Model Predictive Path Integral (MPPI) Control for Remote Control Cars

Dominique Escandon Valverde
Mechanical Engineering
Carnegie Mellon University
descando@andrew.cmu.edu

Sahil Chaudhary
Mechanical Engineering
Carnegie Mellon University
stchaudh@andrew.cmu.edu

Darwin Mick
Mechanical Engineering
Carnegie Mellon University
dmick@andrew.cmu.edu

***Abstract*—The aim of this project is to implement a Model Predictive Path Integral (MPPI) control algorithm for an RC car platform, and compare its performance with an existing Iterative Linear Quadratic Regulator (iLQR) controller. Traditional approaches like iLQR involve decoupling the planner and the controller. However, MPPI couples both of them, eliminating the need for a separate planner. The iLQR implementation uses the FALCO planner, which uses a fixed set of pre-generated paths that are computed offline. In contrast, our MPPI implementation randomly samples control sequences and rolls them out, which can lead to paths that could not have been achieved by the FALCO planner. This enables the MPPI to attain more aggressive paths. In this report, we show our implemented MPPI controller outperforming our system's current iLQR/FALCO control/planning stack.**

***Index Terms*—Model Predictive Control, Iterative Linear Quadratic Regulator, Model Predictive Path Integral**

I. Introduction

Autonomous driving has taken the world by storm in the past decade. Companies like Tesla, Rivian, Waymo, Uber, and many others are continuing to invest in self-driving technology. As such, autonomous driving has been a large focus of research across academia. In 2007, Carnegie Mellon University laid its claim as the premier institution for autonomous driving by competing in and winning the DARPA Urban Challenge [1]. In this competition, many of the techniques and standard approaches used for planning and control of autonomous vehicles were first introduced and tested.

Common planning and control stacks separate control and planning into two distinct parts where the planner will find a feasible path and the controller will help the car follow that path [1], [2]:

1) Planning: Planners find possible paths through a given environment. In autonomous driving, there are usually two layers of planners. First, a global planner that will help construct longer paths through an entire course or environment. Global planners usually run at a lower rate but may give some guarantee on optimality (i.e. A*, D*, RRT*). Whereas local planners (the focus of this work) plan "around" the car and primarily seek to avoid obstacles. To this end, lattice-based planners are commonly used [3].

2) Control: Once a feasible path is found by the local planner, the controller will control towards that path. Model Predictive Control (MPC) has been very useful in this regard for cars [4]. This is an optimal controller that calculates a control sequence over a finite horizon and then executes a single step. After step execution, the controller recalculates a control sequence over the finite horizon, and then executes a command once more.

Fig. 1. Project RC Cars

In more recent years, the focus on aggressive off-road autonomy has increased. Programs like DARPA Racer have accelerated academic research in this domain. In doing so, some traditional planning and control methods hit their limits due to challenging new terrain. For example, finding aggressive feasible paths that systems can avoid can be incredibly difficult [5]. As such, new methods have been derived to increase the speed and versatility of off-road autonomy systems.

To this end, Model Predictive Path Integral (MPPI) control was introduced to fuse the planning and control steps [5]. It does so by combining planning and control by sampling control sequences and rolling out paths for each sequence. Then, a final control sequence is calculated by averaging sampled sequences weighted by resultant path costs. Like traditional MPC, the first control is executed before running the full sequence again.

Our team thought this would be very interesting to study because it relates to ongoing projects in our lab. We have a multi-robot system of RC cars (Fig. 1) that we drive to explore tunnels for military and search and rescue applications.

Currently, each car is controlled with an iLQR controller with the FALCO local planner running on the vehicles [6]. Our current system is functional, but we were interested in running MPPI on our system to see how it compares to the current autonomy stack. As such, in this work, we have developed MPPI for a simulated RC Car, and compared it to an iLQR and FALCO planning stack.

II. Related Work

In [5], MPPI is first introduced. The relationship between free energy and relative entropy is established to prove the optimality of MPPI. The controller is then implemented on a fifth-scale auto-rally vehicle, and GPU was leveraged to adequately sample the full control space and solve the weighted cost-average problem in real-time. Since the original publication, MPPI has had many extensions. One such of these extensions enabled MPPI to work for non-affine dynamics [7].

