Autonomous race car

Rajat Chand
rajatc@uw.edu

1 Race Car Platform

The platform is based on 1/10th sized rally cars and the Nvidia Jetson TX 2 embedded super-computer. The platform houses the following major components, **1.** RealSense ZR-300 RGB-D Camera which could get images up to 1080p @ 30Hz and get depth upto 628×628 @ 60fps **2.** Hokuyo URG-04LX-UG01 Laser having a scanning angle of 30° and a sampling rate of 10Hz and **3.** Razor IMU containing the accelerometer, gyroscope and magnetometer programmed for sample rate of 50Hz and **4.** Fully programmable VESC (Vedder Electronic Speed Controller) which controls both the brushless motor (forward/backward) and servo motor (steering).

Over the course of the quarter this remote controlled car was programmed using ROS, Python, PyTorch, OpenCV and scikit-learn to handle autonmous driving one step at a time. First, particle filter was implemented to allow the car to localize itself. Second, incoming visual data from the car's RGBD camera was used to implement local control algorithms and an end to end deep learning based imitation learning solution. Third Model Predictive path control (MPPI) algorithm was implemented to integrate local control with a global plan. Finally a global planning module is implemented that integrates all the above components with an efficient gloabl search over the map to compute a trajectory (plan) that the car would follow.

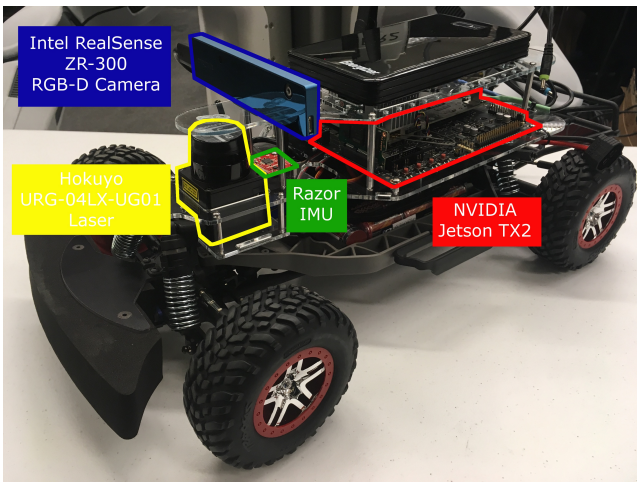


Figure 1: Racecar platform

2 State Estimation

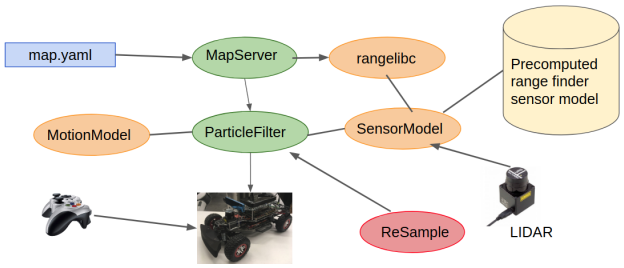


Figure 2: Complete system architecture for state estimation

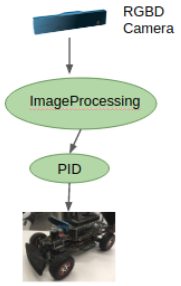
The state estimation implements a particle filter to localize the car within a known map. Different parts of the particle filter, namely the motion model, sensor model and the overall filtering framework (i.e the particle filter itself and the resampling algorithm) is implemented here. The code for the state estimation module can be found at https://www.github.com/rajatsc/autonomous-race-car/tree/master/lab1

