Setting the new record in the ROAR Simulation Racing Series

Virgile Foussereau[1] and Quang Huynh[2]

***Abstract*— This paper presents the record-breaking solution developed by the team Laplace Racing for the Robot Open Autonomous Racing (ROAR) Simulation Racing Series. Our solution outperforms all previous competitors in the autonomous racing domain by leveraging a model-free combination of Pure Pursuit for lateral control and a PID controller for longitudinal dynamics. Emulating the racing vehicle in the CARLA Research Environment, our approach focuses on achieving maximum speed and efficiency on the Monza track. We detail the architecture of our winning controller, but also the different approaches we have taken and their potential. Furthermore, we provide insights into our methodologies and optimization techniques that contributed to our success in the competition. This paper serves as a comprehensive analysis of our winning solution, and discusses potential future developments for further enhancing autonomous racing performance.**

I. INTRODUCTION

Autonomous driving technology has seen significant advancements in recent years, offering promising prospects for safer and more efficient transportation systems. A critical component of autonomous driving systems is the development of robust control algorithms. Racing offers a unique opportunity to test these algorithms in an environment where vehicles are pushed to their extreme limits. Due to the high-risk nature of autonomous racing competitions, they are often conducted via simulation.

The ROAR Lab was established in 2019 with the mission to advance solutions of Autonomous Systems, Intelligent Machines, and extreme robotics applications [1]. The Simulation Racing Series is a competition organized by the ROAR Lab and held on the CARLA simulator [2]. The objective is to achieve three loops on the Monza Circuit as fast as possible without collisions.

In this paper, we present a comprehensive study focused on control strategies for the ROAR Simulation Racing Series and we set a new record for the competition. Our final solution combines an offline global trajectory optimization, a lateral Pure Pursuit controller and a longitudinal Proportional–Integral–Derivative (PID) controller.

[1] Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.
virgile.foussereau@berkeley.edu
[2]Department of Mechanical Engineering, University of California, Berkeley. huuquang_huynh@berkeley.edu

II. MODEL PREDICTIVE CONTROL

Model Predictive Control (MPC) is an advanced control technique widely used in autonomous driving [3]. It operates by utilizing a dynamic model of the vehicle to predict its future behavior over a finite time horizon. By formulating an optimization problem incorporating system dynamics, constraints, and performance objectives, MPC calculates an optimal control sequence to minimize a predefined cost function. This predictive capability enables MPC to proactively anticipate and adapt to system uncertainties and disturbances, resulting in precise and robust control actions. Notably, MPC is adept at handling nonlinearities, constraints, and multi-input-multi-output (MIMO) systems. However, it can be complex to implement and requires a good dynamic model of the vehicle.

As the competition takes place in a simulated environment, we decided to start with an MPC-based solution. Indeed, a simulation can offer, in theory, perfect predictability which would give a huge advantage to an MPC solution.

A. Dynamic models

The first step for MPC is to have an accurate model to predict future states of the vehicle given its current state and the applied controls. In our case, the controls are:

- δ: the car's steering angle.
- th: the car's throttle.

Both have a value in $[-1, 1]$. The steering angle control can be converted to radians using the maximum steering angle of the car but the throttle is more difficult to interpret. Indeed, most dynamic models use the acceleration as input. While they are strongly related, CARLA does not provide any documentation on the relation between the throttle and the acceleration. Due to this lack of information, our first step was to model the acceleration using experiments.

1) Modeling the acceleration: We start by running the car with a dummy control to collect data on acceleration. In figure 1, we plot the acceleration at throttle=1.0 as a function of the vehicle speed.

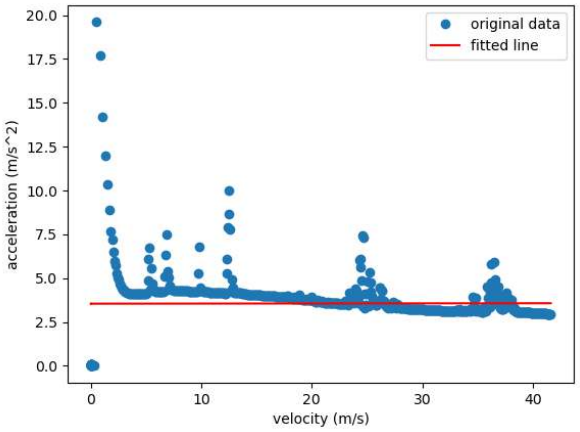

Fig. 1: Linear model of acceleration

Figure 1 seems to indicate that acceleration for a throttle of 1 decreases linearly with speed. However, trying to fit directly a linear model fails due to outliers when the vehicle starts. In order to overcome this issue, we use a RANSAC algorithm as presented in figure 2. It discards outliers and allows us to get a better model of the acceleration.

