MPPI for Robust & Efficient Racing

Aman Goel
Carnegie Mellon University
Pittsburgh, USA
amangoel@andrew.cmu.edu

Thomas Detlefsen
Carnegie Mellon University
Pittsburgh, USA
tdetlefs@andrew.cmu.edu

Brian Park
Carnegie Mellon University
Pittsburgh, USA
brianp2@andrew.cmu.edu

***Abstract*—Autonomous driving requires vehicles to make safe decisions quickly while navigating a complex dynamic environment. Autonomous racing faces similar challenges, but at a much higher speed. A higher velocity leads to a heavier reliance on accurate vehicle dynamics and requires faster decision making. To tackle this problem, we propose the implementation of a GPU-enabled MPPI on the F1Tenth autonomous racing platform tested in simulation and the real world. Additionally we propose comparing the performance of MPPI to traditional algorithms such as RRT in terms of lap-time and computational speed. Our implementation is available at: https://github.com/t-detlefsen/f1tenth_mppi**

***Index Terms*—Autonomous Racing, Model Predictive Control, Model Predictive Path Integral Control (MPPI), Rapidly Exploring Random Trees (RRT)**

I. Introduction

Autonomous vehicles in the real-world are expected to handle dynamic, robust environments. These environments include paths with complex geometry, aggressive drivers, and obstacle avoidance. In these scenarios, control algorithms must be computed quickly, while also taking into account vehicle dynamics, in order to safely maneuver through these scenarios.

Our project aims to represent this issue that self-driving vehicles face through autonomous racing. Autonomous racecars must handle difficult environments at extremely high speeds. Due to its ability to efficiently handle complex spaces, Model Predictive Path Integral (MPPI) control is a powerful algorithm for real-time control for autonomous systems. Unlike traditional planning algorithms, MPPI leverages sampling to evaluate numerous trajectories, which helps alleviate the amount of uncertainties that our system faces in dynamic environments. This approach has proven advantageous in high-speed autonomous applications, such as racing vehicles [6], drones [7], and robotic manipulators [8], where rapid decision-making and robustness are paramount.

In autonomous racing scenarios, conventional planning methods like RRT [2] often face limitations because of its efficiency and slow responsiveness to dynamic environments. These methods heavily relies on pre-computation, so it cannot react to real-time changes to the environment. To address these shortcomings, our research explores the efficacy of MPPI by exploiting its computational strengths, particularly its ability to harness parallelization through GPU acceleration. We aim to investigate the efficiency of our MPPI algorithm ran across different types of hardware. It is known to be faster when ran on the GPU, but we also compare its efficiency when ran on the CPU against other planning methods. Ultimately, we seek to leverage MPPI's strengths to enable safer, faster, and more reliable autonomous racing performance.

II. Related Works

Rapidly exploring Random Trees (RRTs) [2] are widely used in robotic motion planning for efficiently navigating high dimensional and complex spaces. By incrementally building a tree through random sampling, RRTs can quickly explore feasible paths toward a goal, making them suitable for environments with dynamic constraints. While RRTs are simple, flexible, and probabilistically complete, they often produce suboptimal paths and require parameter tuning and post processing for refinement. Variants like RRT* [3] have been developed to address these limitations and improve path quality.

A* [5] is a classic graph based path planning algorithm that guarantees optimal paths using a cost function combining path cost and heuristic estimates. It performs well in grid based and structured environments but struggles in high dimensional or continuous spaces. In such cases, sampling based methods like RRT or probabilistic approaches like MPPI are more efficient.

There has also been many works [6] that have applied MPPI on autonomous race cars. This work targets the problem of multi-vehicle interactions using a modified version of MPPI, called Best-Response MPPI (BR-MPPI).

Arruda et al. [8] applied Model Predictive Path Integral (MPPI) control to robustly plan push manipulations using learned forward models. By leveraging uncertainty aware learning methods like Gaussian Processes and Mixture Density Networks, their planner actively avoids regions of high model uncertainty.

Model Predictive Path Integral (MPPI) control has garnered increasing attention for its ability to handle nonlinear dynamics and high-speed, constrained planning tasks in real time. Williams et al. [10] introduced a foundational MPPI-based framework for aggressive driving, demonstrating real-time trajectory optimization on a fifth-scale autonomous rally car using GPU-accelerated stochastic sampling. Their formulation merges information-theoretic principles with stochastic optimal control, enabling aggressive maneuvers without reliance on predefined trajectories.

