

Model Predictive Control for F1Tenth Autonomous Racing

Elio Saade
School of Electrical, Computer and
Energy Engineering
Arizona State University
Tempe, USA
esaade@asu.edu

Abstract—F1Tenth Autonomous Racing is a competition that serves as both a research and educational platform for autonomous systems. This project implements a Model Predictive Control (MPC) algorithm to track a reference racing line on the track. The algorithm is evaluated in simulation using the F1Tenth Gym environment.

Keywords—F1Tenth, Autonomous Racing, Model Predictive Control, Constrained Optimization

I. INTRODUCTION

F1Tenth Autonomous Racing is a semi-regular competition organized by an international community of researchers, engineers, and autonomous systems enthusiasts. The competition consists of racing 1/10 scale RC cars around a small bounded track in two formats: Time Trials and Head-to-Head race. Time Trials is similar to the “qualifying” format in racing, in which the goal is to drive through the track as fast as possible. In the Head-to-Head race, two cars compete on the track at the same time, and the first car to complete 10 laps wins. F1Tenth is much more than a competition; it has evolved into a well-established education and research platform on autonomous algorithms with an accurate simulator allowing rapid development.

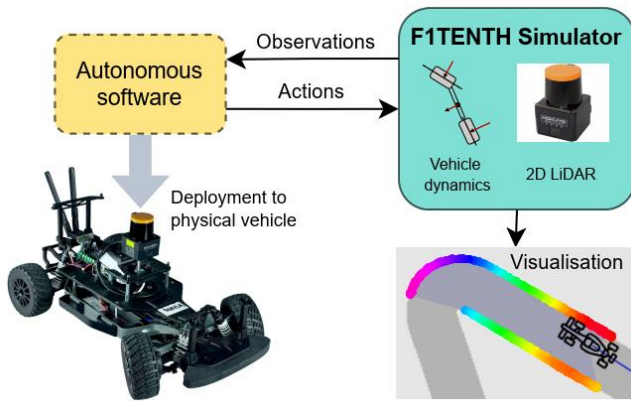


Figure 1. F1Tenth development platform for autonomous racing research: accurate simulator enables research and rapid deployment on hardware

This project consists of implementing a Model Predictive Control algorithm for an F1Tenth car to track a reference racing line on a given map. This approach is suitable for the Time Trials race, where no other cars are on the track. The controller is implemented in ROS2 and tested in the F1Tenth Gym simulation environment running in a Docker container. The project does not include any deployment in hardware.

II. MODELING

A. Vehicle Kinematics

The RC car is modeled using a single-track bicycle kinematics model, where the symmetry of the car is used to reduce it to a bicycle (Figure 2) [1].

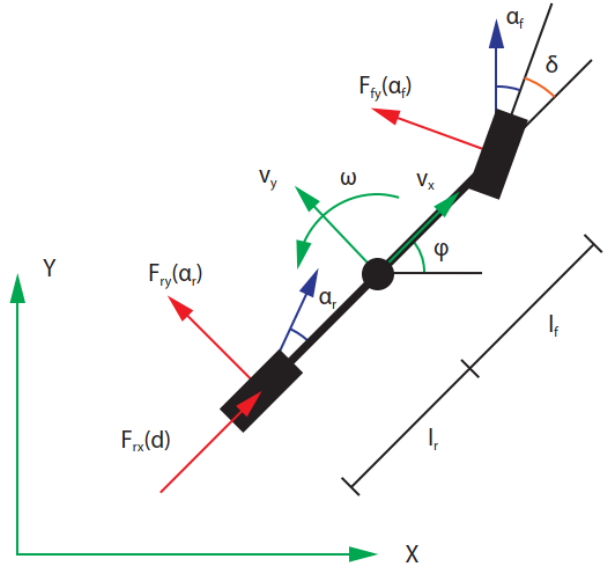


Figure 2. Single-track bicycle kinematics model [2]

The states of the vehicle are: x-coordinate on the map (x), y-coordinate on the map (y), velocity(v) and yaw angle

(φ). The continuous-time, kinematic model, nonlinear Ordinary Differential Equations (ODE) are as follows:

$$\begin{aligned}\dot{x} &= v \cos(\varphi) \\ \dot{y} &= v \sin(\varphi) \\ \dot{v} &= a \\ \dot{\varphi} &= \frac{v \tan(\delta)}{L}\end{aligned}$$

where δ is the steering angle and L is the wheelbase of the vehicle.

