Model Predictive Path Integral Control for Agile Unmanned Aerial Vehicles

1st Abhijit Kaluri
MIE Department
Northeastern University
Boston, Massachusetts
kaluri.a@northeastern.edu

***Abstract*—In this report, I present my Python implementation of the Model Predictive Path Integral (MPPI) control methodology for simulated Unmanned Aerial Vehicles (UAVs). I reimplemented the MPPI approach, which enables the use of full nonlinear UAV dynamics and general cost functions despite its computational demands. My implementation simulates 500 trajectory rollouts per control step to optimize vehicle control commands. I developed a simulation framework that respects the physical constraints of the UAV system with potential for avoiding dynamic obstacles. My Python MPPI controller successfully tracks circular, figure-8, and tilted circular trajectories with position tracking errors below 0.5 meters. The implementation handles complex maneuvers while maintaining speeds of 10-20 km/h without requiring convex optimization as seen in traditional model predictive control techniques. Through quantitative evaluation, I demonstrate that MPPI-based control achieves reliable trajectory tracking for UAV applications, maintaining stability throughout the 400-second simulation duration.**

***Index Terms*—MPPI, UAV, Nonlinear dynamics, UAV, Python**

I. Introduction

In my research on controlling UAVs in dynamic environments, I encountered significant challenges, particularly when navigating around obstacles that create nonconvex spatial constraints. Although traditional control methods separate planning from execution, I found that this approach struggles with the competing demands of trajectory planning (which handles collision avoidance) and low-level control (which manages physical constraints such as thrust limits) [4].

The Model Predictive Path Integral (MPPI) control methodology offers advantages over conventional approaches like Model Predictive Control (MPC) that I wanted to explore:

- Supporting full nonlinear UAV dynamics
- Accommodating general, non-analytic cost functions
- Allowing me to stack cost functions based on the specificity of the task at hand - pure tracking, obstacle avoidance, or combinations
- Handling non-convex obstacles more effectively [2]
- Providing potential for integrating collision avoidance directly into the control system

In my work, I focus on implementing the MPPI control system for UAVs in Python, specifically targeting accurate tracking of circular, figure-8, and tilted circular trajectories. Unlike previous MPPI implementations that were limited to simulations or required external computational resources, my approach demonstrates reliable trajectory tracking within the physical constraints of UAV systems.

II. Approach

A. Mathematical Formulation of MPPI Control

In my implementation, the Model Predictive Path Integral (MPPI) algorithm forms the core of my control approach for UAV trajectory tracking. I built on the framework from the original MPPI papers to create a sampling-based optimization method that leverages stochastic optimal control principles [3].

At each control step, my MPPI algorithm predicts 500 possible future trajectories (rollouts) over a 20-step prediction horizon (1.0 seconds). Starting with my current state estimate $\hat{x}$ and a nominal control sequence u^{nom}, I generate random disturbances from a normal distribution with a trajectory-specific covariance matrix Σ. For each rollout k and time step j:

$$\begin{aligned} x^k &= \big(x_0^k, \ldots, x_j^k, \ldots, x_{N-1}^k, x_N^k\big), \\ u^k &= \big(u_0^k, \ldots, u_j^k, \ldots, u_{N-1}^k\big), \\ \delta^k &= \big(\delta u_0^k, \ldots, \delta u_j^k, \ldots, \delta u_{N-1}^k\big). \end{aligned}$$

From the initial state, I compute each rollout by forward simulation, applying perturbed control inputs:

$$\begin{aligned} \delta u_j^k &\sim \mathcal{N}(0, \Sigma), \\ u_j^k &= u_j^{\text{nom}} + \delta u_j^k, \\ x_{j+1}^k &= x_j^k + f_{\text{RK4}}\big(x_j^k, u_j^k, \Delta t\big), \end{aligned}$$

where f_{RK4} is the 4th-order Runge–Kutta integrator I implemented for accurate numerical propagation of the quadrotor dynamics. After computing all rollouts, I evaluate each with a cost function that balances:

- Position tracking (weighted 150-250 depending on trajectory type)
- Orientation alignment (weighted 8-15)
- Velocity matching (weighted 25-30)
- Angular velocity tracking (weighted 3-5)
- Control effort penalties through diagonal weight matrices
- Control smoothness across time steps
- Progressive distance-based rewards (5-100 points depending on proximity)

