as allowing us to initialize without knowing the robot's actual location, we prioritized efficiency instead.

We store our particles in a vector P. If N is the number of particles, then P takes the form

$$P = \begin{bmatrix} \hat{p_1} \\ \hat{p_2} \\ \vdots \\ \hat{p_N} \end{bmatrix} = \begin{bmatrix} X_1 & Y_1 & \Theta_1 \\ X_2 & Y_2 & \Theta_2 \\ \vdots & \vdots & \vdots \\ X_N & Y_N & \Theta_N \end{bmatrix} \tag{16}$$

2.3.2 Particle Updates

When we receive `LaserScan` data from our LiDAR, we use the sensor model to evaluate the probability of each particle existing given the new data, which we downsampled to 100 points for efficiency. This is the update step of Monte Carlo Localization. We store the probabilities in a vector W_N.

$$W_N = \begin{bmatrix} p(\hat{p_1}) \\ p(\hat{p_2}) \\ \vdots \\ p(\hat{p_N}) \end{bmatrix}$$

where $p(p_i)$ is the probability that the ith particle exists. We choose whether to resample during this iteration according to these probabilities. Let S be a Bernoulli random variable with parameter p ($S \sim \text{Ber}(p)$). If $S = 1$, then we don't resample and multiply the weights of each particle by the probabilities that they exist. Suppose that w_i is the current weight of the ith particle. The new weights will be

$$W = \begin{bmatrix} w_1 \cdot p(\hat{p_1}) \\ w_2 \cdot p(\hat{p_2}) \\ \vdots \\ w_N \cdot p(\hat{p_N}) \end{bmatrix} \tag{17}$$

If $S = 0$, then we resample. First, we transform the particle probabilities by taking their cube roots and normalizing the result. In other words, we have

$$W_N = \begin{bmatrix} \frac{p(\hat{p_1})^{\frac{1}{3}}}{\sum_{i=1}^{N} p(\hat{p_i})^{\frac{1}{3}}} \\ \frac{p(\hat{p_2})^{\frac{1}{3}}}{\sum_{i=1}^{N} p(\hat{p_i})^{\frac{1}{3}}} \\ \vdots \\ \frac{p(\hat{p_N})^{\frac{1}{3}}}{\sum_{i=1}^{N} p(\hat{p_i})^{\frac{1}{3}}} \end{bmatrix} \tag{18}$$

This process prevents a divide by zero error that arises when averaging with weights that are tiny compared to others. We take a new sample of N particles from our current particles which are distributed according to their probability of existing as given by W_N. P will be updated according to the particle we take and the corresponding entry in W will be updated with the particle's weight in

W_N. For example, if our first sampled particle is $\hat{p_{50}}$, then we update $P^{(1)} = \hat{p_{50}}$ and $W^{(1)} = W_N^{(50)}$. Note that notation such as $W^{(i)}$ is used to denote the ith entry of W. Resampling increases the accuracy of our robot's pose estimation because it is likely to keep the particles that have high probabilities of existing while filtering out particles that aren't as accurate, discarding unimportant information.

When we receive odometry data, we update our particles, which is the prediction step of Monte Carlo Localization. From the `Odometry` message, we extract the linear velocity in the x and y direction, which is given by $\dot{x}$ and $\dot{y}$, respectively. Then, we extract the angular velocity along the z axis, given by $\dot{\theta}$, and compute the amount of time that passed since we last received odometry data. Using all these parameters, our motion model updates the pose of each particle.

2.3.3 Pose estimation

After updating our particles, we can estimate our robot's pose $r = \begin{bmatrix} x & y & \theta \end{bmatrix}$. Our robot's current x and y coordinates are given by a weighted average of the respective coordinates of the particles. In other words, we have

$$x = \frac{\sum\limits_{i=1}^{N} w_i X_i}{\sum\limits_{i=1}^{N} w_i} \text{ and } y = \frac{\sum\limits_{i=1}^{N} w_i Y_i}{\sum\limits_{i=1}^{N} w_i} \tag{19}$$

where w_i is the weight of the ith particle. Our robot's current yaw is computed using a weighted circular average of the yaw of each particle.

$$\theta = \arctan\left(\sum_{i=1}^{N} w_i \sin\Theta_i, \sum_{i=1}^{N} w_i \cos\Theta_i\right) \tag{20}$$

We published this estimated pose to both `Odometry` and `Transform` messages, where the position of each message is given by $\begin{bmatrix} x & y & 0 \end{bmatrix}$ and the orientation by calling the built-in `quaternion_from_euler` function from the `tf_transformations` library.

$$q_x, q_y, q_z, q_w = \text{quaternion\_from\_euler}(0,\ 0,\ \theta) \tag{21}$$

We store all x, y coordinates of the robot in a `Marker` message of type `LINE_STRIP` to visualize previously estimated robot poses in real time.