Building on this, Testouri et al. [11] proposed a refined MPPI framework aimed specifically at autonomous driving systems, emphasizing safety critical scenarios. Their approach

introduces obstacle-aware cost functions and utilizes circular approximations to represent obstacles in real-time planning. Validated on a full sized autonomous KIA Soul EV, the method proved capable of executing smooth merges, obstacle avoidance, and vehicle following behaviors, demonstrating safety and feasibility under real world constraints.

These contributions underscore the potential of MPPI as a robust, scalable solution for both racing and urban autonomous scenarios. Our work seeks to extend these foundations by applying MPPI to high-speed autonomous racing on the F1Tenth platform, comparing its performance to classical planners like RRT under constrained computational and dynamic conditions.

III. Approach

MPPI is a sampling-based form of MPC which solves an optimization problem, takes the first action, and repeats this process for every time step. The MPPI algorithm is summarized in **Fig. 1** which shows the race line and environment map as inputs. The algorithm consists of a trajectory generator which feeds trajectories to the cost function which returns the best control action. Each of these components will be described in further detail below. A major advantage of MPPI is that it is highly parallelizable. We tested by attempting to accelerating the algorithm on the GPU. The overal process is summarized in **Fig. II**

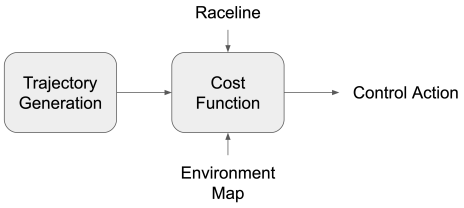

Fig. 1. Overview of the MPPI algorithm. Trajectories are generated and measured using a cost function which relies on a race line and environment map to compute the best control action.

A. Inputs and Outputs

• **Race line** - MPPI functions as a local planner which tracks a global race line. The race line consists of position (x, y), orientation yaw, and velocity v.

• **Environment Map** - A representation of the environment is required to determine which areas are drivable. This is represented by an binary occupancy grid where areas are marked as free or obstacles. Cells in the occupancy grid may also be obstacle probabilities or a lower cost can be added around obstacles to encourage clearance.

• **Control Action** - The goal of MPPI is to determine the control action for the next time step. In the case of the F1Tenth autonomous racing platform, the control message is a velocity v and steering angle ω.

B. Trajectory Generation

Trajectories are generated by applying zero-mean gaussian noise ϵ to a control input sequence u which controls the steering input ω and velocity v. The standard deviations σ_v and σ_ω are tunable parameters which control how far the trajectories deviate from the original control input sequence. Each of the control input sequences are clamped to ensure that the trajectories are physically feasible.

We utilize the kinematic bicycle model updated by euler step where the state space is $X = [x, y, \theta]$ and control space is $U = [v, \omega]$, where x, y are coordinates in 2D space, v is velocity, θ is heading angle, ω is steering angle, and L is the wheelbase. The state transition for the model is as follows:

$$\begin{aligned} \dot{x} &= v\cos(\theta) \\ \dot{y} &= v\sin(\theta) \\ \dot{\theta} &= v\frac{\tan\omega}{L} \end{aligned} \tag{1}$$

C. Cost Function

For each point in the trajectory, a quadratic cost is applied to encourage conformity to the given race line. The closest point in the race line is used as a reference point for position (x, y), orientation (yaw), and velocity (v). A high cost is applied for any point in the trajectory that overlaps with the environment map. The environment map is dilated to encourage obstacle avoidance and reduce scrapes. The cost function below is applied to every step in the trajectory for the stage cost, as well as the terminal cost at the last step to encourage the vehicle to end on the race line where k is the trajectory and t is the timestep.

$$\begin{aligned} S(k) = Q_f(z_{T,ref} - z_T)^2 + Q_{collision} is\_collided(z_T) + \\ \sum_{t=0}^{T} Q(z_{t,ref} - z_t)^2 + Q_{collision} is\_collided(z_t) \end{aligned} \tag{2}$$