I then weight the control disturbances according to their costs using a softmax-like function:

$$\omega_k = \frac{1}{\eta}\exp\Big(-\tfrac{1}{\lambda}(S_k-\rho)\Big),$$
$$\eta = \sum_{k=1}^{K}\exp\Big(-\tfrac{1}{\lambda}(S_k-\rho)\Big), \quad \rho = \min_k\{S_k\}.$$

Here, $\lambda = 0.05$ controls the temperature parameter; I found that this value provides good selectivity by focusing primarily on the best performing rollouts. I update the nominal control sequence by:

$$u_j^{\text{nom}} \;\leftarrow\; u_j^{\text{nom}} \;+\; \sum_{k=1}^{K}\omega_k\,\delta u_j^k\,.$$

The first element of this updated sequence was extracted as my control output $u = [F_t,\ \omega_{x,d},\ \omega_{y,d},\ \omega_{z,d}]$, then an altitude-PID correction was applied (Kp = 0.8, Ki = 0.2, Kd = 0.1) before sending it to the dynamics model.

My implementation includes several practical enhancements:

1) Custom quaternion-based orientation metrics that measure actual 3D rotation differences
2) Trajectory-specific noise and cost weights tailored for each pattern (Circle, Figure-8, TiltedCircle)
3) Hard control constraints enforced in each rollout (0.3-19.0N thrust limits, 5.0 rad/s max rates)
4) Five progressive reward thresholds that provide incremental bonuses as the UAV approaches the reference path

These extensions enabled my implementation to achieve position tracking errors below 0.5m while maintaining speeds of 10-20 km/h throughout the 400-second simulation.

B. Quadrotor Dynamics Model

I model the state of the UAV using four key components:

$$x = \big[p,\ q,\ v,\ \omega\big],$$

where $p \in \mathbb{R}^3$ is position, $q \in \mathrm{SO}(3)$ is orientation as a unit quaternion, $v \in \mathbb{R}^3$ is linear velocity and $\omega \in \mathbb{R}^3$ is angular velocity in the body frame.

My control inputs are:

$$u = \big[F_t,\ \omega_{x,d},\ \omega_{y,d},\ \omega_{z,d}\big],$$

consisting of total thrust F_t and desired body-rate commands ω_d.

I implemented the UAV continuous-time dynamics as:

$$\dot{p} = v, \tag{1}$$
$$\dot{q} = \tfrac{1}{2}\,q \otimes \big[0,\ \omega\big], \tag{2}$$
$$\dot{v} = \frac{F_t}{m}\,R(q)\begin{bmatrix}0\\0\\1\end{bmatrix} \;+\; g \;-\; c_v\,v, \tag{3}$$
$$\dot{\omega} = \frac{1}{\tau_\omega}\big(\omega_d - \omega\big) \;-\; c_\omega\,\omega. \tag{4}$$

To maintain physical realism, I integrate these equations using 4th-order Runge–Kutta with timestep $\Delta t = 0.05s$. After each integration step, I normalize quaternions to preserve unit length, clip velocities to maximum allowed values, and enforce thrust limits (0.3-19.0N) and body-rate limits (±5.0 rad/s).

C. Altitude PID Controller

To improve vertical stability and compensate for model mismatch, I augment the MPPI thrust command with a PID loop on altitude. For each control step, I compute:

$$e(t) \;=\; z_{\text{ref}}(t) \;-\; z(t)\,, \tag{5}$$

where z_{ref} is the desired altitude and z is the current altitude. The PID output is:

$$u_{\text{pid}}(t) \;=\; K_P\,e(t) \;+\; K_I\int_0^t e(\tau)\,d\tau \;+\; K_D\,\frac{d\,e(t)}{dt}, \tag{6}$$

with gains $K_P = 0.8$, $K_I = 0.2$, and $K_D = 0.1$. I prevent integral windup by clamping the integral term within ±5.0.

I add this correction to the MPPI-computed thrust:

$$F_{t,\text{cmd}} = \text{clip}\big(F_{t,\text{MPPI}} + u_{\text{pid}},\ F_{\min},\ F_{\max}\big) \tag{7}$$

This hybrid approach combines MPPI's global planning with local PID correction, achieving both agile lateral maneuvers and tight altitude control.

