Autonomous Exploration of Unknown Environments

Gavin Hua
California Institute of Technology
Pasadena, CA
ghua@caltech.edu

Gio Huh
California Institute of Technology
Pasadena, CA
ghuh@caltech.edu

Taekyung Lee
California Institute of Technology
Pasadena, CA
tlee2@caltech.edu

***Abstract*—This report describes an integrated system for autonomous exploration of unknown environments using simultaneous localization and mapping (SLAM) with an exploration-oriented planning strategy. Our approach combines a particle-filter-based localization algorithm with occupancy grid mapping and an A$*$ based planner driven by an entropy metric. The system is validated within a physics-based simulation environment and demonstrates robust performance despite sensor noise and dynamic challenges. This report explains our methodology, implementation details, design trade-offs, and lessons learned.**

***Index Terms*—SLAM, Robotics**

I. Introduction

The primary goal of this project is to enable a UAV (modeled by its state $x = \left(p_x, p_y, \dot{p}_x, \dot{p}_y\right)^\top$) to autonomously explore unknown 2D environments while building an accurate map and localizing itself in real time. The map is composed of a rectangular boundary, polygonal obstacles, and beacons representing room/terrain features used in localization. Key challenges included managing sensor noise, ensuring robust localization in cluttered environments, and generating safe, exploration-driven paths. To meet these challenges, our system integrates several core components:

- **Localization**: A particle-filter approach that leverages room feature ("beacon") measurements.
- **Mapping:** An occupancy grid framework using log-odds updates and ray-tracing to incrementally build the environment's map.
- **Planning:** An A$*$ based planner augmented with an entropy map to select informative goal points.
- **Simulation & Control:** A physics simulation node models robot dynamics and sensor noise, while separate control and teleoperation nodes provide both autonomous and manual command inputs.

This modular design facilitates testing of individual components and allows for straightforward integration of additional sensors or planning strategies.

II. Notation

For multi-dimensional Gaussian distributions, we overload the notation $x \sim N(p, \sigma^2)$ where p, σ are scalars to denote $x \sim \mathcal{N}\left(\begin{pmatrix} p \\ \vdots \\ p \end{pmatrix}, \mathrm{diag}(\sigma^2, ..., \sigma^2)\right)$. Unless denoted otherwise, all positions are in the world frame.

III. Approach

A. Simultaneous Localization and Mapping

Our SLAM implementation is a variant of the particle filter. This enables us to incorporate non-Gaussian distributions that commonly arise in distributions of the robot's position. However, the room features' positions (p_x^i, p_y^i) are determined with Kalman filter updates (details in the next section), since sensor noise is assumed to be Gaussian.

Compared to applying an Kalman filter to the full state consisting of the robot's position and all known features' positions, our hybrid method does not assume the form of the distribution of x. Moreover, it does not require the full covariance matrix (nor its inverse, which is computationally expensive to acquire), nor does it require dynamically resizing the state vector as more beacons are discovered.

Compared to applying a particle filter to track beacon positions, performing Kalman updates allows us to leverage key assumptions (white noise on sensors) to optimize mapping with minimal computation.

B. Autonomous Planning and Trajectory Tracking

We deploy an information-theory-informed planner to maximize exploration efficiency. Our system autonomously identifies information-rich areas, generates collision-free paths using the A$*$ algorithm, and precisely tracks trajectories to explore unknown environments. The A$*$ algorithm provides optimal path planning by efficiently searching the configuration space while avoiding obstacles. The entropy-gradient method effectively resolves the exploration problem by identifying boundaries between known and unknown regions, rejecting areas with uniformly high entropy (inaccessible regions) and uniformly low entropy (already explored regions). A PD controller with waypoint tracking provides robust trajectory following that effectively rejects process noise in our simulation.

The entropy-gradient method effectively resolves the exploration problem as it rejects regions with uniformly high entropy (unknown and inaccessible to planner) and uniformly low entropy (known regions). A simple PD controller suffices to reject process noise in our simulation. The following flowchart provides a high-level overview. Technical details are discussed in the following section.

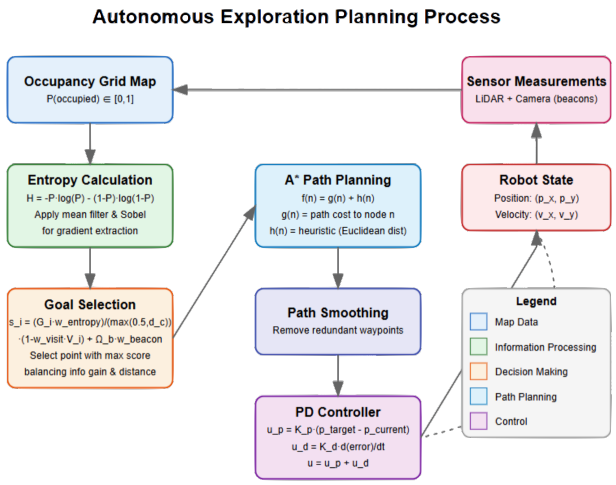

Fig. 1: Autonomous Exploration Planning Information Flowchart

IV. Implementation

A. ROS Node Structure

We utilize RVIZ for visualization.

