F1tenth Autonomous Racing: Project Report

Shivangi Misra
shivangi@seas.upenn.edu

Dingding Zheng
zddkj@seas.upenn.edu

Weiyi Tang
tangwy@seas.upenn.edu

I. Introduction

”F1tenth” is an open-source, small-scale racing car platform widely used for teaching and research in safe autonomy. Maneuvering a racing car to finish loops in minimum time has been studied for decades. However, it’s always computationally expensive and infeasible to solve this problem by real-time trajectory planning. [1] In this project, we generated a velocity profile to find the maximum permissible speed of racing car on each waypoint on the path. Then, we use CMA-ES (Covariance Matrix Adaptation - Evolution Strategy) to generate the desired path for vehicle to track. The generated path has a relatively small curvature which allows the racing car to run at high, steady speed. Pure pursuit is used as the vehicle controller. For obstacle avoidance part, we implemented ODG-PFM (Obstacle-Dependent Gaussian Potential Field) and compared its performance VS. RRT*. In order to measure the performance of these two algorithms, we setup several testing maps and added noise to the environment.

II. Trajectory Generation

In order to obtain an optimum trajectory on the given map, we try trajectory optimization using the Covariance Matrix Adaptation - Evolution Strategy (CMA-ES). However, we did not include velocity of the racecar over the path as part of the CMA-ES optimization problem. Instead we chose to estimate speed profile separately for each path generated iteratively to ensure the car operated at speeds that considered tire-road friction and prevented it from slipping.

A. Velocity Profile Estimation

We generated velocity profile in order to find the maximum permissible steady state vehicle speed when given zero longitudinal force. [1] To generate a quick racing trajectory, we parameterized the path as a curvature K, which is a function of distance along the path s. [2] We divide our reference trajectory into smaller segments that have piece-wise constant curvature. The curvature is decided for these segments by fitting a circle to the arc segment.

U_x is the vehicle forward velocity. We ignored vehicle mass transform and other topography effects and got the maximum speed as following equation by: [3]

$$U_x(s) = \sqrt{\frac{\mu g}{|K(s)|}} \tag{1}$$

Here, μ is the tire-road friction, equal to 0.523 and g is teh acceleration due to gravity. We know that for real racing car, there is a maximum engine force constraint. Thus, we first used forward and then backward integration to smooth the velocity profile curve. In this method, the velocity of a given point is determined by the velocity of the previous point and it’s maximum available longitudinal force for acceleration $F_{x,accel,max}$ and force $F_{x,deccel,max}$ for deceleration. The speed update format is shown as below [4]:

$$U_x(s+\Delta s) = \sqrt{U_x^2(s) + 2a_{x,\text{ accel,max}}\,\Delta s} \tag{2a}$$

$$U_x(s-\Delta s) = \sqrt{U_x^2(s) - 2a_{x,\text{ decel,max}}\,\Delta s} \tag{2b}$$

The plot of velocity profile we got is shown in Fig. 1.Our reference path has 10,000 waypoints. And the blue curve represents the original velocity profile while the red one represents the updated profile after being ”smoothed”. The acceleration limits are set to be $7.51ms^{-2}$ and for deceleration as $8.26ms^{-2}$.

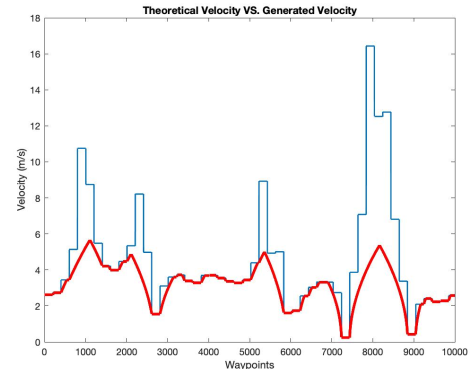


Fig. 1: Velocity Profile

The final speed profile we obtain is scaled and processed so that is constrained within speed limits of the race. The resulting speed profile is shown in Fig. 2.

