Final Race Lab

Team: Virtual Fast and Keyboard Furious

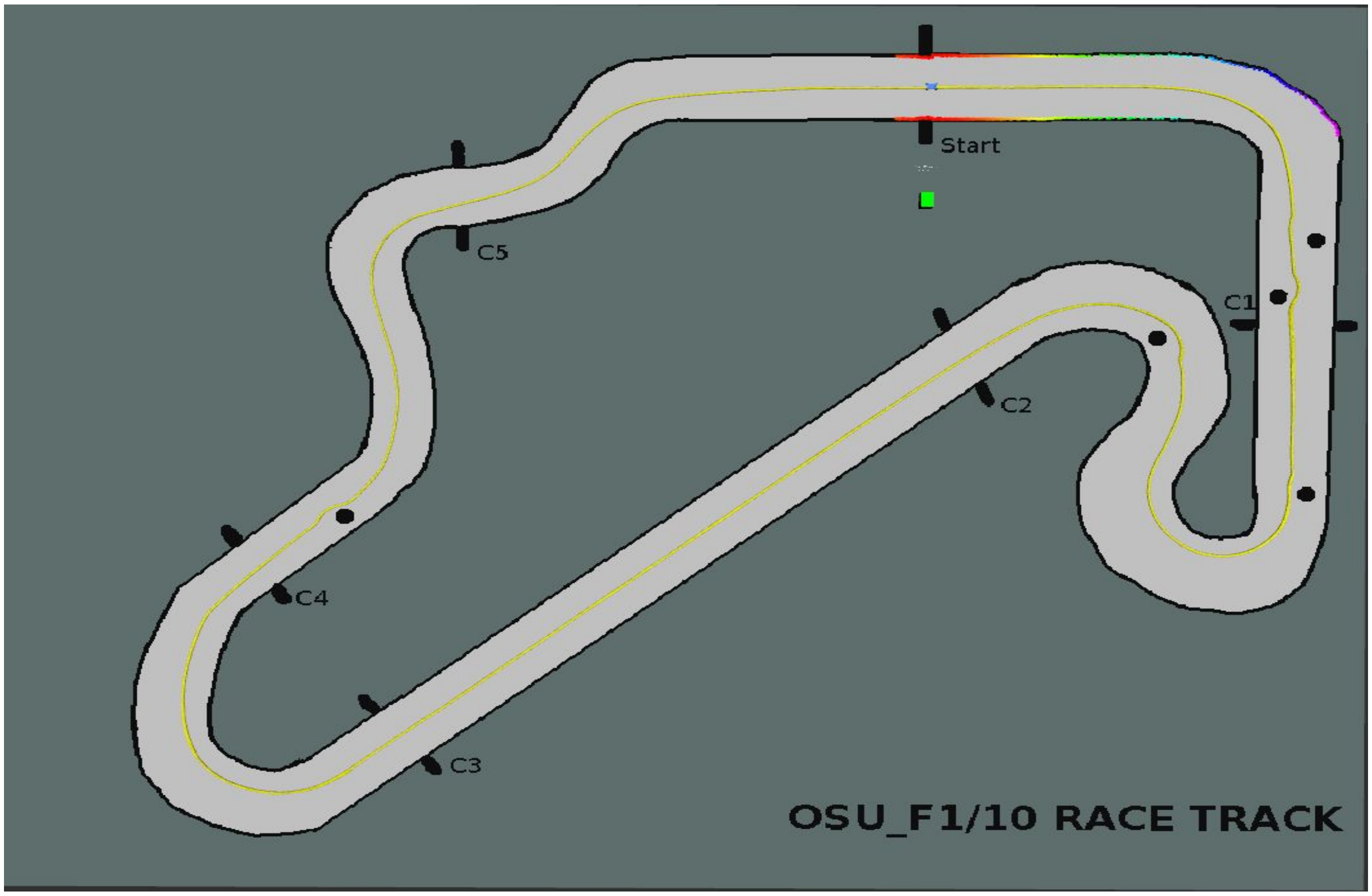
In the final race, we followed three directions to get the best result.