B. Physics Simulation

We assume the controller has first-order control of the plant. Every time step, given the control input u, the physics simulation performs an Euler integration of the form

$$\begin{pmatrix} p \\ \dot{p} \end{pmatrix} = \Delta t \begin{pmatrix} 0 & I \\ 0 & 0 \end{pmatrix} \begin{pmatrix} p \\ \dot{p} \end{pmatrix} + \begin{pmatrix} 0 \\ u + w \end{pmatrix} \tag{1}$$

where $w \sim \mathcal{N}(0, \sigma_c^2)$ represents the controller noise.

If a collision were to occur, the line from p_t to p_{t+1} is traced and the robot is placed on the edge of the obstacle.

Moreover, the physics simulation computes the sensor measurements. The LiDAR sensor is modeled by ray-casting, which returns the distance to the nearest obstacle in the direction of the ray if the obstacle is within $r_{\max}$. The camera sensor is modeled by a Gaussian distribution with mean p^i and covariance $\sigma_b^2 I$, where p^i is the position of the ith beacon. The camera sensor is able to detec.t beacons if the line of sight is not obstructed.

All geometry functionality is provided by the `Shapely` library

C. Mapping

We maintain three data structures for mapping: the occupancy grid, the list of beacon positions, and the list of beacon position covariances.

We perform a log-odds ratio update for the occupancy grid. The occupancy grid is represented by a 2D array of cells, each with a log-odds value theoretically from $(-\infty, +\infty)$. In practice, however, clipping the values to $[-10, 10]$ improved numerical stability. The occupancy grid is initialized to 0 for all cells. The log-odds update is performed as follows:

- for every ray cast by the LiDAR sensor, we perform Bresenham's line algorithm to find the cells that the ray intersects.
- for every cell before the ray's endpoint, we add L_{FREE} to the log-odds value.
- if the ray intersects an obstacle, we add L_{OCCUPIED} to the log-odds value of the cell at the ray's endpoint. Otherwise, we add L_{FREE} to the log-odds value of the cell at the ray's endpoint.

Given a log-odds ratio, the occupancy probability is computed as $p = \frac{1}{1+e^{-l}}$.

We introduce two different approaches to updating the beacons: Particle-Based Updating and Kalman Filter Updating.

For the particle-based updating, we represent each beacon as $B = \{p_1, p_2, ..., p_n\}$ where each $p_i \in \mathbb{R}^3$ with the z-axis set to 0. Intuitively, each beacon is then represented as a cluster of particles computed from measurements across all cycles to the present. From this, we estimate the position of each beacon by $\hat{\mu} = \frac{1}{n}\sum_{\{i=1\}}^{n} p_i$ with a covariance measured by $\hat{\Sigma} = \frac{1}{n}\sum_{\{i=1\}}^{n}(p_i - \hat{\mu})(p_i - \hat{\mu})^T$. Notice through this representation we are able to handle non-Gaussian, non-parametric distributions in contrast to the Kalman Filter update. At each cycle, we map each cluster to a beacon already on the map or determine it to be a new landmark through a voting mechanism. This is shown in the following steps:

Algorithm: Particle-Based Beacon Association and Update

Input: Cluster $C = \{c_1, c_2, ..., c_m\}$, Existing beacons $B = \{B_1, B_2, ..., B_n\}$

Output: Updated beacon set

For each cluster C:

1) Calculate votes for each beacon:
   $\text{votes}(j) = \sum_{i=1}^{m} \mathbb{1}[\text{closest}(c_i) = j]$
2) Find best matching beacon:
   $\text{match} = \text{argmax}_j \, \text{votes}(j)$
3) **If** match is found with sufficient votes:
   a. Update beacon with new points:
   $B_{j'} = B_j \cup C$
   b. Recalculate mean:
   $\hat{\mu}_{j'} = \frac{1}{|B_{j'}|}\sum_{p \in B_{j'}} p$
   c. Recalculate covariance:
   $\hat{\Sigma}_{j'} = \frac{1}{|B_{j'}|}\sum_{p \in B_{j'}}(p - \hat{\mu}_{j'})(p - \hat{\mu}_{j'})^T$
4) **Else** create new beacon:
   a. Initialize new beacon:
   $B_{\text{new}} = C$
   b. Calculate mean:
   $\hat{\mu}_{\text{new}} = \frac{1}{|C|}\sum_{p \in C} p$
   c. Calculate covariance:
   $\hat{\Sigma}_{\text{new}} = \frac{1}{|C|}\sum_{p \in C}(p - \hat{\mu}_{\text{new}})(p - \hat{\mu}_{\text{new}})^T$

Note that each particle cluster maintains a coherent structure representing a single beacon. This property enables efficient classification of measurement groups as either existing or new beacons, leveraging preprocessing from the SLAM localization step.