B. Minimum Time

The CMA-ES algorithm is used to solve problems that require non-linear or non-convex numerical optimization. It takes inspiration from biological evolution where candidate solutions are spawned and ranked. The best solutions are then used to spawn a new class of solutions, slowly refining the parameter search in the direction that promises optimality.

We decided to pick a minimum-time trajectory which the race car would follow to finish laps around the race track faster than its opponents. A sample trajectory was obtained by running the car around the track using a reactive planning (follow

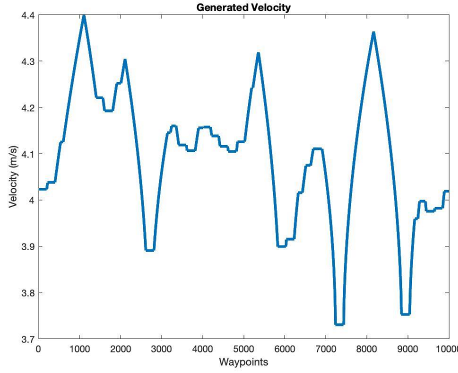


Fig. 2: Processed Velocity Profile

the largest gap) algorithm. To use the CMA-ES optimization algorithm, we parameterize the trajectory using the magnitude of displacement in the direction of normal to the tangent of the curve at each point as shown in Fig. 3. We craft the fitness function to calculate the velocity profile for each of the current candidates and use the velocity profile to calculate how much time the racecar would take to go around the track once. This value helps us rank candidate solutions from best to worst. The covariance matrix is used to express the pairwise dependencies of the variables and the covariance matrix adaptation strategy is used to update this matrix. From the mean and covariance matrix, the distribution at every iteration from which solutions are drawn is defined and on repeatedly updating these values, the algorithm is said to have converged when the fitness value of candidate solutions is less than a threshold minimum or if the user specified total number of iterations is exceeded,

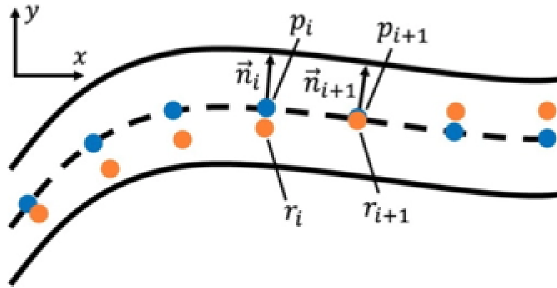


Fig. 3: Trajectory parameters [6]

We use the CMAES MATLAB implementation available online (by Hansen *et al.*) and modify the fitness function used to solve the problem at hand. The resulting optimized trajectory (orange) is shown in Fig. 4 where the original path is drawn in blue.

III. RRT*

RRT* is the optimized version of RRT, which is a path planning algorithm that relies on randomly generating points in an obstacle free area and chaining the root node to its closest neighbor point and so on, to build a graph through which

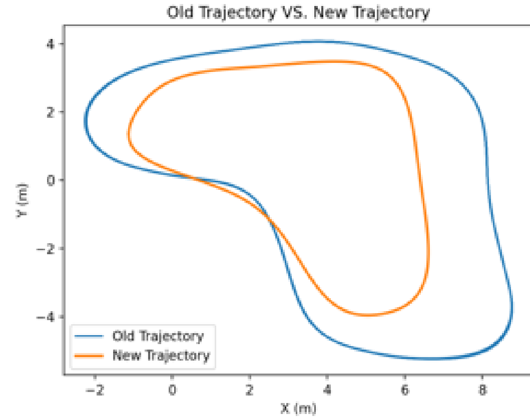


Fig. 4: Minimum time trajectory

a path to the goal is found. RRT* is an improvement as it includes two additional features, unlike RRT. RRT* records the distance each node (sampled point) has travelled relative to its parent node as a cost of the node. If a close-by node with a cheaper cost is found, then it replaces the original node. The second difference is that the tree is rearranged when adding a new node. An existing node of the tree which is closest to the new node, not necessarily having to be at the end of a branch is taken to be the new node's parent. This allows RRT* to generate smoother paths than RRT and considerably assists the pure pursuit controller on the race car, in following the reference path with more stability. The pseudo code is shown in Algorithm 1.