1. **Wall-following (center) and Pure Pursuit tracking with PF odometry**
2. **RRT based waypoint generation and Pure Pursuit tracking with PF odometry**
3. **Manual generated waypoints, heuristics in Pure Pursuit tracking with PF odometry**

As listed above we get an average lap timings as follows:

1. 90s with zero collisions [Wall-following (center) and Pure Pursuit tracking with PF odometry]
2. 40s with 1-2 collisions [Manual generated waypoints, heuristics in Pure Pursuit tracking with PF odometry]
3. Best timing till now is 34s

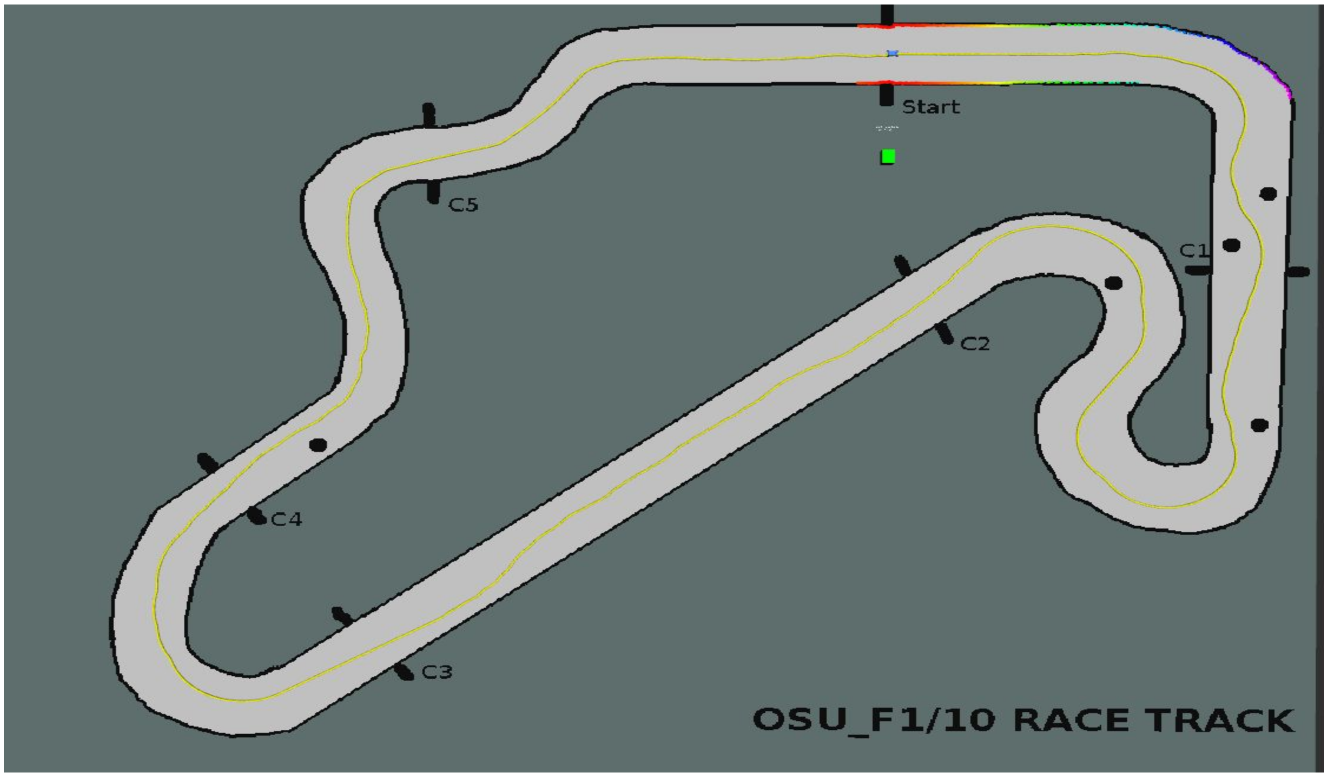
1. **Wall-Following and Pure Pursuit Control tracking with Particle Filter:**
    a. We started with the skeleton code of Pure Pursuit and builded up on that.
    b. We used Wall Following lab code to make waypoints, specifically we used center wall following and with that we used our Waypoint saver nodes in order to store Odometry data to form racing lines.
    c. First we implemented Pure Pursuit on the ground truth Odom frame and noted the performance, after that we included a live particle filter localization method and experimented Pure Pursuit with particle filter odom frame.
    d. We used our updated error analysis code from the Pure Pursuit lab where we calculated the tracking error and saved it to a csv file.
    e. The figure below shows the race line generated through the wall following code.
    f. We noticed that when we created waypoints using wall following the race line was going to close to the obstacles due to this we were seriously limited by the maximum speed and at points if it crashed into the obstacle close to waypoint and it is almost in straight path of the car, it will continuously keep ramping into the obstacle.
    g. When we started to improve the tracking of the race line, we changed many different parameters from Simulator max speed to angle change robot can take, finally we settled on 11 deg max angle change so that if for some reason at high speed if we get angle more than 15 the car skids and loses control.
    h. Then, we decided it’s time to get a better race line.