Alternatively, we can perform and validate a Kalman update for the beacons. If an observed beacon is not within r_b of a known beacon, then its observed position is added to the list of beacons. Its corresponding covariance is set to the covariance

of the point cloud since the camera's covariance is unknown. Otherwise, given known position p_1, known covariance P_1 and observed position p_2, point cloud covariance P_2, we perform the Kalman update:

$$\begin{aligned} P &= \left(P_1^{-1} + P_2^{-1}\right)^{-1} \\ p &= P(P_1^{-1}p_1 + P_2^{-1}p_2) \end{aligned} \tag{2}$$

This optimally fuses the two measurements. The algorithm is explain more in detail in the following section.

Algorithm: Kalman Filter Beacon Association and Update

Input: Observed beacon position p_2, Point cloud covariance P_2, Set of known beacons with positions $\{p_1^i\}$ and covariances $\{P_1^i\}$

Output: Updated beacon set

For each observed beacon (p_2, P_2):

1) Find closest existing beacon: Compute distance $d_i = \|p_1^i - p_2\|$ for all known beacons
2) **If** $\min(d_i) \leq r_b$ (where r_b is the association threshold):
   a. Select the closest beacon (p_1, P_1)
   b. Compute the information matrices: $\Omega_1 = P_1^{-1}$ $\Omega_2 = P_2^{-1}$
   c. Update the beacon's covariance: $P = (\Omega_1 + \Omega_2)^{-1}$
   d. Update the beacon's position: $p = P(\Omega_1 p_1 + \Omega_2 p_2)$
3) **Else** create new beacon:
   a. Add new beacon position: $p_{\text{new}} = p_2$
   b. Set new beacon covariance: $P_{\text{new}} = P_2$

D. Localization (Particle Filter)

The particle filter maintains $N = 1000$ particles to represent the belief of the robot's position. Each particle is represented by its position p_p^i, and they are initialized by sampling $\mathcal{N}(0, \sigma_p^2)$. At each time step, the particle filter performs the following steps:

- Integrate all particles forward in time using Euler's method.
- Add noise $w_p^i \sim \mathcal{N}(0, \sigma_p^2)$ to each particle.
- For the set of observed beacons p_b^o for $o \in [1, m]$ and the corresponding closest known beacons p_{b*}^o compute the score s_i, where

$$s_i = \sum_{o=1}^{m} \frac{1}{\|p_{b*}^o - p_b^o\|} \tag{3}$$

- If any observation does not have a corresponding beacon within r_b, then $s^i = -\infty$.
- Resample the particles according to the $P_i = \frac{e^{s_i}}{\sum_i e^{s_i}}$ (the softmax distribution) with replacement.

Observe that p_b^o is based on particle-based updating, which would be the mean associative B_i, or based on Kalman-filter update, which would be p_1^i.

E. Entropy-Based Goal selection

To maximize autonomous exploration efficiency, we select points where the maximum information gain is expected. Our approach computes the information density **gradient**, representing boundaries between known and unknown regions, and navigates to the most promising locations. The mathematical foundation of our entropy-based exploration is outlined below:

Algorithm: Entropy-Informed Exploration Goal Selection with Visit Memory

Input: Occupancy probability grid P, expected robot position p, set of known beacons $\{p_b^i\}_{i=1}^B$, visited regions grid V

Output: Exploration target

1) compute the information entropy grid element-wise $H = -P\ln(P) - (1-P)\ln(1-P)$
2) apply mean filtering to remove noise artifacts. $H_{\text{av}} = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} * H$
3) apply a Sobel filter to extract gradients

$$\begin{aligned} G_x &= \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} * H_{\text{av}} \\ G_y &= \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * H_{\text{av}} \end{aligned} \tag{4}$$

4) compute gradient magnitude grid $G = \sqrt{G_x^2 + G_y^2}$ and normalize to [0,1]
5) update visited regions grid:
   - decay existing values: $V = V \cdot \gamma$ where $\gamma = 0.995$
   - mark current robot position and surrounding area as visited
6) initialize best_score = $-\inf$, best_cell = null

For a subset of cells with high gradient magnitude G:

1) if cell is occupied ($P > 0.7$), skip to next cell
2) compute cell's world position p_c
3) compute distance to robot $d_c = \|p_c - p\|$
4) if $d_c < 0.5$ or $d_c > 12.0$, skip to next cell
5) compute beacon information $\Omega_b = \sum_{i=1}^{B} \frac{1}{\|p_b^i - p_c\|}$
6) compute cell score with visit penalty $s_i = \frac{G_i \cdot w_{entropy}}{\max(0.5, d_c)} \cdot (1 - w_{visit} \cdot V_i) + \Omega_b \cdot w_{beacon}$