Finally a multitude of extensions to increase robustness of the algorithm and make systems better follow paths has been introduced. For example, Tube-MPPI adds a second optimization layer that uses Differential Dynamic Programming (DDP) as another tracking layer on the MPPI optimization [8]. Robust-MPPI extends this work by adding information from the DDP controller into the MPPI step [9]. Additional methods build off of Tube-MPPI by adding Constrained Covariance Steering control algorithms to the controller, thus making the car more closely follow selected paths [10], [11].

Other extensions in MPPI seek to better generate paths. One such of these methods smooths paths [12]. Additionally, work is being done to extend MPPI into uncertain cases, where people have looked at planning in partially observable environments [13].

III. Contributions

Our contribution is to implement MPPI in our autonomous robot simulation stack. We run a comparison between our MPPI implementation and the FALCO/iLQR stack currently implemented in our system. We compare the performance of the controllers on the following metrics: control accuracy and time taken to achieve a way point.

IV. Methodology

This section will briefly describe the simulation environment in which we did all of our testing. Then, it will discuss the Kinematic Bicycle Model that we used to model our system and give a brief background into the existing work on our stack, i.e., the FALCO planner and the iLQR. Finally, we delve into detail on MPPI and implementation details of our controller.

A. Simulation Environment

We are running a Gazebo simulation of an RC Car with Ackermann steering. Finally, using the simulation environment, we have complete observability of our current state. We leverage the simulation to send goal way points to the controller. We can send a series of consecutive way points for the car to follow. This series of way points represents the desired trajectory that we want the car to follow.

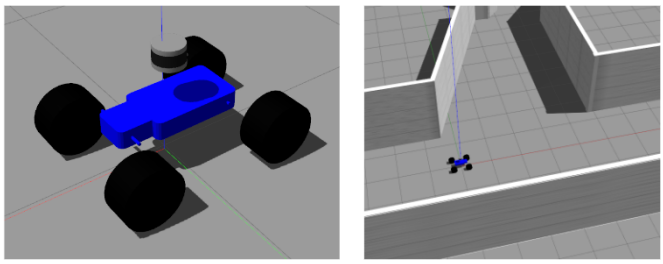

Fig. 2. Vehicle in Gazebo virtual world

B. Kinematic Bicycle Model (KBM)

Kinematic Bicycle models are often used to simplify the complex model of a car by approximating it as a bicycle with non-holonomic constraints on the wheels [2]. A common parameterization of the bicycle model is depicted below in Fig. 3.

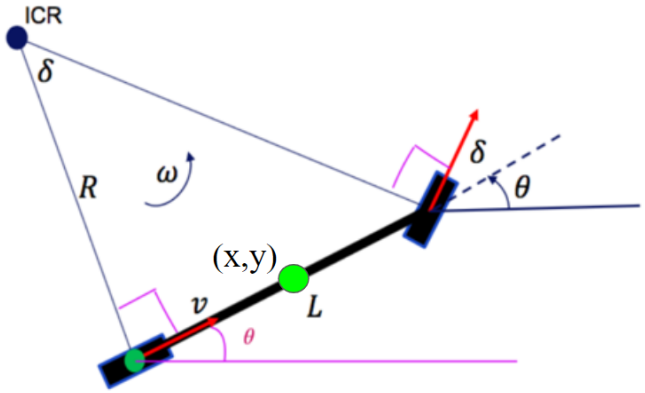

Fig. 3. Kinematic Bicycle Model

Using the above parameterization we define the state of the car as its position (x, y), its yaw (θ), and its velocity (v). The commanded inputs are the velocity (v) and the steering angle (δ). Finally, our goal state is the x, y position of the car.

$$\begin{aligned} x &= \begin{bmatrix} x & y & \theta & v \end{bmatrix}^\top \\ u &= \begin{bmatrix} v & \delta \end{bmatrix}^\top \\ goal &= \begin{bmatrix} x & y \end{bmatrix}^\top \end{aligned} \tag{1}$$

As such, all state and control inputs are elements of $\mathbb{R}^1$, and L is the length of the bicycle, we can write the state update equations as follows, where Δt is our timestep:

$$\begin{aligned} x_{k+1} &= v_{in} \cos(\theta_k) \Delta t \\ y_{k+1} &= v_{in} \sin(\theta_k) \Delta t \\ \theta_{k+1} &= v_{in} \tan(\delta_{in}) \frac{\Delta t}{L} \\ v_{k+1} &= v_{in} \end{aligned} \tag{2}$$

C. Prior Work

As mentioned above, we are leveraging systems already in our current stack. These tools will be used as a baseline to compare our MPPI implementation. Our current stack uses a FALCO planner to select a reference path, then uses the iLQR to follow that path.