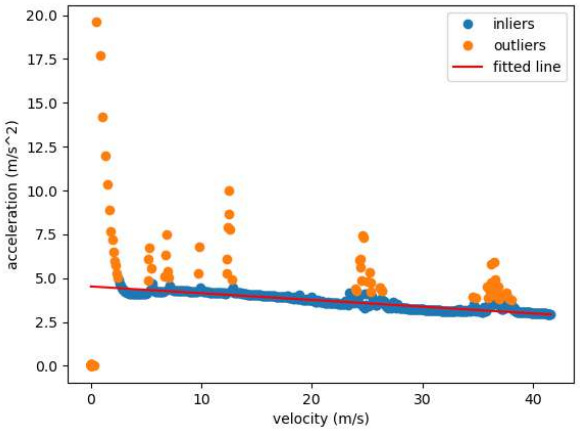

Fig. 2: RANSAC model of acceleration

Performing the same operation for different values of the throttle shows that the slope is more or less conserved, the line being simply shifted vertically. Therefore we can have the following model of the acceleration: $a = th * (\alpha - \gamma * v)$ with a the acceleration, th the throttle, v the speed, and α and γ two positive coefficients. This assumes the speed is less than $\frac{\alpha}{\gamma}$. In our case, we find $\alpha = 4.5$ and $\gamma = 0.037$.

In figure 3, we present the ground truth acceleration and the model prediction for a full lap. The model is inaccurate in transition phases such as the start or when changing gears but seems to be a decent approximation in nominal phases.

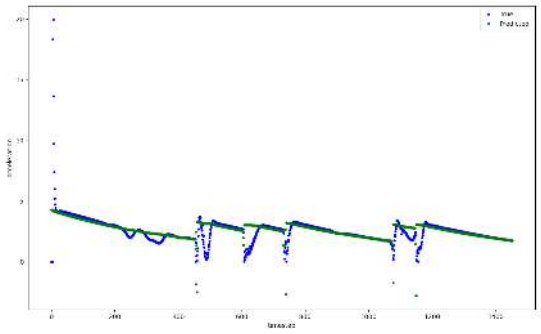

Fig. 3: Acceleration model prediction over a full loop

2) Bicycle model: The first model we used for the MPC is the bicycle model [4]. This model presents several advantages: it simplifies the complex dynamics of a vehicle into a manageable set of equations, making it easier to analyze and design control systems, and due to its simplicity, the bicycle model can be implemented in real-time control systems with low computational overhead.

The bicycle model represents a vehicle with only two wheels, where the front and rear wheel pairs are each lumped into one wheel. This simplification is justified since the roll dynamics and tire slip are not considered, so that the velocity vector v at the center of the rear axle is always aligned with the link between the front and rear wheel.

To model this problem, let introduce the variables s_x and s_y for the position of the rear wheel, v the linear velocity, v_δ the velocity of the steering angle, δ the steering angle, Ψ the heading and l_{wb} the wheelbase.

The differential equations of the bicycle model are defined as follows:

$$\begin{aligned}
\dot{\delta} &= v_\delta \\
\dot{\Psi} &= \frac{v}{l_{wb}} \tan(\delta) \\
\dot{v} &= a_{long} \\
\dot{s_x} &= v \cos \Psi \\
\dot{s_y} &= v \sin \Psi
\end{aligned}$$

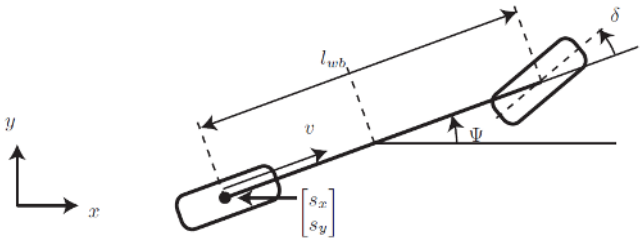

Fig. 4: Bicycle Model

After researching the Internet and the CARLA simulator, we came up with the following parameters:

Vehicle Parameter	Symbol	Unit	Tesla Model 3
Vehicle Length	l	[m]	4.719
Vehicle Width	w	[m]	2.09
Wheelbase	l_{wb}	[m]	2.875

TABLE I: Vehicle parameters for Bicycle Model

The limitation of our implementation of this model lies in its neglect of tire slip, leading to the omission of critical effects like understeer or oversteer. Consequently, when the vehicle operates near its physical limits, these essential effects are absent from the model's predictions: at high speed, the vehicle slips too much and crashes.