2.4 Path Planning

After successfully localizing our robot in Stata Basement, our next goal was to autonomously navigate the basement using path-planning and control algorithms. This section focuses on path planning. Given two points on the map, we wish to develop a path for our robot to follow to get from point A to point B safely and efficiently.

2.4.1 Map Representation

Christopher Liem

Before attempting to construct a path from point A to point B, we need a graph of nodes to traverse. When we launch the racecar simulator, an `OccupancyGrid` message is published, which contains the grid representation of Stata Basement as a 2D matrix. In this grid representation, the (i, j) entry will be 1 if the real-world-coordinate (i, j) contains an obstacle and 0 otherwise. Thus, we can use this matrix to easily tell whether or not a node can be traversed through indexing. This message is only published when we run the simulator, so we only need to preprocess the map once before running our path and motion planning algorithms.

Since our robot isn't a point mass, attempting to construct a shortest path may result in cutting corners which aren't feasible in the real-world. To deal with this issue, we first create a kernel using a fixed radius of 0.3 meters, which we chose, to represent the neighborhood of each coordinate. Using this kernel, we dilate the map with OpenCV, expanding the region of potential obstacles on the map. For example, coordinate (i, j) in the matrix could've been 0 before dilation but turned into 1 after dilation because coordinate $(i, j+1)$ contained an obstacle. Dilating the map before running our planning algorithms prevented us from creating paths that were too close to walls, allowing our robot to reach our goal safely.

2.4.2 Search-Based Planning (A^*)

Christopher Liem

Algorithm

A^* is a search-based algorithm, which means it attempts to find a path from start to end by iteratively expanding its search space. Our implementation relies on two data structures: an open set as a min-heap and a closed set as a hashmap. The open set keeps track of which nodes we still need to process, meaning that they are potential shortest-path candidates, while the closed set tracks which nodes we have finished processing. At every step of our algorithm, we pop the node with lowest cost $C(n_i)$ and add the coordinates in its octagonal neighborhood

$$n_{i+1} \in \left\{ (n_{i_x} + \delta_x, n_{i_y} + \delta_y) \,|\, (\delta_x, \delta_y) \in \{-1, 0, 1\}^2 \setminus \{(0,0)\} \right\} \tag{22}$$

to the open set, given that they don't contain an obstacle and that they aren't in the closed set. To determine the order in which we pop nodes from the open set, we need to define our cost function $C(n_i)$. Let s, e, and n_i be our start, end, and current coordinate that we are searching respectively. Then we have

$$C(n_i) = S(s, n_i) + h(n_i, e) \tag{23}$$

where $C(n_i)$ is the current cost of node n_i, $S(s, n_i)$ is the current shortest path weight from s to n_i, and $h(n_i, e)$ is a heuristic which we define later. To make analysis well-defined, assume $S(s, n_i)$ is initialized to infinity for every node n_i. To evaluate $S(s, n_i)$ at each iteration, we use Euclidean Distance which is given by

$$E(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2} \tag{24}$$

Assuming that we are searching node n_{i-1} and n_i is its neighbor, $S(s, n_i)$ is computed using a minimum.

$$S(s, n_i) = \min\{S(s, n_{i-1}) + E(n_{i-1}, n_i),\ S(s, n_i)\} \tag{25}$$

After fully processing n_{i-1}, we add it to the closed set and set the parent of node n_i to n_{i-1} if $S(s, n_{i-1}) + E(n_{i-1}, n_i)$ was the minimum in (25), otherwise we keep the parent of n_i unchanged. After we reach our goal state e, we back-track through the parents to reconstruct the shortest path, completing our A^* algorithm.

