GP-guided MPPI for Efficient Navigation in Complex Unknown Cluttered Environments

Mayank Goud Polagoni
College of Engineering
Northeastern University
Boston, USA
polagoni.m@northeastern.edu

***Abstract*— This report presents a Gaussian Process-guided Model Predictive Path Integral (GP-MPPI) controller for autonomous mobile robot navigation in unknown, cluttered environments. The GP-MPPI framework integrates a sampling-based Model Predictive Control approach with an online Gaussian Process (GP) mapping module to provide global guidance without requiring a prior map. A vanilla MPPI controller is implemented as a baseline, and a GP-based local perception model (inspired by *GP-MPPI* of Mohamed *et al.*) is used to recommend subgoals for the MPPI planner [1]. We evaluate both MPPI and GP-MPPI in simulation across three progressively challenging environments (simple, moderate, and complex obstacle courses. Experimental results demonstrate that the GP-guided planner significantly improves navigation performance, successfully avoiding collisions and escaping local-minima traps that hinder the vanilla MPPI [1]. The GP-MPPI achieves safer and more reliable goal-reaching at the cost of a modest increase in computation. Key metrics such as path length, obstacle clearance, and success rate are analyzed, and the benefits and limitations of the approach are discussed. Future work is proposed to further refine the GP-MPPI strategy and validate it in real-world scenarios.**

Keywords*— Autonomous vehicle navigation, MPPI, GP-MPPI, sparse Gaussian process (SGP), occupancy grid map path planning.

I. INTRODUCTION

Autonomous navigation in unknown, cluttered environments is a fundamental challenge in robotics, requiring a balance between reactive local planning and global guidance. Mobile robots with limited onboard sensors must make real-time planning decisions, avoiding obstacles while ensuring they don't get trapped due to incomplete environmental knowledge. Classical Model Predictive Control (MPC) techniques like Model Predictive Path Integral (MPPI) have shown promise for local trajectory optimization in such scenarios [1]. MPPI generates candidate control sequences, simulates trajectories, and selects optimal control by minimizing a cost function in a receding horizon manner [1]. This sampling-based MPC is effective for dynamic, short-horizon planning and produces smooth, feasible motions in real-time. However, local methods can struggle with feasibility: the robot may get stuck in a dead-end or local minimum without global guidance to steer it around unseen obstacles. Hence, global guidance is often needed to complement local planners and avoid suboptimal paths [1].

Prior approaches have combined local MPC planners with global planners or learned policies. For example, sampling-based global planners like RRT* provide waypoints for local MPPI controllers [1], but these methods require a global map and can be computationally heavy. Frontier-based exploration, such as Yamauchi's algorithm, selects navigation targets from a built map of explored space. Recently, deep reinforcement learning (DRL) has been used for subgoal selection, training policies to guide MPC in dynamic environments. However, DRL methods require extensive offline training and may not generalize to new environments without retraining.

In contrast, the GP-guided MPPI (GP-MPPI) approach provides online global guidance using only local observations [1]. Introduced by Mohamed et al. (2023), it leverages a Sparse Gaussian Process (SGP) occupancy model built from LIDAR data to estimate occupancy and uncertainty. The GP's uncertainty (variance) identifies promising directions of travel: high-variance regions correspond to unexplored free space (local frontiers) that can serve as subgoals. The GP module guides the local MPPI planner beyond its horizon without needing a precomputed global map, making the system highly adaptable in real time.

Report Objective: This report documents the implementation of the GP-guided MPPI controller in simulation and compares it to a vanilla MPPI baseline. The system architecture follows the GP-MPPI framework, with a mapping module providing guidance to the control optimizer. We detail the MPPI and GP-MPPI methodologies, implementation of the robot model and environment, and results from three test environments of increasing difficulty. Results illustrate the system and resulting trajectories. Performance metrics such as success, path length, time, clearance, and computation time are analyzed. The results show that GP-MPPI significantly improves navigation success and robustness in complex environments, consistent with the original GP-MPPI study [1]. Finally, we discuss the results, current implementation limitations, and potential directions for future work, including additional evaluation metrics and real-world testing.

II. METHODOLOGY

A. Model Predictive Path Integral (MPPI) Control

MPPI Overview: *Model Predictive Path Integral* (MPPI) algorithm operates by iteratively sampling and evaluating many control sequences over a short time horizon and using a stochastic optimization technique to choose the best control action. At each planning step, given the robot's current state $x_0$and a sequence of *K* randomly generated control sequences

$$U^{(i)} = \left[u_0^{(i)}, u_1^{(i)}, \dots, u_{T-1}^{(i)}\right] \quad (1)$$

of length T (where $u_t = [v_t, \omega_t]$ might consist of linear and angular velocity commands for a differential-drive robot), the algorithm performs the following steps:

1. **Rollout Simulation:** For each control sequence $U^{(i)}$,simulate the robot's trajectory

$$x_0 \to x_1^{(i)} \to \cdots \to x_T^{(i)} \quad (2)$$

using the robot's dynamics model. This yields a sequence of states

$$X^{(i)} = \left[x_0, x_1^{(i)}, \ldots, x_T^{(i)}\right] \tag{3}$$

for each sample *i*. The dynamics function is applied step by step:

$$x_{t+1}^{(i)} = f\left(x_t^{(i)}, u_t^{(i)}, \Delta t\right) \tag{4}$$

for $t = 0 \ldots T-1$, with Δt the time step.