3) Single-Track model: Given that in racing, our vehicle is close to its physical capabilities, we decided to use the Single-Track (ST) model which considers tire slip. We leverage the implementation made for the CommonRoad benchmark [5]. In addition to the variables already introduced in the bicycle model, we additionally require the slip angle (at the center of gravity) β, the yaw rate $\dot{\Psi}$, the longitudinal acceleration a_{long}, the moment of inertia about z axis I_z, the distance from center of gravity to front and rear axles l_f and l_r, the center of gravity height h_{cg}, the cornering stiffness coefficients for front and rear $C_{S,f}$ and $C_{S,r}$ and the friction coefficient μ. The Single-Track Model is defined as follows:

$$
\begin{aligned}
\dot{\delta} &= v_\delta \\
\dot{\beta} &= \frac{\mu}{v(l_r+l_f)}(C_{S,f}(gl_r - a_{long}h_{cg})\delta - (C_{S,r}(gl_f + a_{long}h_{cg}) \\
&+ C_{S,f}(gl_r - a_{long}h_{cg}))\beta + (C_{S,r}(gl_f + a_{long}h_{cg})l_r \\
&- C_{S,f}(gl_r - a_{long}h_{cg})l_f)\frac{\dot{\Psi}}{v}) - \dot{\Psi} \\
\ddot{\Psi} &= \frac{\mu m}{I_z(l_r+l_f)}(l_f C_{S,f}(gl_r - a_{long}h_{cg})\delta \\
&+ (l_r C_{S,r}(gl_f + a_{long}h_{cg}) - l_f C_{S,f}(gl_r - a_{long}h_{cg}))\beta \\
&- (l_f^2 C_{S,f}(gl_r - a_{long}h_{cg}) + l_r^2 C_{S,r}(gl_f + a_{long}h_{cg}))\frac{\dot{\Psi}}{v}) \\
\dot{v} &= a_{long} \\
\dot{s_x} &= v\cos(\beta + \Psi) \\
\dot{s_y} &= v\sin(\beta + \Psi)
\end{aligned}
$$

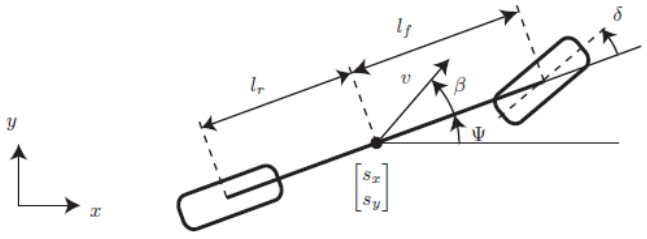

Fig. 5: Single-Track Model

After conducting research on the Internet and exploring the CARLA simulator, as well as formulating certain hypotheses, we have determined the following parameters.

Vehicle Parameter	Symbol	Unit	Tesla Model 3
Vehicle Length	l	[m]	4.719
Vehicle Width	w	[m]	2.09
Total Vehicle Mass	m	10^3[kg]	1.845
Moment of Inertia	I_z	$10^3[kgm^2]$	1.5
Distance to Front Axle	l_f	[m]	1.5
Distance to Rear Axle	l_r	[m]	1.5
Center of Gravity Height	h_{cg}	[m]	0.45
Stiffness Coefficient (Front)	$C_{S,f}$	[1/rad]	20.89
Stiffness Coefficient (Rear)	$C_{S,r}$	[1/rad]	20.89
Friction Coefficient	μ	[-]	1.048

TABLE II: Vehicle parameters for ST Model

Predictions using the ST Model were on average better than with the Bicycle model, especially in sharp turns as seen on figure 6.

