Optimizing For Off-Road Navigation

Antoine Bergerault
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
abergera@andrew.cmu.edu

Mukhtar Maulimov
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
mmaulimo@andrew.cmu.edu

Eric Youn
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213
eyoun@andrew.cmu.edu

***Abstract*—This document is summarizing our project on off-road car navigation from cost maps of real-world data. Based on data collected by the AirLab at Carnegie Mellon University, we were able to design a trajectory planner using RRT# and determine optimal controls to follow this trajectory. We experiment with iLQR and MPPI to track the reference trajectory using simple bicycle model dynamics.**

***Index Terms*—off-road navigation, trajectory optimization, RRT, iLQR, MPPI**

I. Introduction

Autonomous off-road driving on complex terrain is a challenging task. In this project, we were interested in the trajectory optimization of a vehicle in an environment derived from real-world data. To describe the vehicle we used bicyle model dynamics, and the data we used for the maps has been collected with a Yamaha vehicle at Gascola, PA by AirLab researchers. This laboratory researchers have generated all maps using various sensors such as cameras, IMU, and Lidar. Once the observations and cost maps are generated, we can derive a near-optimal trajectory by the means of a sample-based planner. In order to track the trajectory in this complex environment, we wanted to compare a sampling-based method called Model Predictive Path Integral (MPPI) to the classic iLQR-LQR pair.

II. Problem description

In this project, an off-road driving Yamaha vehicle has been used as the main data collection platform. The vehicle is equipped with multiple sensors such as Velodyne HDL-64E Lidar, multisense camera, and Novatel IMU. A simple illustration of the vehicle is shown in Figure 1.

Fig. 1. Yamaha vehicle at Gascola, PA test site.

The data has been collected at the Gascola, PA test side outside of Pittsburgh, PA by AirLab researchers. During the data collection, the vehicle has been driven with teleop. The illustration of the environment during the data collection can be observed in Figure 1.

The raw data, specifically Lidar points and IMU data, has been post-processed to generate registered point cloud along with an accurate localization. As a SLAM solution, a state-of-the-art work by [3] has been used.

As a next step, a perception mapping module developed for an off-road driving by the AirLab researchers have been used to build a map. The registered point clouds and the localization states have been used as an input to the perception module to generate map features. In this project, only 3 map features have been used: terrain map, observed map, and object height map.

The perception mapping module is based on 3D voxels. It is 2D grid cells along x-y with configurable number of voxels along the z axis of each cell. In order to populate the 3D voxel mapping, a consecutive registered point cloud is inserted to a corresponding voxel and it updates number of hits, number of pass through, and density of the voxel. When there is a point registered as a hit in a cell, it is marked as an observed (known) cell. Points with no hits will stay as no data. Based on recursive mean neighbor averaging, a certain number of no data (unknown) cells right next to the observed (known) cells are estimated and marked as inferred cells. The observed map is then estimated using these three kinds of cells (observed, inferred, and no data). Next, using the lowest valid voxel elevation data in each grid cell, the terrain map is estimated. The terrain (elevation) map is generated based on Markov Random Field (MRF). Finally, using the highest valid hit voxel elevation data in each grid cell and the terrain map, the object height map is calculated as the difference between the two.

Figures 2, 3, and 4 show the maps used in this project. The map size is 200m x 200m robot-centered with 0.5m of grid size resolution.

a) The object map: It is shown in Figure 2. The object map shows where the lethal obstacles are in the map, and is used to navigate away from them.

b) The observed map: It is shown in Figure 3. The observed map shows where the occluded and not-occluded regions are in the map.

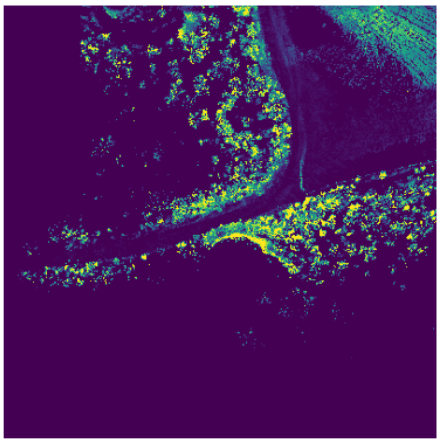


Fig. 2. Object height map. Dark is low, bright is high.