Heuristic Selection

The efficiency of A^* is dependent upon choosing an optimal heuristic. We had two choices for $h(n_i, e)$: the Euclidean Distance that we used for the shortest path weight and the Manhattan Distance which is defined by

$$M(a, b) = \|a_x - b_x\| + \|a_y - b_y\| \tag{26}$$

To decide between the two heuristics, we conducted three test cases: straight line, turn, and full-map paths. First, we evaluated performances when the start and end goal could be traversed by a straight line. Figure (1) shows that the performance of both heuristics was identical, which makes sense because the Euclidean and Manhattan heuristics are equivalent in one dimension.

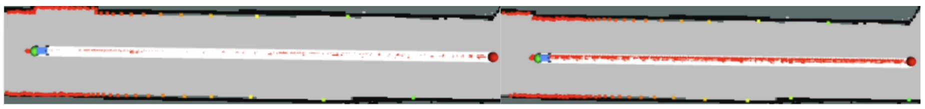

Fig. 1: **Manhattan (left) and Euclidean (right) heuristics have similar performance for straight paths**. The shortest paths found are highlighted in white and the coordinates searched are in red. Both heuristics cause A^* to converge to a straight-line path with a similar number of nodes traversed.

Next, we evaluated performance in the case where the shortest-path involved a turn. Figure (2) shows that Manhattan A^* was optimal since it only traversed coordinates that were southeast of the initial pose which improved runtime. Despite this, both heuristics converged to similar paths that only differ due to variance in pose initialization.

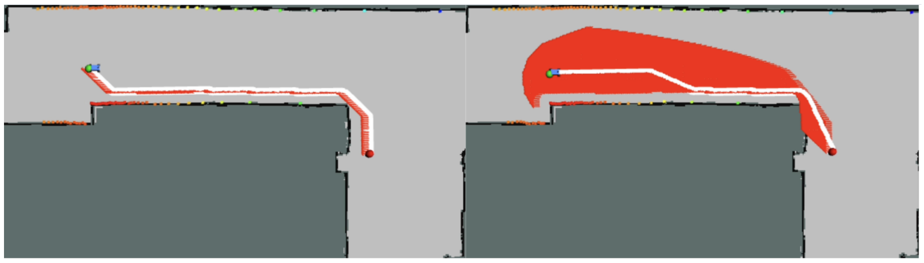

Fig. 2: **Manhattan A^* (left) searches less nodes than Euclidean A^* (right) on paths with a turn**. The shortest paths found are highlighted in white and the coordinates searched are in red.

Finally, we tested both heuristics on start and goal poses that were across the entire basement. Figure (3) shows that Manhattan A^* pruned the entire left-most hallway while Euclidean A^* attempted to traverse through it, significantly increasing runtime.

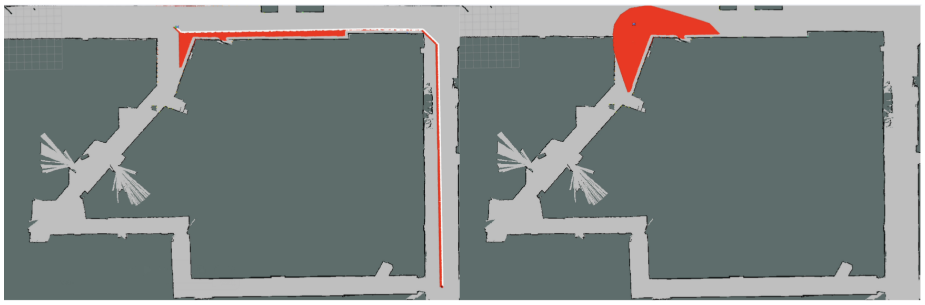

Fig. 3: **Manhattan A^* (left) is optimal for full-map paths**. The shortest paths found are highlighted in white and the coordinates searched are indicated in red. We didn't allow Euclidean A^* to converge due to time constraints.

.

Through these three test cases, we deduced that Manhattan was the optimal heuristic for our A^* implementation.