B. Cost Function

Once a satisfying dynamic model has been chosen, a well-defined cost function is necessary. For a race, the general objective is to complete the circuit in the shortest time possible without collisions. This can be difficult to directly optimize. A possibility would be to track the progress along the circuit and maximize it. To avoid collisions, it could be added to an exponential cost on the distance to the circuit boundaries. In this scenario, the MPC component would have to plan its own trajectory and speed to optimize these objectives. In theory, it would be doable but would require a very long time horizon in order for MPC to foresee turns and adapt the speed accordingly. In practice, such a long time horizon would make the computations intractable in real-time. An easier possibility is to perform the path planning offline, and then have the MPC component follow the planned trajectory. We discuss the offline trajectory optimization in IV. For such a case, we derived the following cost function:

$$
\begin{aligned}
\text{Cost Function} = \sum_{i=0}^{N-1} \Bigg(& w_{\text{cte}} \cdot \text{cte}_i^2 + w_{\text{epsi}} \cdot \text{epsi}_i^2 + w_{\text{v}} \cdot \Delta v_i^2 \\
& + w_{\text{actuations}} \cdot (\delta_i^2 + th_i^2) \\
& + w_{\text{rate}} \cdot \left(\left(\frac{\delta_i - \delta_{i-1}}{\Delta t} \right)^2 + \left(\frac{th_i - th_{i-1}}{\Delta t} \right)^2 \right) \Bigg)
\end{aligned}
$$

Where:

- cte_i is the cross-track error at time step i.
- epsi_i is the orientation error at time step i.
- Δv_i is the speed error at time step i.
- δ_i and th_i are the steering angle and throttle inputs at time step i.
- δ_{i-1} and th_{i-1} are the previous steering angle and throttle inputs.
- Δt is the time step.
- w_{cte}, w_{epsi}, w_{v}, $w_{\text{actuations}}$, and w_{rate} are the weights for each term.

In this cost function, there are three main terms and four secondary terms. The cross-track error term corresponds to the distance between the current vehicle position and the closest point of the reference trajectory. It ensures that the vehicle will not deviate too much from the reference trajectory, hence avoiding collisions. Being on a valid position is not sufficient to follow correctly the trajectory, a correct orientation is also necessary, which is ensured by the orientation error term. Finally, the speed error forces the vehicle to follow the trajectory at a reference speed in order to finish the circuit in the shortest time possible.

Normalization terms are added on the actuations as, for the same result, lower use of the actuators will be preferred. Finally, normalization terms are also added on the actuations rate to keep the control commands smooth and avoid oscillations.

C. Results

We were able to implement the MPC described in this section and successfully test it on the Monza track. The bicycle model from section II-A.2 was sufficient to control the vehicle to follow the track at a speed up to 90 KPH. However, higher speed would result in too much slipping and a crash. The ST Model allowed us to take the slip angle into account and make better predictions. On offline data, we saw that the prediction from the ST Model would outperform the bicycle model by a large amount in specific cases as shown in figure 6. In the presented example, a turn at high speed is incorrectly predicted by the Bicycle Model due to the vehicle's slipping. However, the ST model also appeared to be more complex and less stable when used with standard Euler integration [6], sometimes resulting in exploding gradients. Using a more stable solver such as LSODE [7] via the Scipy library [8] avoids this issue. However, the computational cost of a MPC step is then very large and takes up to three seconds on our hardware. As a simulation step is 0.05s, this is both unrealistic and impractical for tuning. For these reasons, we decided to change our approach.

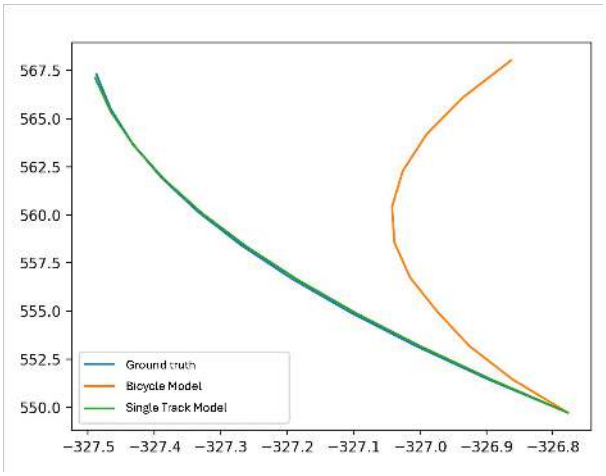

Fig. 6: Comparison of dynamic predictions

III. MODEL-FREE CONTROL

In theory, Model Predictive Control seemed ideally suited to our scenario as a simulation environment can offer a fully predictable system. However, the lack of CARLA documentation on its dynamics nullified this advantage. Additionally, the MPC high computational complexity, made it arduous to optimize for real-time situations. Consequently, we pivoted towards model-free methods, which often exhibit superior computational efficiency, as they do not require the iterative optimization process characteristic of MPC, making them suitable for real-time control applications where dynamics are unknown or uncertain.