B. Discretization

The state equations need to be discretized in order to formulate the Model Predictive Control (MPC) problem and perform simulations. Forward Euler discretization is chosen because of its simplicity. The discretized nonlinear equations with sampling time dt are as follows:

$$\begin{aligned}x_{k+1} &= x_k + v_k \cos(\varphi_k) dt \\ y_{k+1} &= y_k + v_k \sin(\varphi_k) dt \\ v_{k+1} &= v_k + a_k dt \\ \varphi_{k+1} &= \varphi_k + \frac{v_k \tan(\delta_k)}{L} dt\end{aligned}$$

The nonlinear discrete model can also be written in matrix form, where $z = [x, y, v, \varphi]^T$ is the state vector and $u = [a, \delta]^T$ is the control input vector:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ v_{k+1} \\ \varphi_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cos(\varphi_k) dt & 0 \\ 0 & 1 & \sin(\varphi_k) dt & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{\tan(\delta_k)}{L} dt & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \\ v_k \\ \varphi_k \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ dt & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a_k \\ \delta_k \end{bmatrix}$$

C. Linearization

The MPC problem will be formulated as a Quadratic Programming optimization. Therefore, the model is linearized using first-order Taylor series expansion around the operating point z_0, u_0 .

The linearization of the state x is as follows:

$$\begin{aligned}x_{k+1} &= x_k + v \cos(\varphi) dt \\ \text{The nonlinear term is } f(v, \varphi) &= v \cos(\varphi) \\ x_{k+1} &= x_k + dt \left[f(v_0, \varphi_0) + \frac{\partial f(v, \varphi)}{\partial v} \Big|_{v_0, \varphi_0} (v - v_0) + \frac{\partial f(v, \varphi)}{\partial \varphi} \Big|_{v_0, \varphi_0} (\varphi - \varphi_0) \right] \\ x_{k+1} &= x_k + dt [v_0 \cos(\varphi_0) + v \cos(\varphi_0) - v_0 \sin(\varphi_0) - \varphi v_0 \sin(\varphi_0) + v_0 \varphi_0 \sin(\varphi_0)] \\ x_{k+1} &= x_k + [dt \cos(\varphi_0)]v + [-dt v_0 \sin(\varphi_0)]\varphi + dt v_0 \varphi_0 \sin(\varphi_0)\end{aligned}$$

Similarly, the linearization of the state y is as follows:

$$y_{k+1} = y_k + v \sin(\varphi) dt$$

The nonlinear term is $f(v, \varphi) = v \sin(\varphi)$

$$\begin{aligned}y_{k+1} &= y_k + dt \left[f(v_0, \varphi_0) + \frac{\partial f(v, \varphi)}{\partial v} \Big|_{v_0, \varphi_0} (v - v_0) + \frac{\partial f(v, \varphi)}{\partial \varphi} \Big|_{v_0, \varphi_0} (\varphi - \varphi_0) \right] \\ y_{k+1} &= y_k + dt [v_0 \sin(\varphi_0) + v \sin(\varphi_0) - v_0 \cos(\varphi_0) + \varphi v_0 \cos(\varphi_0) - v_0 \varphi_0 \cos(\varphi_0)] \\ y_{k+1} &= y_k + [dt \sin(\varphi_0)]v + [dt v_0 \cos(\varphi_0)]\varphi - dt v_0 \varphi_0 \cos(\varphi_0)\end{aligned}$$

The state v has a linear state equation; therefore, it is used as is in the linear model. Finally, the linearization of the state φ is as follows:

$$\varphi_{k+1} = \varphi_k + \frac{v \tan(\delta)}{L} dt$$

The nonlinear term is $f(v, \delta) = v \tan(\delta)$

$$\begin{aligned}\varphi_{k+1} &= \varphi_k + \frac{dt}{L} \left[f(v_0, \delta_0) + \frac{\partial f(v, \delta)}{\partial v} \Big|_{v_0, \delta_0} (v - v_0) + \frac{\partial f(v, \delta)}{\partial \delta} \Big|_{v_0, \delta_0} (\delta - \delta_0) \right] \\ \varphi_{k+1} &= \varphi_k + \frac{dt}{L} \left[v_0 \tan(\delta_0) + v \tan(\delta_0) - v_0 \frac{v_0}{\cos^2(\delta_0)} \delta + \frac{v_0 \delta_0}{\cos^2(\delta_0)} \right] \\ \varphi_{k+1} &= \varphi_k + \left[dt \frac{\tan(\delta_0)}{L} \right] v + \left[dt \frac{v_0}{L \cos^2(\delta_0)} \right] \delta - dt \frac{v_0 \delta_0}{\cos^2(\delta_0)}\end{aligned}$$