- **MotionModel**
  The MotionModel library implements the Odometry motion model (`OdometryMotionModel`) and the kinematic motion model (`KinematicMotionModel`). The odometry model is based on sesnor odometry information i.e the pose information provided by the car. The controls for our car are the speed of the car and the steering angle and the kinematic model explicitly models the effect of this action on the previously estimated pose, predicting a velocity for the 2D position and orientation which we integrate forward in time to get the future pose. Noise is added to both the models to allow our models to be probabilistic thus more robust. Later, a neural network(f) is trained to learn the motion model of the car. This is termed as Model learning. This is done by collecting control and pose data while driving the car around, the pose data collected here is estimated via a particle filter provided by the TAs which is very accurate and thus the estimated pose is therefore used as ground truth for the pose values. The neural network (f) predicts the change in the robot's state from the applied controls and its previous change in state i.e. $state_{t+1} = f(state_t, controls_t)$ The script for model learning can be found at https://github.com/rajatsc/autonomous-race-car/blob/master/lab3/src/Trainer.py.
- **SensorModel**
  The LIDAR sensor shoots out rays into the environment at fixed angular intervals and returns the measured distance along these rays (or nan for an unsuccesful measurement). Therefore, a single LIDAR scan $\mathbf{z}$ has a vector of distances along these different rays $\mathbf{z} = [z^1, z^2, ..., z^N]$. Given the map of the environment $\mathbf{m}$, these rays are conditionally independent of each other, so we can rewrite the sensor model likelihood as follows: $p(\mathbf{z_t}|\mathbf{x_t}, \mathbf{m}) = p(z^1, z^2, ..., z^N|\mathbf{x_t}, \mathbf{m}) = p(z^1|\mathbf{x_t}, \mathbf{m}) * p(z^2|\mathbf{x_t}, \mathbf{m}) * .... * p(z^N|\mathbf{x_t}, \mathbf{m})$. To evaluate the likelihood we will first generate a simulated observation $\hat{\mathbf{z}}_\mathbf{t}$ given that the robot is at a certain pose $\mathbf{x_t}$ in the map. We do this by casting rays into the map from the robot's pose and measuring the distance to any obstacle/wall the ray encounters. This is very much akin to how laser light is emitted from the LIDAR itself. This is done by using the ray casting library called `range_libc` We then quantify how close each ray in this simulated observation is close to the real observed data z_t^i providing us an estimate of $p(z^i|\mathbf{x_t})$. The `precompute_sensor_model` method of the SensorModel class returns a numpy array containing a tabular representation of the sensor model. This array is used as a look-up table to quickly compare new sensor values to expected sensor values, returning the final sensor model likelihood: $p(\mathbf{z_t}|\mathbf{x_t})$
- **ReSample**
  This script implements the naive and low-variance sampling which is used by the ParticleFilter class to sample new particles.
- **ParticleFilter**
  Particle filter connects the MotionModel, the SensorModel and ReSample objects together

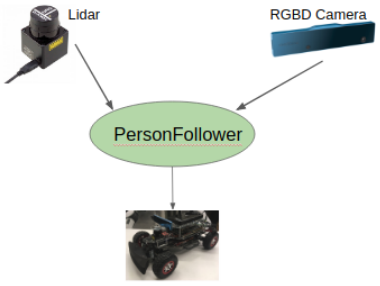
in a tight loop that runs in real-time. The particle filter uses a number of hyper-parameter settings that we had to tune for good performance, such as the number of particles to use, the amount to sub-sample the LIDAR data, when to resample, etc.

- **MapServer**
  Mapserver is a ROS node that reads a map and offers it via a ROS Service.