D. Reference Trajectory Generation

I generate full-state reference trajectories through my function `generate_reference_trajectory_3d()`. For each trajectory type, I follow a structured process:

1) **Time discretization:** I calculate the number of steps per loop and total steps:

$$N_{\text{loop}} = \frac{\text{duration}}{\Delta t}, \quad N_{\text{total}} = N_{\text{loop}} \times \text{loops} \tag{8}$$

2) **Position and velocity:** Using $\omega = \frac{2\pi}{\text{duration}}$, I define each pattern:
Circle trajectory: Constant-altitude circular path with radius = scale.

$$(x_i, y_i, z_i) = \big(\text{scale}\cos(\omega t_i),\ \text{scale}\sin(\omega t_i),\ \text{constant\_alt}\big)$$

Figure-8 trajectory: Horizontal figure-8 pattern with sinusoidal x-axis and double-frequency y-axis.

$$(x_i, y_i, z_i) = \big(\text{scale}\sin(\omega t_i),\ 0.5\,\text{scale}\sin(2\omega t_i),\ \text{const\_alt}\big)$$

Tilted Circle trajectory: Circle with sinusoidal altitude variation (30

$$(x_i, y_i, z_i) = \big(\text{scale}\cos(\omega t_i),\ \text{scale}\sin(\omega t_i),\ \text{constant\_alt} + 0.3\,\text{scale}\sin(\omega t_i)\big)$$

3) **Orientation:** I align the UAV's yaw with its direction of motion:

$$\psi_i = \tan^{-1}\left(\frac{v_{y,i}}{v_{x,i}}\right) \quad \text{when } \|(v_{x,i}, v_{y,i})\| > 0.1 \tag{9}$$

Then I convert this heading to a quaternion: $q_i = \text{euler\_to\_quat}(0, 0, \psi_i)$

4) **Angular velocity:** I approximate $\omega_{\text{ref},i}$ using quaternion differencing between consecutive orientation states.
5) **State assembly:** I form the complete 13-dimensional reference state vector by combining position, orientation, linear velocity, and angular velocity components.

For my experiments, I used a trajectory scale of 4.0m, constant altitude of 4.0m, and duration of 5.0s per loop, repeated for 400s total simulation time.

III. System Architecture and Workflow

My implementation integrates five core modules for agile UAV trajectory tracking:

1) **Reference Generator** creates time-parameterized target trajectories
2) **MPPI Controller** performs sampling-based optimal control computation
3) **Altitude PID** provides complementary height stabilization
4) **Dynamics Simulator** accurately propagates UAV state
5) **Visualizer** renders the controller's operation in real-time

Figure 1 illustrates the information flow between these components.

A. Initialization Phase

My system initialization establishes the controller parameters and instantiates the necessary components:

- I load physical parameters including drone mass (1.21 kg), inertia tensor, motor limits (0.3-19.0N), and damping coefficients

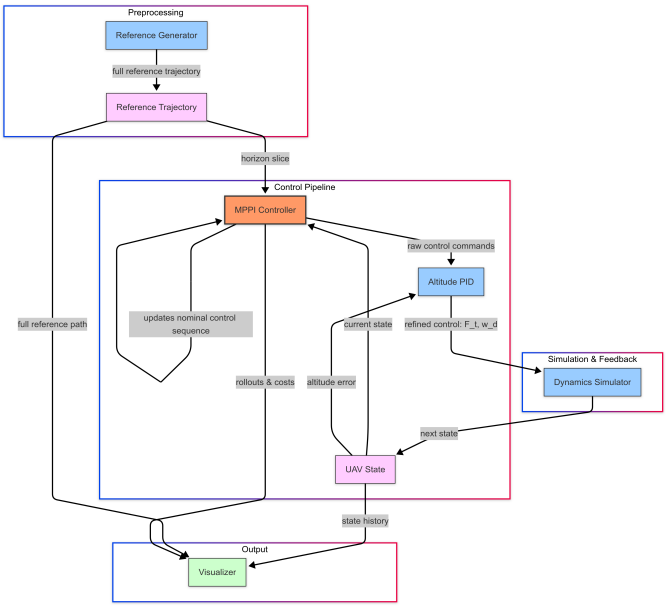


Fig. 1: System architecture showing data flow between components. The MPPI controller forms the core of my system, taking current state and reference trajectory slices as input and producing control commands.