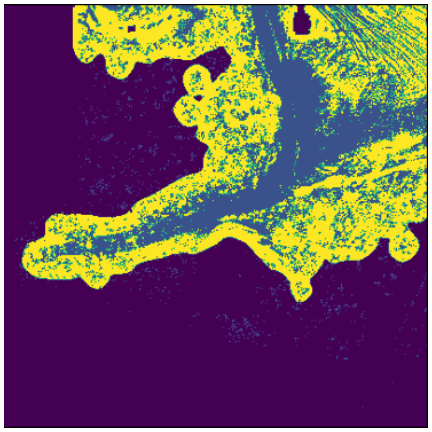


Fig. 3. Observed map. Purple is no data, yellow is inferred, blue is observed.

c) The terrain map: It is shown in Figure 4. The terrain map shows elevation of the terrain. It can be used to calculate slope of the terrain and is important for the vehicle roll-over considerations.

We describe our car with the very classical bicycle model with wheelbase w, whose state and controls are given as follows:

$$x = \begin{pmatrix} p_x \\ p_y \\ \theta \\ v \end{pmatrix}, \qquad u = \begin{pmatrix} a \\ \alpha \end{pmatrix} \tag{1}$$

Where $p \in \mathbb{R}^2$ denotes the position of the car, $\theta \in \mathbb{R}$ its yaw angle, $v \in \mathbb{R}$ its velocity (magnitude), $a \in \mathbb{R}$ its acceleration (magnitude) and $\alpha \in \mathbb{R}$ its steering angle.

In this configuration, the dynamics are then the following:

$$\dot{x} = f(x,u) = \begin{pmatrix} v\cos\theta \\ v\sin\theta \\ v\frac{\tan\alpha}{w} \\ a \end{pmatrix} \tag{2}$$

Fig. 4. Terrain/elevation map.

Whose jacobians are:

$$\frac{\partial f(x,u)}{\partial x} = \begin{pmatrix} 0 & 0 & -v\sin\theta & \cos\theta \\ 0 & 0 & v\cos\theta & \sin\theta \\ 0 & 0 & 0 & \frac{\tan\alpha}{w} \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{3}$$

$$\frac{\partial f(x,u)}{\partial u} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & \frac{v}{w\cos^2\alpha} \\ 1 & 0 \end{pmatrix} \tag{4}$$

We form the discrete dynamics by using a midpoint integration, which, if we denote $x_m \doteq x + f(x,u)\frac{\Delta t}{2}$ is defined as follows:

$$M(f,x,u) \doteq x + f\left(x_m, u\right)\Delta t \tag{5}$$

We also derive explicitly the Jacobians with respect to x and u respectively:

$$\frac{dM(f,x,u)}{dx} = I_n + \Delta t \frac{\partial f\left(x_m,u\right)}{\partial x}\left(I_n + \frac{\Delta t}{2}\frac{\partial f\left(x,u\right)}{\partial x}\right) \tag{6}$$

$$\frac{dM(f,x,u)}{du} = \Delta t\left(\frac{\Delta t}{2}\frac{\partial f\left(x_m,u\right)}{\partial x}\frac{\partial f(x,u)}{\partial u} + \frac{\partial f(x_m,u)}{\partial u} I_n\right) \tag{7}$$

III. RRT PLANNING

In this project, we used RRT# to generate the sub-optimal trajectory between start and goal states. Implementation details can be found in [1]. RRT# is a tree based planning algorithm used to approximate value function at its nodes where we expanded the tree backwards starting from the goal point. In our implementation, dynamic feasibility has not been taken

into account. Instead, each node is expanded along a straight line. Dynamic feasibility is planned to be added as a future work as an extension.