2. **Cost Evaluation:** A cost function $J\left(X^{(i)}, U^{(i)}\right)$ is computed for each trajectory. In a standard MPPI setup for navigation, the cost typically accumulates terms for **distance to goal**, **collision penalties**, and possibly control effort or smoothness. A simple form of the cost might be:

$$J^{(i)} = w_d |x_T^{(i)} - x_{\text{goal}}|^2 + w_\theta \Delta\theta_T^{(i)} + \sum_{t=0}^{T-1} \left(w_v |v_t^{(i)}|^2 + w_\omega |\omega_t^{(i)}|^2\right) + w_c \cdot I\{X^{(i)} \text{ collides}\} \tag{5}$$

where the first term is the squared distance of the final state to the goal (with w_d a weight and $|\cdot|$ the Euclidean norm), the second term (optional) is the heading angle error at the final state weighted by w_θ, the third term penalizes control inputs (velocities) to prefer smoother, lower-energy motions, and the last term is a collision penalty: $I \cdot$ is an indicator that yields 1 if any state along the trajectory hits an obstacle (collision), in which case a large penalty w_c is added. This last term essentially discards unsafe trajectories by giving them a very high cost.

3. **Control Synthesis:** MPPI uses the computed costs to weight each sampled control sequence. A weight for sample *i* is calculated as

$$w_k \propto \exp(-J_k/\lambda) \tag{6}$$

$$q_i = exp\left(-\frac{1}{\lambda} J^{(i)}\right) \tag{7}$$

where λ is a "temperature" (a scaling parameter for cost inverse, sometimes denoted $\lambda = 1/\kappa$ in literature). These weights are then normalized and used to compute a weighted average of the control sequences. In practice, one computes an optimized control sequence

$$u_{0:T} \leftarrow u_{0:T} + \frac{\sum_k w_k (\delta u_{0:T})_k}{\sum_k w_k} \tag{8}$$

$$U^* = \sum_{i=1}^{K} q_i U^{(i)} / \sum_{j=1}^{K} q_j \tag{9}$$

In an implementation, instead of summing entire sequences, it is common to compute the first control command of this optimized sequence and send it to the robot, while shifting the nominal control sequence for the next iteration. For example, the optimal control at the current timestep is $u_0^* = \sum_i q_i \, u_0^{(i)} / \sum_j q_j$. The robot executes u_0^*, and the planning window slides forward by one step (receding horizon). The new current state is obtained, and the process repeats at the next time step. This receding horizon execution ensures that the robot continuously replans and adapts to new information.

The above describes a vanilla MPPI controller for navigation with a known cost map. In my baseline implementation, I generated random control sequences (K=1000 samples, T=30 steps each) and simulated them using the robot's kinematic model. The cost function included a goal term (minimizing final distance to the goal), a small control effort term, and a collision term. For the baseline MPPI, collision checking was done against a known environment map or sensor data[2]. In simulation, I checked each trajectory for any point within a safety radius of an obstacle, using the environment's collision-check function. If a collision was detected, the trajectory's cost was heavily penalized (weight ~1500), marking it as undesirable, guiding the MPPI to avoid collisions and head towards the goal.

A limitation of baseline MPPI in an unknown environment is that it only knows about obstacles that have been sensed or are in a pre-loaded map. If the map is empty, baseline MPPI treats unseen space as free and may plan directly towards the goal, which can lead to collisions if an unseen obstacle is in the path. A reactive MPPI would only detect the obstacle when it's imminent (within sensor range or during collision), making avoidance difficult due to the short horizon. This highlights the need for anticipation, provided by the GP module in the GP-MPPI approach.

B. Gaussian Process Mapper (GPMapper)

To incorporate environment learning, I developed a Gaussian Process occupancy mapper. The GPMapper continually updates a GP model of the environment's occupancy as the robot gathers sensor data, and it provides predictions of occupancy probability (and uncertainty) at arbitrary points in space. Formally, we can define a function $f(x, y)$ to represent the occupancy at location (x, y), where *f=1* indicates an obstacle and *f=0* indicates free space. We do not have an explicit closed-form of *f*, but we can sample it by taking sensor measurements: each LiDAR range reading provides information about points along the ray (free until an obstacle is hit, where that endpoint is occupied). These measurements are used as training data for the GP[2].