The linearized model around the operating point z_0, u_0 is then written in matrix form:

$$z_{k+1} = A z_k + B u_k + C$$

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ v_{k+1} \\ \varphi_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & dt \cos(\varphi_0) & -dt v_0 \sin(\varphi_0) \\ 0 & 1 & dt \sin(\varphi_0) & dt v_0 \cos(\varphi_0) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & dt \frac{\tan(\delta_0)}{L} & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \\ v_k \\ \varphi_k \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ dt & 0 \\ 0 & dt \frac{v_0}{L \cos^2(\delta_0)} \end{bmatrix} \begin{bmatrix} a_k \\ \delta_k \end{bmatrix} + \begin{bmatrix} dt v_0 \varphi_0 \sin(\varphi_0) \\ -dt v_0 \varphi_0 \cos(\varphi_0) \\ 0 \\ -dt \frac{v_0 \delta_0}{L \cos^2(\delta_0)} \end{bmatrix}$$

III. MODEL PREDICTIVE CONTROL FORMULATION

A list of reference points is generated by driving the car manually around the track and recording the state of the vehicle throughout the lap. The goal of the MPC controller is to generate the necessary control inputs in order to track the reference points over a finite horizon of length N while satisfying a set of constraints. The MPC algorithm is detailed in the subsequent sections.

A. Objective Function

The objective function of the MPC algorithm consists of minimizing the sum of four terms [2].

$$\begin{aligned}\min: & (z_{N,ref} - z_N)^T Q_F (z_{N,ref} - z_N) + \sum_{k=0}^{N-1} (z_{k,ref} - z_k)^T Q (z_{k,ref} - z_k) \\ & + \sum_{k=0}^{N-1} u_k^T R u_k + \sum_{k=0}^{N-2} (u_{k+1} - u_k)^T R_d (u_{k+1} - u_k)\end{aligned}$$

The first term of the objective function is the terminal deviation from the reference point, weighted by Q_F . The

second term is the sum of the stage deviations from the reference points, weighted by Q . The third term is a penalty on control inputs, weighted by R . And the fourth term is a penalty on the change in control input, weighted by R_d . The last term is used to limit aggressive changes in acceleration and steering input.

B. Constraints

The set of constraints for the MPC problem are stated below:

- $z_0 = z_{curr}$: the initial state in the MPC planned trajectory must match the current state of the vehicle
- $z_{k+1} = A z_k + B u_k + C$: vehicle states must follow the linearized model
- $a_{min} \leq a_k \leq a_{max}$: acceleration limits
- $\delta_{min} \leq \delta_k \leq \delta_{max}$: steering angle limits

C. Assumptions

Two assumptions are needed in order to implement the controller in simulation:

- A map of the racetrack with exact dimensions is known
- The state of the vehicle, including position on the map, velocity and orientation, can be fetched from the simulation environment at any time

If the controller were to be implemented in hardware, additional algorithms are needed to generate a map (SLAM), localize the vehicle on the map, and estimate its states.

IV. IMPLEMENTATION

A. Simulation Environment

The simulator used in this project is the open-source F1Tenth Gym. It is a python-based simulator, which also includes a Robot Operating System (ROS) bridge to enable the interaction of ROS2 control code with the simulator. Because ROS2 runs natively on Linux, some form of virtual Linux environment is needed to run on Windows, and two popular options are available. The first option is to set up a Linux Virtual Machine (or Windows Subsystem for Linux) and install ROS2 and Python inside it. The downside of this option is that it involves a tedious installation process, and the simulation environment is not shareable; it needs to be set up from scratch on any other machine. The second option is to use Docker by defining the whole simulation environment as a **docker-compose** file with **Docker images**. This way, the simulation environment can be replicated on any machine that has Docker, without having to install ROS and Python, or worry about dependencies and software versions.