D. Control Action

Each of the control input sequences are weighted based on the optimal information theoretic control law described in eq. 22 of [9] and shown belos. Where each control input sequence is weighted by the cost of the minimum cost input sequence and the weighting is controlled by a parameter λ which is more restrictive as it gets smaller. A low $lambda$ creates a highly reactive control algorithm, with lower stability and a high $lambda$ gives greater stability, but with less reaction. These weights are used to compute a weighted average of noise to add to the original control input sequence which is updated and used for the next control action.

$$\begin{aligned} w(V) = \frac{1}{\eta} exp(-\frac{1}{\lambda}(S(V) + \lambda \sum_{t=0}^{T-1} (\hat{u_t} - \tilde{u_t})^T \Sigma^{-1} v_t)), \\ u_t = \mathbb{E}_{\mathbb{Q}_{\tilde{U},\Sigma}}[w(V)v(t)] \end{aligned} \tag{3}$$

IV. Experiments

We implemented MPPI using three different methods. We used NumPy on the CPU, CuPy on the GPU, and PyTorch on the GPU.

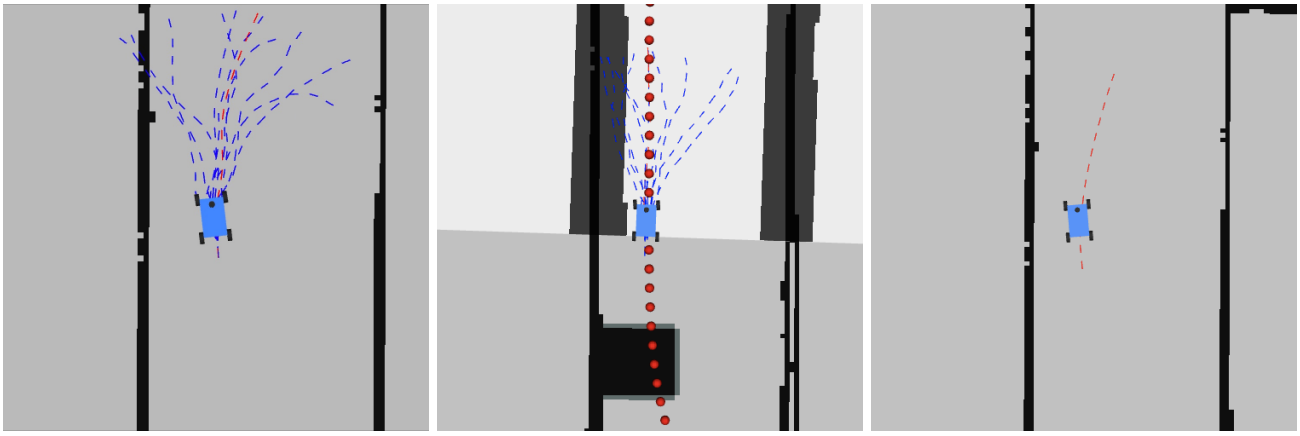

Fig. 2. A visualization of the MPPI algorithm on the F1Tenth vehicle in simulation. In the first image multiple blue trajectories are rolled out using the KBM model. These trajectories are evaluated as shown in the second image based on their deviation from the raceline and collision with obstacles denoted by the black and white overlay. In the third image weighted sum is performed for each trajectory to produce the final trajectory in red.

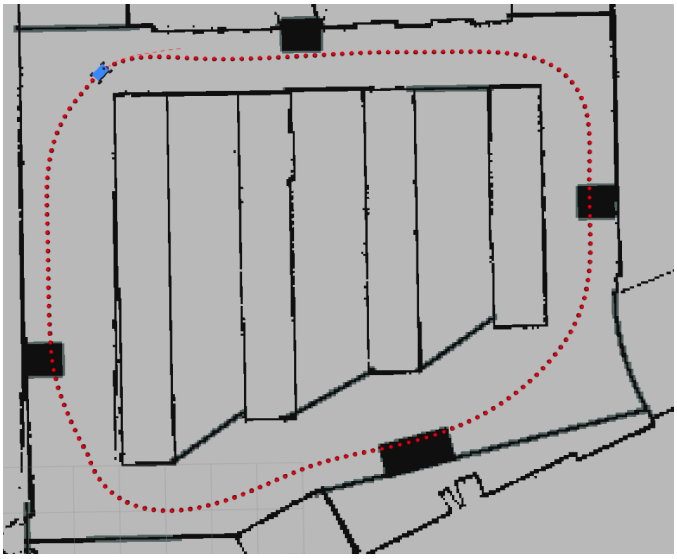

Fig. 3. AIMS lockers map with obstacles.