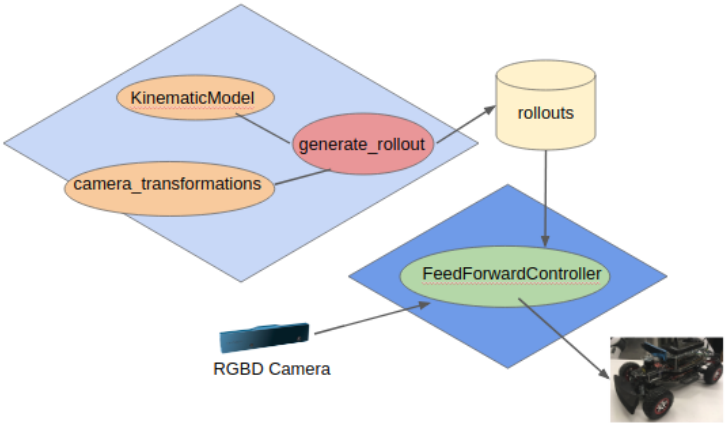
3 Local control



(a) PID Controller



(b) Person follower using data from LIDAR and RGBD Camera



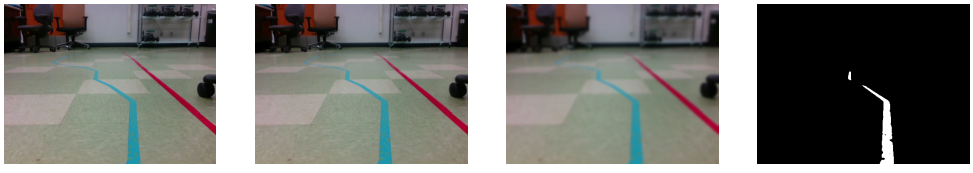
(c) Feedforward controller using template matching

Figure 3: Complete Image Processing pipeline

Here, we design three local controllers that operates on visual observations coming from RGBD camera. The three local controllers are PID controller using region of interest, Feedforward controller using template matching and Person follower using a combination of sensor data from the camera and the LIDAR. The code for the state estimation module can be found at https://github.com/rajatsc/autonomous-race-car/blob/master/lab2/src/Trainer.py.

- **ImageProcessing**
  The car is equipped with an RGBD camera. The raw image from the camera is first reduced to 80% of the original image to reduce computation. The image is then blurred using a Gaussian function to remove noise. Now since color information in RGB space can change depending on lighting conditions it makses it difficult to extract this information. The Gaussian blurred RGB image is thus converted to more light invariant HSV space. We then binarize the image and create a pixel mask that isolates the colors you specifically care about to extract the path that we wish to follow. After you have extracted the path from the original image, we select a region of interest. This region represents the 'feature' the robot car seeks to perform local control on. The region of interest is chosen to be the lower 130

rows of pixels, offset 50 pixels above the bottom of the frame. The centre of the region of interest is calculated by calculating the centre of all non-zero pixels.



(a) Raw image

(b) Resize image to 80% to reduce the amount of computation

(c) Gaussian blur (kernel size 15x15 pixels) to reduce noise

(d) Threshold colors to compute mask

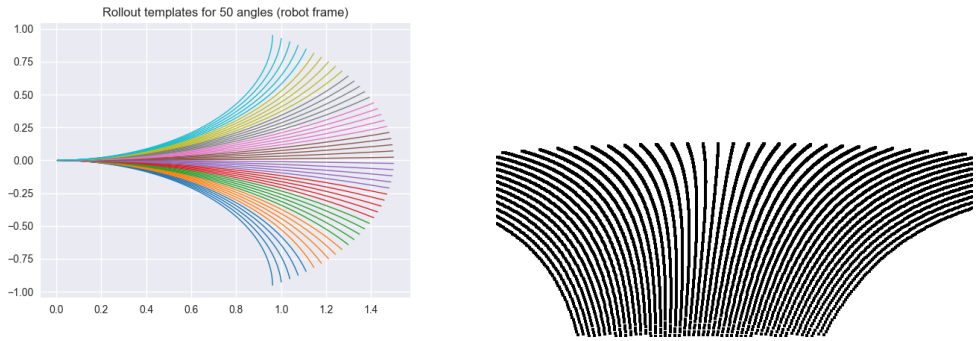
(e) Mask the original image to find the blue line

(f) Crop to region of interest

(g) Find center coordinate (shown with green dot)

Figure 4: Complete Image Processing pipeline

- **PIDController**
  The error term is then computed by taking the difference between x coordinate of a centred line (determined empirically) and the x coordinate of the centre of the region of interest.The image processing code can process process roughly up to 52.6hz which is fast enouh to handle the 30hz publishing rate of the camera. The error calculated is then used by the PIDController to set the car controls. We let the car drive at a constant speed and the error only determines the steering angle.
- **generate_rollout**
  `generate_rollout` script is used to create rollouts of the car's path using the kinematic car model, then display that motion in the camera's 2D pixel frame. We transformed the rollouts first from the kinematic frame to the camera's 3D frame, and then projected the rollouts into the camera's pixel frame using camera intrinsics. We measured the car to tune accurate transformations between the robot and camera frames.



(a) Several templates overlayed on one plot in the robot frame.

(b) Several templates overlayed on one plot in the camera frame.