1) FALCO Planner: Our FALCO implementation uses the Kinematic Bicycle Model above to build a parameterized set of possible paths offline. Paths are parameterized by speed. So, in the planning step, using the estimated robot speed, a set of paths is rolled out in "front" of the current estimated position of the robot. Assuming deterministic knowledge of obstacles, paths that intersect obstacles are removed. Finally, the path that minimizes distance to the goal is selected and used as the reference path. The generated paths and the selected path can be seen below in Fig. 4 where the generated paths are red and the selected path is green.

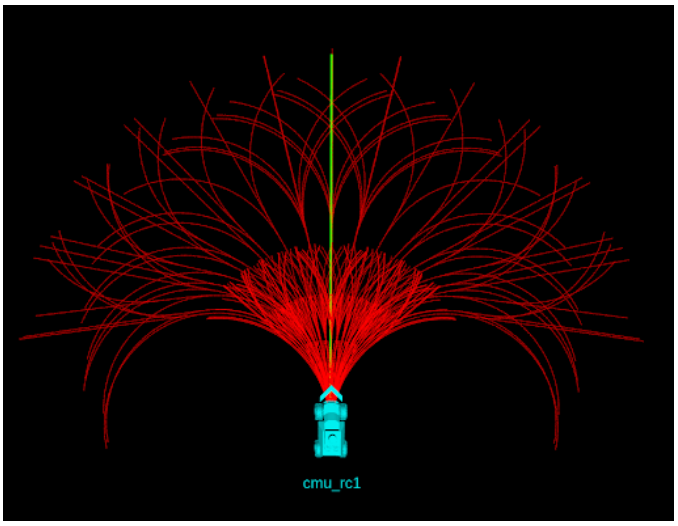

Fig. 4. Visualization of pre-generated reference paths (red) and FALCO selected path (green).

2) iLQR: iLQR is used to control the car to the reference path. This allows for optimal control to the paths generated by the nonlinear KBM (Eqn. 2). Additionally, this supports inequality constraints on steering and velocity inputs in our system. At the rate of the system update, the system will receive a new path from the FALCO planner, and rerun the iLQR controller.

D. Model Predictive Path Integral (MPPI) Control

Rather than including a planning and control step, MPPI combines these steps. MPPI does this by rolling out a sequence of paths. To roll out one path, MPPI samples a sequence of control inputs over the steps. Then, using a predefined cost function (Eqn. 3), the cost of each rollout is calculated. This process occurs for N number of rollouts. Then, the final control sent to the system is calculated using the weighted average of all control inputs, where the weight is based directly on the cost of the path. Mathematically, this is demonstrated in the pseudocode reported in Algorithm 1. Here, an additional hyperparameter, λ, is introduced to further influence the weighted average. For our purposes, we set $\lambda = 1$.

Algorithm 1: MPPI Pseudocode

Given : K: Number of samples;
N: Number of timesteps;
($\mathbf{u}_0$, $\mathbf{u}_1$, ... $\mathbf{u}_{N-1}$): Initial control sequence;
A, **B**: System dynamics;
λ: Cost Parameters;
$\mathbf{u}_{init}$: Value to initialize new controls to;

while *task not completed* **do**
  Calculate cost per path, $\tilde{S}_k$, from sampled control, $\tilde{\mathbf{u}}_i$
  for $k \leftarrow 0$ ***to*** $K-1$ **do**
    $\mathbf{x} = \mathbf{x}_{t_0}$;
    for $i \leftarrow 1$ ***to*** $N-1$ **do**
      $\mathbf{x}_{i+1} = \mathbf{A}\mathbf{x}_i + \mathbf{B}\tilde{\mathbf{u}}_i$
      $\tilde{S}_k = \tilde{S}_k + calculate\_cost(\mathbf{x}_i, \mathbf{u}_i)$
    end
  end
  Average control input based on weight
  for $i \leftarrow 1$ ***to*** $N-1$ **do**
    $\mathbf{u}_i = \sum_{k=1}^{K} \left(\frac{exp(\frac{-1}{\lambda}\tilde{S}_k)\mathbf{u}_i}{\sum_{k=1}^{K} \exp(\frac{-1}{\lambda}\tilde{S}_k)} \right)$
  end
  send_to_actuators($\mathbf{u}_0$)
  Update current state after receiving feedback;
  Check for task completion;
end