The full simulation and development environment consists of two Docker services (containers) running as part of one Docker application. The first service, labelled **sim-1**, is where the simulations take place. It includes the F1Tenth Gym, as well as ROS, python and RVIZ. This container

includes a Docker Volume, which is a memory location on the host system that provides a way to store and share data between the container and the host system. In other words, it provides a “mapping” between a memory location on the host system and the memory of the Docker container. This is the folder where the control code is developed and saved on the host system, and can then be run from inside the container. Also, port 8765 of this container is mapped to port 8765 of the host system in order to be able to connect Foxglove to the container for live telemetry. The second service, labelled **novnc-1** provides a remote connection at port 8080 using noVNC. Through this remote connection, the host system is able to view RVIZ, which is running in **sim-1**. To make that possible, port 8080 of **novnc-1** is mapped to port 8080 of the host system. The RVIZ visualization can therefore be accessed on <http://localhost:8080/vnc.html> on the host system. Figure 3 shows a diagram of the simulation setup and the relationship between its different components.

Step-by-step instructions on how to set up the simulation environment and run it are included in the *README.md* file in the GitHub repository (VI).

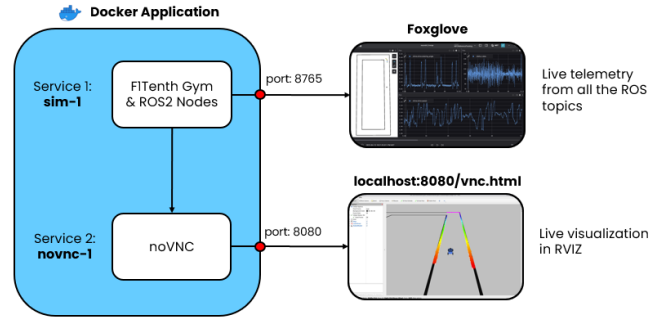


Figure 3. Simulation environment setup: 2 Docker services running as part of 1 Docker application

A simple map is used in this project: it represents the corridor of a building. Other maps of scaled down Formula 1 race tracks are available with the F1Tenth Gym; however, these tracks are significantly longer than the corridor and require a few minutes to complete a single lap, which increases the load on telemetry and recording file size, and slows down the MPC tuning cycle. Figure 4 shows a top view of the corridor map.

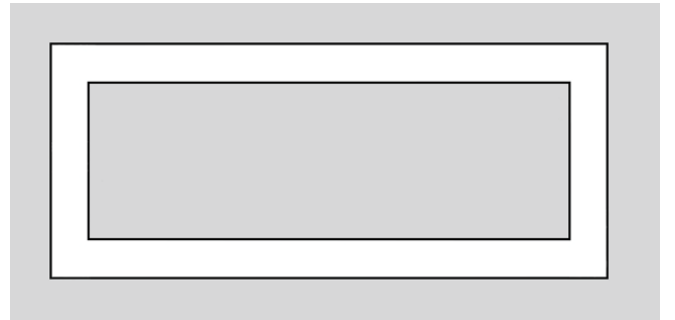


Figure 4. Map of the corridor (racetrack): shorter track for faster development and smaller telemetry files

B. Reference Points

Reference points that include the state of the vehicle around the track are needed for the MPC algorithm. For that, a ROS2 node, *waypoints_logger.py* is implemented. The node subscribes to the odometry messages of the vehicle and stores the state information (x, y, v, φ) in a csv file. The following steps are taken to generate the reference points:

1. The *waypoints_logger* node is launched
2. The vehicle is driven manually around the track using keyboard commands, while trying to remain around the centerline of the track. The *waypoints_logger* node saves the states in a csv file. Figure 5 shows the first 15 reference points in the csv file.
3. The csv file is manually processed to reduce sampling and remove overlapping points at the start/end of the lap. Figure 6 shows the resulting x and y coordinates of the reference points after processing.
4. Save the processed reference points in a new csv file and store it in the simulation directory. Then the reference points can be loaded with any ROS2 node.

	A	B	C	D	E
1	x	y	yaw	v	
2	0.13027	-0.1	0	2	
3	0.32497	-0.1	0	2	
4	0.58348	-0.1	0	2	
5	0.80336	-0.1	0	2	
6	1.01335	-0.1	0	2	
7	1.22835	-0.1	0	2	
8	1.42335	-0.1	0	2	
9	1.62335	-0.1	0	2	
10	1.81835	-0.1	0	2	
11	2.02835	-0.1	0	2	
12	2.22835	-0.1	0	2	
13	2.42335	-0.1	0	2	
14	2.64835	-0.1	0	2	
15	2.87835	-0.1	0	2	