Figure 5

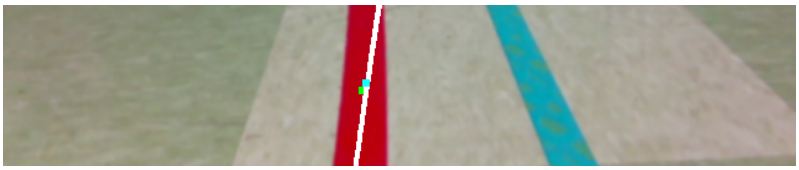
- **FeedForwardController**

  Each template represents a generic path that the robot may encounter while going around a track. A template is an image that you will compare to the extracted color mask from the ImageProcessing object in order to find a match. Each template has a height equal to that of the mask, and a width that is a fraction of the mask's width. In addition, each template has an associated control that should be performed when that template is observed. Fig 6 illustrates paths that the robot may find when driving. Given this path, the robot must decide which of its pre-specified templates best matches the path.

  We computed the rollouts for each template based on fixed control values for steering angle and velocity. Each template is assigned the fixed control values used to generate it. When a template matched, the feedforward controller would publish the corresponding control. Each template only contains a single fixed control value, but if we are unable to detect a line in the camera frame we will continue to publish the most recently chosen control until the line is found. This makes going around sharp corners more robust, in case the robot overshoots and the line leaves the camera frame temporarily.

(a) A straight line in the left of the frame matches a left turn template.

(b) At the corners the matched template is the one with the sharpest right turn.



(c) A straight line in the middle of the frame corresponds to a forward template.

Figure 6

- **ImitationLearning**

  This end-to-end approach trains a Deep Convolutional Neural Network to geenrate steering angle based on the input camera observation. This requires a human driver to drive around the track and collect data which is then used to train the CNN.

- **PersonFollower**

  We first read in the color image and extract out the portion that matches the color of interest. We then find the center of the colored region and compute an error for the PID controller. We use this to determine the steering angle. To make the reverse motion more robust, we negated the steering angle when going in reverse. This helps to keep the target in the frame of the camera.

  We use the LIDAR data to determine the speed using another PID controller. First we filter the LIDAR ranges to get the ranges that are primarily in front of the car (approximately the center third of the sweep). We filter out ranges that are below some threshold to help reduce the noise in the reading. We then look at the closest 70 ranges and take the mean to try to determine the distance to the target.

  We take the difference between this distance and the desired distance (75 centimeters) to produce an error for the PID controller. This will create a speed control to move either toward the target forwards, or away from the target backwards.

Because the LIDAR publishes slower than the camera, control messages are only sent when the camera publishes an update. This lets the robot stay slightly more up to date in its steering angle even when the speed control hasn't had a chance to update.
Due to the amount of variance between successive filtered images, we found large variations between successive steering angles. We smoothed this out with exponential smoothing, taking 70% of the previously computed control and 30% of the current control. This significantly smoothed the steering angle without significantly impacting performance. Since we maintained an update rate of approximately 30hz, this was expected.