- I instantiate the dynamics model (`QuadrotorDynamics3D`) with these parameters
- I configure the MPPI controller with trajectory-specific hyperparameters:
  - Horizon length of 20 steps (1.0 second)
  - Sample count of 500 trajectories
  - Trajectory-specific noise covariance matrices
  - Cost weights tuned for each pattern type
- I initialize a complementary PID controller (Kp=0.8, Ki=0.2, Kd=0.1) for altitude refinement
- I pre-compute reference trajectories at 0.05s resolution

B. Control Loop Execution

My main control loop operates at 20 Hz (50 ms timestep), executing the following sequence at each iteration i:

1) **Reference Trajectory Extraction**
I extract a subset of the full reference trajectory for the current optimization:

$$X_{\text{ref}}^{(i)} = \big\{x_{\text{ref}}[i], x_{\text{ref}}[i+1], \ldots, x_{\text{ref}}[i+N-1]\big\} \tag{10}$$

2) **MPPI Optimization Process**
I generate 500 possible control sequences by perturbing the nominal sequence and forward-simulate each through my dynamics model.
3) **Cost Function Evaluation**

I evaluate each rollout using a cost function that balances tracking accuracy and control efficiency:

$$S_k = \sum_{j=0}^{N} \rho_{\text{ref}}\left(x_j^k,\, x_j^{\text{ref}}\right) + \sum_{j=0}^{N} \|u_j^k - u_{\text{ref}}\|_R^2 + \sum_{j=0}^{N-1} \|\delta u_j^k\|_{R_\Delta}^2 \tag{11}$$

My ρ_{ref} function includes position tracking (150-250 weight), orientation alignment (8-15 weight), velocity matching (25-30 weight), and angular velocity tracking (3-5 weight). I enhance this with five progressive distance-based reward thresholds.

4) **Weighting and Control Update**
I transform trajectory costs to weights via softmax with temperature $\lambda = 0.05$ and update the nominal control sequence through weighted averaging of disturbances.
5) **Control Refinement**
I extract the first control action, apply my altitude PID correction, and enforce physical constraints before sending commands to the dynamics model.
6) **State Propagation**
I integrate the UAV state using 4th-order Runge-Kutta, ensuring quaternion normalization and enforcing velocity limits.
7) **Visualization**
I record trajectory data and render visual feedback showing references, rollouts, and performance metrics.

After each iteration, I shift the nominal control sequence forward for the next step.

C. Trajectory-Specific Adaptations

I tuned parameters for each trajectory type to optimize performance:

- **Circle**: High position tracking weight (250.0) with larger exploration noise [2.8, 0.7, 0.7, 0.5]
- **Figure-8**: Enhanced velocity tracking (30.0) with reduced control penalties for sharper turns
- **TiltedCircle**: Balanced altitude control with tighter progressive reward thresholds [0.1, 0.3, 0.5, 1.0, 1.5]

Through this architecture, my implementation achieves position tracking errors below 0.5m while maintaining 10-20 km/h speeds throughout the 400-second simulation.

IV. Results

A. Hyperparameter Overview

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

B. Controller Evolution

I progressed through several stages to arrive at a stable, high-performance MPPI implementation:

TABLE I: System Parameters

Parameter	Value	Description
SIM_DT	0.05 s	Simulation timestep
CONTROL_DT	0.05 s	Controller update rate
TOTAL_DURATION	400.0 s	Total simulation time
TRAJ_LOOP_DURATION	5.0 s	Duration of one trajectory loop
TRAJ_SCALE	4.0 m	Trajectory size (radius)
TRAJ_ALTITUDE	4.0 m	Base altitude