Figure 5. Logged reference points in the csv file

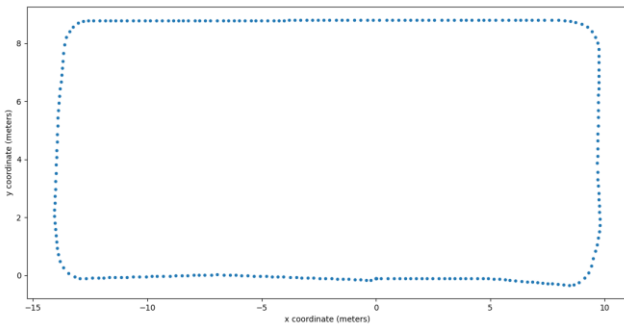


Figure 6. x and y coordinates of the processed reference points: equally distant points, no overlapping, and continuity on the start/finish line

A ROS2 node called *waypoints_marker* is implemented in order to visualize the resulting racing line in RVIZ. Figure 7 shows the racing line formed by the reference points after running the *waypoints_marker* node.

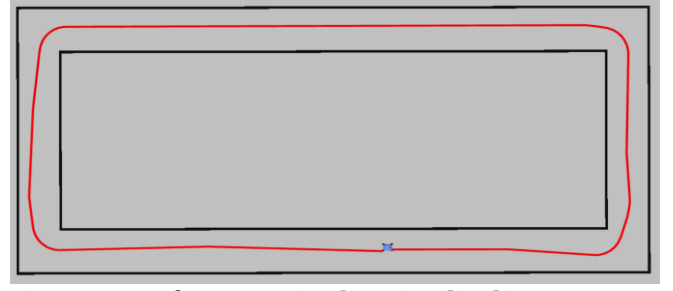


Figure 7. Reference racing line visualized in RVIZ

C. MPC Implementation

The code for the Model Predictive Controller is implemented in a ROS node called *mpc_node*. The node includes a subscriber to the Odometry topic (*/odom*) which updates a current state vector every time a new odometry message is received. Also, the node includes a timer with a period of *100ms* which calls the MPC algorithm periodically. At every callback, four steps are taken. First, a set of 6 reference points is extracted from the reference trajectory based on the current state. Second, the linearization matrices A , B and C are evaluated at the current state in order to provide an updated linear model to the MPC. Then, the MPC optimization problem is solved. Finally, the first point of acceleration and steering in the solution are published to the */drive* topic to actuate the car.

The MPC optimization problem is implemented using the *cvxpy* Python library and the *Operator Splitting Quadratic Program (OSQP)* solver is chosen.

D. Tuning of MPC

Several variables can be tuned in order to modify and optimize the behavior of the Model Predictive Controller.

First, the discrete time step, which is also the period of the timer in the ROS node, is set to *100ms*. Initially, the target was to run the loop at a period of *50ms*. However, because the simulation is running in a Docker container, it has limited resources (4 cores) from the host system. So, at *50ms*, the simulation environment could not keep up with the timing constraint, which affected the performance of the MPC since it was running slower than it is supposed to. With the period of the timer set to *100ms*, the loop is consistently running around *100ms*, along with the ability to open RVIZ and record ROS bag sessions. Figure 8 shows the recorded time difference between consecutive calls to the MPC function when the timer is set to *100ms*.

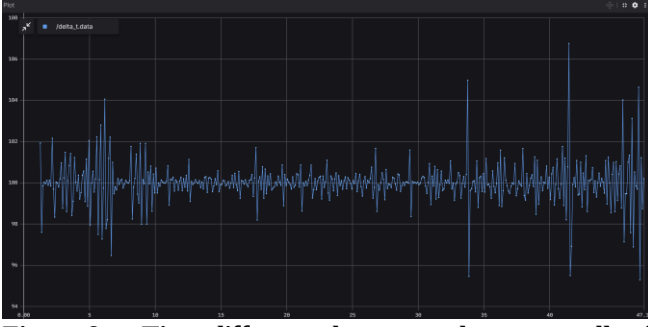


Figure 8. Time difference between subsequent calls of the MPC function: MPC runs consistently at a period of approximately 100ms