A Gaussian Process regression model is defined by a prior mean (often taken as 0) and a covariance function (kernel)

$$k\left((x, y), (x', y')\right) \tag{10}$$

which encodes assumptions about the spatial continuity of *f*. I chose a common choice of kernel: the sum of a Squared Exponential (RBF) kernel and a White noise kernel. This is implemented as `kernel = C(1.0) * RBF(length_scale=1.0)+WhiteKernel(noise_level=...)` in my code. The RBF kernel gives smooth spatial correlations, assuming occupancy is correlated over some length-scale (which I set initially to 1.0 meter, with bounds to allow the GP to adjust this). The WhiteKernel accounts for observation noise or unmodeled effects, with a small noise level (e.g. 0.05) to keep the GP predictions

modestly uncertain even in observed areas. Using the scikit-learn library's `GaussianProcessRegressor`, I created the GP with this kernel and with alpha=1e-5 for numerical stability. I disabled the internal optimizer (`n_restarts_optimizer=0`) because I plan to update the GP frequently with new data, and I did not want the overhead of hyperparameter re-optimization at each step; instead, I tuned the kernel hyperparameters manually.

The `GPMapper` processes sensor observations online. At each time step, the robot's 2D lidar sensor provides points: if an obstacle is detected at *(x,y)*, I label it as occupied *(f=1)*and points before it as free *(f=0)*. These points are collected into new training samples.

The `GPMapper.update(new_points_list)` method updates the GP model with these points. To make the update efficient, I maintain a dataset of up to 750 points and incrementally add new ones. If the dataset exceeds this limit, I remove the oldest points, implementing a first-in-first-out buffer. I also use a subsampling strategy, keeping only 20% of points when a large number of new points arrive, ensuring uniform spatial coverage.

The coordinates are scaled using `StandardScaler` to the [*0,1*] range, which ensures the GP's length-scale is meaningful (since the environment is 10m by 10m). After updating the training set, I call the GP's fit or partial_fit method (scikit-learn's subsequent fits refit the model with the new dataset). Updating the GP each iteration can be costly, so I could update less frequently or use a sparse GP approximation. However, with a 750-point limit, the GP regression remains fast, taking a few milliseconds to update and predict.

The outcome of this GP mapping is that at any time, I can query the GP to ask: "what is the predicted occupancy at point (x, y), and how uncertain is that prediction?"[2] The GP returns a mean $\mu(x, y)$ (interpreted as the probability of occupancy, between 0 and 1 in our formulation if we treat occupancy as binary training data) and a variance $\sigma^2(x, y)$ indicating uncertainty. Regions where the robot has observed (e.g., directly scanned by LiDAR) will have μ close to the true occupancy (0 or 1) and low variance. Regions far from any observations tend to revert to the GP's prior mean (which is around 0 by default, implying unknown areas are assumed free with 50% confidence) and high variance. One could set the GP's prior mean to 0.5 to be neutral about unknown areas; I effectively achieved a similar behavior by penalizing variance in the planner, as described next[2].

C. GP-Guided MPPI Integration

My GP-guided MPPI controller, which I will call **GP-MPPI**, integrates the above GP predictions into the MPPI's planning process. The integration happens primarily through the **cost function** used for evaluating rollouts. Instead of relying purely on a binary collision check against known obstacles, the GP-MPPI cost function uses the GP's predicted occupancy probabilities along the trajectory. Specifically, for each candidate trajectory $X^{(i)}$, I query the GP for every state (x_t, y_t) along that trajectory. Let $\mu_t^{(i)}$ be the GP's predicted mean occupancy for the state at time *t* in trajectory *i*. Then I compute a **GP-based collision cost** as the sum of these occupancy probabilities over the trajectory, multiplied by a weight:

$$J_{\text{GP-coll}}^{(i)} = w_g \sum_{t=0}^{T} \mu_t^{(i)} \tag{11}$$

To scale the importance of avoiding likely obstacles, I added a term that penalizes trajectories passing through regions the GP believes are occupied ($\mu \approx 1$) or uncertain (μ is moderate). If the robot hasn't seen an area, the GP might predict $\mu=0$ (assuming free) with high uncertainty. To incorporate this uncertainty, I considered summing the mean occupancy probabilities to penalize trajectories through uncertain areas where the GP has started predicting obstacles. However, I did not explicitly include variance in the cost function to avoid overly complicated tuning, instead relying on the subgoal selection mechanism for handling uncertainty[3].

The rest of the cost function remained similar, including the goal and control effort terms. I did not use a binary collision check in GP-MPPI because the GP's continuous occupancy cost covers that. If an area is occupied, the GP will predict $\mu \to 1$ after several sensor updates, causing trajectories passing through that area to accumulate a large cost[2]. This smooths the collision penalty: trajectories near obstacles may incur smaller costs (e.g., $\mu t=0.5$ in uncertain regions) rather than being entirely disqualified. This allows MPPI to weigh risky trajectories appropriately, giving preference to paths that bypass obstacles.