Algorithm 1 RRT*

1: **procedure** RRT*
2: $\quad G(V, E)$ $\triangleright$ Initialize graph with root node
3: $\quad$ **for all** iter $\leq$ max iterations **do**
4: $\quad\quad X_{new}$=RandomPosition()
5: $\quad\quad$ **while** Obstacle(X_{new}) == True **do**
6: $\quad\quad\quad X_{new}$ = RandomPosition()
7: $\quad\quad$ **end while**
8: $\quad\quad X_{nearest}$ = Nearest($G(V, E), X_{new}$)
9: $\quad\quad$ Cost(X_{new}) = Distance($X_{new}, X_{nearest}$)
10: $\quad\quad X_{best}, X_{neighbors}$=findNeighbor($G(V, E)$,$X_{new}$,$\theta_{limit}$)
11: $\quad\quad$ Link = Chain(X_{new}, X_{best})
12: $\quad\quad$ **for all** X' in $X_{neighbors}$ **do**
13: $\quad\quad\quad$ **if** Cost(X_{new})+Distance(X_{new}, X')<Cost(X') **then**
14: $\quad\quad\quad\quad$ Cost(X')=Cost(X_{new})+Distance(X_{new}, X')
15: $\quad\quad\quad\quad$ Parent(X') = X_{new}
16: $\quad\quad\quad\quad$ G += X_{new}, X'
17: $\quad\quad\quad$ **end if**
18: $\quad\quad$ **end for**
19: $\quad$ **end for**
20: $\quad$ **return** G
21: **end procedure**

Some additional details to note about our implementation of the algorithm is that we added an angular constraint θ_{limit} that constraints the growth of branches in all radial directions and focuses computational effort in building a path towards the front of the car. The frequency at which RRT* is able to return valid paths to the car is of importance because the car's movements become more unstable if there is any latency in the drive topic publishing rate. Therefore, when far away from an opponent car the goal node for the ego car is set as a point "look-ahead" distance away from its current position on the optimal reference trajectory and RRT* needs lesser and focused sampling near this goal. When the opponent car is close to the ego car, the RRT* algorithms sampling iterations are increased and the region to be sampled around the goal node is also widened. In addition, to ensure that the car does not aggressively oscillate in following a shorter path returned by RRT*, the ego car chooses to follow points that are further up the tree.

IV. ODG-PFM

In this section, we describe the Obstacle-Dependent Gaussian Potential Field works, how did we implement this in detail as well as what did we do to improve behavior of ODG-PFM.

A. Method Theory

Just like traditional Potential Field methods, ODG-PFM also utilizes an artificial potential field consisting of an attractive field and a repulsive field to push car to destination without hitting any obstacles. Unlike any other method using vector to guide car to some specific direction, ODG-PFM finds the angle with minimum values from the total field function, that is to say, we can directly collect data from 2D laser scan, then compute total field based on the each beam angle, finally find the best angle to be our steer angle.

Before talking about deep theory, we need to clarify how our laser scan collect data and what kind of data format we have. We are using 2D laser, which can be illustrated in figure 5. Each scan beams will give us distance from car to end point, so we will have a range of distant data like figure 6 shows. In our situation, we will have 1080 laser beams with angle ranging from -2.7 to 2.7.

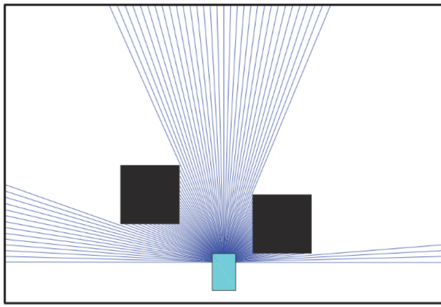


Fig. 5: Laser Scan Finder [5]

So let's beck to ODG-PFM theory part, the main idea behind this method is that, after receiving distance data from the range sensor, enlarge the obstacles with regard to the vehicle's width, and construct a Gaussian (repulsive) potential field from them. Then it calculates the attractive field from the yaw angle information from odom topic. The total field is made of these two field, and, from it, it chooses the angle with the minimum total field value.