Second, the Q , Q_F , R and R_d matrices that appear in the objective function (Section III.A), were manually tuned following an iterative process, which resulted in the below values:

$$Q = \begin{bmatrix} 20 & 0 & 0 & 0 \\ 0 & 20 & 0 & 0 \\ 0 & 0 & 5.5 & 0 \\ 0 & 0 & 0 & 20 \end{bmatrix}$$

$$Q_F = \begin{bmatrix} 20 & 0 & 0 & 0 \\ 0 & 20 & 0 & 0 \\ 0 & 0 & 5.5 & 0 \\ 0 & 0 & 0 & 20 \end{bmatrix}$$

$$R = \begin{bmatrix} 5 & 0 \\ 0 & 85 \end{bmatrix}$$

$$R_d = \begin{bmatrix} 5 & 0 \\ 0 & 85 \end{bmatrix}$$

The states x , y and ϕ are given a more significant weight compared to v because it is more important to follow the reference and stay on track than to match the velocity exactly. This stems from the fact that, in the Time Trials, the lap time gets deleted if the car crashes; so, it is crucial to finish the lap without crashing. Regarding the control inputs, the steering angle required a higher penalty value to reduce its oscillations compared to acceleration.

Third, the finite horizon was tuned by trial and error and was set to $N=6$, which resulted in the best compromise between control performance and computation time. This finite horizon corresponds to a distance of 1.2 meters given that the car is driven at 2m/s with a period of 100ms.

V. RESULTS

The results are obtained by running the simulation with the tuned parameters detailed in IV.D and the reference racing line in Figure 7 with a reference velocity of 2m/s. The MPC has a finite horizon of $N=6$ and runs at a rate of 100ms (Figure 8). The simulation is performed for a total duration of 47s, which is slightly less than one and a half laps of the track.

Figure 9 shows the steering control input generated by the MPC (in rad) as a function of time. Every peak in the

steering command corresponds to a corner; so, in total, the car completes 6 corners. All the peaks have a positive magnitude because all the corners on the track are counter-clockwise and the car only steers to the left. The plot shows a very smooth turning profile where, at every turn, the steering angle increases almost linearly before reaching the peak, then decreases in an approximately linear trend back to 0. This indicates that the controller is not performing steering corrections in the middle of the corner, nor is it overshooting the required steering then compensating for it. This is a desired behavior in racing and matches the telemetry of professional racing drivers, when their car does not oversteer or understeer. Moreover, the plot shows minimal steering oscillations while driving in a straight line. The small oscillations that appear around 0 in the plot are in the order of 0.005 rad.

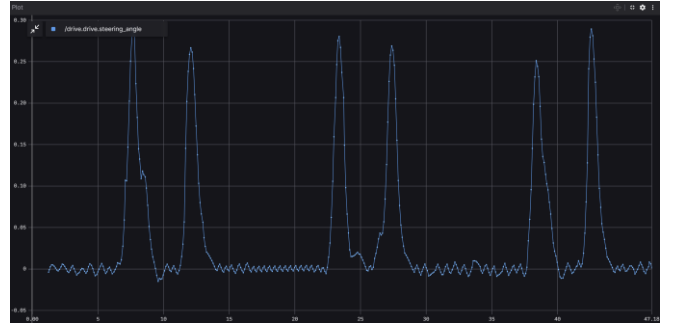


Figure 9. Steering control input from the MPC. The steering profile shows minimal oscillations and smooth turns

Figure 10 shows the acceleration control input generated by the MPC (in m/s^2) as a function of time, while Figure 11 shows the velocity profile of the car that results from applying the acceleration commands (in m/s). The acceleration starts at a high value when the car has low velocity, then decreases as the car gains velocity and reaches the reference of 2m/s. The rise time of the velocity is a bit long in the context of racing; so, the acceleration is desired to be even higher at low velocities. However, while tuning the controller, a more aggressive acceleration reduced the velocity rise time, but resulted in oscillations in the velocity curve around the reference. Therefore, the results of Figure 10, Figure 11 and Section IV.D were the best compromise between velocity rise time and oscillations around the reference.



Figure 10. Acceleration control input from the MPC

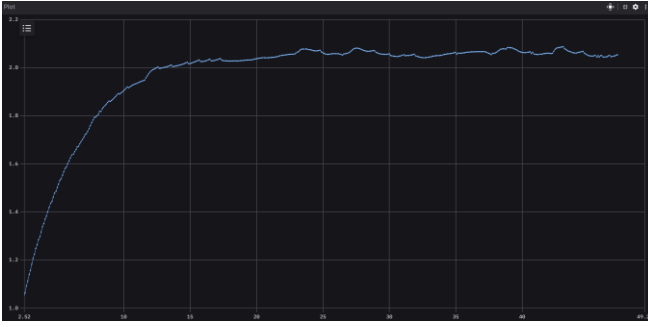


Figure 11. Velocity profile of the car. Velocity is smooth and tracks the 2m/s reference

Figure 12 compares the reference racing line in red to the racing line of the car using MPC in blue. It is a visual metric that shows the tracking accuracy of the states x and y . The MPC matches the reference racing line in almost all the parts of the lap, except in small sectors during tight turns.