A. PURE PURSUIT LATERAL CONTROLLER

Instead of the MPC, we then tried to implement a pure pursuit controller which is a path tracking algorithm to follow a desired path [9]. The pure pursuit controller is an automatic steering method that computes the angular velocity command that moves the vehicle from its current position to reach some look-ahead point in front of the vehicle. The algorithm moves the look-ahead point on the path based on the current position of the vehicle until the last point of the path. We can think of this as the vehicle constantly chasing a point in front of it.

The controller continuously measures the current position and heading of the vehicle as an input. Because the desired path that the vehicle should follow is known, the algorithm then draws a circle around the current position. The intersection between the circle and the desired path determines the target point. The Look Ahead Distance, which represents the radius of this circle, is the main tuning property for the controller. It corresponds to how far along the path the vehicle should look from the current location to compute the angular velocity commands. The faster the vehicle goes, the greater the look ahead distance must be.

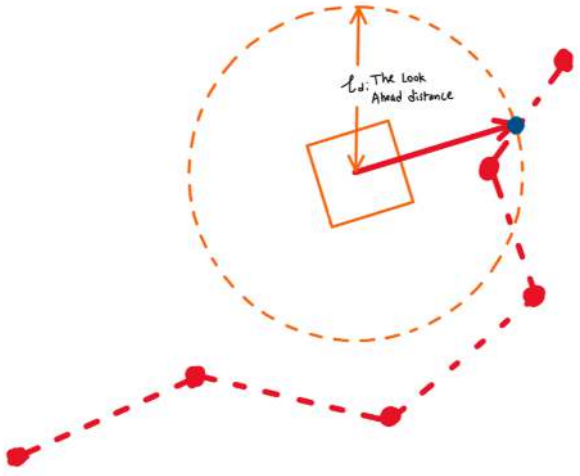

Fig. 7: Pure Pursuit controller operating principle [9]

When looking for the target point, there are 3 possibilities: no intersection between the circle and the desired path, one intersection between the circle and the desired path, or two intersections between the circle and the desired path.

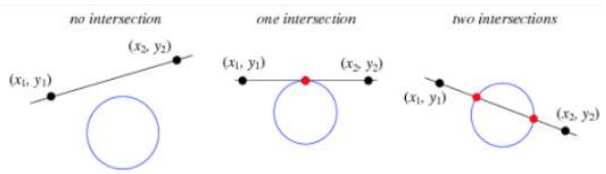

Fig. 8: Line-Circle Intersection [9]

In geometry, a line meeting a circle in exactly one point is known as a tangent line, while a line meeting a circle in exactly two points is known as a secant line.
Defining:

$$d_x = x - 2 - x_1$$
$$d_y = y_2 - y_1$$
$$d_r = \sqrt{d_x^2 + d_y^2}$$
$$D = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1$$

gives the points of intersection as

$$x = \frac{D d_y \pm sgn^*(d_y) d_x \sqrt{r^2 d_r^2 - D^2}}{d_r^2}$$
$$y = \frac{-D d_x \pm |d_y| \sqrt{r^2 d_r^2 - D^2}}{d_r^2}$$

where the function $sgn^*(x)$ is defined as

$$\text{sgn}^*(x) = \begin{cases} -1 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$$

The discriminant $\Delta = r^2 d_r^2 - D^2$ therefore determines the incidence of the line and the circle, as summarized in the following table.

Δ	Incidence
$\Delta < 0$	No intersection
$\Delta = 0$	Tangent
$\Delta > 0$	Intersection

TABLE III: Classification based on Δ value

In order to prevent the vehicle from going backward in the path, we can create a variable *lastFoundIndex* to store the index of the point it just passed. Every time the loop runs, it will only check the points that are located after *path[lastFoundIndex]*. This way, the segments the vehicle has already traveled through will not be checked again for intersections. In the cases that no new intersection has been found (vehicle deviates from the path), the vehicle will follow the point at *lastFoundIndex*.