2.4.3 Sample-Based Planning Yibo Cheng

Alternatively, we also implemented the Probabilistic Roadmap (PRM) algorithm to plan a feasible path for the vehicle. First, we sample a thousand points within the free space of the map. For each sampled point, we attempt to connect it to its ten nearest neighbors, ensuring the resulting edge is collision-free. To guarantee that the path is not only safe but also kinematically feasible for the vehicle to follow, each edge is constructed using Dubins curves. These curves

respect the vehicles minimum turning radius and avoid sharp turns that would be difficult or impossible to navigate.
After constructing the graph of nodes and edges, we apply the A* algorithm to compute the shortest collision-free path from the vehicles initial pose to the desired goal pose.

2.5 Pure Pursuit Control

Richard Yeboah

The Pure Pursuit algorithm is a geometric path-tracking controller widely used in mobile robotics and autonomous ground vehicle navigation. It operates by selecting a *lookahead point* P_{target} at a fixed Euclidean distance L_d from the current vehicle position, along the reference trajectory. The algorithm computes the required curvature κ of a circular arc that connects the robot's current position to this target point, thereby generating the appropriate steering command to follow the path.
Assume the vehicle is modeled as a non-holonomic unicycle with a rear-axle reference point located at position (x, y) and orientation θ. Given the lookahead distance L_d, the Pure Pursuit algorithm calculates the angle α between the vehicles heading and the line connecting its position to P_{target}. The curvature κ of the circular path to the target point is given by:

$$\kappa = \frac{2\sin(\alpha)}{L_d}$$

For a vehicle with wheelbase L, the corresponding steering angle δ (assuming a bicycle model) is:

$$\delta = \tan^{-1}(L \cdot \kappa)$$

The algorithm continuously updates P_{target} and recalculates δ at each control cycle, enabling the vehicle to iteratively "pursue" the path. While it is computationally efficient and suitable for real-time applications, its performance is sensitive to the choice of L_d, which must balance tracking accuracy and stability.

3 Experimental Evaluation

3.1 A* vs. PRM (Simulation)

Audrey Lee

We compared A* Search (a search-based method) to PRM (a sampling-based method) to empirically evaluate each method's strengths and weaknesses. In general, search-based methods like A* guarantee completeness and optimality (i.e., the shortest path, if a path exists) given an admissible heuristic. They can also incorporate dynamics given cost functions and heuristics that reflect those dynamics. However, A* sacrifices computational complexity due to its reliance on explicit state space enumeration, which becomes computationally expensive in large environments. As a single-query planner, it also requires a full search for each new start-goal pair.

Sampling-based methods account for these deficiencies by building a reusable roadmap of sampled points from the environment, making it more scalable and efficient for multiple queries. However, this sampling method means that PRM is only asymptotically optimal for a large number of samples. Furthermore, PRM assumes straight-line connections between configurations, making it difficult to incorporate complex dynamics and limiting its effectiveness for systems like our car, which does not necessarily travel between two points in a straight line.

We observed many of these properties through simulation testing. Fig 4 and Table 3.1 show the path length and planning time for both A* and PRM through common environment features: a straight hallway, a sharp turn, and a tight passage. We see this optimality vs. computational complexity tradeoff as A* plans slowly but achieves shorter paths, whereas PRM plans faster but produces less efficient paths. For example, in the "Straight Hallway", the path length for A* is the exact straight line distance between the start and end points (i.e., the optimal path), whereas PRM oscillates around the optimal line.

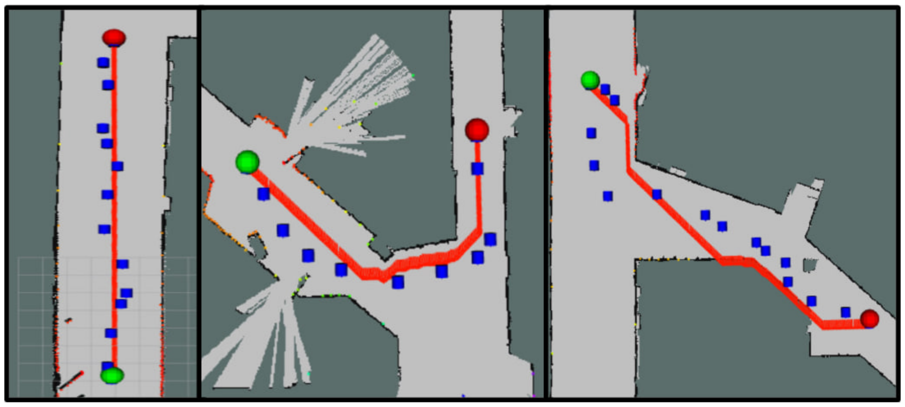

Fig. 4: **Qualitative Comparison of A* vs. PRM.** A* (red) produces smoother paths with less clearance, while PRM (blue) produces oscillating paths with inefficient detours.