Return $\arg\max_i s_i$

Since further points require more control effort and time to reach, we discount information gradient by d_c to incentivize exploring closer points. The penalty for previously visited regions V_i ensures the robot explores new areas rather than revisiting familiar territory. Since beacons provide localization in our setting, we to reward staying close to beacons (to ensure effective localization), which gives rise to the $\Omega_b \cdot w_{beacon}$ term. w_{beacon} is a tunable parameter that weights the importance of good localization. Additionally, $w_{entropy}$ (= 1.0) controls how much importance is given to information-rich

boundaries and w_{visit} (= 0.8) determines how strongly previously visited regions are penalized. This prevents the robot from repeatedly exploring the same areas, even if they have high information content.

F. A∗ Path Planning

Given the exploration target from our entropy-based goal selection, we use the A∗ algorithm to generate an optimal collision-free path. A∗ combines the completeness of Dijkstra's algorithm with the efficiency of heuristic-guided search, making it well-suited for our grid-based environment. The algorithm was implemented as the following:

Algorithm: A∗ for Optimal Path Planning
Input: Occupancy probability grid P, current position p_{start}, goal position p_{goal}, grid origin o_g, resolution r
Output: Path from start to goal, or null if no path found

1) convert start and goal to grid coordinates
2) initialize:
   - open_set = priority queue with start node (priority = heuristic(p_{start}, p_{goal}))
   - closed_set = empty set
   - g_score[start] = 0
   - f_score[start] = heuristic(start, goal)
   - came_from = empty map
3) **While** open_set is not empty:
   - current = node with lowest f_score in open_set
   - if current = goal:
     1) path = reconstruct_path(came_from, current)
     2) return smooth_path(path)
   - remove current from open_set
   - add current to closed_set
   - **For** each neighbor of current:
     1) if neighbor in closed_set or is occupied (P[neighbor] > 0.5): **Continue**
     2) tentative_g_score = g_score[current] + movement_cost(current, neighbor)
     3) if neighbor not in open_set or tentative_g_score < g_score[neighbor]:
        a) came_from[neighbor] = current
        b) g_score[neighbor] = tentative_g_score
        c) f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal)
        d) if neighbor not in open_set: add neighbor to open_set

Return null (no path found)

Here, for the heuristic function, we use the Euclidean distance. The movement cost is 1.0 for cardinal directions (horizontal/ vertical) and 1.414 for diagonal movements when diagonal movement is enabled. After finding a path, we apply path smoothing to reduce unnecessary waypoints while maintaining safety. The smoothing process significantly reduces the number of waypoints while ensuring the path remains collision-free, leading to more efficient and natural robot motion.

G. PD Controller

After the A∗ algorithm generates a collision-free path from the robot's current position to the goal, a proportional-derivative (PD) controller computes the sequence of control inputs to make the robot follow the planned path. The controller generates velocity commands based on position errors and their rates of change.The control input u is obtained from the combinations of two terms - the proportional term (P) and the derivative term (D). The proportional term generates a control input proportional to the current error, which is the difference between the target position p_{target} and the robot's current position $p_{current}$:

$$u_p = K_p \cdot \left(p_{target} - p_{current}\right) \tag{5}$$

where K_p is the proportional gain. Since a higher K_p results in faster corrections but risks overshooting, the nominal gain value was chosen to be 1.0.

The derivative term dampens oscillations by responding to the rate of error change, preventing abrupt movements and smoothing the robot's motion:

$$u_d = K_d \cdot \frac{\mathrm{d}}{\mathrm{d}t}\left(p_{target} - p_{current}\right) \tag{6}$$

where K_d is the derivative gain. The nominal gain value was chosen as 0.1. The total control input is the sum between the two terms, yielding $u = u_p + u_d$.

Based on this design, the controller is implemented as the following:

Algorithm: PD Control for Path Following
Input: Path waypoints $[w_1, w_2, ..., w_n]$, current position p, previous error e_{prev}, time step dt
Output: Control command $u = \left[v_x, v_y\right]$

1) if path is empty or all waypoints reached: **Return** [0, 0]
2) target = current waypoint from path
3) compute position error: e = target $-p$
4) compute error magnitude: dist_to_target = $\|e\|$
5) if dist_to_target < waypoint_threshold:
   1) advance to next waypoint
   2) if no more waypoints: **Return** [0, 0]
   3) update target and recompute e
6) compute error derivative: $\dot{\mathrm{e}} = \frac{e - e_{prev}}{dt}$
7) compute control command: $u = K_p \cdot e + K_d \cdot \dot{\mathrm{e}}$
8) limit control magnitude:
   - if $\|u\| >$ max_speed: u = max_speed $\cdot \frac{u}{\|u\|}$
9) update previous error: $e_{prev} = e$