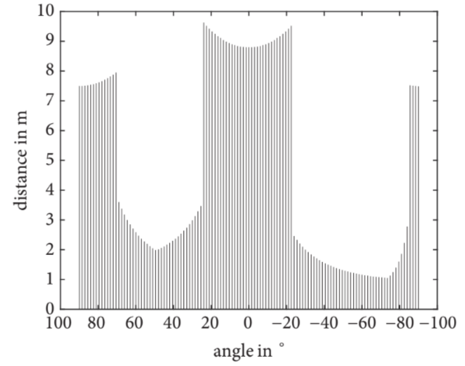


Fig. 6: Corresponding Sensor Data [5]

As figure 7 shown, if some data are spatially continuous within the threshold, they are considered as obstacles. So in this case we show, there are two obstacles. In order to implement ODG-PFM, we need to compute three values for each obstacles: the average distance to each obstacle (d_k), the angle occupied by each obstacle (ϕ_k) and half of the angle occupied by each obstacle (σ_k).

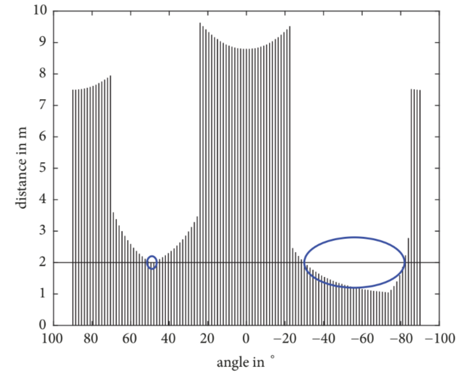


Fig. 7: Sensor data and threshold distance [5]

In real system, car's width should be taken into account. Then we should recalculate ϕ_k:

$$\phi_k = 2\sigma_k = 2\, atan2(d_k\, tan\frac{\phi_k}{2} + \frac{w_{car}}{2}, d_k) \tag{3}$$

where d_k is the average distance to the k_{th}obstacle, and w_{car} is the car's width. The illustration can be seen in figure 8. By enlarging the ϕ_k, car will perform well when avoiding relative small obstacles.

Gaussian likelihood function (repulsive field) of obstacles are calculated as:

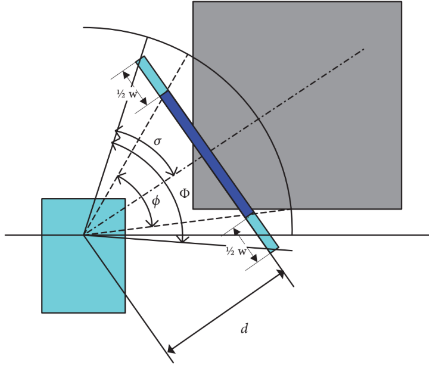


Fig. 8: Enlarge the angle occupied by obstacles [5]

$$f_k(\theta_i) = A_k\, exp(-\frac{(\theta_k - \theta_i)^2}{2\sigma_k^2}) \tag{4}$$

This is a function of θ_i. And the subscript i means that the i_{th} data segment of the sensor data and it is the sequence number of each angle. Each Gaussian likelihood function becomes a component of the repulsive field.The coefficient A_k should be set in order that the Gaussian likelihood of each obstacle fully embraces the angle occupied by each obstacle. Subsequently,

$$\widetilde{d_k} = A_k\, exp(-\frac{(\theta_k - (\theta_k \pm \sigma_k))^2}{2\sigma_k^2}) = A_k\, exp(-\frac{1}{2}) \tag{5}$$

Thus,

$$A_k = \widetilde{d_k}\, exp(\frac{1}{2}), \tag{6}$$

where $\widetilde{d_k} = d_{max} - d_k$. d_{max} is the maximum detection range of the range sensor.