In MPPI, the nonlinear system dynamics (Eqn. 2) are directly calculated for each time step. Thus, for each path, we can directly roll out the system parameters to create our reference path, and the final x, y position in the rollout is $\mathbf{x}_{ref}$. Since we are only concerned about reaching the goal x and y position, with no regard to yaw, our cost is defined as:

$$J = (\mathbf{x}_{ref} - \mathbf{x}_{goal})^\top \mathbf{Q} (\mathbf{x}_{ref} - \mathbf{x}_{goal}) + \mathbf{u}^\top \mathbf{R}\mathbf{u} \tag{3}$$

where $\mathbf{x}_{ref} \in \mathbb{R}^{2x1}$, $\mathbf{x}_{goal} \in \mathbb{R}^{2x1}$, $\mathbf{u} \in \mathbb{R}^{2x1}$, $\mathbf{Q} \in \mathbb{R}^{2x2}$, $\mathbf{R} \in \mathbb{R}^{2x2}$. We chose this metric for cost because we can individually penalize the x and z error through Q. It would have also been reasonable to use an Euclidean distance metric to penalize the cost. One advantage of MPPI is that our cost function does not have to be quadratic. However, squaring the terms is necessary in this case so our cost stays positive. Examples of generated paths are shown in Fig. 5.

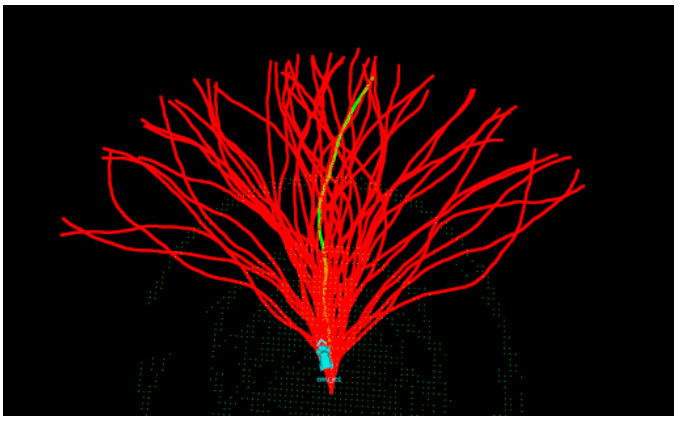

Fig. 5. Visualization of sample-based roll outs (red) and ultimate MPPI-executed path (green).

We implemented second-order constraints on the controls. Perturbations were executed on throttle and steering effort and multiplied by the maximum rate of change of the controls and the time step of the rollout to more simply implement second-order constraints on the control inputs. Additionally, to encourage backward driving in tight corridors (rather than completing a forward turn-around), we ensured that a minimum number of rollouts were executed with negative acceleration. Examples of the paths after second order smoothing, planned over a shorter horizon are shown in Fig. 6.

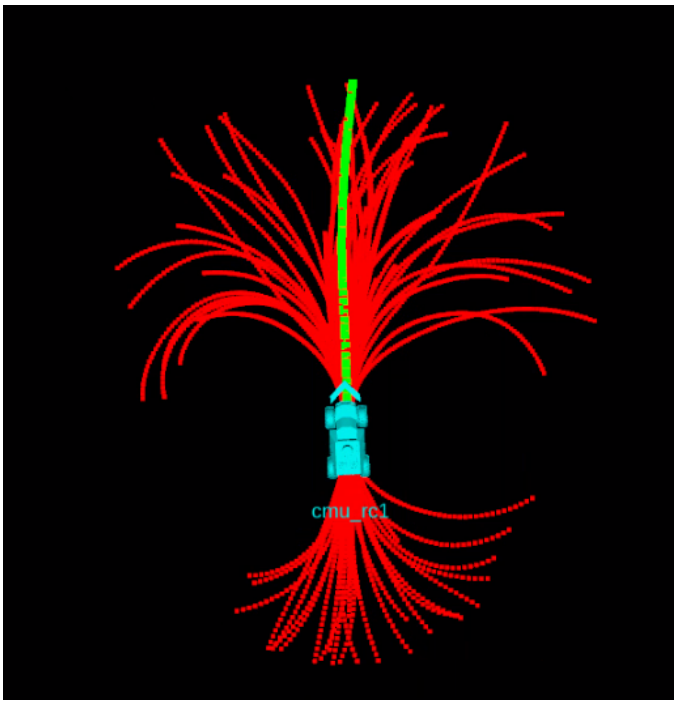

Fig. 6. Visualization of sample-based roll outs (red) and ultimate MPPI-executed path (green), after additional constraints.