The metrics that we use to evaluate our performance will be the laptimes of RRT and MPPI. We will use our map of the locker area outside of AIMS for both simulation and real-world environments. For the racetrack, we will test our RRT and MPPI algorithms on a clear, empty racetrack around the AIMS lockers. We also place obstacles along the racetrack to demonstrate obstacle avoidance for both environments. An image of the simulation obstacles is shown in Fig. 3. For the real-world environment, we placed boxes to block the reference raceline that the car would usually take.

The second performance metric that we will evaluate is computation time. We believe that an advantage of MPPI is that we are able to parallelize the heavy computation on the GPU. Therefore, we will measure the publishing frequency of RRT, MPPI on the CPU, and the two implementations of MPPI on the GPU.

V. Results

A. Lap time

Based on the results in Table I, we can see that MPPI was able to achieve a faster lap time than RRT for both racetracks with/without obstacles. These values were measured based on the performance in the real-world environment. From the videos, we can see that the overall movement of MPPI is much smoother than that of RRT. Additionally, MPPI is able to avoid obstacles in a much more efficient way than RRT, which takes lots of unnecessary turns.

B. Computation time

The computation times for all the methods that we implemented are shown in Table II. The values represent the publishing frequency, or how many times our algorithm published an AckermannDriveStamped message to the car. The parameters that we varied for these experiments were the number of MPPI trajectories we generated and the number of RRT iterations. Based on these values, we show that for the same number of MPPI trajectories and RRT iterations, the CPU MPPI implementation achieves a slightly higher frequency than RRT. This may be because RRT has a higher algorithmic time complexity than the vectorized version of MPPI, since we have to construct and search a tree of N nodes.

Our GPU implementations on CuPy and PyTorch were not able to speed up the computation time from the CPU MPPI. In order to transfer our CPU MPPI onto the GPU, we made sure to convert all of the large matrices into CuPy/PyTorch data structures. This included the matrices that stored the optimal raceline reference points, the noise we generated to sample trajectories, and the state of the robot based on the sampled trajectories. With these data structres on the GPU, we were able to perform computations in parallel. We believe that the main reason for the performance decrease is the overhead caused by transferring data from the CPU to the GPU. The advantages of the GPU over CPU operations appears once we have very large data structures that we can operate on in batch.

In MPPI, we are working with a relatively low number of trajectories and trajectory steps, so the overhead of transferring data to the GPU dominates and creates a bottleneck.

Furthermore, we tested MPPI by generating a large number of trajectories, as shown in Table III. Based on this table, we can see that with generating 1,000 trajectories, the NumPy method suffers from a massive performance decrease, whereas the CuPy and PyTorch publishing frequency remains relatively constant from the Table II values. With 10,000 trajectories, the CPU MPPI continues to slow down and the fastest method becomes the PyTorch-based MPPI implementation. This demonstrates that the advantage of MPPI occurs when we are operating with a very large number of trajectories. With this in mind, however, we noticed that a higher number of trajectories does not correlate to better racing performance. The simulation and real-world experiments were only using 100 sampled trajectories. Therefore, we believe that for this domain of autonomous racing, a CPU-based MPPI algorithm with a relatively low number of trajectories is optimal.

Method	**Time (seconds)**
MPPI	23.83
MPPI with Obstacles	25.33
RRT	42.93
RRT with Obstacles	43.91

TABLE I
LAP TIMECOMPARISON OF MPPI AND RRT WITH AND WITHOUT OBSTACLES

Method	**50 traj/iter**	**100 traj/iter**	**150 traj/iter**
MPPI (CPU)	35.515 Hz	23.658 Hz	18.196 Hz
MPPI (CuPy)	3.771 Hz	4.265 Hz	3.996 Hz
MPPI (PyTorch)	4.339 Hz	4.303 Hz	4.546 Hz
RRT	34.211 Hz	15.411 Hz	8.616 Hz

TABLE II
COMPUTATION TIME (PUBLISHING FREQUENCY) COMPARISON.