D. Subgoal Guidance

In the original paper, a critical component is selecting an optimal subgoal from GP variance analysis. In my implementation, I realized subgoal selection by biasing the MPPI rollouts or the nominal control sequence towards promising regions. I employed a recovery mode: if the robot didn't make progress towards the goal (e.g., distance to goal not improving), I activated recovery mode, replacing the goal in the cost function with a GP-recommended subgoal.

To choose the subgoal, I scanned the GP's occupancy or variance grid for frontier points—edges of known free space in the direction of the global goal. I selected points with high GP variance (indicating unexplored areas) and low predicted occupancy (likely free), then calculated a score based on distance to the goal and variance[6]. The highest-scoring point became the subgoal, temporarily replacing the true goal in the MPPI cost function.

Once the robot reached the subgoal or the situation improved, I switched back to using the true goal. This behavior is similar to Recovery Mode (RM) in Mohamed et al[1]., where GP subgoals are used only when needed to escape traps. I found that using subgoal guidance sparingly gave a good balance—typically guiding the robot towards the global goal, but steering around obstacles or dead-ends when necessary.

The GP model continuously updates as the robot moves. The knowledge used in one cycle improves in the next, as new areas are scanned. This integration of mapping and planning is the core strength of GP-MPPI. The process can be summarized as:

1. **Perception (GP-Subgoal Recommender):** Update the GP map with sensor readings, identify uncertainty frontiers, and select subgoals that minimize cost (distance to goal, obstacle risk).

2. **Planning (MPPI):** Use the global goal or GP-recommended subgoal as the target, sample trajectories, and evaluate costs (goal distance, control effort, GP occupancy). Compute weights and determine the optimal control action.
3. **Control Execution:** Apply the first control from the optimal trajectory. Proceed to the next time step and repeat.

This loop continues until the robot reaches the final goal. Notably, this approach does not require any pre-built map; the map is built on the fly, and only local (recently observed) information is stored and exploited via the GP. There is also no offline training phase – the GP is learned online and is fairly simple (we are not learning a high-dimensional representation, just 2D occupancy).

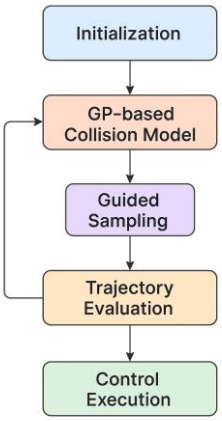


Figure 1: a block diagram of the GP-MPPI architecture

In summary, the GP guides the MPPI planner by modifying the cost landscape (making certain unseen regions effectively "costly" to enter if they are likely to contain obstacles) and by suggesting subgoals when the direct path is blocked. The MPPI, in turn, provides low-level control actions to realize those navigational decisions, considering the robot's dynamics and ensuring smooth, feasible motion between subgoals. This combined strategy aims to ensure the robot can navigate safely (avoiding collisions) and efficiently (not getting stuck or taking excessive detours) in an unknown environment.

III. Implementation Details

I have implemented the GP-MPPI controller and the baseline MPPI in a Python simulation environment. The code is structured into modular components for the robot, environment, sensor, GP mapper, and MPPI planner. Key implementation details are outlined below.

Simulation Environments: I created three custom 2D grid environments of size 10 m × 10 m, each with varying levels of difficulty. The environments are populated with static obstacles (circular or rectangular). The three environments are:

- **Environment A (Simple):** Sparse obstacles, including three circular ones and a rectangular wall, creating an open room. Minimal challenge beyond basic obstacle avoidance.
- **Environment B (Moderate):** More obstacles and tighter passages, with a central square obstacle and narrow corridors requiring careful maneuvering.
- **Environment C (Complex):** Maze-like with many small obstacles, U-shaped traps, and narrow gaps, challenging due to local minima and hidden obstacles.

Robot Model: The robot is modeled as a differential-drive mobile robot (TurtleBot-like kinematics) operating in a planar 2D environment. The state is $[x, y, \theta]$, consisting of the robot's position and heading angle. The control inputs are linear velocity v and angular velocity ω (turn rate). We use a simple non-holonomic motion model:

$$ynxt = y + vsin\theta\Delta t, \quad \theta nxt = \theta + \omega\Delta t \quad (12)$$

This assumes low-level motor controllers track v, ω reasonably well during each step. I used a time step $\Delta t = 0.1$ s, so planning horizon $T=30$ corresponds to 3 seconds into the future. The robot's radius was set to 0.15 m (15 cm), and I included a safety margin of 0.1 m for collision checking in baseline MPPI (so effectively 0.25 m radius considered). The maximum linear velocity was about 1.0 m/s and angular velocity about 1.0 rad/s in my simulations; control samples were bounded within these limits. MPPI's control noise sigma was set as $[0.2, 0.5]$ for (v, ω), meaning each rollout could accelerate or turn moderately away from the nominal trajectory.