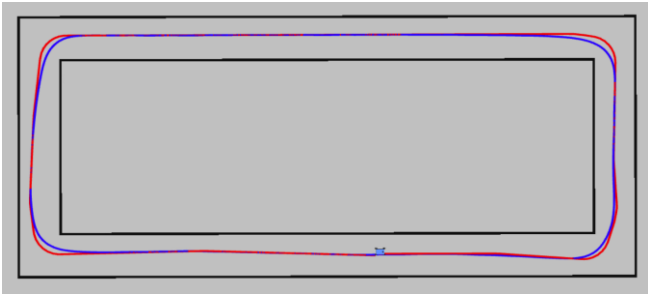


Figure 12. Comparison between the reference trajectory (red) and car trajectory with MPC (blue)

Longer simulations were performed, where the car was left running for durations between 5 and 10 minutes; but the data is not included in the report because the plots would be significantly compressed in the time axis and would not show the details of the driving trend. However, these simulation runs prove that the car does not crash even after running for prolonged periods.

A video of the full simulation that corresponds to the results presented in this section played in Foxglove is uploaded to

https://arizonastateu-my.sharepoint.com/:v/g/personal/esaade_sundevils_asu_edu/EfP1E42p0rZAqsvmCTSwOEQBGsuQnTXjOg2hvONY_a06AA.

On the map in this video, the red points correspond to the reference trajectory, while the green points correspond to the solution of the MPC in the finite horizon.

VI. CODE

The full code for the project is available in my GitHub repository at:

<https://github.com/ElioSaade4/RoboCar-Racing>

The GitHub repository is organized as follows:

- **README Images:** folder that contains all the images for the *README.md* file

- **recording:** folder that contains a ROS bag recording of the simulation run presented in Section V, which can be opened and replayed in Foxglove
- **sim_ws:** folder that contains all the code of the project. It is also the folder that is accessible by the **sim-1** container. It includes the following 3 folders:
 - **f1tenth_gym_ros:** ROS2 python package which contains the code for the ROS bridge to the F1Tenth Gym, as well as the keyboard teleoperations node. **This package is available open-source along with the F1Tenth Gym and was not developed or modified by me.**
 - **mpc:** ROS2 python package that includes the code for *mpc_node*, which implements the MPC.
 - **waypoints:** ROS2 python package that includes the code for the nodes *waypoints_logger* and *waypoints_marker*.
- **Dockerfile:** defines the image of the **sim-1** Docker container
- **README.md:** includes a detailed step-by-step documentation on how to set up the simulation environment and perform simulations
- **docker-compose.yml:** defines the structure of the multi-container Docker application, along with the containers, images, memory volume and port mappings.

VII. IDEAS FOR FUTURE WORK

This project is the steppingstone for a full software stack for autonomous racing and potentially participating in the F1Tenth competition or conducting more research on the topic in the future. In keeping with that theme, and to conclude this work, below is a list of ideas that can be implemented or researched in the future:

- add a velocity profile to the reference waypoints to improve lap time: faster velocities in the straights, slower velocities in the turns
- add more telemetry channels to track more variables
- test the algorithm on other racetracks
- SLAM to generate a map of the environment instead of loading the picture
- localization on the map instead of using the Odometry message from the simulation
- race line optimization to generate better racing line instead of driving the car manually on the track
- add obstacle avoidance constraints to the MPC problem after detecting obstacles using LiDAR
- compare other algorithms like Reinforcement Learning, or MPC that generates the reference line instead of following it

REFERENCES

- [1] B. D. Evans, R. Trumpp, M. Caccamo, F. Jahncke, J. Betz, H. W. Jordaan and H. A. Engelbrecht, "Unifying F1TENTH Autonomous Racing: Survey, Methods and Benchmarks," 2024
- [2] A. Liniger, A. Domahidi and M. Morari, "Optimization-Based Autonomous Racing of 1:43 Scale RC Cars," 2017