Waypoints Generated through Center Wall Following

2. **Manual generated waypoints, heuristics in Pure Pursuit tracking with PF odometry**
    a. In order to start manually driving the car in the simulator and store those waypoints, we first reduced the simulator keyboard steer angle and max speed to 0.2 rad and 0.8 m/s. So that we can generate denser waypoints and control is easy.
    b. We tried a bunch of different parameters of keyboard steer angle and speed in order to get the best race line.
    c. We used waypoints_saver to save odom data to form a racing line.
    d. After getting the racing line we noticed that due to lack of experience with the keyboard control of the robot in the simulator we were not able to get exact best race line we could achieve.
    e. Then, we tried bunch of different ways to optimize the waypoints, we ran the waypoints saver in different parts of the track and saved those parts and when tried to stitch those parts together or change a portion from the whole waypoints, but since this was a dense waypoints set (waypoints going upto 7000 or so points), we always had some variation in either joining of the path or getting rest segments which are not required.

f. Finally, we decided to go with the one in which we could drive to the best of our knowledge and experience and use that as the test for the racing line.
g. We used the same Pure Pursuit algorithm from the previous wall following section.



Manually Generated Waypoints

3. RRT based waypoint generation and Pure Pursuit tracking with PF odometry

For RRT based path planning we used the vanilla version as described in the algorithm

```
function rrt($x_{init}$, max_iter):
    V ← {$x_{init}$}                                      // vertices
    E ← ∅                                              // edges

    for i = 1:max_iter:                                   // maximum number of expansions
        $x_{rand}$ ← SampleFree()                           // collision free random configuration
        $x_{nearest}$ ← Nearest(G=(V,E), $x_{rand}$)           // closest neighbor in the tree
        $x_{new}$ ← Steer($x_{nearest}$, $x_{rand}$)             // expanding the tree

        if ObstacleFree($x_{nearest}$, $x_{new}$):            // the edge is collision-free
            V ← V ∪ {$x_{new}$}
            E ← E ∪ {($x_{nearest}$, $x_{new}$)}

    return G=(V,E)
```

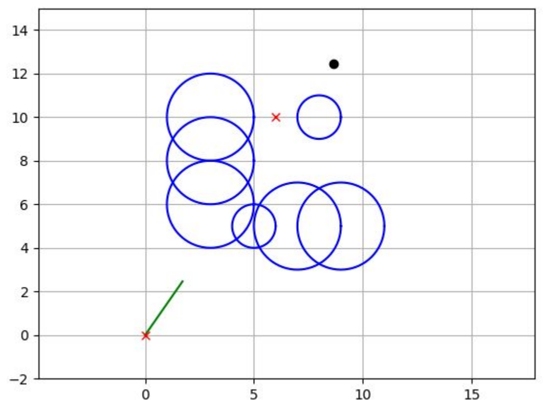
First, we developed a toy environment to test our RRT algorithm. Here we develop the code which will help us plan in the ROS final race environment.