Return u

Combining all, the following (Fig 1) is an image of how the robot operates:

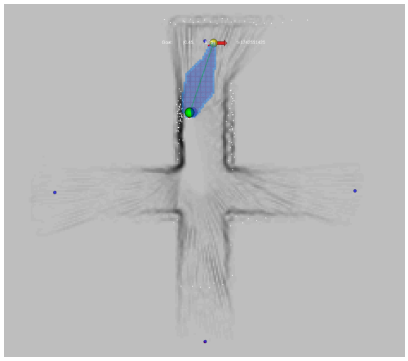

Fig. 2: The Operation of the Entropy-Informed A∗ Algorithm

V. Evaluation

We perform a systematic ablation study across key components of autonomous SLAM study, through the following experiments:

A. *SLAM versus Naive Odometry Localization and Mapping*

To demonstrate the effects of our SLAM algorithm, we compare localization errors over a round trip around the top-left obstacle. The blue sphere represents the naive odometry, the green sphere the ground truth position, and the red sphere is the position estimate returned by the SLAM.

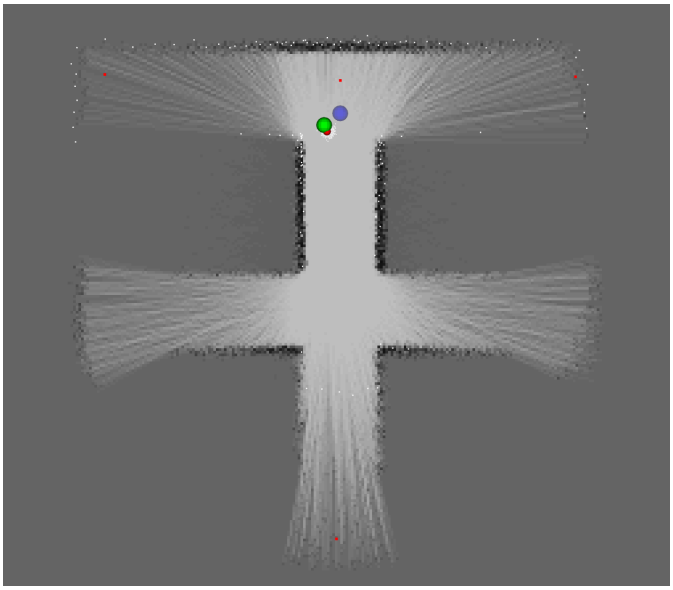

Fig. 3: This figure shows the components of the basic system: the ground truth position (green), estimated position (red), integrated position (blue), lidar points (white), and particle filter's particles (white around red)

The error is defined as distance between the localization's result and the ground truth robot position, i.e., $e(t) = \|p_{\text{loc}(t)} - p(t)\|$. Error accumulation is shown below.

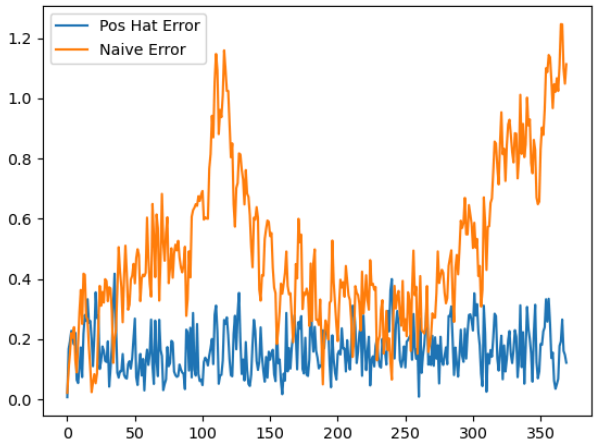

Fig. 4: Error comparison between naive localization and particle filter

As shown, the naive localization's error continues to grow. This can be attributed to the fact that at every time step noise is injected into the system; theoretical results show that the total error scales with $\sqrt{N}$, where N is the number of time steps. However, the SLAM algorithm tracks the ground truth position much better.

We note that this path was chosen to be collision-free. Since the naive localization is a simple integrator, it is not aware of the obstacles, nor collisions. Therefore, the naive localization will have a much greater (possibly unbounded as $t \to \infty$) error if the robot continues to drive against an obstacle.

Qualitatively, we may also make a "round trip" around the map by tracking the outer walls, and compare the maps generated by the two localization methods.

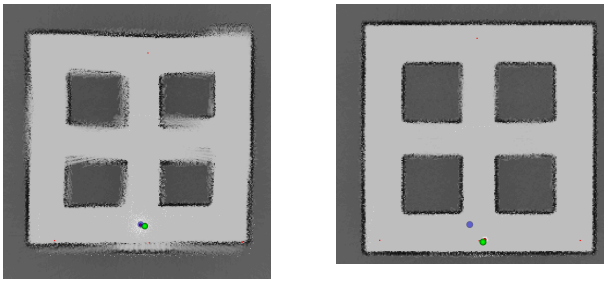

Fig. 5: Screenshots of the maps created by two localization methods. The left figure is created by naive integration and the right is created by particle-filtering.