The observed map shown in Figure 3 and the object height map shown in Figure 2 are used as an input to the RRT# algorithm to calculate the approximate value function. The cost function calculation details between two nodes is as follows:

```
cost = segmentLength

for each grid cell along segment:
  cellCost = exp(objectHeightSquare)

  if cell is inferred:
    cellCost += 1.0

  cost += cellCost
```

The cost includes length of the segment between the nodes. Additionally, it will be penalized based on the object height as exponential of square of the object height value at each grid cell. The reason is to highly penalize cells with lethal obstacles while not or slightly penalize the cells with no or low obstacles. Additionally, inferred cells are given an extra penalty. The reason to do it that way is to not encourage the search towards the inferred region but instead towards observed (known) regions.

For the RRT# it has been decided to only include observed and inferred cells as search space. There is no reason to expand the tree towards the no data region. But, in the future, if the goal state lies within the no data region the search space potentially can be expanded to include the no data region as well. Another potential to limit the search space is to exclude cells with high object height (> object height threshold).

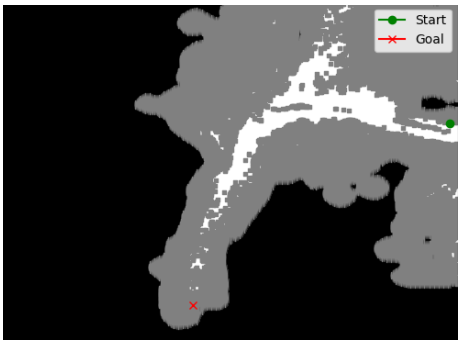


Fig. 5. RRT# search space with start and goal states.

In Figure 5, the RRT# search space along with start and goal states are shown. In the figure, white is observed (known) region, grey is inferred region, and dark is no data region. The white and grey cells are chosen to be the search space for the RRT#.

In Figure 6, the RRT# tree expansion is shown. From the figure, it can be observed that the RRT# optimal path mostly stays along the observed (known) region.

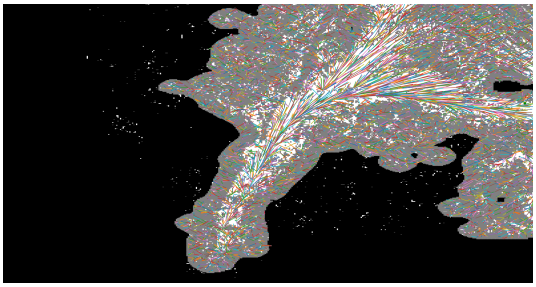


Fig. 6. RRT# tree expansion.

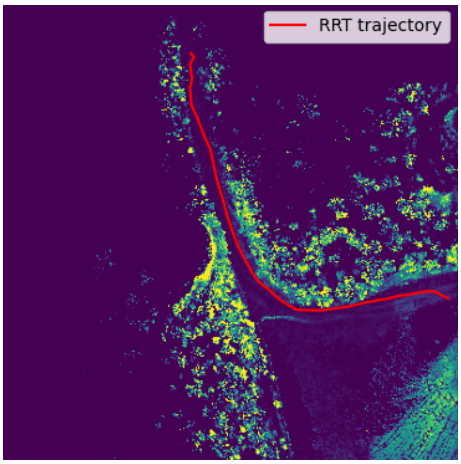


Fig. 7. RRT# trajectory on the object height map.

The Figure 7 shows the sub-optimal RRT# trajectory on the object height map.

RRT# is implemented in C++ and integrated into the the rest of the project (python) via python boost wrapper. In terms of performance:

Iterations	Nodes	Time [ms]
10,000	1200	400-800
20,000	2200	1000-1500
50,000	5500	3400-3800

The RRT# algorithm runs at a slower speed. As a future work, RRT# can get faster incorporating parallel computing for cost calculations between the sample and neighboring nodes and saving some of the computed edge costs for a future re-planning.