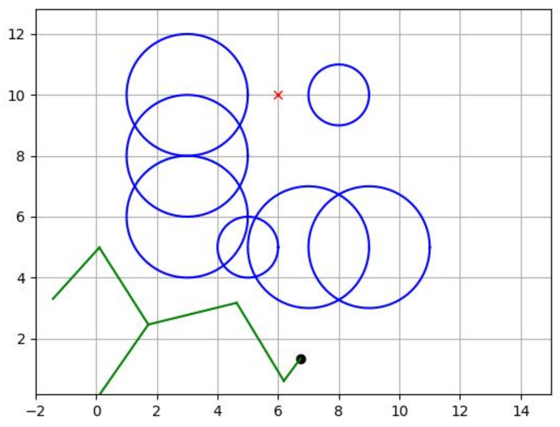
You can try out in scripts/rrt_toy.py

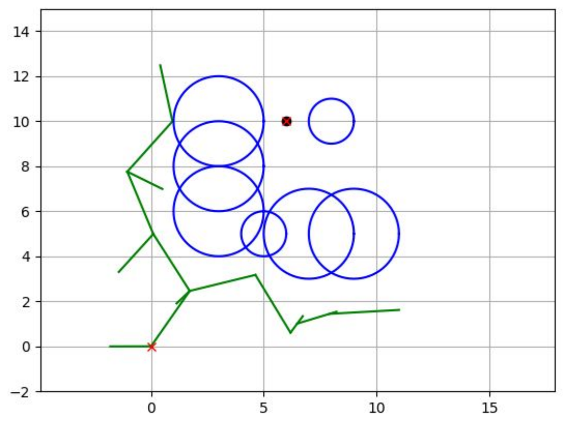
```
python rrt_toy.py
```

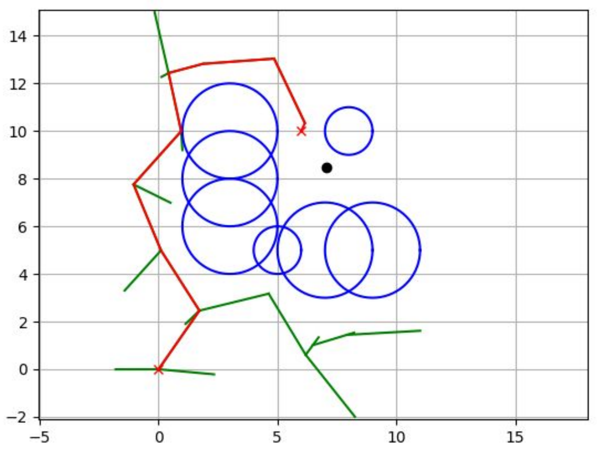
The simulation images are stored in imgs/ and also the video in final_race/lab/videos/

Shown below are some snapshots in toy environments









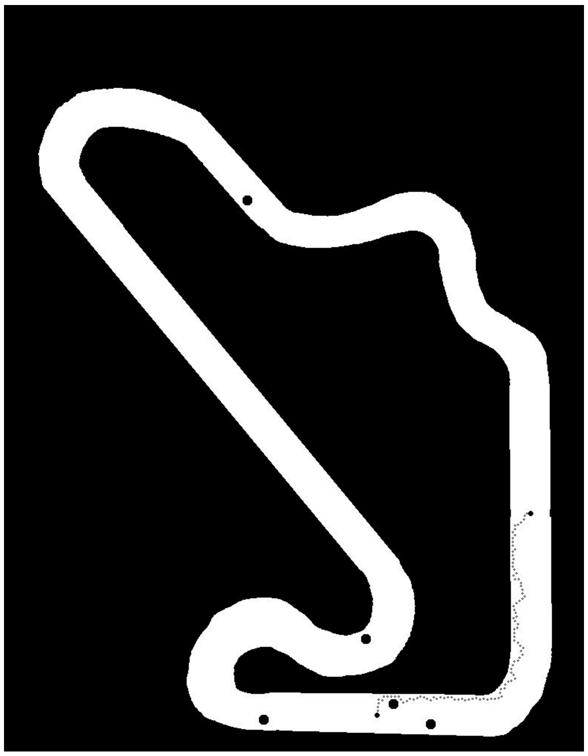
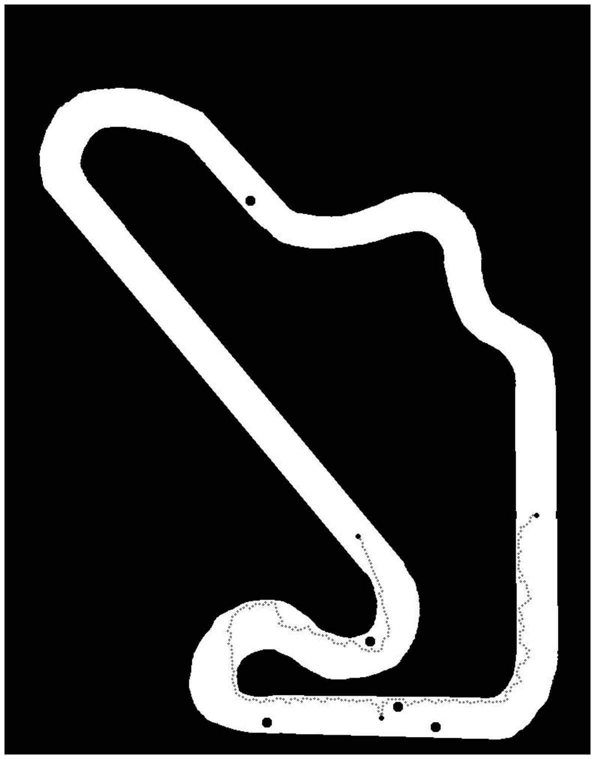
As shown above, the starting point is (0,0) and the goal point is (6,10). The RRT generates a plan shown in the red line.