It is obvious that without SLAM, the robot's predicted position gradually drifts away from the ground truth, and the map becomes extremely distorted. In comparison, the SLAM-generated map's roughness originates only from sensor measurement noise, and the shape of the map remains close to ground truth.

In summary, it can be shown that for the noise levels in our simulation, SLAM is the critical foundation for higher-level functionality such as automatic exploration and beacon position determination.

B. Particle-based versus Kalman filter-based beacon mapping

To compare the particle-based beacon mapping with the Kalman filter-based mapping, we construct the map of the following form. Notice that we constructed the map to be relatively simple with multiple beacons in order to demonstrate the ability of the methods to handle the multiple beacons across successive measurements.

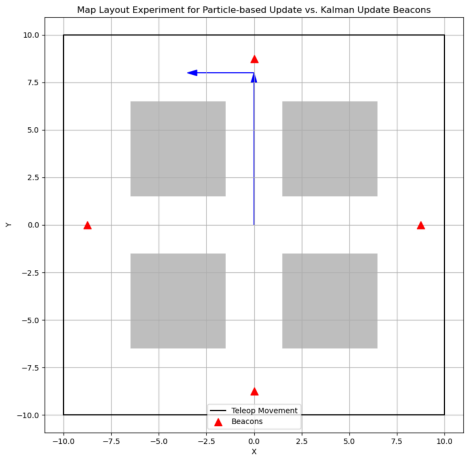

Fig. 6: Experimental map to compare the performance between particle-based beacon mapping vs. Kalman filter-based.

We run 2 experiments, one for particle-based beacon mapping and Kalman filter-based, where for each cycle, we compute the MSE w.r.t. the estimated beacon from the two methods. That is,

$$\mathrm{SE}\left(\hat{b}\right) = \min_{b_i \in \mathcal{B}} \|\hat{b} - b_i\|^2 \tag{7}$$

$$\mathrm{MSE}_{\mathrm{particle}} = \frac{1}{m} \sum_{i=1}^{m} \mathrm{SE}(\mu_i) \tag{8}$$

$$\mathrm{MSE}_{\mathrm{kalman}} = \frac{1}{l} \sum_{i=1}^{m} \mathrm{SE}(p_i) \tag{9}$$

where m is the number of beacons in the mapping of the robot.

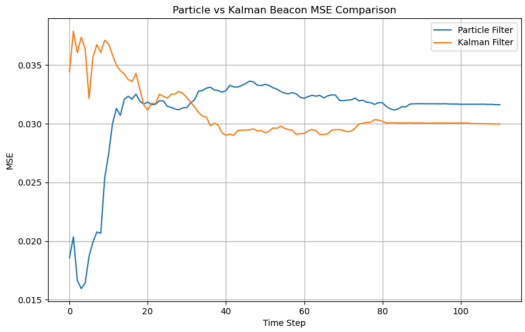

Fig. 7: Experimental map to compare the performance between particle-based beacon mapping vs. Kalman filter-based.

Time Period	Metric	Particle Filter	Kalman Filter
Overall	MSE	**0.0309**	0.0310
	Std Dev	0.0037	**0.0022**
After Stable (>20 steps)	MSE	0.0321	**0.0301**
	Std Dev	**0.0006**	0.0009
Before Stable (≤20 steps)	MSE	**0.0253**	0.0351
	Std Dev	0.0063	**0.0017**

In the early time cycles, we notice that both the particle file and Kalman Filter fluctuate significantly in MSE as they are both unsure about their mapping. However, after around 20 cycles, as the number of measurements accumulates for both methods, both become sure that the locations of the beacons converge, as seen in Fig 3. Interestingly, in the early cycles, the particle filter outperforms the Kalman filter; this could be due to an information bottleneck and the empirical particle clouds being able to capture of the non-linear distributions. As more measurements are found, due to the simiplicity of the map constructed, the estimation of the the beacons resembles that of a Gaussian distribution. This is further validated by the RVIZ visualization during experimentation.

Another interesting observation can be seen in the table. We see that when one method has a lower MSE, the other has a lower standard deviation. This would indicate a tradeoff between the two methods between the mean and variance of the MSE.

In general, this initial comparison shows that one method of beacon mapping does not necessarily outperform the other on simple map cases, as exemplified in Fig 5. However, it reveals complementary strengths between the two. Potentially, when measurements are few and far between the particle filter as shown in early cycles is preferable. With more data that resembles a Gaussian distribution, Kalman Filters may fare better, as they converge to a slightly lower MSE in Fig 6. Further testing can be conducted on more complex mappings that capitalize on the flexibility of the particle filter.

