PD and RRT* in GRAIC Racing Simulator

ECE484: Principles of Safe Autonomy, Software Group 03

Hongqing Liu *
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, USA
hl85@illinois.edu

Michael Vilsoet *
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, USA
mvilso2@illinois.edu

Jingwei Bao *
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, USA
jb52@illinois.edu

Yiyang Xu *
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, USA
yiyangx6@illinois.edu

Abstract—We have created a racing pipeline for the GRAIC simulation competition which involves the implementation of PD controllers, graph configuration space building, and RRT* search algorithms. The agent (Ego) is able to traverse many different types of racetracks while avoiding obstacles. Perception data such as static obstacle waypoints, agent velocity, transformation matrices, and boundary locations are assumed to be "perfectly" grounded as truth. Therefore, this paper will focus on planning and control of the agent. The novelty of our pipeline comes from the combination of established algorithms and our path classification and the secondary PD control of the throttle to match a specified velocity. Our controller is able to achieve an impressive time which is outlined in the results section after implementations of various optimizations. Finally, we evaluate our pipeline on several tracks where obstacles are present. In these cases, success is defined by completion of the race without hitting any obstacles which we are successful in doing for every racetrack scenario.

I. INTRODUCTION

The purpose of this project is to further explore two areas of autonomy. We would like to improve our knowledge of both planning and control since we have access to a robust simulator GRAIC. Our purpose in conducting this study is to exercise our understanding of the inner workings of software design for autonomous vehicles. Autonomous driving companies use these high-level robotics concepts as possible decisions in their final products. We acknowledge that outside the simulation, control and safety is far more robust. However, we also assert that the basic underlying concepts behind vehicle planning and control are very similar to what is outlined in our project. To fulfill the purpose of our project we have two main goals: 1) to implement integration and custom decision-making in a PD controller, and 2) to build an RRT* graph which would allow the car to avoid obstacles. The PD controller is ubiquitous in the realm of robotic control for its simplicity and accuracy. RRT* is known for its speed and is often compared with A*, both commonly implemented in industrial robotics.

II. SYSTEM

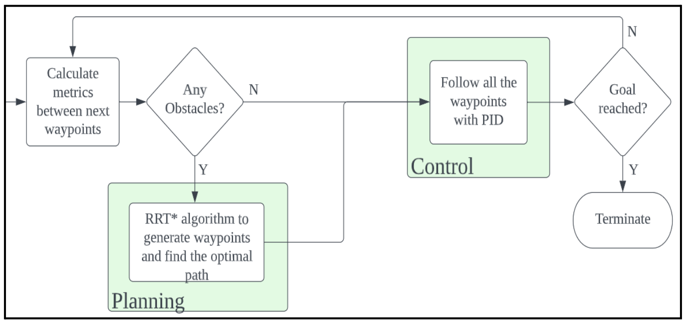

Calculate metrics between next waypoints
Any Obstacles?
N
Y
RRT* algorithm to generate waypoints and find the optimal path
Planning
Follow all the waypoints with PID
Control
Goal reached?
N
Y
Terminate

Fig 1. Overview of our planning (II.A) and control (II.B) strategies.

A. Control in autonomy pipeline

For our control, we used a PD controller to ensure the vehicle followed the waypoints given on the roadmap. Variables x, y, θ, as well as waypoints are imported into this system and follow the matrix formula:

$$u = K * x \tag{1}$$

Where x is the input, K is the progress gain, and u is the result after control. In each run step, our goal is to find the difference between Ego's current value of both position and velocity and the desired value at the next waypoint. Ego's current parameters x, y, θ, and v are given by the Carla Vehiclecontrol library. These values are then compared with the reference value which comes from the top of the waypoints queue [W]. We have simplified our vehicle

* All authors are of equal contribution and order was decided by a series of coin flips.

dynamics to a 2D system since our vehicle never moves above or below the plane of the road. Since we get a 1x4 matrix as the input of the PD control, we designed the progress gain as a 2x4 matrix which are multiplied together:

$$u = \begin{bmatrix} kx & 0 & 0 & kv \\ 0 & ky & k\theta & 0 \end{bmatrix} * [\Delta x \;\; \Delta y \;\; \Delta\theta \;\; \Delta v]^T \tag{2}$$

Where x is the linear motion, y is the angular motion, θ is the angular velocity, and v is the linear velocity.
We then manually tuned the parameters in k matrix to make the vehicle follow the waypoints smoothly, which corresponds to "Example I" in Table I.