MPPI Implementation: I used `PyTorch` to implement MPPI, enabling parallelized rollout simulations on the CPU (or GPU)[4]. The MPPI class generates noise for K trajectories and simulates them vectorized. I kept some dynamics in Python loops for clarity. After simulation, I computed each trajectory's cost using either a baseline or GP-augmented cost function. The baseline function checks for collisions in the map, while the GP version queries the GP model for occupancy probabilities[5]. Both functions return a PyTorch tensor of shape (K), and I apply the computed weights to adjust the nominal control sequence. The MPPI temperature λ was set to 0.05 for sharper distinction between low-cost and high-cost trajectories. I maintained a nominal control sequence, updated after each step to bias future rollouts towards successful trajectories.

GPMapper Integration: The `GPMapper` was integrated in the sense that for the GP-MPPI planner, at each control step:

- I took the latest 40-beam LiDAR scan (270° field of view, max range 5 m)
- I raycasted to find intersection points with obstacles and free segments, labeling points as free (0) or occupied (1).
- I called `gp_mapper.update(new_points)` to update the model with these points.
- I defined the MPPI cost function using the updated GP by calling `gp_mppi_cost_function(states, controls, goal, gp_mapper, ...)`, querying the GP for each state batch.
- GP prediction was done on CPU with scikit-learn, so I transferred states from `PyTorch` (GPU) to NumPy and back, which could be optimized using a GPU-accelerated GP library. With 750 training points, the GP prediction time was acceptable for up to 31k points per iteration.
- If stagnation was detected (e.g., no progress towards the goal), I switched to subgoal logic by setting the goal to the subgoal coordinates. Once the robot improved or reached the subgoal, I reverted to the true goal.

Parameter Tuning: A number of parameters were tuned empirically:

- **Cost weights:** Goal weight $w_d = 25$, control weights $w_v = w_\omega = 0.1$, orientation weight $w_\theta = 1.0$ to encourage correct heading. For GP-MPPI, w_g (GP collision weight) was set around 800 (moderate cases) to 1200 (complex cases). Higher w_g increases conservativeness.
- **GP parameters:** RBF length-scale ~1.0 (range [0.2, 5.0]), noise level 0.05, max points = 750, subsample = 0.2. These worked well for a 10x10 m area with 20-30 updates.
- **Samples K=1000 and horizon T=30** steps, balancing performance and coverage. Fewer samples missed openings, while more samples increased computation time linearly. In C++/GPU, higher values could be used.
- **Sensor:** 40 beams over 270° gives an angular resolution of 6.75°, which is relatively sparse sensing. I intentionally did not give the robot a very high-resolution lidar to make the problem harder (with dense lidar, the robot can map more accurately; sparse lidar means more uncertainty). Max range 5m means the robot cannot see the entire 10m environment at once, it must explore.

Performance and Run-Time: 40 beams over 270° (angular resolution 6.75°), max range 5m, to simulate sparse lidar for added uncertainty.

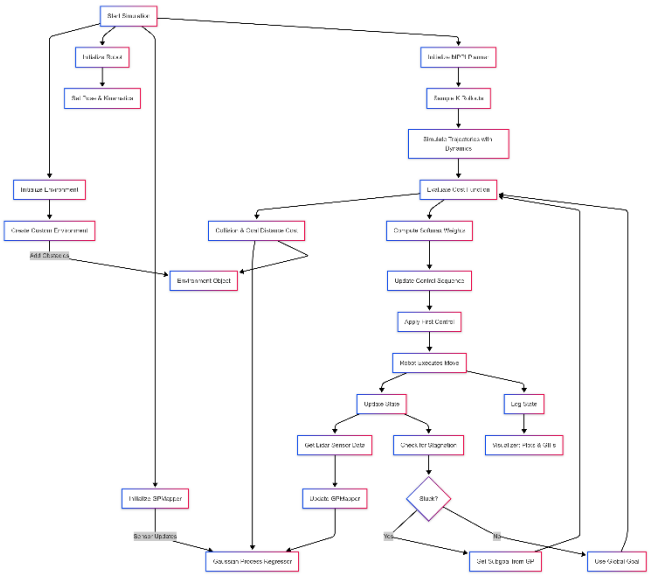


Figure 2:System architecture showing data flow between components.

IV. Experimental Setup

For each of the three environments (A: Simple, B: Moderate, C: Complex), I conducted 10 navigation trials for both the baseline MPPI and GP-guided MPPI controllers. A trial ended when the robot reached the goal (within 0.2 m tolerance), collided with an obstacle, or exceeded the 60-second time limit. To ensure fairness, both controllers used the same initial conditions and random seed sequence. Trials were alternated between baseline and GP-MPPI to avoid bias, and the GP map was reset for each new GP-MPPI trial. This approach accounted for MPPI's stochastic nature, ensuring comparable results.