C. Human Teleoperation against Autonomous Exploration Models (over Entropy Sum)

Comparing entropy reduction characteristics between human teleoperation and our autonomous exploration implementation reveals distinct patterns in information acquisition strategies. Human teleoperation (Fig 7) demonstrates a steady entropy reduction profile over its 220-second runtime. The algorithm begins with an immediate steep entropy decrease and maintains a relatively consistent reduction rate with several subtle changes in gradient throughout the mission.

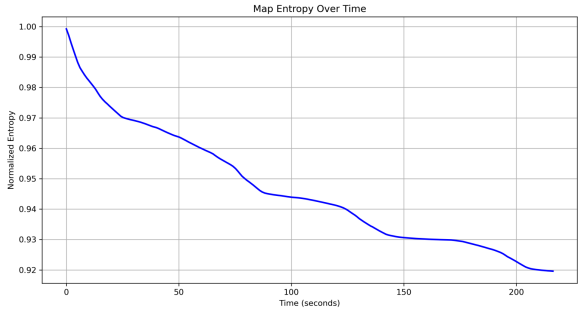

Fig. 8: Map Entropy Reduction over Time for the Human Controller

Our autonomous entropy-based exploration algorithm (Fig 8) similarly shows an immediate entropy reduction from mission start, but achieves a more significant total reduction (0.12 units versus 0.08 units) over a comparable timeframe (250 seconds versus 220 seconds). This translates to approximately 38% greater exploration efficiency for the algorithm when measured by total entropy reduction. The algorithm's entropy curve maintains a more consistent gradient throughout most of the exploration period, only beginning to level off after t≈200s.

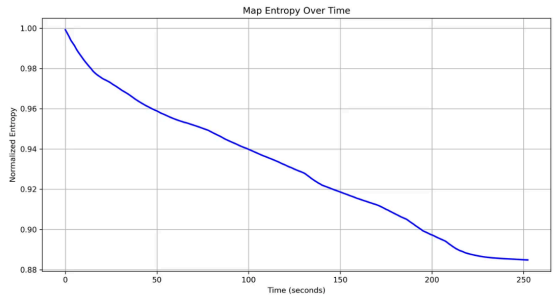

Fig. 9: Map Entropy Reduction over Time for the Autonomous Entropy-Based Exploration Algorithm

While the human teleoperator ultimately reaches a normalized entropy of 0.92, our autonomous algorithmachieves a notably lower final entropy of 0.88, representing a 50% improvement in overall uncertainty reduction.

D. Proportional-derivative Control vs. Open-loop

As an experimental evaluation, we compared the performance of proportional-derivative (PD) control against open-loop control for autonomous exploration tasks. The controllers demonstrated similar path tracking accuracies, but the PD controller tended to achieve a mean cross-track error of 3.13 cm, while the pen-loop controller exhibited a comparable 3.21 cm mean error.

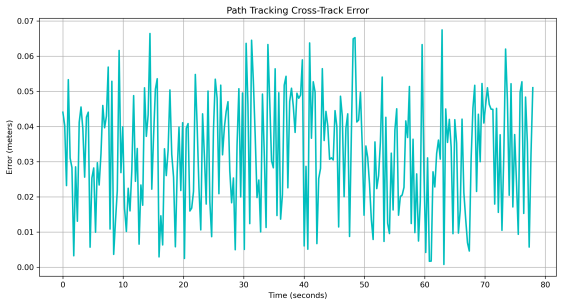

Fig. 10: Cross-Track Error Distribution for Open-Loop Controller.

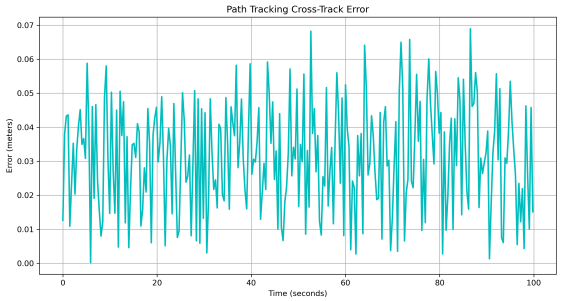

Fig. 11: Cross-Track Error Over Time for PD Controller.

However, analysis of the cross-track error profiles reveals similar behavioral patterns between both controllers. The open-loop controller maintained tracking errors primarily between 0-6.7 cm across its 77.93-second mission duration, while the PD controller exhibited errors in the 0-7.0 cm range over a longer 99.79-second mission. The error distribution histograms for both controllers show remarkably similar profiles, with identical mean and median values within each controller's data (3.1 cm for PD and 3.2 cm for open-loop). This suggests that despite their fundamentally different approaches to control, both systems achieved comparable path-following precision in our structured exploration environment.