A. *Iterative LQR*

Quadratic cost functions are often considered for trajectory optimization problems, as they enable the use of very cheap and well-behaving solvers. LQR is a popular and powerful feedback-based controller that we can easily implement on our car. The iterative LQR or iLQR algorithm is an offline trajectory optimization solver for trajectory tracking under dynamics constraints only.

$$\min_{\substack{\Delta x_{1:N} \\ \Delta u_{1:N-1}}} \sum_{k=1}^{N-1} {\Delta x_k}^T Q \Delta x_k + {\Delta u_k}^T R \Delta u_k + {\Delta x_N}^T Q_f \Delta x_N$$

$$\text{s.t.} \quad x_{ref,k+1} + \Delta x_{k+1} = f(x_{ref,k} + \Delta x_k, u_{ref,k} + \Delta u_k)$$

Given a reference trajectory, it determines a feasible trajectory with associated controls minimizing the given cost. From the values $x_{ref,k}, \Delta x_k, u_{ref,k}, \Delta u_k$, the output trajectory and controls are:

$$x_k = x_{ref,k} + \Delta x_k$$
$$u_k = u_{ref,k} + \Delta u_k$$

To solve this problem, we used the standard approach of iteratively solving backward the state-value and action-value functions to satisfying the Bellman principle of optimality.

First, the state-value function or cost-to-go is defined as the cost associated to a given state assuming we follow an optimal policy.

$$V_N(x) = {\Delta x_N}^T Q_f \Delta x_N \tag{8}$$

$$V_{k-1}(x) = \min_{\Delta u_{k-1} \in \mathcal{U}} {\Delta x_{k-1}}^T Q \Delta x_{k-1} + {\Delta u_{k-1}}^T R \Delta u_{k-1} + V_k(x) \tag{9}$$

Second, the action-value function is the cost associated to a given pair of state and action.

$$S_{k-1}(x,u) = {\Delta x_{k-1}}^T Q \Delta x_{k-1} + {\Delta u_{k-1}}^T R \Delta u_{k-1} + V_k(x) \tag{10}$$

When the state is fixed, the controls are still free of choice and we can select the ones that minimize this cost. Under this choice, the action-value function is equal to the state-value function.

iLQR is performing quadratic approximations of these two functions, and gives as an end result the following policy for the controls:

$$\Delta u_{k-1} = -d_{k-1} - K_{k-1} \Delta x_k \tag{11}$$

Where:

$$d_k = \left(\nabla^2_{u_k,u_k} S_k \big|_{x,u} \right)^{-1} \left(\frac{\partial S_k}{\partial u_k} \bigg|_{x,u} \right)^T$$

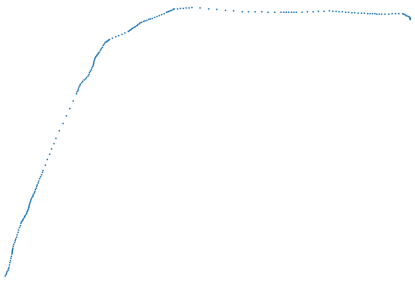
$$K_k = \left(\nabla^2_{u_k,u_k} S_k \big|_{x,u} \right)^{-1} \left(\nabla^2_{u_k,x_k} S_k \big|_{x,u} \right)$$

These gradients and essians are obtained iteratively by linearizing the discrete dynamics around the points x_k, u_k. Several iterations of iLQR are needed to refine x_k, u_k because of the approximations introduced above. The new values of u_k are determined by (11), and the new values of x_k can be determined by forward pass on the dynamics, using the updated controls u_k. For online control, LQR can use the gain matrices K_k as the linear feedback policy for each step k.

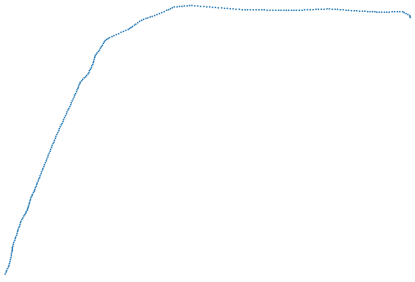
For the initial values and the reference trajectory, we augment the result of RRT using rough approximations on the controls, the yaw angles and the velocities. These are only approximations, and this is why we preferred to choose Q, R and Q_f to give a higher penalty from deviating from the reference 2D trajectory (p_x and p_y). These approximations are the following:

$$v_k = 4.5$$
$$a_k = 0$$
$$\rho_k = \tan^{-1}\left(\frac{p_{y,k+1} - p_{y,k}}{p_{x,k+1} - p_{x,k}} \right), \; k \leq N$$
$$\rho_N = \rho_{N-1}$$
$$\rho_{-1} = \rho_0$$
$$\theta_k = (\rho_{k-1} + \rho_k)/2$$
$$\alpha_k = \theta_{k+1} - \theta_k$$