The paths shown in Fig. 5 involve no clamping, so they don't take into consideration the actuator limits and the maximum acceleration and steering rate that is physically possible. As a result, these paths are more widely spread out. In contrast, the paths shown in Fig. 6 have first order and second order clamping. Since these constraints take into consideration the actuator limits and maximum rate of change that is physically possible, they are a lot less widely spread out. Also, the paths facing the rear of the car correspond to negative velocity paths.

V. Results

We compared the results of waypoint following between iLQR and MPPI. We set 8 waypoints at different combinations of x and y positions. We tracked the distance from the final position of the car to the goal (Table I). Additionally, we tracked the time it takes for the car to get to the goal position between the methods (Table II).

TABLE I
Comparison of final position errors.

Waypoint	MPPI Error (m)	ilQR Error (m)
(50, 0)	0.16	1.39
(0, 50)	0.12	1.21
(50, 50)	0.28	1.17
(0, -50)	0.50	1.72
(-50, 0)	0.26	1.70
(-50, -50)	0.60	1.64
(-50, 50)	0.26	1.76
(50, -50)	0.43	1.17

TABLE II
Comparison of time taken to waypoint

Waypoint	MPPI Time (s)	ilQR Time (s)
(50, 0)	15.4	12.3
(0, 50)	14.6	13.2
(50, 50)	18.6	15.8
(0, -50)	15.2	23.1
(-50, 0)	15.0	23.7
(-50, -50)	18.9	31.6
(-50, 50)	18.75	29.8
(50, -50)	18.1	15.9

Overall, our results indicate that the final error of MPPI is lower. Where the average final error over all of the runs is 0.33m, and the average final error for iLQR is 1.47m. On average, iLQR took a bit longer, with an average of 20.67 seconds to completion while the time for MPPI was only 16.81 seconds.

Over the course of one path, we compare the path taken by the car (Fig. 7). We also compare the controlled velocity output (Fig. 8) and the steering output (Fig. 9).

MPPI seems to have taken a straighter and more direct path to the waypoint. We believe this is because MPPI isn't relying on pre-computed paths and is sampling new paths. So, it can sample a direct path to the waypoint, whereas with iLQR and FALCO, we are restricted to the limited set of pre-computed paths, so it is possible that none of those paths aligned directly with the most direct path to the waypoint.

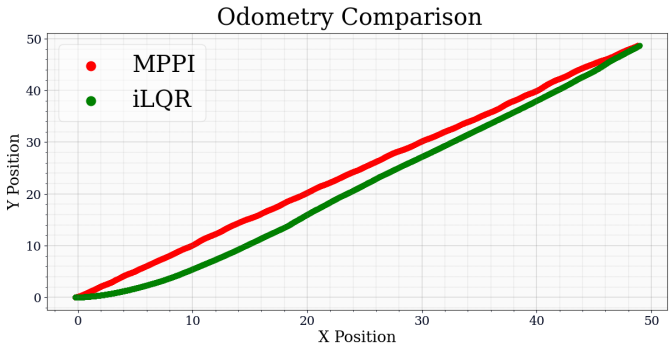

Fig. 7. Velocity comparison of MPPI and iLQR.

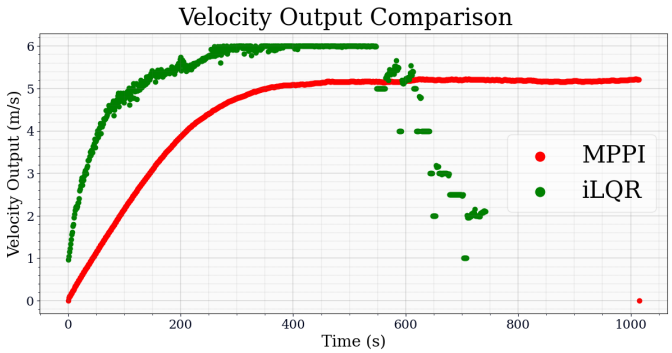

Fig. 8. Velocity comparison of MPPI and iLQR.