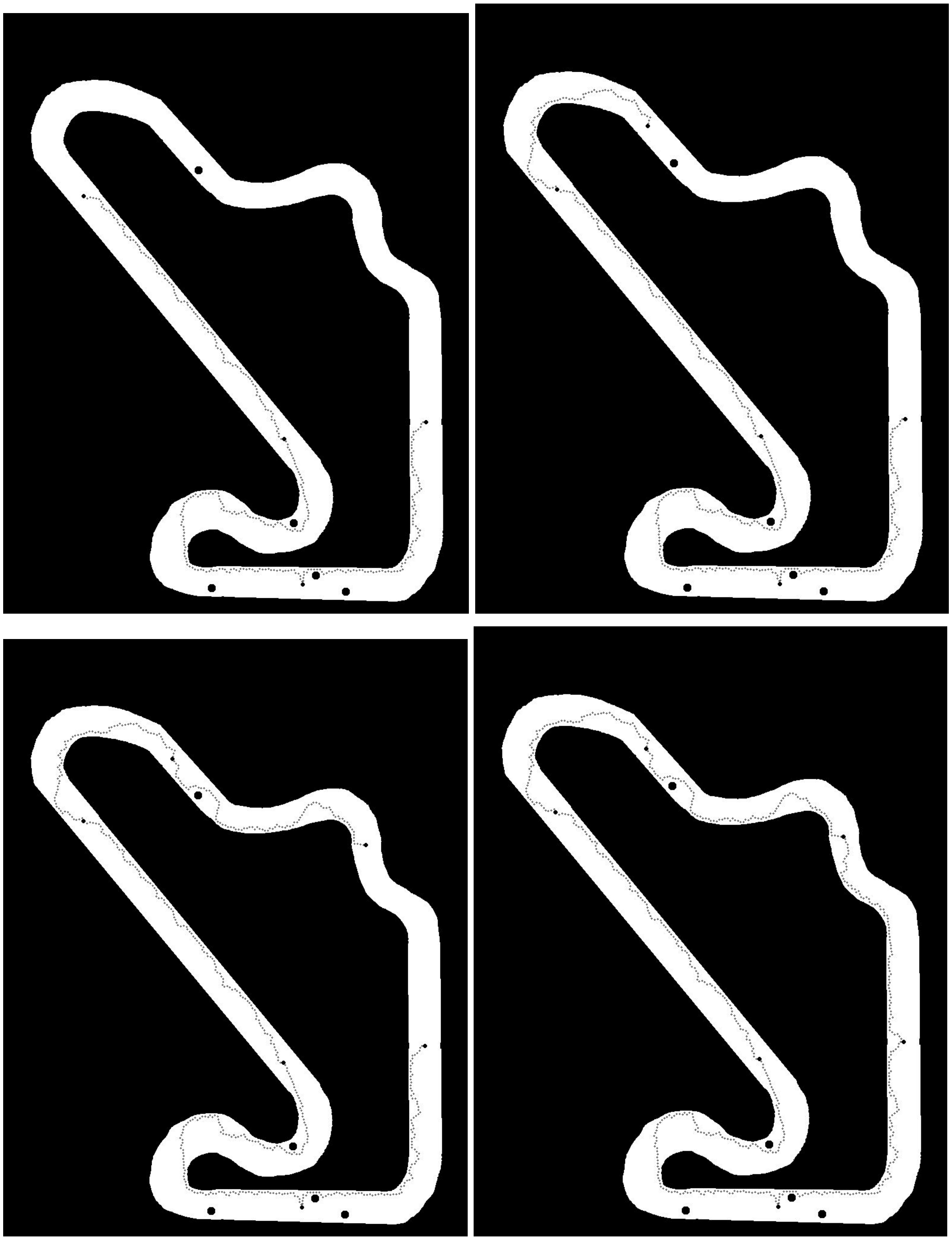
With the same algorithm, we apply this to the offline race map provided in pgm format. We also use the map resolution and scale parameters provided in the yaml file. We read the map as an image and plan a path in image coordinates. This will help us convert the grid coordinate system to map coordinates system and vice-versa. We pick points from start to checkpoint 1,2,3,4,5 and convert them to grid coordinates. Since the obstacles in the toy version were circles, for the map, we threshold the image to have 0 and 255 values.