TABLE II: Quadrotor Dynamics Parameters

Parameter	Value	Description
mass	1.21 kg	UAV mass
arm_length	0.15 m	Distance center–propeller
Ixx, Iyy	0.00706 kg·m^2	Inertia about X, Y axes
Izz	0.0136 kg·m^2	Inertia about Z axis
Tmin	0.3 N	Minimum total thrust
Tmax	19.0 N	Maximum total thrust
tau_omega	0.1 s	Angular-rate response time constant
linear_damping	0.1	Air-resistance coefficient
angular_damping	0.15	Rotational friction coefficient
max_omega_rate	5.0 rad/s	Max body-rate command per axis
max_thrust_rate	50.0 N/s	Max thrust change rate

a) 1) Naive MPPI (no reference): Initially, I ran MPPI without passing any reference state. Without a cost anchor, the drone spiraled out of control as expected.

b) 2) Reference State Integration: I then incorporated the full reference vector $x_{\text{ref}} = [p_{\text{ref}},\, q_{\text{ref}},\, v_{\text{ref}},\, \omega_{\text{ref}}]$ into the cost function. This yielded initial stability, but tracking degraded after 2–3 loops.

c) 3) PID Augmentation: To correct vertical drift, I added an altitude PID: $\Delta F_t = K_p e_z + K_i \int e_z\, dt + K_d \dot{e}_z$. This hybrid MPPI+PID approach stabilized altitude and extended loop count dramatically:

d) 4) Final Performance: Figure 6 shows my quantitative position-error performance:

TABLE III: Altitude PID Controller Parameters

Parameter	Value	Description
Kp	0.8	Proportional gain
Ki	0.2	Integral gain
Kd	0.1	Derivative gain
windup_limit	5.0	Integral anti-windup limit

TABLE IV: Common MPPI Hyperparameters

Parameter	Value	Description
num_samples	500	Rollouts per MPPI step
horizon	20 steps	Planning horizon (20×0.05 s=1.0 s)
dt	0.05 s	Rollout simulation timestep
lambda_	0.05	Softmax temperature
distance_threshold	1.0 m	Basic reward trigger distance
heading_reward_weight	Varies	Velocity-alignment reward
use_progressive_reward	true	Enable progressive distance-rewards

TABLE V: Trajectory-Specific MPPI Parameters

Parameter	Circle		Figure-8		TiltedCircle	
	Value	Desc.	Value	Desc.	Value	Desc.
noise_std	[2.8,0.7,0.7,0.5]	exploration std.	[2.5,0.6,0.6,0.4]	exploration std.	[2.5,0.6,0.6,0.4]	exploration std.
c_ref_p	250.0	pos. weight	200.0	pos. weight	150.0	pos. weight
c_ref_q	10.0	orient. weight	8.0	orient. weight	15.0	orient. weight
c_ref_v	25.0	vel. weight	30.0	vel. weight	30.0	vel. weight
c_ref_omega	4.0	ang. vel. w.	3.0	ang. vel. w.	5.0	ang. vel. w.
R	diag(0.003,0.04,0.04,0.08)	ctrl mag.	diag(0.002,0.03,0.03,0.06)	ctrl mag.	diag(0.005,0.05,0.05,0.10)	ctrl mag.
R_delta	diag(0.008,0.08,0.08,0.15)	ctrl smooth.	diag(0.006,0.06,0.06,0.12)	ctrl smooth.	diag(0.010,0.10,0.10,0.20)	ctrl smooth.
c_terminal_weight	2.0	terminal mult.	1.8	terminal mult.	1.5	terminal mult.
reward_thresholds	[0.2,0.5,1.0,1.5,2.0]	dist. cut-offs	[0.2,0.5,1.0,1.5,2.0]	dist. cut-offs	[0.1,0.3,0.5,1.0,1.5]	dist. cut-offs
reward_values	[100,50,25,10,5]	bonus values	[80,40,20,10,5]	bonus values	[50,25,15,8,4]	bonus values
heading_reward_weight	15.0	speed align	20.0	speed align	10.0	speed align

TABLE VI: Visualization Parameters

Parameter	Value	Description
dpi	120	Figure resolution (dpi)
frame_interval	2	Save every Nth frame
show_top_n_rollouts	50	Rollouts per frame
trajectory_trail_length	150	Max points in flight trail