4 Global Planner

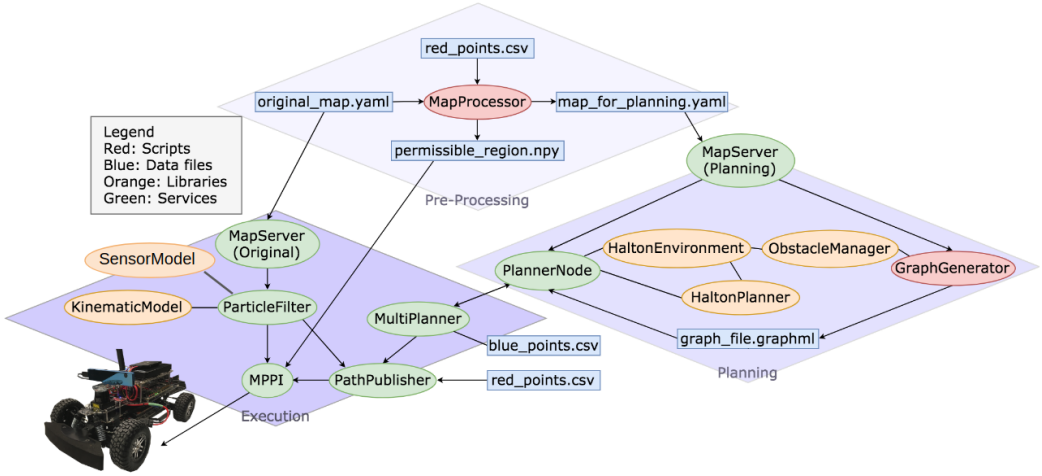


Figure 7: Complete system architecture for global planning

Our system architecture divides the global planning tasks into three main modules to be run sequentially. The code for the global planning system can be found at https://www.github.com/rajatsc/autonomous-race-car/tree/master/lab4

1. **Pre-processing**
   Given input data, this module handles generating map and graph files used for planning and execution. This is only run once.
2. **Planning**
   The planning module maintains the state of a search graph to use for local planning. A service is provided that will produce a path between two different configurations.
3. **Execution**
   The execution module contains the controller, MultiPlanner, that handles planning and execution requests. Given a series of waypoints to visit, this module will request local plans from the Planning module and stitch them together to create a global path. The grand plan will then be sent to be executed by the MPPI controller.

4.1 Description of Components

- **MapProcessor**
  The MapProcessor reads in the `red_points.csv` and the original provided map. The first step is to process the map using cv2 to remove noise and smooth out the walls. We then add the obstacles as circles centered at the provided coordinates. The processor will output `map_for_planning.yaml`, which expands the obstacles and walls slightly. This will later be used to create the motion path. The processor will then output

permissible_region.npy, which is an occupancy grid that exaggerates the walls and obstacles, so the MPPI controller is more likely to avoid the red points.

- **MapServer**

  We run two separate MapServers in different namespaces. One of them will publish the original map to help us visualize the global position of the robot. This is used in the particle filter, and ideally matches the actual layout of the evironment as closely as possible. The second map server reads in map_for_planning.yaml so we can overlay the exaggerated walls onto our visual output and provide an occupancy grid for the ObstacleManager used by A* search. The visualization is useful for debugging to make sure our robot was actually avoiding the walls and bad waypoints.

- **GraphGenerator**

  The GraphGenerator creates a graph structure on top of the map of the room. It reads in the map_for_planning data from the MapServer, and uses the Halton generator to choose the node locations. Nodes are randomly generated in bounds, and connected if close enough to each other. The weights on each edge are calculated as the shortest Dubins path that connects the two poses. The resulting graph is saved into graph_file.graphml.

- **PlannerNode**

  The PlannerNode provides a search service that runs A* between two given poses on the map. The planner reads in graph_file.graphml and uses the HaltonEnvironment to maintain the state of the graph. Information about the layout is read from the MapServer, so edges that go out of bounds will not be included in a result path.

- **MultiPlanner**

  The MultiPlanner is the primary interface into the Execution phase of the global planning. This component reads in all desired waypoints from blue_points.csv and computes a global path to connect them sequentially. The MultiPlanner calls out to the PlannerNode service to compute shortest path between each waypoint, and builds up the grand plan one waypoint at a time. Each path is a list of intermedate goal positions, which follow the Dubins path between each graph node. Once computed, the entire path will be published to the topic /mp/mppi_path to be read in by the PathPublisher.

- **PathPublisher**

  This module controls the MPPI node and decides when to publish the next goal position for the racecar to move towards. It also controls the max velocity of MPPI depending on how far the race car is from an obstacle. This node subscribes to /mp/mppi_path to get the list of paths from the multi-planner node and publishes goals one at a time to /pp/path_goal which is in turn listened to by the MPPI node. The path publisher decides to publish a new goal whenever the car is within a given distance and theta threshold from the current goal. The path publisher subscribes to /pf/ta/viz/inferred_pose to get the current position of the race car. The inferred pose is published by the particle filter node. In order to have an overview of the obstacles, it reads in the red_points.csv file.