All 0 values or black regions in the image are obstacles (collisions) and all the white regions are free space or 255 values. So our collision checking is checking whether a node is free/colliding given 0 or 255 in image. Finally to plan paths. we break the start and end points into the six checkpoints and then stitch together the 6 paths as one final path. This can be seen below. The six images are basically

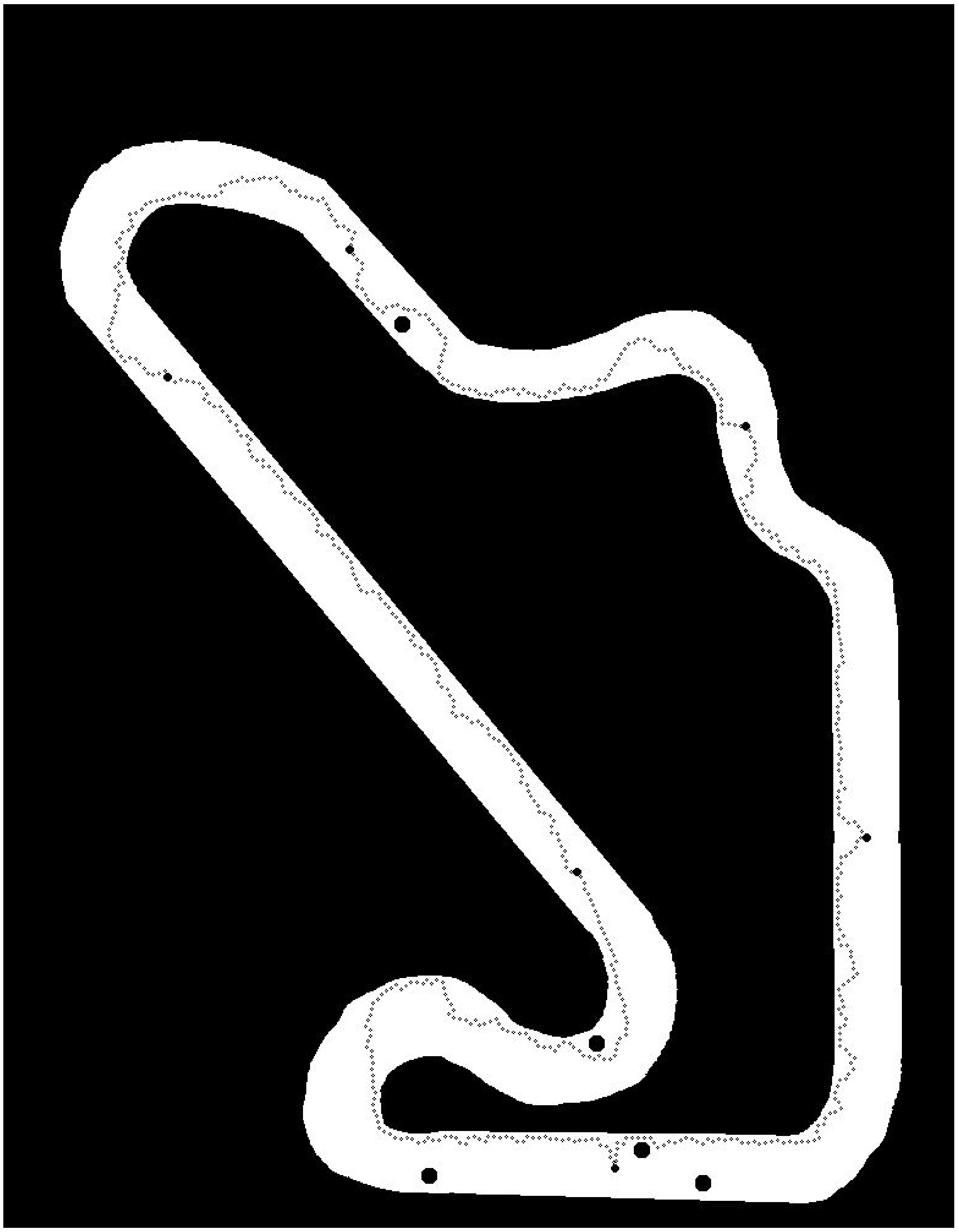
1. start to checkpoint1
2. Checkpoint1 to checkpoint2
3. Checkpoint2 to checkpoint3
4. Checkpoint3 to checkpoint4
5. Checkpoint4 to checkpoint5
6. Checkpoint5 to start.