	Straight Line		Sharp Turn		Tight Hallway	
	A*	PRM	A*	PRM	A*	PRM
Time	0.88 ± 0.34	0.94 ± 0.08	1.44 ± 0.18	0.99 ± 0.12	1.14 ± 0.05	0.98 ± 0.06
Path Length	19.18 ± 0.11	19.77 ± 0.23	16.53 ± 0.66	17.87 ± 0.86	20.89 ± 0.29	50.41 ± 48.65

Table 1: **Numerical Comparison of A* vs. PRM.** A* has higher planning times and lower path lengths - PRM is the opposite (fast planning, slow paths).

A key limitation of PRM is its lack of consistency between runs. Due to its random sampling, the resulting path can vary significantly, which is what happened between the two runs of Fig. 5. This stochastic behavior made debugging quite difficult, as an error sometimes just wouldn't get reproduced.

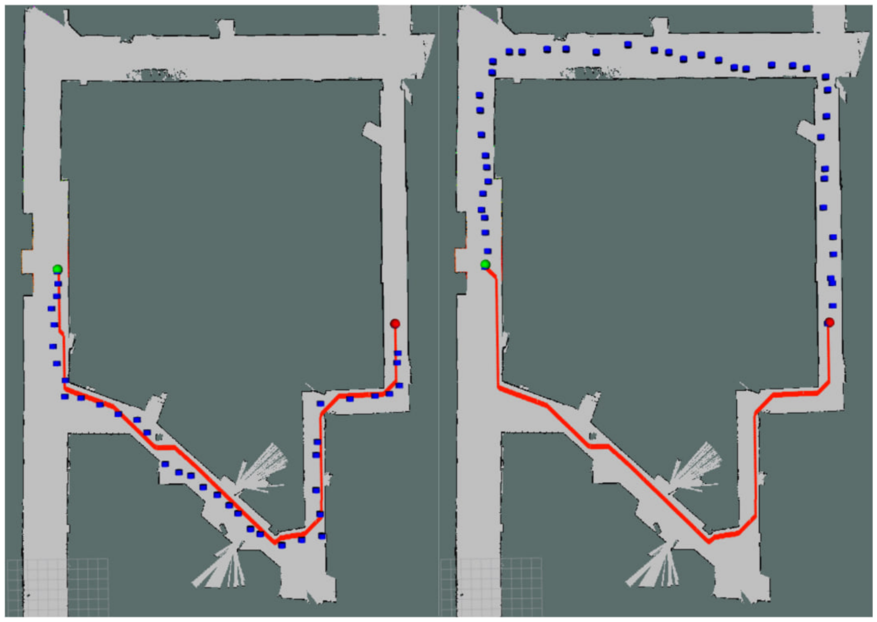

Fig. 5: **Consistency Comparison of A* vs. PRM.** A* (red) produces the same path every time, whereas PRM's probabilistic nature can produce very different paths on different runs.

Furthermore, the jaggedness of PRM paths often made our real robot's trajectory-following more inconsistent, as well as PRM's tendency to swing close to a wall at random. While A* did have a similar issue of skirting close to walls, we were able to mitigate these problems with the dilation methods mentioned in Section 2.4. Overall, A* produced smoother, more consistent, and more optimal paths, leading us to select this method for our on-robot testing.

3.2 Pure Pursuit Evaluation (Simulation) Audrey Lee

The main parameter to tune on our pure pursuit controller was the lookahead distance - shorter lookahead distances made our robot react too slowly, while longer lookahead distances caused the robot to cut corners. We tested several lookahead distances on a path that included both a sharp turn and a long hallway, to try to find a distance that balanced between quick reactions and accurate following. The results of this testing can be seen in Fig. 6, which shows Lookahead Distance = 3.0 m as our optimal value.