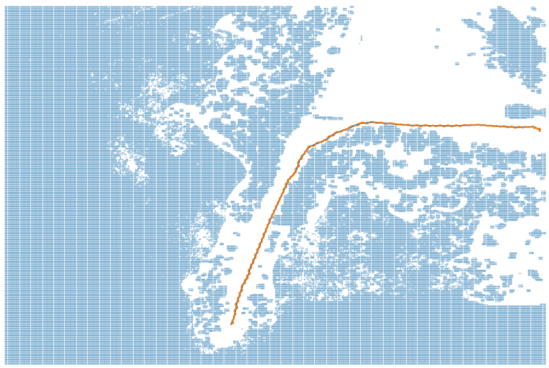
In order to help iLQR converge, we experimented two upsampling methods. The first one is adding equidistant points between every consecutive points in the path. This is a straightforward upsampling method to implement, but has the drawback to keep the same repartition of points in the path.



This repartition is characteristic of RRT/RRT# outputs for this kind of terrains. However, it is penalizing iLQR when it comes to finding a feasible path with equal time steps between each output point. This heterogeneity results in great variations on the velocities and accelerations, which prevents us from closely fitting to the reference RRT path. To alleviate this phenomenon, we used instead an umpsampling method that produces a path with equidistant consecutive pairs of points.

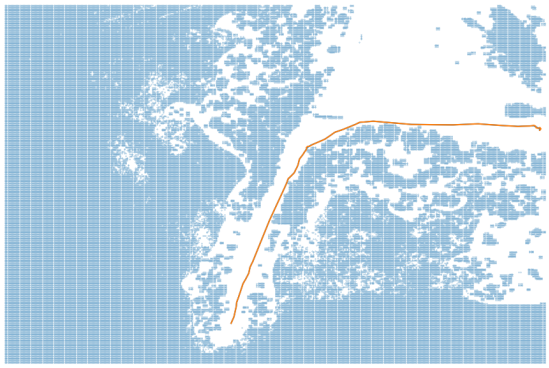


1) Full iLQR: We first tried running iLQR to optimize the entire path. This results in reasonable paths, but the consequent horizon length deteriorates the resulting path.