Now that our goal point is determined, the next step is to make the vehicle move toward that point. Therefore, the controller has to calculate the steering angle required to guide the vehicle toward the selected target point. The turn error, in blue on figure 9, is the difference between the current heading and the target direction, normalized to be between $[-\pi, \pi]$. We then apply a gain K of the form $\frac{K_p}{\sqrt{v}}$, with v the linear velocity, so that the faster we go, the more we reduce the gain K and therefore the steering angle to avoid deviating too quickly. The Pure Pursuit Controller finally returns the desired steering angle: $K * TurnError$.

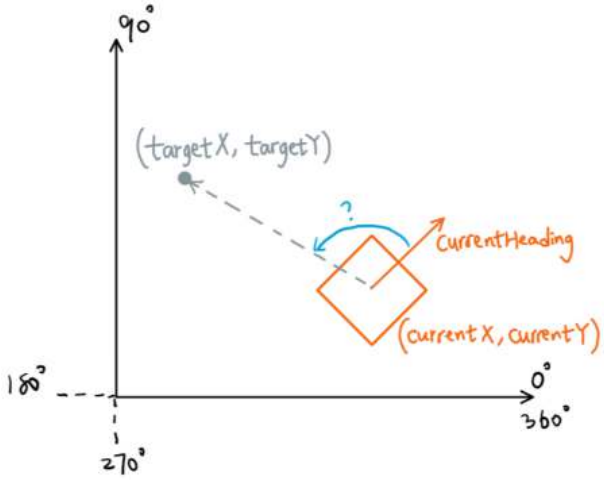

Fig. 9: Illustration of the turn error [9]

The pure pursuit controller operates as a feedback loop, it constantly senses the vehicle's position, compares it to the desired path, and makes steering adjustments to minimize the error between the actual and desired trajectories. In figure 10, the dotted gray line is the pre-computed path that the vehicle needs to follow and the solid orange line is the vehicle's trajectory. As we can see, the pure pursuit controller performs decently well.

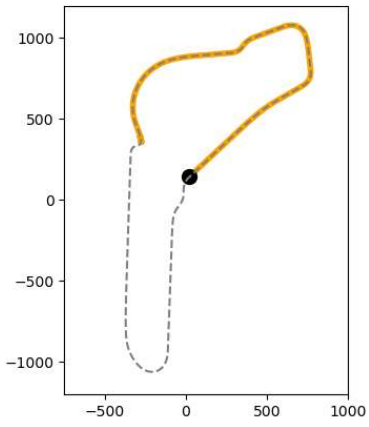

Fig. 10: Illustration of the vehicle's path tracking

B. PID LONGITUDINAL CONTROLLER

The pure pursuit controller allows us to control the steering angle of the car, but we still need to control the throttle. For this, we use a Proportional-Integral-Derivative (PID) feedback control loop mechanism to match a reference speed.

A PID adjusts the output based on the difference between a desired setpoint and the measured process variable. The proportional term responds to the current error, the integral term integrates past errors, and the derivative term predicts future errors based on the current rate of change. The integral term helps to eliminate steady-state errors by continuously summing past errors over time, effectively reducing any remaining offset between the desired setpoint and the actual speed. This allows the car to maintain consistent velocity even in the presence of external disturbances or variations in the environment (e.g. wind). The derivative term especially helps to avoid overshooting which is especially useful in racing to prevent collisions.

IV. RACE TRAJECTORY OPTIMIZATION

Separating the trajectory planning from the control can help reduce the online computation burden. In general, it also provides a more globally optimized solution. As the race track is known, we can compute an optimized trajectory offline. We use the time-optimal trajectory planning by *Christ et al.* [10]. In this method, the minimum lap time problem is described as an optimal control problem, converted to a nonlinear program using direct orthogonal Gauss-Legendre collocation and then solved by the interior-point method IPOPT [11].

Solving the nonlinear program leads to an optimized trajectory. Our MPC-based solution uses directly this optimized trajectory. Our Model-Free solution only uses the speed profile and follows the center line of the race track as a Pure Pursuit controller cuts corners. The optimized speed profile is presented in figure 11. In long straights, the vehicle quickly accelerates to its maximum speed (about 270 KPH). Before sharp turns, the vehicle decelerates to avoid a potential crash.