However, there are further optimizations that we decided to implement. The car would crash once we set higher speed for the vehicle no matter how we tune the k matrix, and our time was slower than we had hoped. We realized that vehicle velocity can not be constant during the whole driving process. Instead, we classify two different path types "normal" and "turning" by calculating the difference in headings between the current waypoint and the next waypoint. A "normal" path is mostly straight and should have a high velocity goal, while a "turning" path is more dangerous and will cause Ego to slow down.

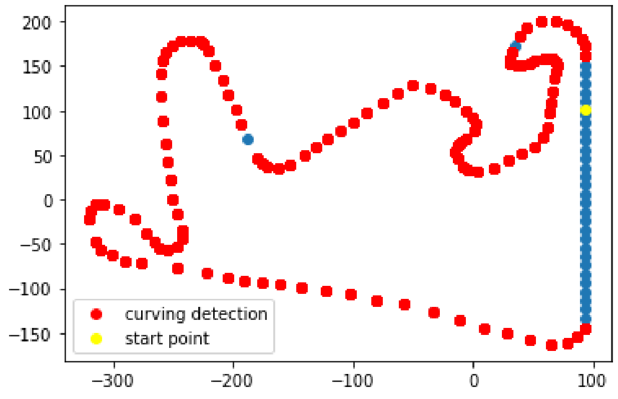


Fig. 2. Filtered points which the system considers as curving with threshold radius = 1. Blue points are "normal" while red points are "turning".

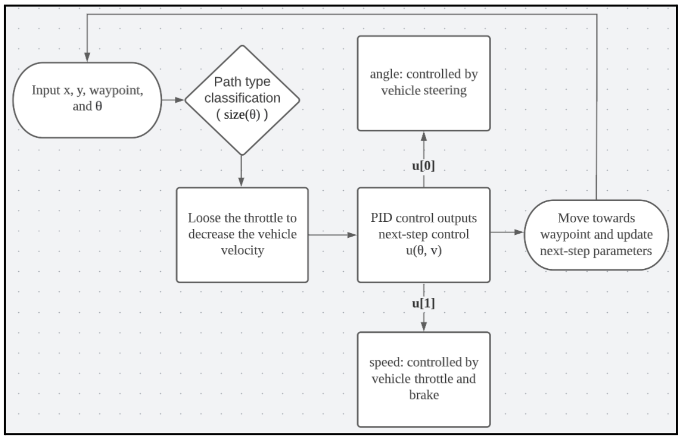
After we tune the value of the threshold radius, two sets of parameters with better performance come out which are able to finish the race with no human intervention.

TABLE I.

PD Tuning Parameters			
	Example I	*Example II*	*Example III*
k_X	10	0.2	0.2
k_Y	0.75	1.5	1.5
k_θ	1	0.135	0.135
k_V	1	1	1
Reference Velocity	10	20	15
Throttle: normal	1	0.6	0.8
Brake: normal	1	1	1
Throttle: turning	none	0	0
Brake: turning	none	1	1
Score: Lap Time	143.57	125.31	122.50

Table.1. Parameters of PD controller.

The total time for both sets is nearly the same in one racetrack, but results are much different when the environment is changed. The parameters in Example II have the advantage of smoother turns by the vehicle. However, the cost is that its maximum velocity is limited to 15 meters per second. If the straight sections of the racetrack were longer by making our "turning" classification threshold less sensitive, the score may improve at the cost of safety.



Input x, y, waypoint, and θ
Path type classification (size(θ))
angle: controlled by vehicle steering
u[0]
Loose the throttle to decrease the vehicle velocity
PID control outputs next-step control u(θ, v)
Move towards waypoint and update next-step parameters
u[1]
speed: controlled by vehicle throttle and brake

Fig. 3. Block diagram describing system PD control structure. After a path is classified, tuning parameters are altered to maximize speed and safety.