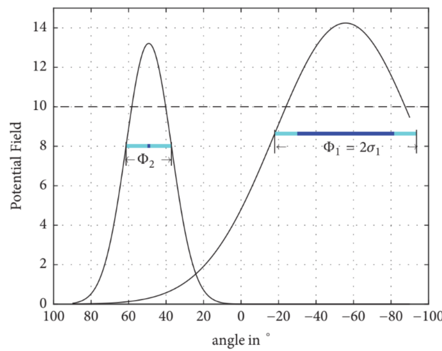


Fig. 9: Gaussian likelihood of each obstacle [5]

In the figure 9, the angle occupied by and the Gaussian likelihood of the obstacles are shown. This explains that the Gaussian likelihood of each obstacle fully embraces the occupied angle at $\widetilde{d_k}$.

Then we can obtain the repulsive field by summing up all of the Gaussian likelihood function of obstacles into Eq.7.

$$f_{rep}(\theta_i) = \sum_{k=1}^{n} A_k\, exp(-\frac{(\theta_k - \theta_i)^2}{2\sigma_k^2}) \tag{7}$$

This is also a function of angle θ_i. The attractive field is calculated as

$$f_{att}(\theta_i) = \gamma|\theta_{goal} - \theta_i| \tag{8}$$

The total field is calculated by adding these two fields. Since both of these two fields are function of θ_i, the total field is also a function of θ_i:

$$f_{total}(\theta_i) = f_{rep}(\theta_i) + f_{att}(\theta_i) \tag{9}$$

Finally, the angle minimizing the total field is chosen as the heading angle. The small circle in figure 10 points out this angle.

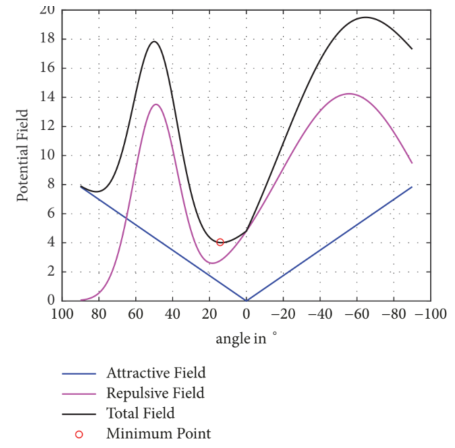


Fig. 10: ODG-PF force field [5]

In conclusion, this method has some features that are not like in classical potential field-based methods, First, instead of putting the range sensor data into equation, ODG-PF defines obstacles (occupied angle, average distance, etc.) from range sensor data and then calculates the repulsive fields of the obstacles. Second, the repulsive field and the attractive field of ODG-PF are functions of angle; that is to say, they are not vectors. In the classical potential field, the direction is decided by the direction of the total vector of the repulsive field and the attractive field. Thus, ODG-PF is very robust that even if there are small changes in the environment, the trajectory is not so much affected by them. Mathematically, the main difference between PFM and ODG-PF is that the attractive field and the repulsive field (as well as the total field) are angle functions while all of the fields in PFM are vector values.

B. Implementation and Simulation

In this part, we will introduce how we implement OG-PFM in Rivz simulator by using ROS.

First, we need to find three important variables for each obstacles: average distance (d_k), the angle occupied by obstacles (ϕ_k) and half of the angle occupied by obstacles (σ_k). We first

Parameter	Value
Velocity (m/s)	4.5
$d_{max}(m)$	50
γ	5
Look Ahead Distance (m)	2
Angle Gain	0.8
$w_{car}(m)$	0.2
Obstacle Threshold (m)	1.0

TABLE I: Parameters of Simulation

wrote a for loop to go through all the sensor range to find the number of obstacles, the first index of each obstacle and the last index of each obstacles. Then utilizing the information we got above to compute the above three variables we need to have. After this we would have three list storing d_k, ϕ_k and σ_k respectively. Then writing another for loop to go through all the θ_i to compute the repulsive field.