VI. Discussion

We suspect that one reason there is lower error from MPPI is that we have more dynamic paths. This allows us to generate paths that move more directly towards the waypoints. On the other hand, FALCO-generated paths may not align directly with the way point because they are static and pre-computed. This behavior is exhibited in 7, where we can see the position of the car from the MPPI travel more directly to the waypoint when compared to iLQR.

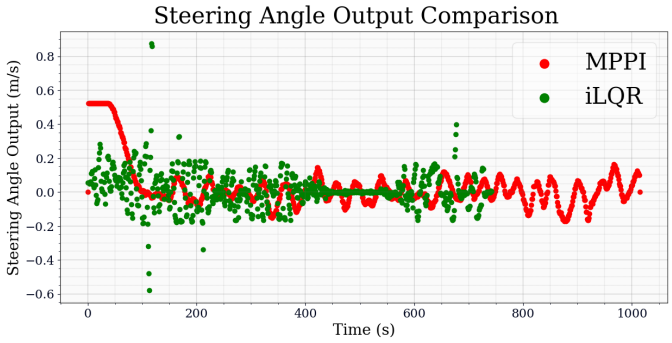

Fig. 9. Steering angle comparison of MPPI and iLQR.

It is worth noting that there are some implementation details that could also result in this behavior. For example, there are hard coded constraints that disable the controller near a goal position. This is to enable the system to stop moving, rather than oscillate infinitely around the goal position. It is possible this constraint is looser for the iLQR controller, meaning the controller stops sooner than in our MPPI.

Additionally, the cost function between the two methods is calculated differently. In the FALCO planner, the path with a final point that has the shortest Euclidean distance to the goal is selected, whereas MPPI's cost is based directly on x and y positions. Additionally, the hyperparameter tuning between the two systems is different, as such it is possible that different Q and R matrices are impacting the performance. However, since our team has been using iLQR for several years now, we are comfortable making the assumption that the iLQR has been tuned to work its best in the software stack. So, we are satisfied with the comparison we have thus far, and the performance of our MPPI controller.

There are also differences in the commanded velocity and steering angles between the two controllers. It is apparent that the MPPI never quite hits the goal speed of 6 m/s. As such, it is likely that if we decreased our control cost, we would see more aggressive velocities. On the other hand, the iLQR implementation hits the target speed more rapidly, then decreases. This decrease is the result of other engineering decisions in the iLQR implementation that decelerate the car as it gets closer to a waypoint, whereas the MPPI has a zero velocity output at the last step.

Next, there are also differences in the commanded steering angle. In the MPPI controller, the steering angle is oscillatory. This behavior is unideal, as such we may want to increase the control cost of the steering angle to try and better damp the response. However, we can see that the iLQR output is not smoother nor less oscillatory. It is likely these controllers, in the current tune, may have a hard time in real world scenarios since they are commanding steering angle changes that may not be physically possible.

A major issue in our current implementation is that we rarely drive backward. We believe this is due to our sampling about the last control input because we do generate backward paths (Fig. 6). Since our next control commands are always sampled about the previous control input, once the system starts moving forward, it is likely to stay moving forward.

Unfortunately, our system can not avoid obstacles. We had an occupancy grid implementation set up through the stack already, and we wanted to have obstacle avoidance implemented. However, we were unable to integrate this correctly into our cost function. The system can detect obstacles, however, it can not maneuver around them. We gave the particular path that crossed an obstacle an infinite cost, which would in turn imply a close to zero weight given to that path while calculating the weighted average. We thought that this would be equivalent to cancelling the path, since the weight would effectively be zero. However, this approach does not work well. Even though an obstacle is detected and the paths have a high cost, the car still prioritizes turning around (and hence crashing into the obstacle in front) instead of stopping and reversing. We believe that that our weighted averaging is not allowing negative paths to be taken.

VII. Conclusion

This exploratory work sought to investigate the differences between MPPI and FALCO/iLQR for use in autonomous vehicle navigation with obstacle avoidance. Our experiments demonstrate that MPPI has a lesser final error and faster speed when compared to the iLQR. This is because MPPI is able to generate more direct and straight paths to the goal waypoint as compared to iLQR and FALCO, by virtue of sampling controls randomly and generating new and unique paths in real time.