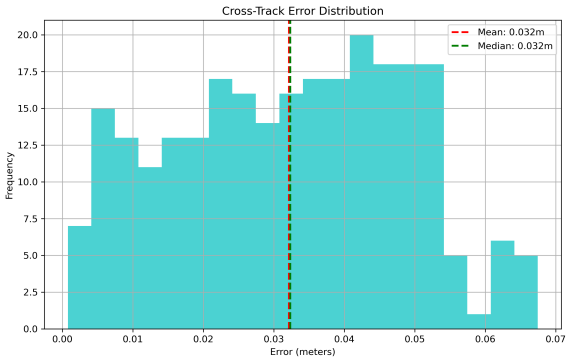

Fig. 12: Cross-Track Error Over Time for Open-Loop Controller.

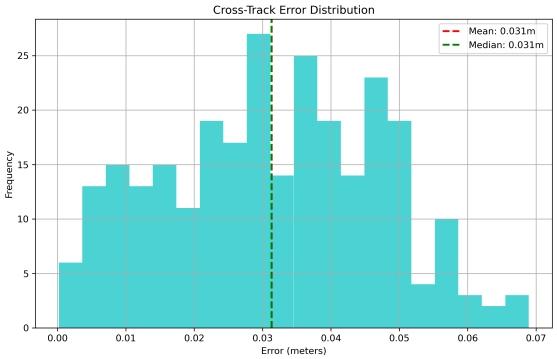

Fig. 13: Cross-Track Error Distribution for PD Controller.

VI. Conclusion

In summary, our integrated approach demonstrates that robust autonomous exploration in unknown environments critically hinges on effective SLAM techniques. The fusion of particle filter-based localization with occupancy grid mapping, augmented by an entropy-driven exploration strategy and rapid trajectory planning (via RRT) and control (via a PD controller), significantly mitigates the challenges posed by sensor noise and dynamic uncertainties.

Key takeaways include:

- Importance of SLAM: Compared to naive odometry, SLAM effectively limits error accumulation, ensuring that the robot's estimated position remains close to the ground truth, even in cluttered or collision-prone scenarios.
- Complementary Mapping Approaches: The comparative analysis between particle-based and Kalman filter-based beacon mapping reveals that while the particle filter can initially better capture the non-Gaussian, non-linear uncertainties in early measurements, the Kalman filter converges to a slightly lower mean squared error as more Gaussian-like data accumulates. This suggests that a hybrid or adaptive strategy might offer further performance improvements.
- Enhanced Exploration Efficiency: The autonomous exploration algorithm demonstrated a significant advantage in entropy reduction compared to human teleoperation. By consistently selecting high-information-gain waypoints, the algorithm achieved approximately 38% greater exploration efficiency within a similar timeframe. This result highlights the effectiveness of entropy-based exploration strategies in rapidly reducing map uncertainty.
- Modularity and Scalability: The modular system architecture not only facilitates independent testing and optimization of each component but also allows for easy integration of additional sensors or alternative planning methods. This modularity is crucial for adapting the system to a wide range of real-world applications and environments.

VII. Appendix

A. Code Appendix

For the full code base, see Github.

B. A $*$ algorithm v.s. Rapidly-exploring Random Tree (RRT) algorithm

We also employed the Rapidly-exploring Random Tree (RRT) algorithm, which efficiently handles complex environments and non-holonomic constraints for generating a collision free path. The implementation is like the following:

Algorithm: RRT for Exploration Planning
Input: Occupancy grid P, start position p_{start}, goal position p_{goal}, grid origin o_g, resolution r, max iterations N
Output: Path from start to goal, or null if no path found

1) Initialize tree T with root node at p_{start}
2) For $i = 1$ to N:
   a. With probability 0.1, set p_{rand} = p_{goal} (goal biasing) Otherwise, randomly sample p_{rand} from the environment
   b. Find nearest node p_{near} in T to p_{rand}
   c. Compute new point "p_{new}" by stepping from p_{near} toward p_{rand} by step_size
   d. If CollisionFree(p_{near}, p_{new}, P, o_g, r):
   i. Add p_{new} to T with parent p_{near} ii. If $\|p_{new}$ - $p_{goal}\|$ < threshold:
   - Add p_{goal} to T with parent p_{new}
   - Extract path by traversing T from p_{goal} to p_{start}
   - **Return** reversed path

Return null (no path found within iteration limit)

The CollisionFree function uses interpolation to check if the path segment between two points intersects obstacles:

Function: CollisionFree(p_1, p_2, P, o_g, r)

1) Convert P to binary occupancy map M where $M(i, j) = 1$ if $P(i, j)$ > threshold
2) Interpolate n points $\{q_k\}$ between p_1 and p_2
3) For each point q_k: a. Convert q_k to grid coordinates (i, j) = world_to_grid(q_k, o_g, r) b. If $M(i, j) = 1$, return false
4) **Return** true

The reason why we didn't use RRT in the final implementation was that it was too computationally intensive and sensitive to the occupancy grid that even if it found a path, the post checking of whether it is collision-free denied that it was a possible path, making it go through numerous runs of path generation.