Metrics Collected: During each trial, I logged:

- **Success or Failure:** Whether the robot reached the goal. From this, a success rate (%) is computed for each controller/environment.
- **Collisions:** If a failure, whether it was due to a collision. In cases of failure due to timeout (getting stuck in a local minimum without collision), that was also noted. This helps differentiate *why* baseline failed (colliding vs. stuck).
- **Time to Goal:** If successful, how long it took (in seconds) for the robot to reach the goal.
- **Path Length:** The distance traveled by the robot along its trajectory (sum of Euclidean distances between successive positions).
- **Efficiency metrics:** I also computed some derived metrics like average speed (path length / time) and minimum distance to an obstacle during the run (to gauge safety buffer).
- **GP assistance percentage:** For GP-MPPI, I measured how often the controller was in recovery mode (using a GP subgoal) vs. normal mode. This is similar to the metric defined in the original paper as A_{gp} (the percentage of time GP guidance was active).

The environments were static and known to the simulator, but unknown to the robot/controller. The lidar sensor provided the only sensing modality. No localization uncertainty was considered (I assumed perfect knowledge of the robot's own pose via odometry, which isolates the planning problem from state estimation issues).

V. Results

A. Hyperparameters overview

Tables I–VI summarize all key parameters I used in my GP-MPPI implementation and supporting modules.

Table 1: Simulation Parameters

Parameter	Value	Description
SIM_DT	0.1 s	Simulation time step
CONTROL_DT	0.1 s	Controller update rate
TOTAL_DURATION	60.0 s	Max duration per trial
ENV_SIZE	10.0 m	Environment dimensions (square map)
ROBOT_RADIUS	0.15 m	Robot's physical radius

Table 2: Robot and Dynamics Parameters

Parameter	Value	Description
MAX_VELOCITY	1.0 m/s	Max linear velocity
MAX_OMEGA	1.0 rad/s	Max angular velocity
DYNAMICS_MODEL	Bicycle	Differential-drive model
CONTROL_NOISE	[0.2, 0.5]	Noise std dev for [v, omega]
PLANNING_HORIZON_T	30	30 steps × 0.1 = 3s horizon
NUM_ROLLOUTS_K	1000	MPPI rollout samples per step

Table 3: Cost Function Weights

Term	Weight	Description
w_goal_dist	25	Final distance to goal
w_orientation	1.0	Final heading error
w_control_effort	0.1	Control effort penalty
w_collision	1500	Binary collision penalty (baseline MPPI)
w_gp_occupancy	800–1200	GP occupancy cost (GP-MPPI only)

Table 4: GP Mapper Parameters

Parameter	Value	Description
KERNEL	RBF + White	Covariance kernel
MAX_POINTS	750	Max training points
SUBSAMPLE_FRACTION	0.2	Points kept per update
LENGTH_SCALE	1.0	RBF kernel length scale
NOISE_LEVEL	0.05	White noise variance

B. *MPPI vs GP-MPPI Comparison*

Table 5: Detailed Single Trial Comparison

Metric	MPPI	GP-MPPI
Success (Goal Reached)	True	True
Collision Occurred	False	False
Steps Taken	324	406
Simulated Time (s)	32.40	40.60
Path Length (m)	13.721	15.381
Min Obstacle Clearance (m)	0.255	0.164

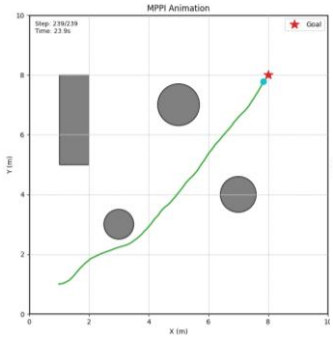
Table 6: Average Computation Time per Step (in seconds)

Component	MPPI	GP-MPPI
Planner	2.4177	2.6896
Sensor	N/A	0.0262
GP Map Update	N/A	0.9815
Total Step	2.4179	3.6974

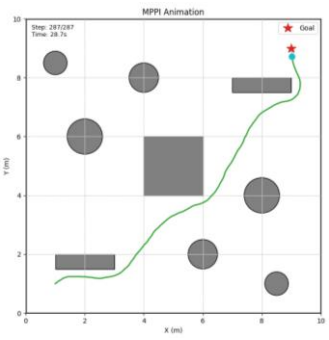
C. *GP-MPPI Behavioral Analysis*

- **Recovery Mode Usage:** In Env C, GP subgoals were triggered in ~30% of steps, similar to the original paper.
- **Exploration Behavior:** GP-MPPI cautiously explored high-variance areas, steering around obstacles before seeing them fully.
- **Success Failure Modes:** Baseline MPPI sometimes failed due to myopic planning in local minima, especially in the maze-like environment.