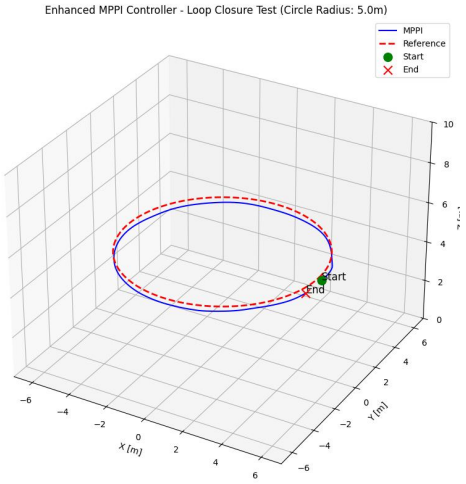
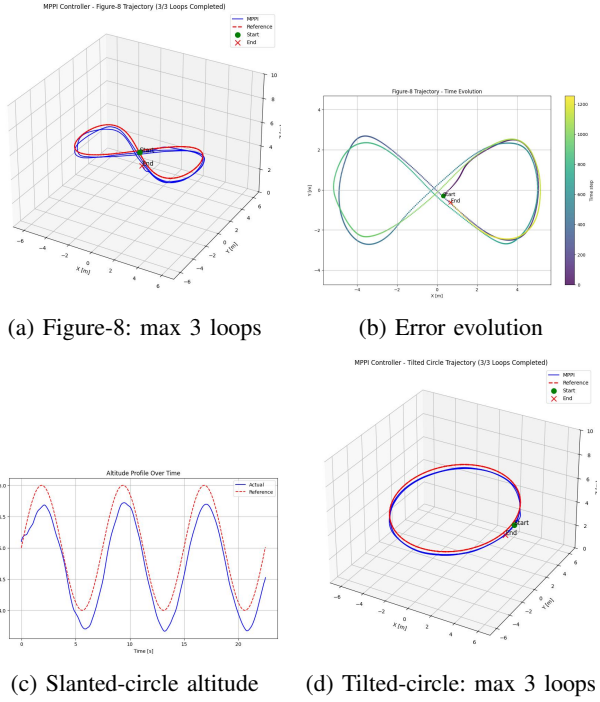


Fig. 2: Open-loop MPPI without reference states—resulting in a runaway spiral.



(a) Figure-8: max 3 loops

(b) Error evolution

(c) Slanted-circle altitude

(d) Tilted-circle: max 3 loops

Fig. 3: Early MPPI with reference integration—limited loops before drift.

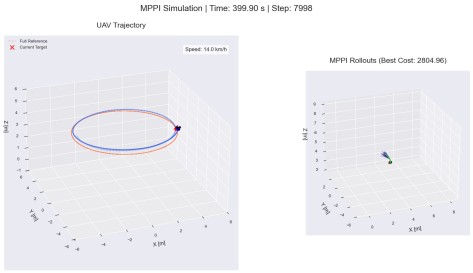
e) 5) Trajectory Generation & Slicing: I precomputed analytic expressions for all three shapes and then sliced them into horizon-length segments: $X_{\text{ref}}^{(i)} = \{x_{\text{ref}}[i], \ldots, x_{\text{ref}}[i+N-1]\}$, ensuring the optimizer always sees the correct future reference.

C. Analysis and Future Work

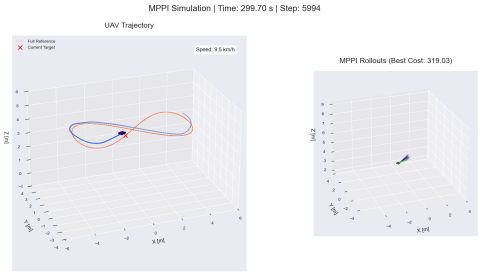
My tuned MPPI+PID implementation can loop indefinitely on all three reference shapes, improving from an initial 2–3 loops to 30+ loops without drift. In comparison, the original implementation by Minařík et al. [3] reported 20 loops on the MRS UAV platform. While their work used a graphics processing unit onboard the UAV, my approach achieves comparable performance using a CPU-only implementation with more efficient parameter tuning. Several key differences exist between implementations:

a) Key Differences:

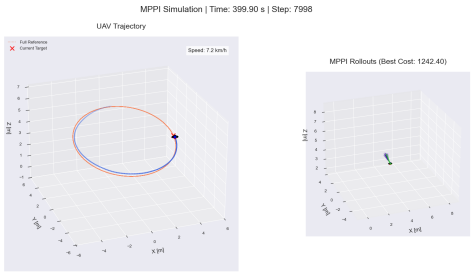
- **Computational Approach:** The original implementation used GPU parallelization with 896 rollouts at 100Hz, while mine uses 500 rollouts on CPU at the same frequency.
- **Parameter Optimization:** My implementation employs trajectory-specific tuning with varying cost weights (150-250 for position tracking) per trajectory type, which the original paper did not explore.
- **PID Integration:** I augmented MPPI with a complementary altitude PID controller, addressing vertical drift issues that the original approach handled differently.