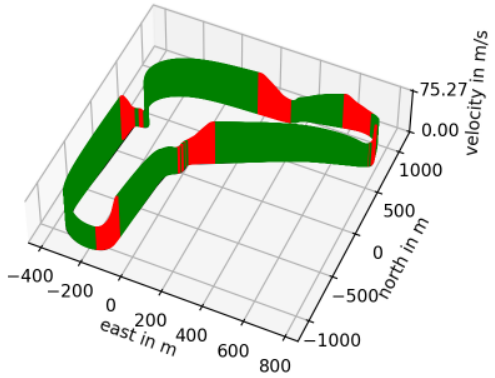

Fig. 11: Optimized speed profile for the race trajectory. Green indicates acceleration while red represents deceleration.

V. RESULTS AND CONCLUSIONS

Our final model combines a lateral Pure Pursuit Controller, a longitudinal PID controller, and a globally optimized speed profile. It outperforms all previous submissions to the ROAR Simulation Racing Series as presented in table IV. Note that this is the case as of May 6, 2024.

TABLE IV: Contest Results

Rank	Solution	Total Sim time Elapsed (s)
1	Ours	342.1
2	Mark Menaker	356.9
3	Yuehang Yang, Audrey Han, et al.	385.2
4	Ryan Chow	386.15
5	Derek Chen	418
6	Eric Li, Wayne Li, et al.	462.55
7	Krishay Garg	519.2
8	Avinash Karthik, Sourodeep Deb	1658.9

Future work might improve this result by continuing research on MPC. Especially, an implementation in C++ or using an automatic differentiation package such as CasADi [12] could vastly increase the MPC step speed, making it viable for the competition.

Reinforcement Learning is also an interesting future exploration. However, a team from the ROAR lab has already dedicated its research to this avenue [13] but has not yet reached a level close to traditional control methods in this competition.

REFERENCES

[1] "Robot Open Autonomous Racing at UC Berkeley," https://roar.berkeley.edu/, [Accessed 02-28-2024].

[2] A. Dosovitskiy, G. Ros, F. Codevilla, *et al.*, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, S. Levine, V. Vanhoucke, and K. Goldberg, Eds., vol. 78. PMLR, 13–15 Nov 2017, pp. 1–16. [Online]. Available: https://proceedings.mlr.press/v78/dosovitskiy17a.html

[3] C. E. García, D. M. Prett, and M. Morari, "Model predictive control: Theory and practice—a survey," *Automatica*, vol. 25, no. 3, pp. 335–348, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0005109889900022

[4] R. Rajamani, *Vehicle dynamics and control*. Springer Science & Business Media, 2011.

[5] M. Althoff, M. Koschi, and S. Manzinger, "Commonroad: Composable benchmarks for motion planning on roads," in *Proc. of the IEEE Intelligent Vehicles Symposium*, 2017.

[6] K. Atkinson, *An Introduction to Numerical Analysis, 2nd Ed*. Wiley India Pvt. Limited, 2008. [Online]. Available: https://books.google.com/books?id=lPV8Fv2XEosC

[7] K. Radhakrishnan and A. C. Hindmarsh, "Description and use of lsode, the livermore solver for ordinary differential equations," 1993. [Online]. Available: https://api.semanticscholar.org/CorpusID:53752439

[8] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[9] S. Xiang and V. team ILLINI, *Basic Pure Pursuit*. Purdue Sigbots, 2021. [Online]. Available: https://wiki.purduesigbots.com/software/control-algorithms/basic-pure-pursuit

[10] A. H. Fabian Christ, Alexander Wischnewski and B. Lohmann, "Time-optimal trajectory planning for a race car considering variable tyre-road friction coefficients," *Vehicle System Dynamics*, vol. 59, no. 4, pp. 588–612, 2021. [Online]. Available: https://doi.org/10.1080/00423114.2019.1704804

[11] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, pp. 25–57, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:14183894

[12] J. Andersson, J. Åkesson, and M. Diehl, *CasADi: a symbolic package for automatic differentiation and optimal control*, 01 2012, pp. 297–307.

[13] Y. Cao, T. Zhang, F. Huang, *et al.*, "Reniforcement learning platform and sample solutions for robot open autonomous racing(roar) simulation race," https://github.com/augcog/ROAR_PY_RL, 2023.