As for attractive field, firstly, we should find the goal point from the waypoints we have. We set a look ahead distance to find the points we want to track and recorded the speed we would use, and if there's no point lie in this look ahead distance, we used interpolation to generation a new point to track. After finding the goal point in global coordinate, we subscribed orientation data from odom topic, then computed rotation matrix to convert global coordinate to car local frame. Finally, we were able to find θ_{goal} and construct attractive field function.

After repulsive field and attractive field are construct, we could add them together to get total field. In order to compute the heading angle, we wrote a for loop to go through all the elements in total field to find the minimal one. Finally, multiplying it by a gain to get steer angle.

We also made some improvement. First of all, in order to decrease the vibration of car, we wrote an algorithm to stabilize our car. When we tried to find the best angle to be our heading angle, we set a range above the global minimal, and inside this range there are few candidates. We would pick the one which is closest to goal point. To implement this, we first computed the mean value of total field, then we kept elements whose value is less than mean*0.02, which would give us roughly 10 candidates. Finally, we find one has the smallest index distance to θ_{goal}. After implementing this, our car would be more stable than before.

In the simulation, some parameters we set can be seen in TABLE I. γ is the value that we should choose. If γ is too small, the vehicle will avoid obstacles but the path will be inefficient whereas if it is too large, the vehicle will be more likely to collide with obstacles. The γ value was chosen as 5.0 by executing several simulations. Look ahead distance is also a very important parameter. If it's set small, the car will track the waypoints more aggressively, that is to say, the car will make more sharp turning to track each point. And if it is relative large, the car will have a more smooth performance.

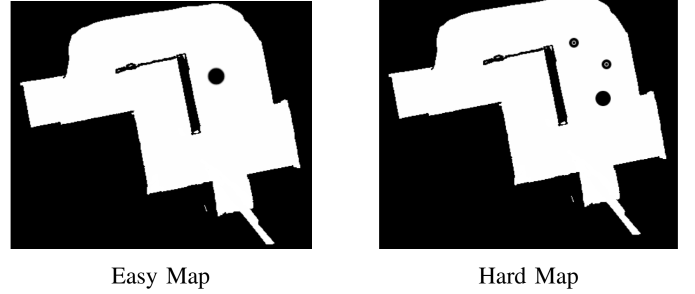
V. TEST RESULT

After making OG-PFM local planner work, we need to compare its performance with another local planner: RRT* which we already implemented before.

In order to compare their performance, we set three different tasks. The first task will test how good they are when avoiding static obstacles. The second task will test how fast they can be when there's no obstacle. The third task will test how robust they are when we added noise to laser scan data.

A. Task1

In this task, we will test how good OG-PFM and RRT* are when avoiding static obstacles. So we make two different level maps, as you can see in figure 7.



Easy Map Hard Map

Fig. 11: Test Maps

Also we want to test how fast these two methods will converge when speed increases, so we set three different speed level: 1.5 m/s, 3 m/s and 4.5 m/s. As a result, we run our car with different speed level in two maps respectively, then see their performance. The test result can be seen in TABLE II.

From the result, we can see RRT* did a great job and passed all the case, while OG-PFM isn't able to pass hard map when speed increases, which suggests that OG-PFM has a relative small converge rate, that is to say, it will require more time to converge to a optimal path, so when speed increases, it doesn't have enough time to generate a path to avoid obstacles.

Also, during test, we found that OG-PFM is so sensitive to obstacles. For instance, if we set obstacle threshold to be 1m, our car will have some avoiding behavior when the obstacle is 1m away from it, which may cause vibration and even loss control sometimes.

B. Task2

In this task, we will let our car run without any obstacle and record the time they will take for 2 laps. And the speed is set as 4.5 m/s for both methods. The result is in TABLE III.

As the table shown, OG-PFM even performed better than RRT*. When there's no obstacle, OG-PFM can be considered as pure pursuit method, since only attractive field works. So compared to RRT*, OG-PFM may be more stable when on obstacle appears in their path.