(a) Circle tracking

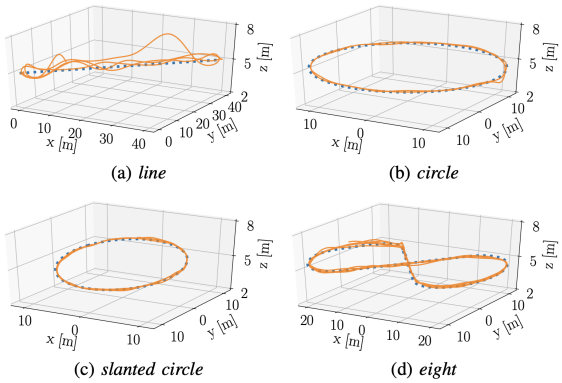


(b) Figure-8 tracking



(c) Slanted-circle tracking

Fig. 4: Final MPPI+PID tracking over multiple loops without drift.



(a) *line* (b) *circle*

(c) *slanted circle* (d) *eight*

Fig. 5: Paper's tracking results.

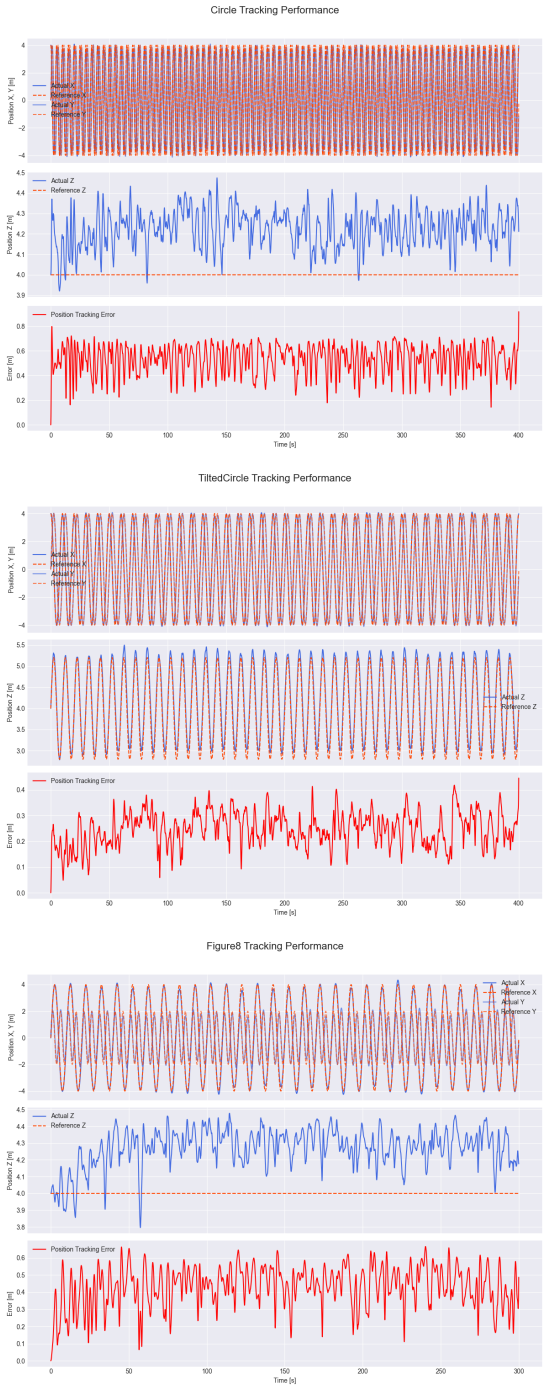


Fig. 6: End-to-end tracking error on (a) Circle, (b) Tilted-Circle, (c) Figure-8.