An extension of this work could involve sampling over the knot points on a set of splines for trajectories, instead of directly sampling the control sequence. This could possibly lead to much smoother trajectories, as splines guarantee at least C2 smoothness. This will also be computationally more efficient, as we need to sample a lot fewer points. This is because the number of knot points to be sampled is a lot lesser than all the control inputs over the time horizon.

Current implementations of FALCO and iLQR autonomy are limited by a reliance on reference trajectories. For this reason, the system model for RC vehicles has remained a simple four-state kinematic bicycle model which must be differentiable. MPPI does not require a differentiable system model, which makes it compatible with non-differentiable models which may be necessary with complex neural networks.

One key limitation of this experiment is its lack of real-world testing. As MPPI has a high known computational cost, control executions may likely differ on-vehicle. Parallel computing leveraging GPUs will be necessary for optimal real-time performance on hardware. In real-world environments with high dynamical variance and non-linearity (e.g. changing or slippery terrain), MPPI is more likely to execute optimal trajectories through its broader sampling of control outcomes which inherently assess risk.

VIII. Acknowledgments

We would like to thank Professor Manchester, and all of the TAs for their guidance and support throughout the course. Additionally, we would like to thank Adam Johnson, Burhan Shirose, Rohan Chandrasekar, Joshua Spisak, and the Mattlab for their work with the FALCO planner, iLQR, obstacle map, and simulation support.

IX. Code Repository

All code produced for this project can be found in the following repository:

`https://github.com/dpmick/ocrl_mppi`

References

[1] C. Urmson, J. A. Bagnell, C. Baker, M. Hebert, A. Kelly, R. Rajkumar, P. E. Rybski, S. Scherer, R. Simmons, S. Singh *et al.*, "Tartan racing: A multi-modal approach to the darpa urban challenge," 2007.

[2] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann *et al.*, "Stanley: The robot that won the darpa grand challenge," *Journal of field Robotics*, vol. 23, no. 9, pp. 661–692, 2006.

[3] H.-y. Zhang, W.-m. Lin, and A.-x. Chen, "Path planning for the mobile robot: A review," *Symmetry*, vol. 10, no. 10, p. 450, 2018.

[4] T. M. Vu, R. Moezzi, J. Cyrus, and J. Hlava, "Model predictive control for autonomous driving vehicles," *Electronics*, vol. 10, no. 21, p. 2593, 2021.

[5] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Aggressive driving with model predictive path integral control," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 1433–1440.

[6] J. Zhang, C. Hu, R. G. Chadha, and S. Singh, "Falco: Fast likelihood-based collision avoidance with extension to human-guided navigation," *Journal of Field Robotics*, vol. 37, no. 8, pp. 1300–1313, 2020.

[7] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Information-theoretic model predictive control: Theory and applications to autonomous driving," *IEEE Transactions on Robotics*, vol. 34, no. 6, pp. 1603–1622, 2018.

[8] G. Williams, B. Goldfain, P. Drews, K. Saigol, J. M. Rehg, and E. A. Theodorou, "Robust sampling based model predictive control with sparse objective information." in *Robotics: Science and Systems*, vol. 14, 2018, p. 2018.

[9] M. S. Gandhi, B. Vlahov, J. Gibson, G. Williams, and E. A. Theodorou, "Robust model predictive path integral control: Analysis and performance guarantees," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1423–1430, 2021.

[10] I. M. Balci, E. Bakolas, B. Vlahov, and E. A. Theodorou, "Constrained covariance steering based tube-mppi," in *2022 American Control Conference (ACC)*. IEEE, 2022, pp. 4197–4202.

[11] J. Yin, Z. Zhang, E. Theodorou, and P. Tsiotras, "Trajectory distribution control for model predictive path integral control using covariance steering," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 1478–1484.

[12] T. Kim, G. Park, K. Kwak, J. Bae, and W. Lee, "Smooth model predictive path integral control without smoothing," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 10 406–10 413, 2022.

[13] I. S. Mohamed, G. Allibert, and P. Martinet, "Model predictive path integral control framework for partially observable navigation: A quadrotor case study," in *2020 16th International Conference on Control, Automation, Robotics and Vision (ICARCV)*. IEEE, 2020, pp. 196–203.