B. Planning in autonomy pipeline

For the path planning part, firstly, we thought about using some graph search algorithm like Dijkstra's algorithm. However, Dijkstra's and similar algorithms are more useful for large datasets rather than a few waypoints like in our case. Since we need our car (Ego) to find the path in a very short time (T_reaction), we want the algorithm to create a graph and find the optimal path simultaneously. Thus, we opted to use RRT* to find the optimal path around obstacles. To cycle through our list of waypoints, our system always checks the first element (index zero) in the *waypoints* list , which is also the closest waypoint to the car currently. The obstacles' vertices are added to an obstacle list based on the distance from the vehicle for collision check. We then use RRT* to find the optimal path around the obstacle . When entering RRT* algorithm, we take the vehicle's current position as the starting point and next waypoint (element with index one in the *waypoints* list) as the goal point.

RRT Algorithm Generate RRT* graph

```
T.init($x_{init}$); E ← None; T ← (V, E);
For i = 1 to K do
    $x_{rand}$ ← Sample(i);
    $x_{near}$ ← Near(V, $x_{rand}$);
    if $x_{near}$ = none then
        $x_{near}$ ← Nearest(V, $x_{rand}$);
    end if
    $x_{parent}$ ← FindBestParent($x_{near}$, $x_{rand}$);
    If $x_{parent}$ != NULL then
        V ← V U {$x_{rand}$};
        E ← E U {($x_{parent}$, $x_{rand}$)};
        E ← Rewire(E, $x_{near}$, $x_{rand}$);
    end if
end for
Return T = (V, E);
```

First, we build the working environment for the RRT* algorithm. When detecting obstacles, the system will locate the waypoint that is the closest to the obstacles. When the vehicle is moving to this particular waypoint, we set the vehicle's current position as the starting point, and its next waypoint as the goal point. We then take three points(first, middle, and last) from each boundary to form a space as the shape of a polygon. Delta is the proportion constant that the entire space will shrink into, and the remaining space will be the configuration space of the RRT* algorithm.

Second, the RRT* randomly generates a point in the configuration space, and the point has a probability equal to goal_sample_rate to be generated at the position of the goal point. Then, it searches for its nearest neighbor using brutal force search. After going through the collision check with all the obstacles, it builds an edge from the neighbor if it's collision free. The length of the edge is never greater than the step_length. The new vertice added becomes a new node in the tree, and it has its neighbor as the parent. If time permits, we want to create different step_lengths depending on the distance between each waypoint. After that, it calculates the cost from each node inside the search_radius of the new node to the start node and adds it to the cost from it to the new node and sets the node with minimum total cost as the new parent.

Third, we build a circle and take the goal node as the center. Then, we find a set of nodes which are within the circle to calculate the cost from the start node to itself and add it to the cost from itself to the goal node. We take the node with minimum cost into the tree.

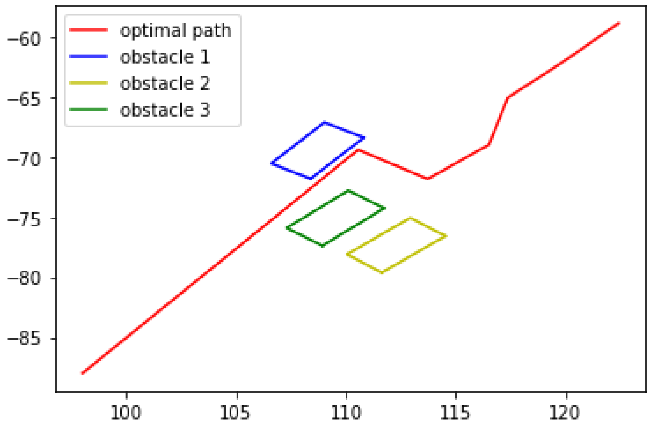


Fig. 4. A visualization of the obstacles' location and planned path .