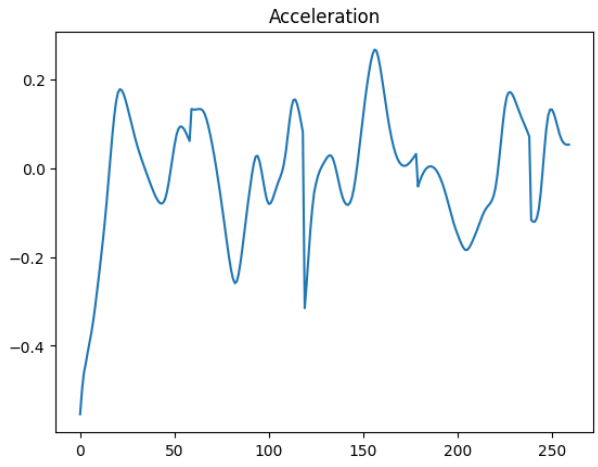


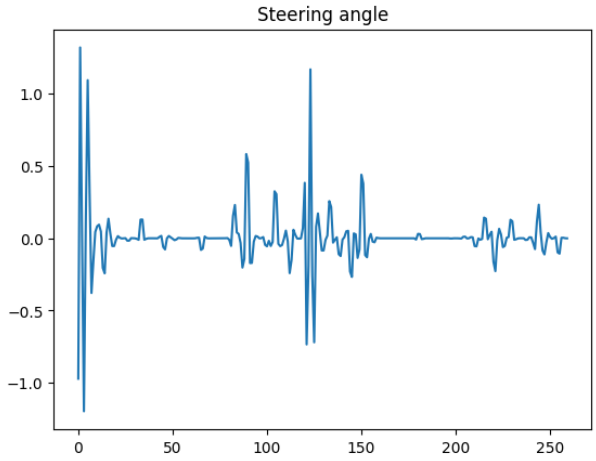
2) iLQR by parts: Shortening the horizon over which iLQR is requested to optimize helps it converge quickly and to more desirable solutions as it makes the optimization problem easier and the linearization of the dynamics more accurate on the tail of the path.

This achieved a more desirable behavior.



The controls can then be used coupled with an LQR controller to follow this trajectory.





B. MPPI control

MPPI is a sampling based control method that offers flexibility and is able to optimize non-linear systems with complex cost functions. MPPI was first introduced by Williams et al. in [4], which allowed for real time optimization by taking advantage of GPU acceleration. MPPI does not split planning and execution into discrete steps, and instead calculates a trajectory, performs a single control input, and uses the rest of the unused trajectory to warm start the subsequent optimization step.

Since the original work, further works have made improvements or deviations to the base MPPI approach. We used the pytorch-mppi library developed by the University of Michigan Autonomous Robotic Manipulation Lab. This is a PyTorch implementation of the version of MPPI introduced by Williams et al. in [5] which extends the original MPPI control process to not require control-affine dynamics. This is critical as it allows for nonlinear approximate system dynamics (ie neural networks) and a purely data driven approach to model learning for deep reinforcement learning applications for MPC.

While our current work does not include a neural network, we chose this method due to its implementation simplicity and avenues for future work in deep reinforcement learning. We used the same bicycle dynamics model described in Equation 2, the same cost function as used in our iLQR experiments, and used an equidistant reference path as it provided the best solution. We used a laptop with a Nvidia RTX 3080Ti to parallelize and accelerate our sampling process.

We can see in Figure 8 that tracking performance is already excellent at 500 samples, though there are some issues at the start and end of the tracking sequence. Some of this may be mitigated with a stronger termination penalty in our cost function, and a better initial starting guess. We found that increasing the nunmber of samples had little impact on the overall trajectory, as even just 500 samples performed well with our current configurations. Where the number of samples made a large difference is the realism of the controls. We can see in Figure 9 that the acceleration inputs throughout this tracking sequence are quite unrealistic. Though the magnitude is not very large, the inputs are quite jerky. This is because

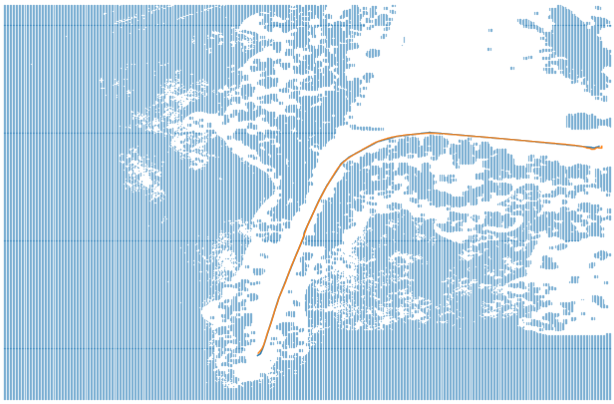


Fig. 8. MPPI Tracking - 500 samples: Blue is the reference trajectory, orange is the tracking trajectory.