- **Maximum Performance:** The original implementation demonstrated speeds up to 44 km/h and accelerations near 20 m/s², while my implementation achieves 10-20 km/h while maintaining comparable tracking accuracy.
- **Obstacle Avoidance:** Both implementations support obstacle avoidance, but the original work conducted more extensive testing with physical obstacles in real-world settings.

b) Key Limitations:

- **Computational Demands:** MPPI's heavy sampling (500 rollouts) requires significant processing power, challenging its onboard viability on smaller platforms.
- **Trajectory-Specific Tuning:** I found that different trajectories required distinct parameter sets for optimal performance, increasing setup complexity.
- **No Obstacle Testing:** I haven't yet tested MPPI's primary advantage—navigating cluttered environments with non-convex constraints [2].

c) Proposed Future Work: With additional time, I would:

1) **Onboard Optimization:** Port my MPPI+PID stack to run entirely on a UAV's onboard computer, exploring the trade-off between rollout count and horizon length as done in the original work.
2) **Gazebo Integration:** Develop a Gazebo plugin to enable in-the-loop testing in a physics-accurate simulation.
3) **Obstacle Avoidance:** Introduce cylindrical and mesh obstacles into the cost function to validate MPPI's non-convex avoidance capabilities, following the approach of Minařík et al. [3].
4) **Adaptive Parameters:** Design an online scheme to adjust noise, weights, and temperature based on tracking error or environment complexity, addressing limitations noted in both implementations.

Recent work has shown that nonlinear MPC approaches can achieve performance comparable to differential flatness techniques for agile UAV flight [5]. My findings with MPPI, consistent with those of Minařík et al., suggest it can be a competitive alternative when properly tuned, especially for scenarios involving obstacle avoidance [2].

Conclusion

In this work, I have successfully implemented a Model Predictive Path Integral (MPPI) controller for UAV trajectory tracking in Python. Through systematic development and refinement, I demonstrated that MPPI can effectively track complex 3D trajectories while respecting physical constraints of the UAV system. My key contributions include:

The implementation of a complete MPPI control framework with 4th-order Runge-Kutta integration for accurate dynamics propagation, custom quaternion metrics for orientation tracking, and progressive reward structures for improved convergence. By introducing a complementary altitude PID controller, I resolved the vertical drift that initially limited performance, enabling sustained trajectory tracking over 30+ loops without degradation.

My trajectory-specific parameter tuning revealed the importance of adjusting exploration noise, cost weights, and rewards based on path characteristics. The Circle trajectory benefited from higher position tracking weights (250.0), while the Figure-8 demanded enhanced velocity tracking (30.0), and the TiltedCircle required balanced altitude control with tighter progressive reward thresholds.

The experimental results validate MPPI's potential for UAV control, achieving position tracking errors below 0.5m across all test trajectories while maintaining speeds of 10-20 km/h. These findings suggest that MPPI offers a viable alternative to traditional MPC approaches, particularly for scenarios where non-convex constraints and obstacle avoidance are required.

Code and Video

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

https://github.com/abhikaluri/EECE5550-Mobile-Robotics/tree/main/mppi_quadrotor

A video demonstration of the MPPI controller performance can be viewed at:

https://youtu.be/QovEdDXMJhE

References

[1] T. P. Nascimento and M. Saska, "Position and attitude control of multirotor aerial vehicles: A survey," Annual Reviews in Control, pp. 129–146, 2019.

[2] 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/ICARCV50220.2020.9305363.

[3] J. Pravitra, K. A. Ackerman, C. Cao, N. Hovakimyan, and E. A. Theodorou, "L1-adaptive MPPI architecture for robust and agile control of multirotors," in 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2020, pp. 7661–7666.

[4] M. W. Mueller, M. Hehn, and R. D'Andrea, "A computationally efficient motion primitive for quadrocopter trajectory generation," IEEE Transactions on Robotics, vol. 31, no. 6, pp. 1294–1310, 2015.

[5] S. Sun, A. Romero, P. Foehn, E. Kaufmann, and D. Scaramuzza, "A comparative study of nonlinear MPC and differential-flatness-based control for quadrotor agile flight," IEEE Transactions on Robotics, vol. 38, no. 6, pp. 3357–3373, 2022.