The dotted line denotes the path from start to goal position.



We store all path planned from start to Checkpoint 1,2,3,4,5, in ../ imgs/
Finally, we store all waypoints in csv format as '../waypoints/rrt_waypoints.csv'.
As you can see below the plath planned by rrt is quite jagged and not good for stable tracking. So we decided to skip the RRT planned path and use our own waypoints that were optimized for the final race.



To run the RRT planner for the race, go to scripts/
You can try out in scripts/rrt_race.py

```
python rrt_race..py
```

Adaptive Pure Pursuit:

All of the Pure Pursuit part until now was running on the fixed lookahead distance set for specific angle range and corresponding velocities.

One thing we realized was the just normal pure pursuit was not enough because with that code we were clocking around 100 sec and if we changed simulator parameter max speed to 20 we were clocking 140 s with the same code and same environment.

Pure Pursuit can only be calibrated or tuned two parameters: Look ahead distance and Velocity
We tried a bunch of different methods or approaches to get timing from 100 secs to 40 secs.

In order to make current stack robust we decided to adopt new adaption or tuning of the pure pursuit algorithm.

1. Firstly, we minimize the max and minimum angle change limit so that the car moves straight when the angle is too small and moves at higher speeds.
2. Secondly, we used data from laser scan in order to get free distance in front of the camera and we used that in order to control the lookahead distance or update it.
3. Since we did not see significant improvement over normal Pure Pursuit, we tried to tune a PID around it so that we can have almost negligence path tracking error, but this idea was not successful.
4. After that we tried instead of tuning the lookahead distance then just why not tune the velocity itself. But this was also not such a successful endeavour.
5. We found an article or a paper (we were not able to find the link again) that used current heading to control lookahead and velocity.
6. We hardcoded the factors used in the equations and with a bunch of different values, and we observed the behaviour of the car and changed parameters to get the best tracking car possible.
7. We turned it such a way that our lookahead distance was revolving around 3m from the current car frame. The velocity was also updated based on the heading change of the car. This worked best for us and ultimately we ended up using this.
8. One more method we were pursuing was using checkpoints and if passed from certain checkpoints it was given a positive reinforcement..