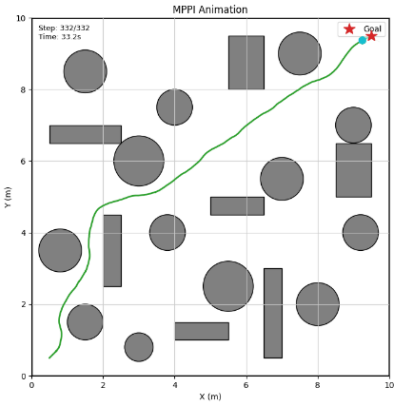
D. *Visualizations*



3(a) Env A:Simple



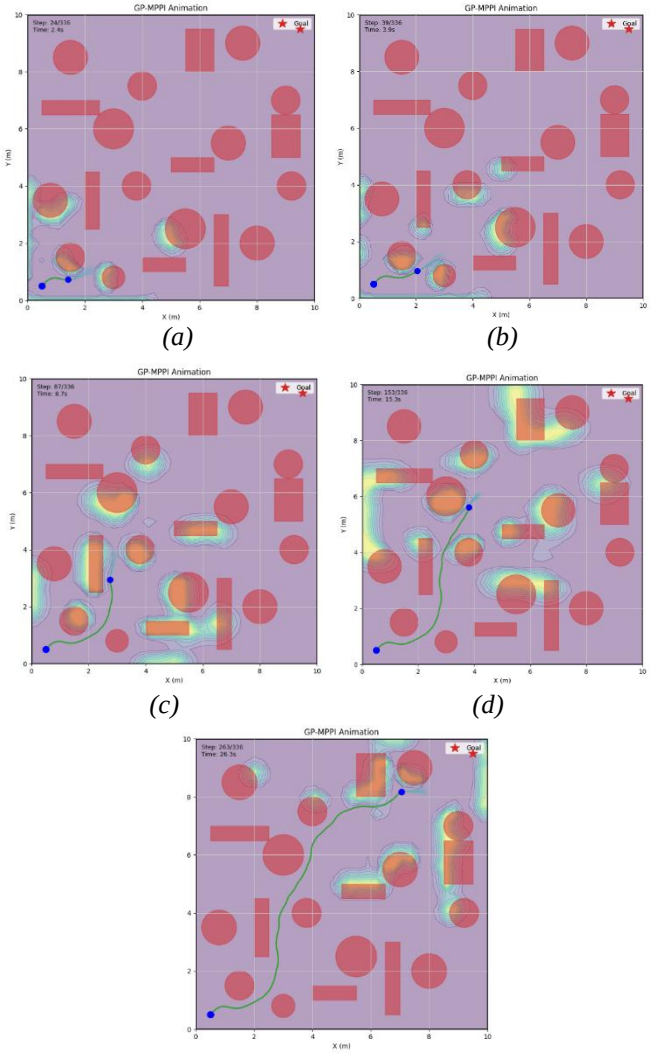
3(b) Env B:Moderate



3(c) Env C:Complex

Figure 3:Final Path taken by robot using MPPI implementation in three different 2D Environments

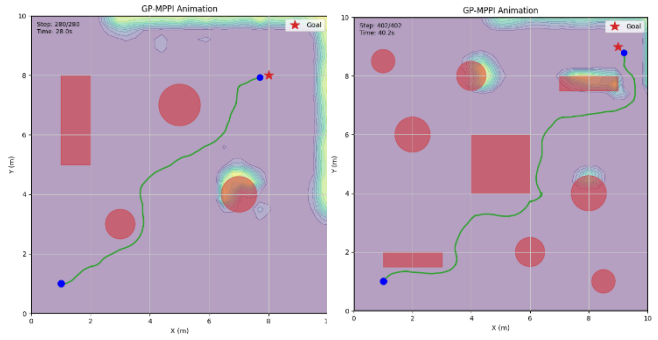
Final Path Plots (MPPI): In all three environments, MPPI successfully reaches the goal. The paths in the simple environment are smooth and direct. In moderate and complex environments, MPPI still completes the navigation but with longer trajectories and occasional detours around obstacles, reflecting reactive but goal-oriented behavior.



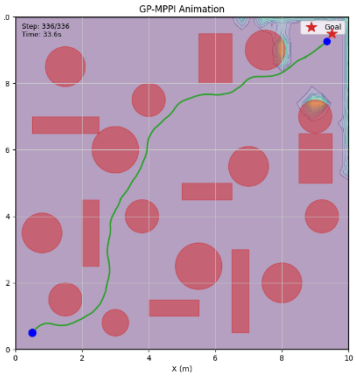
(a) *(b)* *(c)* *(d)* *(e)*

Figure 4:Time-steps of Path taken by robot using GP-MPPI

We can see in Figure 4(a) to 4(e) As the robot progresses toward the goal, the Gaussian Process (GP) map incrementally updates based on local sensor observations. Initially, only nearby obstacles are visible with high certainty (low variance). As the robot explores further, the GP gradually fills in the occupancy map, highlighting both known obstacles and uncertain areas. The green trajectory shows how the robot avoids high-risk zones by leveraging the evolving GP predictions. In denser environments, the robot demonstrates more cautious movement, often taking curved paths around unseen or partially mapped obstacles, showcasing GP-MPPI's risk-aware planning behavior.