- **ParticleFilter**

  We used a variant of the ParticleFilter provided by the TAs to keep the robot localized during execution. The ParticleFilter relies on map data published by the MapServer, as well as the KinematicModel to estimate updates in the robot's position as we send controls. This module continuously publishes the current robot pose to the topic /pf/ta/viz/inferred_pose. We bumped up the number of particles from 4000 to 6880 and noticed no noticeable change in performance due to parallelization on the GPU for the ray casting code.

- **MPPI**

  The MPPI component is where we send controls to the robot. MPPI reads the current robot pose from the ParticleFilter and determines the goal from the topic /move_base_simple/goal as sent by the PathPublisher. The MPPI algorithm computes weighted averages of many trajectories based on a custom cost function which we describe in the Tuning section. To execute MPPI controls smoothly, we run MPPI in a tight loop that executes at approximately 15hz. The pose updates coming from the ParticleFilter will update the pose data within the component asynchronously without interrupting the current execution. As MPPI can update controls faster than it receives particle filter updates, it was

important not to let MPPI wait to publish a control until it receives the most recent position update.

4.2 Running the system

To run the global planning for the final demo, we used the following checklist to prepare the data and execute the controls.

4.3 Prepare Data

1. Download data zip file, unzip.
2. Run script `plot_points_on_map.py` to plot all points on map image from the csv files, including the start point, good waypoints, and bad waypoints.
3. Inspect the points and make an executive decision on the order of the good waypoints to visit while avoiding the bad waypoints. Additional fake blue waypoints are potentially added to assist planning. Change the order of the points in the good waypoints csv file.

4.4 Generate Maps and Graphs

1. Make sure data file paths are correct in MapProcessor.
2. Generate map files with MapProcessor. This generates maps with thicker walls and obstacles for the bad waypoints.
3. Generate graph with GraphGenerator.

4.5 Launching

1. Make sure data file paths are correct in launch files for PlannerNode, MultiPlanner, and PathPublisher.
2. Launch each component:
   - `roslaunch racecar teleop.launch`
   - `roslaunch lab4 ParticleFilter.launch`
   - `roslaunch lab4 PlannerNode.launch`
   - `roslaunch lab4 MultiPlanner.launch`
   - `rosrun lab4 PathPublisher.py`
   - `rosrun lab4 old_MPPI.py # (the newer MPPI was found to be less reliable)`
3. Run Rviz to visualize.

4.6 Interactive Execution

1. Once everything is set up and launched, use the joystick controller to run the programs:
   - **B (red)**: Starts generating a grand plan through the good waypoints from the start point given by the csv file.
   - **Y (yellow)**: Starts generating a grand plan through the good waypoints from the current position of the car.
   - **A (green)**: Start execution of the computed grand plan by sending the plan to PathPublisher, which sends it to the MPPI.
2. Once a grand plan is generated, it is saved in MultiPlanner, and you can execute the same grand plan repeatedly by pressing on button A without having to re-plan. To re-plan, press either B or Y.

Figure 8: A visualization of the original map with waypoints (blue) and obstacles (red)