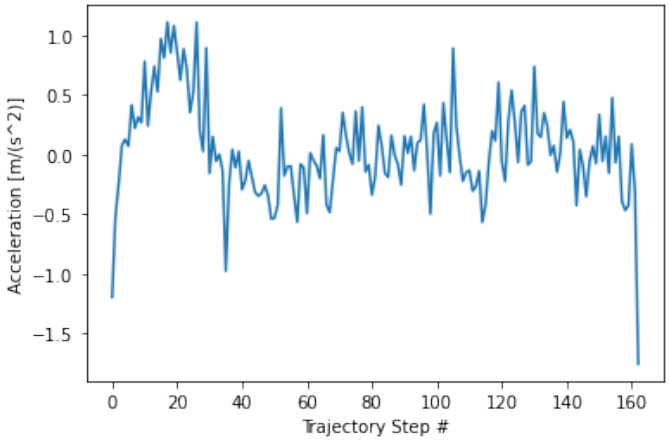


Fig. 9. MPPI Acceleration - 500 samples

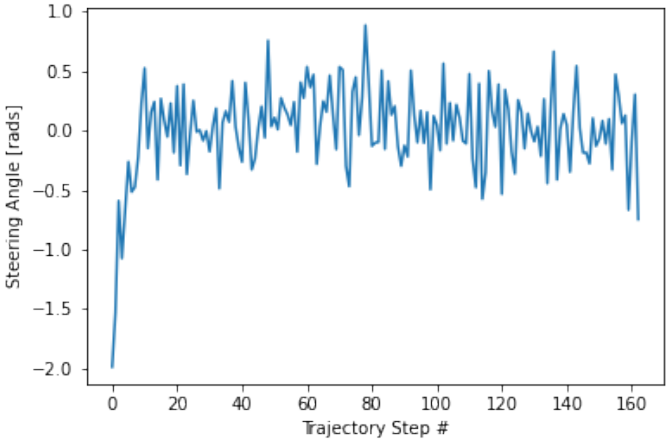


Fig. 10. MPPI Steering Angle - 500 samples

we did not place any additional constraints on the controls to control for smoothness or feasibility. The inputs for the steering angle are also very unrealistic with large swings from one time step to the next (angles are measured in radians).

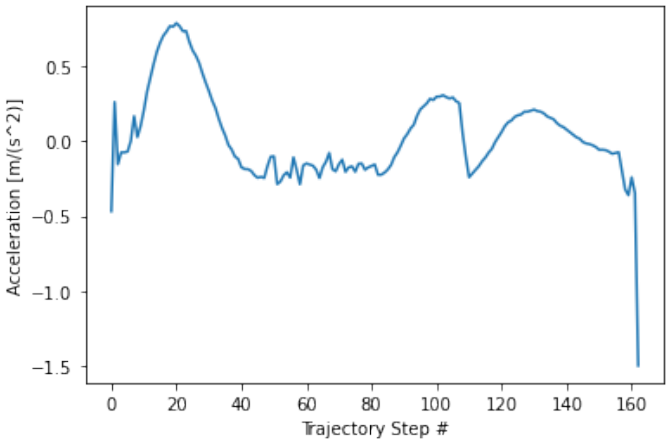


Fig. 11. MPPI Acceleration - 500k samples

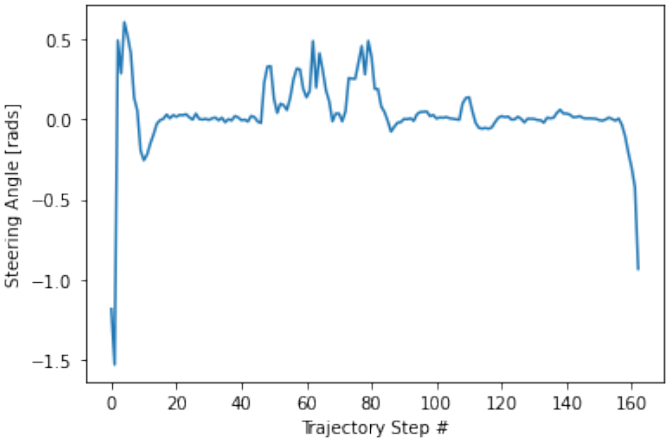


Fig. 12. MPPI Steering Angle - 500k samples

By increasing the sample count to 500,000, we can see in Figure 11 and Figure 12 that our controls are much smoother. There are still extreme values at the beginning and end, but the values and the overall shapes of the control curves are far more realistic. This is without a significant change to the trajectory, indicating that the small sample count produces minor differences in the trajectory by consistently overshooting and overcompensating, leading to the oscillatory and jerky behavior seen in the 500 sample plots. By increasing the sample count, a far greater range of controls are sampled, allowing for more precise control, removing much of this behavior.