C. Task3

For task3, we artificially add random noise to laser scan data collected from LaserScan topic. To be more specific, for each laser angle, we random generate noise between -0.1 and

TABLE II: Task1 Result

Method (speed (m/s))	Easy Map (1.5)	Easy Map (3)	Easy Map (4.5)	Hard Map (1.5)	Hard Map (3)	Hard Map (4.5)
RRT*	pass	pass	pass	pass	pass	pass
OG-PFM	pass	pass	pass	pass	failed	failed

TABLE III: Task2 Result

Method	Time for 2 laps (s)
RRT*	12.96
OG-PFM	11.52

TABLE IV: Task3 Result

Method	1.5 m/s	3.0 m/s	4.5 m/s
RRT*	pass	pass	pass
OG-PFM	pass	failed	failed

0.1. Then running our car in easy map with different speed level. The result is in TABLE IV.

From the result, RRT* easily passed all cases, while OG-PFM failed when speed increases. During the test, OG-PFM method vibrated a lot with noise. As we discussed before, OG-PFM is so sensitive to obstacles, so when noise is added, it would become even more unstable. It did pass one case, but actually it vibrated a lot and it took the advantage of low speed to have more time to compute optimal path. But when speed increases, it failed.

VI. Discussion

In our comparison between RRT* and OG-PFM, we found that overall RRT* would be a more reliable candidate method for static obstacle avoidance. In a racing scenario however, the opponent cars are dynamic obstacles and in a situation when the ego car is close to overtaking the opponent car, it becomes imperative to have a mechanism to ”predict” the opponent car's next step so as to avoid a collision. Some strategies that we introduced into our system are treating the opponent car as a static obstacle and slowing down when overtake is not possible at extremely close quarters using the static obstacle time-to-collision metric, or when more maneuvering space is available, extending the dimensions of the obstacle field around an opponent car so the ego car chooses a longer path at a safer distance from the opponent car to overtake. We believe that RRT* is able to generate safe paths at a sufficient frequency to allow us to use our method of treating dynamic obstacles as blown up static obstacles to avoid collision. However, this method may not be robust enough to handle some edge cases that did not show up in the simulated races between the ego car and an instructor provided opponent car. While the blame for which agent causes a collision remains ambiguous at this point, with the handicap of not having an active predicting strategy for the behavior of an opponent car, the ego car will not be able to avoid collision if the opponent car swerves suddenly into its path. The main controller on the car is pure pursuit and we observe that on careful tuning the car does follow its path somewhat accurately but does not take the car dynamics into consideration, unlike a receding horizon model predictive controller. It is because of this that we observe the car taking wider turns than the reference trajectory dictates. With an MPC controller, much more robust control would have been possible. For future improvements in this project, we would like to implement a MPC controller for these reasons along with a more robust dynamic obstacle avoidance strategy.

References

[1] Nitin R. Kapania, John Subosits, J. Christian Gerdes. *A Sequential Two-Step Algorithm for Fast Generation of Vehicle Racing Trajectories*. DSCC2015-9757, V003T50A005.

[2] Lipp, T., and Boyd, S., 2014. *“Minimum-time speed optimisation over a fixed path”*. International Journal of Control, 87(6), pp. 1297–1311.

[3] Subosits, J. K., and Gerdes, J. C., 2015. *“Autonomous vehicle control for emergency maneuvers: The effect of topography”*. American Control Conference (ACC), pp. 1405–1410.

[4] Velenis, E., and Tsiotras, P., 2008. *“Minimum-time travel for a vehicle with acceleration limits: Theoretical analysis and receding-horizon implementation”*. Journal of Optimization Theory and Applications, 138(2), pp. 275–296.

[5] Jang-Ho Cho, Dong-Sung Pae, Myo-Taeg Lim.et al. *A Real-Time Obstacle Avoidance Method for Autonomous Vehicles Using an Obstacle-Dependent Gaussian Potential Field*. Volume 2018, Article ID 5041401, 15 pages.

[6] Fabian Christ, Alexander Wischnewski, Alexander Heilmeier, Boris Lohmann. (2019) *Time-optimal trajectory planning for a race car considering variable tyre-road friction coefficients*. Vehicle System Dynamics 0:0, pages 1-25.