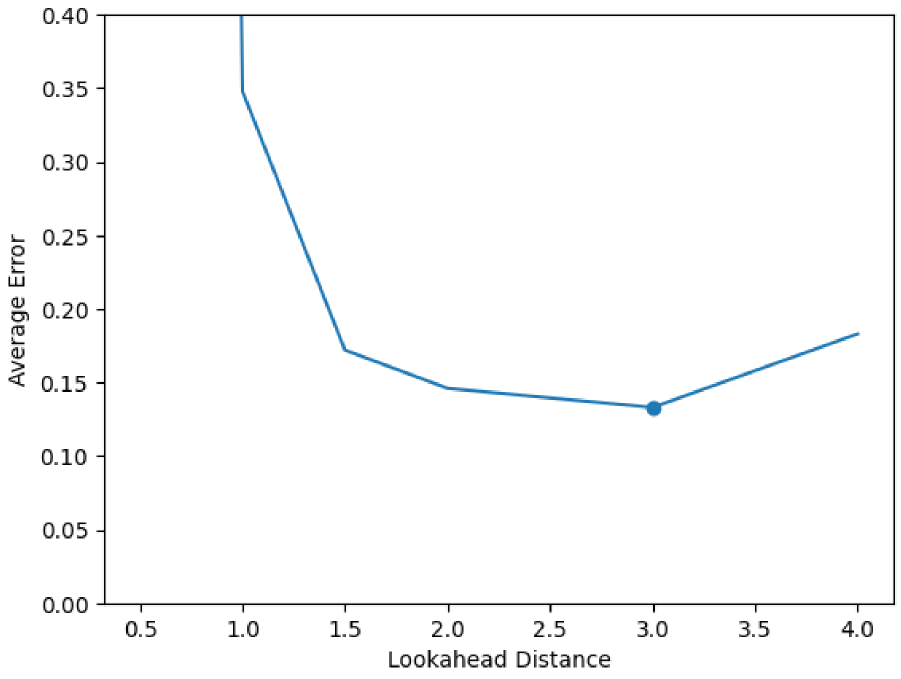

Fig. 6: **Testing various Lookahead Distances**. A lookahead distance of 3.0m gave us the most reasonable reaction time (not too late and not too early).

With this Lookahead Distance, we were able to achieve an average error of 0.13 meters away from the trajectory, which is visualized over time in Fig. 7. This lookahead distance allows us to have very exact tracking on straightaways, but falters and oscillates around turns. This is something that we can certainly continue to work on, perhaps through an adaptable lookahead distance.

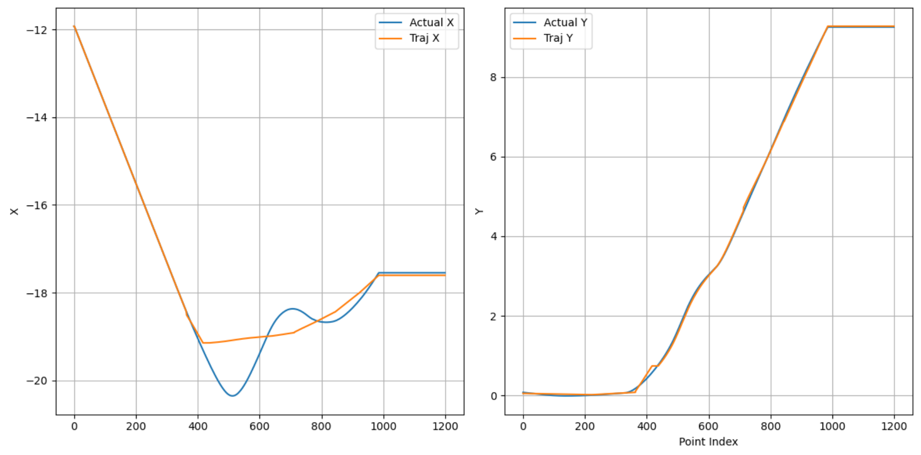

Fig. 7: **Actual (blue) vs. Expected (orange) X and Y**. Straight, smooth sections of the trajectory are followed with a high degree of accuracy, while the robot oscillates when turning.