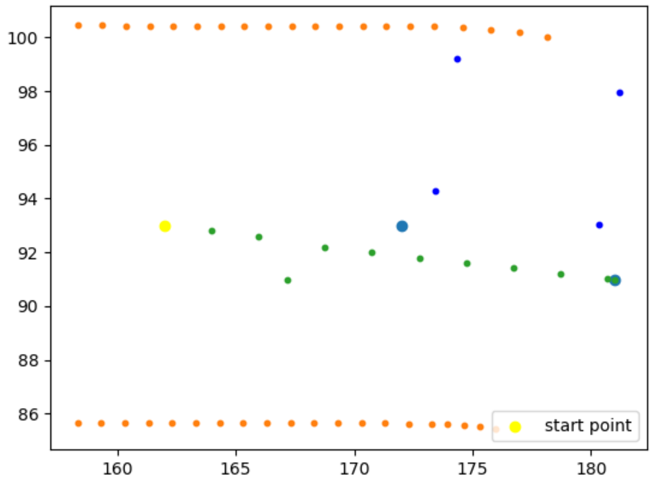


Fig. 5. One example iteration of RRT* Algorithm. Ego plans (green dots) on making a sharp right turn before resuming the normal path. Ego successfully avoided all of the obstacles in blue.

Finally, we rewire Ego's path with the new RRT* path. Each vertex is optimal because it has the lowest cost to go, and is marked as "reached" when the car is within a small (again, less than 2 meters) distance from it. Our resulting reaction time (T_reaction) is kept very small and is discussed soon.

III. EVALUATION AND CONCLUSION

Our score for the Shanghai track with obstacles was 122. seconds and without obstacles it was 114 (before CARLA crashes). We were able to utilize RRT* and our custom algorithms to avoid all collisions on the track as shown in our video. We note that the RRT* search is quite a bit faster than other algorithms we have tried. This allowed us to calculate a very small T_reaction. In practice, we simply need to have a T_reaction small enough to avoid obstacles. But for true safety, we should be able to beat the reaction time of humans so that it is worth it to implement our pipeline instead of the human driver. In the simulation, obstacles appear in the middle of the screen, which mimics the reaction time of the camera or other sensor that finds the object. So, it is our job to swerve out of the way within just a few hundred milliseconds. After trying out many different python functions, we decided that a numpy implementation is far faster and utilizes parallel programming for matrix multiplication. Thus, our T_reaction time, which is dominated by the calculation time of our RRT*, was reduced down to about 150 milliseconds by examining many swerves in the example videos. If given more time, we would implement the integration term in the PID controller. The reason for this is to increase safety by reducing steady-state error between Ego and the waypoints. In real-life, this steady-state error translates into the car traveling between lanes, which is unsafe and illegal. Nevertheless, we are proud of the pipeline we have created and succeeded in our two goals: 1) to implement integration and custom decision-making in a PD controller, and 2) to build an RRT* graph which would allow the car to avoid obstacles.

IV. REFERENCES

A. Citations

[1] S. Sedighi, D.-V. Nguyen, and K.-D. Kuhnert, "Guided Hybrid A-star Path Planning Algorithm for Valet Parking Applications," *IEEE*, 2019. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8813752.

[2]Veronica, "Using Carla Simulator to Realize DQN Self Driving", Veronica1312 CSDN, 21-March-2022. Available: https://blog.csdn.net/weixin_44169614/article/details/120259664

[3] Huiming Zhou, "Path Planning: Commonly Used Path Planning Algorithms With Animations.," *zhm-real GitHub*, 14-Dec-2020. Available: https://github.com/zhm-real/PathPlanning. [Accessed: 02-Dec-2022].

[4] A. Khanal, "RRT and RRT* Using Vehicle Dynamics," *arXiv.org*, 29-May-2022. Available: https://arxiv.org/abs/2206.10533.

[5] M. Jiang, Z. Liu, K. Miller, D. Sun, A. Datta, Y. Jia, S. Mitra, and N. Ozay, "GRAIC: Simulator Framework for Autonomous Racing," *GRAIC*. Available: https://popgri.github.io/Race/. [Accessed: 06-Dec-2022].

B. Video Links and GitHub code

Below is a link to our demonstration of the CARLA race that we ran using our project. We have a scenario with and without obstacles:

➔ https://youtube.com/playlist?list=PLpq6GSTCgUv-IA2JJ5CPfdMqOmwbiuJGa

Here is the link to our updated GitHub repository with instructions on how to run the code. If GRAIC CARLA simulation is already set up on your machine, only the agent.py file is required to run it:

➔ https://github.com/mvilsoet/CARLA_RRT_PD