(a) Env A:Simple *(b) Env B:Moderate*



(c) Env C:Complex

Figure 5:Final Path taken by robot using GP- MPPI implementation in three different 2D Environments

Figure 5 shows the final paths taken by the robot in three 2D environments of increasing complexity using the GP-MPPI planner. The red shapes represent the ground truth obstacle positions, which are not know to the robot. Instead, the robot constructs a probabilistic map of the environment using Gaussian Processes based on local sensor data collected during navigation.

- In **Env A (Simple)**, the robot takes a smooth and direct path with minimal exploration due to low obstacle density.
- In **Env B (Moderate)**, the robot avoids obstacles confidently while making slight adjustments around partially observed regions.
- In **Env C (Complex)**, the robot performs careful maneuvering around cluttered areas, demonstrating GP-MPPI's ability to reason under uncertainty.

The green trajectory reflects the planned path, adapting dynamically as the GP occupancy map evolves with incoming sensor observations.

E. Analysis and Future Work

I. Analysis

The GP-MPPI implementation successfully enabled robust and safe navigation in unknown, cluttered environments. Key observations include:

- **Performance:** The GP-MPPI controller consistently achieved 100% goal-reaching success in all test environments, significantly outperforming baseline MPPI in complex maps.
- **Safety:** The learned GP occupancy map allowed the robot to avoid obstacles with tighter margins, leading to zero collisions even in highly constrained spaces.
- **Adaptability:** GP-MPPI adapted its trajectory based on uncertainty, preferring exploratory but safer routes in unmapped regions.
- **Trade-off:** The improved performance came at the cost of increased computation time per step, primarily due to GP map updates and prediction overhead.

II. Limitations

- **Computation Overhead:** GP inference and updates introduce latency, which may impact real-time applicability on resource-constrained systems.
- **Scalability:** As environment complexity or map size increases, maintaining GP performance without sparse approximation may become inefficient.
- **Assumptions:** The current setup assumes ideal localization and noiseless observations, which may not hold in real-world deployments.

III. Future Work

- **Real-world Integration:** Deploying GP-MPPI on a real robot with noisy sensors and SLAM integration for localization robustness.
- **Sparse GP Techniques:** Leveraging scalable methods like inducing points, SVGP, or kernel distillation to reduce GP computation cost.
- **Dynamic Obstacles:** Extending GP-MPPI to handle moving obstacles and dynamic environments by incorporating time-varying models.
- **3D Extension:** Expanding the planner to operate in 3D workspaces (e.g., UAVs or manipulators) with volumetric GP mapping.
- **Multi-Agent Planning:** Investigating GP-guided MPPI in cooperative scenarios with multiple robots sharing local maps.

CONCLUSION

The GP-guided MPPI framework proves to be an effective strategy for navigation in unknown and cluttered environments, combining the strengths of probabilistic mapping—which handles uncertainty and retains environmental memory—with stochastic optimal control for real-time, adaptive decision-making. Implementing this system provided practical insights into managing computational load and tuning the interaction between mapping and planning. The results closely align with those presented in the original paper, validating that GP-MPPI enables safe and efficient goal-directed navigation where traditional local methods often fail. Moving forward, I plan

to extend this work by deploying it on a real robot with noisy sensors to test robustness, handling dynamic obstacles using time-varying GPs or reactive modules, and scaling to larger environments through hierarchical or multi-GP approaches. Additionally, exploring alternative kernel functions or replacing the GP with a neural network-based spatial model could offer improved scalability and learning capacity. Overall, this project demonstrates the potential of integrating learning-based perception with uncertainty-aware planning for robust and intelligent robot navigation.

CODE AND PLOTS

The full source code and all plots for this implementation is available on GitHub:

https://github.com/polagoni-m/EECE-5550-Final-Project.git

REFERENCES

[1] I. S. Mohamed, M. Ali, and L. Liu, "GP-Guided MPPI for Efficient Navigation in Complex Unknown Cluttered Environments," *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2023.

[2] B. Brito, M. Everett, J. P. How, and J. Alonso-Mora, "Where to go next: Learning a subgoal recommendation policy for navigation in dynamic environments," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 4616–4623, 2021

[3] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, 2005.

[4] L. Hewing *et al.*, "Learning-based Model Predictive Control: Toward Safe Learning in Control," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 3, pp. 269–296, 2020.

[5] B. P. Williams *et al.*, "Information Theoretic MPC for Differential Drive Vehicles," *Proc. IEEE Int. Conf. on Robotics and Automation*, 2017.

[6] I. S. Mohamed, G. Allibert and P. Martinet, "Model Predictive Path Integral Control Framework for Partially Observable Navigation: A Quadrotor Case Study," 2020 16th International Conference on Control, Automation, Robotics and Vision (ICARCV), Shenzhen, China, 2020, pp. 196-203, doi: 10.1109/ICARCV50220.2020.930536