3.3 On-Robot, Full System Evaluation

Drew Tufto

To validate the integrated localization, planning, and control pipeline on hardware, we deployed our full system in the Stata Center basement. We used our particle filter implementation to maintain the robots pose estimate, and our Pure Pursuit controller to follow the planned path. On the robot, the path was generated using A* search, instead of PRM, as discussed in the previous section. This navigation stack was tested in two scenarios: a straight-line trial and a left-hand turn trial. In both cases, we recorded the cross-track error between the robots estimated pose and the planned trajectory. We attribute most of the error to the controllers inability to perfectly follow the path, but note that some error also arises from the inaccurate pose estimate from our localization node.

Our robot performed well in both cases. Along the straight line, we recorded an average cross-track error of 0.170 m, as shown in Fig. 8.

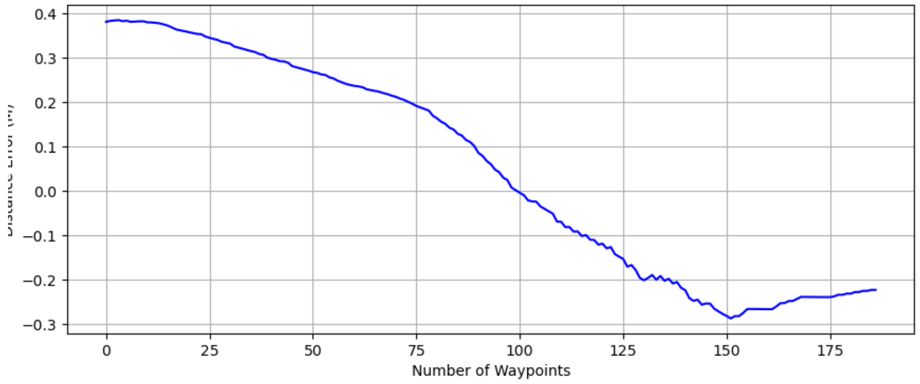

Fig. 8: **Cross-Track Error Between Estimated Pose and Trajectory**

During the left-hand turn, performance degraded slightly, as our average cross-track error increased to 0.318 m With our lookahead distance set to 2 m, the controller cut some turns too tightly, as shown in Fig. 9.

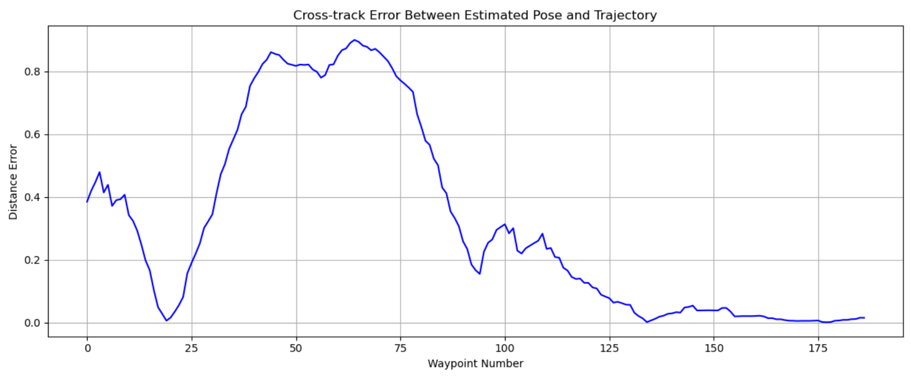

Fig. 9: **Cross-Track Error Between Estimated Pose and Trajectory**

These on-robot results closely match our simulation performance, demonstrating that our navigation system transfers well to real hardware. However, we still experienced issues with following tight turns, which we plan to to compensate against with a redesigned controller, potentially incorporating a more adaptive look ahead parameter.

4 Conclusion

Yibo Cheng

In this lab, we successfully completed the final step of autonomous navigation: motion planning and control. Building upon our earlier perception and localization systems, we implemented a planning pipeline that enables the robot to autonomously navigate the Stata Center basement from any start to any goal location. We explored both search-based (A*) and sampling-based (PRM) planning algorithms, evaluating their respective trade-offs in terms of optimality, runtime, and adaptability to different environments. Based on our evaluation, we selected A* due to its reliable performance and optimal path planning. To follow the planned trajectories, we implemented and tuned a pure pursuit controller, which enabled the robot to execute smooth and accurate navigation in simulation and in the real world.
However, there is still room for improvement. In future iterations, we aim to enhance the robustness of the controller in more complex, dynamic environments, improve real-time planning efficiency, and integrate more advanced planners that better handle vehicle dynamics and uncertainty.