IV. Future Work

A. *RRT#*

As a future work, we would like to achieve RRT# to run around a real time at least at 10 Hz. In order to achieve a faster

compute, first, we would like to run cost calculations between the sample node and its neighboring nodes in parallel. This can be achieved using OMP or TBB libraries if it runs in CPU, or we can try CUDA with a Nvidia GPU. Secondly, those cost calculations can be saved in the memory to be re-used instead of re-computed during RRT# re-planning.

Another improvement we would like to try is to include vehicle dynamics into the node expansion. Currently, the RRT# node expansion is based on a straight line which may greatly limit performance of the RRT# if the cost map is more complex. This can be achieved solving iLQR or DIRCOL or any dynamically feasible MPC algorithm during the node expansion where we could add dynamics and input constraints.

Lastly, in this project, we assumed the map is not changing from the start state to the goal state. But, in fact, on an autonomous vehicle, the map would be evolving as a new measurement is received. As a result, it might necessitate to recompute RRT# trajectory every time when the map is updated. But, new expansion of the RRT# nodes might be a time consuming. Thus, we might need to re-use previous update cycle node expansion in the newly updated map and run a fewer node expansion. This might be achieved with a smarter data structure such as scrolling grid and storing the nodes in the grid cells.

B. MPPI

For MPPI, one possible avenue of exploration is to extend to more complex dynamics models. The bicycle model we used is great for simple applications, but more robust models may be required to adequately capture the dynamics of off-road navigation. More complex 4-wheel models are a possible solution, but another may be to use a neural network to approximate the system dynamics. Depending on the quality and quantity of our data reference, this may be a feasible and interesting avenue for exploration. In particular, many real world problems require real-time performance, limiting possible neural network architectures to very simple models. In traditional deep learning, it is common to train a large model to stabilize and accelerate the learning process, and then shrink the model using a combination of methods such as quantization (low precision representations), pruning (remove redundant neurons), or distillation (train a smaller network to emulate a larger network). An interesting avenue of future work would be to apply these compression techniques to train a relatively large neural network to capture system dynamics, then shrink the model to achieve the runtime performance necessary without a noticeable drop in accuracy.

V. Code Link

All of the code for this project is contained within the following GitHub repositories: https://github.com/Antoine-Bergerault/OCRL-Project and https://github.com/maulim0v/RRT_Sharp_ws/tree/main. The MPPI implementation we used is from https://github.com/UM-ARM-Lab/pytorch_mppi.

References

[1] O. Arslan and P. Tsiotras, "The Role of Vertex Consistency in Sampling-based Algorithms for Optimal Motion Planning," arXiv:1204.6453 [cs], Apr. 2012, Accessed: May 01, 2023. [Online]. Available: https://arxiv.org/abs/1204.6453

[2] N. Hatch and B. Boots, "The Value of Planning for Infinite-Horizon Model Predictive Control," arXiv:2104.02863 [cs], Apr. 2021, Accessed: May 01, 2023. [Online]. Available: https://arxiv.org/abs/2104.02863

[3] S. Zhao, H. Zhang, P. Wang, L. Nogueira, and S. Scherer, "Super Odometry: IMU-centric LiDAR-Visual-Inertial Estimator for Challenging Environments," arXiv:2104.14938 [cs], Aug. 2021, Accessed: May 01, 2023. [Online]. Available: https://arxiv.org/abs/2104.14938

[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.

[5] G. Williams et al., "Information theoretic MPC for model-based reinforcement learning," 2017 IEEE International Conference on Robotics and Automation (ICRA), Singapore, 2017, pp. 1714-1721, doi: 10.1109/ICRA.2017.7989202.