Method	**1000 Trajectories (Hz)**	**10000 Trajectories (Hz)**
NumPy	4.381	0.776
CuPy	1.843	0.327
PyTorch	4.302	1.947

TABLE III
PUBLISHING FREQUENCY FOR A LARGE NUMBER OF MPPI TRAJECTORIES

VI. CONCLUSIONS & FUTURE WORK

This work serves as a baseline implementation of MPPI on the F1Tenth vehicle. Many improvements could be made to improve the performance of the algorithm and push the limits further. Such as better modeling and prediction through the dynamic bicycle model and fourth order Runge-Kutta prediction to create more accurate predictions while accounting for forces on the vehicle. A term could also be added to the cost to limit control effort. A trajectory library could be implemented to pre-compute control inputs and trajectories while maintaining a reasonable distribution of trajectories. The cost map could also be represented by a gradient to encourage trajectories to avoid obstacles better. Finally, the parallelization could be further explored on the GPU to enable faster performance.

We demonstrate that MPPI not only yields faster and smoother lap times than RRT, but also runs more efficiently on a CPU for the moderate trajectory counts ($\sim$ 100) used in practice. We found that GPU-accelerated MPPI only outperforms the CPU implementation at very large trajectory counts ($\sim$ 10,000). However, generating lots of trajectories does not necessarily translate to better racing performance. We believe that deploying MPPI in high-speed autonomous racing contributes in pushing the limits of model-based stochastic control under extreme dynamics, which can accelerate advances in safety-critical applications.

REFERENCES

[1] https://github.com/f1tenth/f1tenth_racetracks

[2] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings , San Francisco, CA, USA, 2000, pp. 995-1001 vol.2, doi: 10.1109/ROBOT.2000.844730.

[3] Karaman S, Frazzoli E. Sampling-based algorithms for optimal motion planning. The International Journal of Robotics Research. 2011;30(7):846-894. doi:10.1177/0278364911406761

[4] G. Williams, P. Drews, B. Goldfain, J. M. Rehg and E. A. Theodorou, "Aggressive driving with model predictive path integral control," 2016 IEEE International Conference on Robotics and Automation (ICRA), Stockholm, Sweden, 2016, pp. 1433-1440, doi: 10.1109/ICRA.2016.7487277. keywords: Trajectory;Optimal control;Entropy;Vehicles;Prediction algorithms;Q measurement;Stochastic processes,

[5] P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968, doi: 10.1109/TSSC.1968.300136.

[6] G. Williams, B. Goldfain, P. Drews, J. M. Rehg, and E. A. Theodorou "Autonomous Racing with AutoRally Vehicles and Differential Games", 2017

[7] J. H. Yang and H. D. Choi, "Cost-Based MPPI: Enhancing the Efficiency of MPPI Controllers in 3D Space for UAV Control," 2024 24th International Conference on Control, Automation and Systems (ICCAS), Jeju, Korea, Republic of, 2024, pp. 147-152, doi: 10.23919/ICCAS63016.2024.10773101.

[8] E. Arruda, M. J. Mathew, M. Kopicki, M. Mistry, M. Azad, and J. L. Wyatt authored the paper "Uncertainty averse pushing with model predictive path integral control". 2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids), doi:10.1109/humanoids.2017.8246918

[9] Williams, G., Drews, P., Goldfain, B., Rehg, J. M., & Theodorou, E. A. (2017). Information Theoretic Model Predictive Control: Theory and Applications to Autonomous Driving. https://arxiv.org/abs/1707.02342

[10] G. Williams, P. Drews, B. Goldfain, J. M. Rehg and E. A. Theodorou, "Aggressive driving with model predictive path integral control," 2016 IEEE International Conference on Robotics and Automation (ICRA), Stockholm, Sweden, 2016, pp. 1433-1440, doi: 10.1109/ICRA.2016.7487277.

[11] Mehdi Testouri, Gamal Elghazaly, and Raphael Frank. Towards a Safe Real-Time Motion Planning Framework for Autonomous Driving Systems: An MPPI Approach. *arXiv preprint arXiv:2308.01654*, 2024. https://arxiv.org/abs/2308.01654.