5 Lessons Learned

5.1 Edward Cheng

Throughout this lab, I gained a much deeper understanding of motion planning, particularly the importance of map representation. I realized that the most critical aspect of planning is not the choice of search algorithm, but how the environment is represented. I also explored the strengths and limitations of different planning strategies, including search-based methods like A* and sampling-based approaches such as PRM. A* guarantees an optimal path if one exists, but can be computationally intensive. In contrast, PRM is generally faster but may produce suboptimal paths or fail entirely in complex environments.
From a teamwork and project management perspective, this lab also taught us valuable lessons. We underestimated the overall workload. Initially, each team member completed their assigned component individually, leaving only the integration for the end. We assumed integration would be quick and straightforward, so we didn't start until the day before the final briefing. During this phase, we discovered that one of the components did not function correctly, and resolving the issue required extensive debugging and coordinationultimately keeping us up until 5 a.m. For future projects, especially the final challenge, we plan to begin integration earlier and allocate time more conservatively, even when tasks initially seem simple.

5.2 Audrey Lee

I would say that this lab was definitely one of the biggest challenges that our team has dealt with. There were a lot of moving pieces and a lot of individual components that had to be integrated at the end, and I think that we really fell into some pitfalls where limited documentation made it difficult for two people working in-person were unable to understand another non-present team member's code and build upon it. I think this lab really highlighted the need for clear documentation and communication, especially in letting people know what progress you made before going offline.
On the technical side, a lot of this lab was actually quite familiar to me, as I have a lot of prior experience with path planning and trajectory following, so I really wanted to take a step back and let some of my teammates handle more of the basic component implementation. I think this beneficial for all parties as I was able to have more of a hand in stitching these components together, which is something that I hadnt necessarily done through past classes and projects. I think that making use of visualizers and plotting results became super useful in this lab in helping us figure out which component was causing the most issues, thereby allowing us to really have targeted debugging. I think this is something that we will definitely continue to use for the final challenge, and I am glad that we were able to take advantage of these strategies for this lab.

5.3 Christopher Liem

Through this lab, I learned about the intricacies of path and motion planning. My main task in this lab was to implement A^*, which could be considered as an extension to the famous Dijkstra's Algorithm. While implementing A^*, I learned about the importance of choosing a good heuristic in order to guarantee optimality, both in terms of yielding the shortest-path and achieving low runtimes. The evaluation between the Manhattan and Euclidean heuristics was interesting and it makes me wonder what other heuristics we could've used for our algorithm. It's natural to think of only using Euclidean distance for our algorithm because it's what we are accustomed to using, but Manhattan distance turned out to be optimal. I learned about A^* during my microinternship last IAP, but this lab solidified my understanding of the algorithm.

From a collaboration and teamwork perspective, we need to check in with each other's implementations to make sure they are fully functional in a reasonable time. We completed our individual tasks, but we realized that one of our components wasn't working when we tried to integrate them all together. This led to us bugfixing for three hours, inevitably redoing the entire implementation before we got everything to work on the real robot. We stayed up until 5AM working on this lab, which wasn't practical. We will work on checking in with each other's implementations for the final project.

5.4 Drew Tufto

I don't think this lab was any more of a technical challenge to our team than any of the other labs. However, the complexity of integrating each of the components was certainly more than we anticipated. We went into the integration step with very solid planning, control, and localization algorithms respectively, but quickly realized that this was not the same as having one very solid navigation algorithm.
I believe a lot of the issues we ran into during this step could be mitigated by a more strategic division of work,

5.5 Richard Yeboah

I worked mostly on the python implementation of the pure pursuit algorithm, which technically, was not tough, but I had many oversights with my code. Planning out the code, optimization should have been part of my priorities, but my initial program ran incredibly slowly, pre-numpyization.
A large lesson that I learned throughout this lab was leaning the best way to combine my conceptual understanding of the pure pursuit algorithm with the implementation and practical testing.