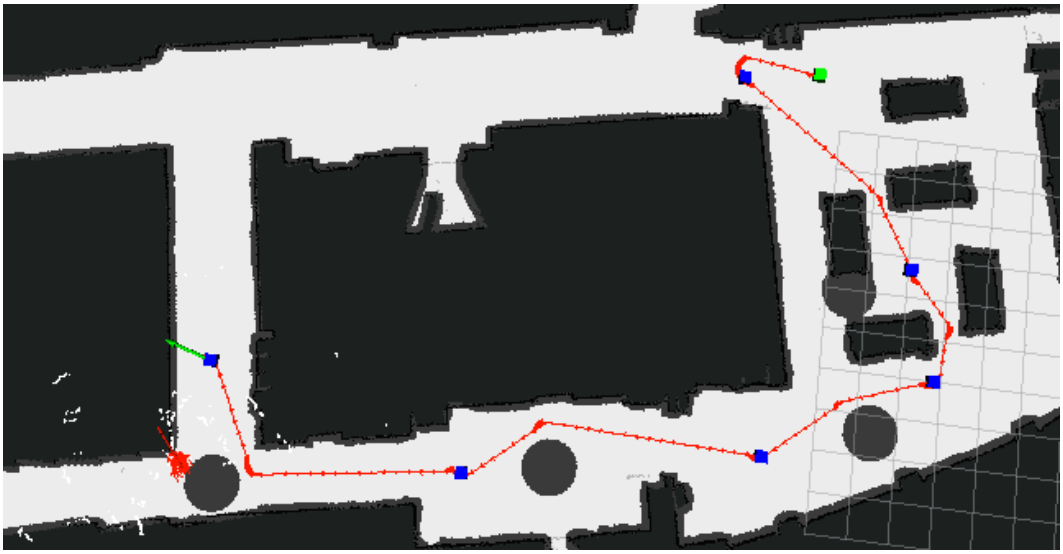


Figure 9: A visualization of the computed path in the final course (Obstacles are marked as circles

4.7 Tuning

Many of the algorithms used in this assignment had parameters or cost functions that required tuning.

- **MPPI parameters**
  The MPPI algorithm creates K proposed trajectories, and does a rollout for T timesteps to predict where the car will end up by following those controls. Each modification to the control is created through random noise with some choice of sigma, it is important to generate enough trajectories that we will find one that brings us closer to the goal. We also need each trajectory to be rolled out far enough ahead that we will avoid running into obstacles. We ended up using the values T=25, K = 2000, σ_{xy}=0.35, and σ_{θ}=0.15.
- **MPPI cost**
  Our group spent much time tuning the MPPI cost function in order to achieve stable behavior in approaching the goals. This required balancing the emphasis on euclidean distance and theta difference when computing the cost of a control.
  The distance component of the cost function was computed as:

$$2.5 \cdot \left(euclidean\_distance \cdot 1.7 + theta\_distance \cdot 0.5 \cdot 1.2^{0.5 \cdot t} \right)$$

where $euclidean\_distance$ and $theta\_distance$ were computed as the differences between the current pose and goal pose. The exponential factor for time means that theta differences become more important in the future, helping to allow the car to rotate when far from the goal but attain the correct orientation later on. The various coefficients were determined by manually testing.

The cost on making large controls was identical to that provided in the MPPI paper. This was given an additional weighting coefficient of 1.9.

Trajectories that went out of bounds were given a cost of 10000 to make them influence the end control very little.

- **Velocity thresholds**

  We also tuned the threshold of our MPPI controller to slow down near a goal for more accurate steering. We also have thresholds in our PathPublisher node to slow down the racecar when it approached obstacles. Finally, we tuned MPPI to run at the highest speed possible with the trained motion model data that was available to us.

- **Goal thresholds**

  The PathPublisher node determined when a new goal was "achieved". As a single path contained both the ultimate goal and intermediary destinations, we had a wider threshold on when the intermediary destinations were goals, and more stringent requirements for the ultimate blue waypoints. We tuned the distance threshold such that when the race car was 80 cm from the next intermediary goal it would advance, but required the race car to be within 25 cm of the ultimate goal.

- **Map and obstacle processing**

  The map processing step used the cv2 function `erode` with a kernel size of (15,15). This expanded the walls by approximately 15cm, to make our car stay sufficiently far away from all obstacles.

  We incorporate the bad waypoints in our first step of pre-processing by creating a circle on the map centered at each coordinate. After testing various circle radii we chose a radius of 0.23 meters. This was sufficient to ensure paths avoid the red points but still allow gaps between the obstacles and the walls. We found it was much more important to generate paths that stay sufficiently far from the bad waypoints, because MPPI was less reliable in its precise movements to avoid obstacle regions.

4.8 Final Run

The objective was to drive autonomously starting at the green marker on the map, navigating to all the blue waypoints and avoiding the red waypoints along the way(shown as black circles on the map) as quickly as possible. We ended up generating a halton graph with 300 nodes. It took approximately 3 minutes to generate the graph. An artificial blue waypoint (third blue marker on the path) was added to generate a shorter Dubin's path between the second and the fourth blue waypoint. This was necessary because the search would otherwise use a longer, safer path. The Multiplanner took a little under a minute to find the entire path. The robot successfully navigated the course (reaching all blue waypoints and avoiding all red waypoints) in 48 seconds.

You can find clips from the class at https://www.youtube.com/watch?v=m0rKojCvP6g&t=179s