Control for Autonomous Vehicles

Creating a Control module for an autonomous robot and adapt it for the EPFL Racing Team

IC Bachelor V
Bachelor Semester Project
Eloi Garandel
05/06/2020

Professor:
Alexandre Alahi

Contents

1 Introduction

For several years now, a various change has started in the vehicle engineering field. Autonomous/self-driving vehicles have become part of the very large market of motorized vehicles. They promise significantly improved safety, convenience and efficiency and they allow more automated circuits, working without human supervision. Because of the multiple possibilities they offer, autonomous vehicles have become a real subject of interest for multinational corporations such as Google, Uber or Tesla.

To be more precise, we called autonomous vehicle a vehicle that satisfies at least SAE's level 3 of automation[1]: *«These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met»* with *«these features»* referring to the Automated Driving System, ADS).

To do so, a self-driving vehicle uses a particular architecture which can be simplified in several modules:

- The *Perception module* recognizes the road and the different elements on it (other vehicles, pedestrians, road signs, traffic lights, etc). It achieves this function using different kinds of sensors, like cameras or radar sensors as well as LiDAR.
- The *Motion Estimation module* estimates the position of the car, its velocity as well as its heading. Like Perception, it uses various sensors to do this task, such as GPS, speed sensors or an Inertial Measurement Unit (IMU). At the end, it generates what we will call here the vehicle's state.
- Given the data generated by the previous modules, the *Mapping module* will create a map of the vehicle's environment. To do so, it uses algorithms such as SLAM: Simultaneous Localization And Mapping. This module is often associated with the previous one.
- Once we know the vehicle's state and its environment, we can finally generate a path to follow: this is the purpose of *Path Planning module*. Given those information, it generates the best possible path for the vehicle giving several environment constraints (not going off the road, not turning on a no-entry street, etc).
- The last module is the *Control module*. Given the path to follow and the vehicle's state, it computes the best vehicle controls to follow this path with the least expensive cost. It is this module that was created for this project. Further explanation will be given in a later part.

Figure 1.1 shows a graph representing the previously described simplified architecture.

[1] *Society of Automotive Engineers Standard J3016*

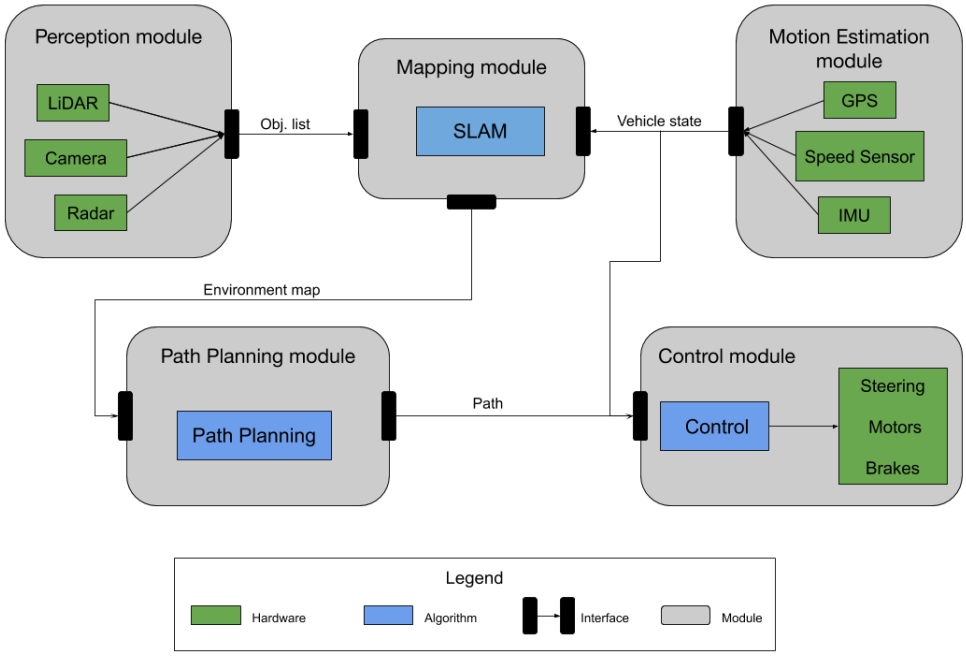

Figure 1.1: Self-driving vehicle architecture

2 Control module for an autonomous robot

The main goal of this project is to create a control module that performs well on an autonomous robot evolving in a 2D-environment. Here, the given robot is a Loomo robot[1]: it is an auto-balanced 2-wheel robot that has built-in ultrasonic sensors, infrared distance sensors, IMUs, etc. In order to create and properly integrate the Control module, I used the framework of another student of the lab working on the same robot. In the end, I just had to add the module.

2.1 Algorithm for the module

As said before, the Control module is responsible for computing the controls of the vehicle (here, the robot) so that it follows the path the most accurately. To do so, I defined the Control input as the solution of a MPC problem. A MPC is Model Predictive Control: it's an *advanced method of process control that is used to control a process while satisfying a set of constraint*[2]. More precisely, in this case, we will extend the formulation from *Optimal Control Applications and Methods*[3].

This formulation optimizes the progress made along the path with respect of the vehicle model and track constraint. It also presents a real advantage: to optimize the solution, it only needs the path and the current vehicle state and it directly takes into consideration the vehicle/track limits when optimizing the solution.

[1] Robot from *Segway*
[2] Wikipedia definition
[3] *Optimization-Based Autonomous Racing of 1:43 Scale RC Cars*, A. Liniger, A. Domahidi, M. Morari

2.2 Robot model

In the used framework, the Control module should output a longitudinal velocity and an angular velocity so that the robot can make a move. Giving the two values, the robot will automatically update it right and left wheel velocity to perform the given move. Hence, we modeled it using the following motion equations:

$$F_{robot}(\mathbf{x},\ \mathbf{u}) = \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \upsilon \\ \omega \end{bmatrix} \tag{2.1}$$

with X and Y the global coordinates of the robot, θ its orientation, υ the translational velocity of the center of the robot, ω its angular velocity and the control inputs $u = [\upsilon, \omega]$.
Figure 2.1 illustrates those equations:

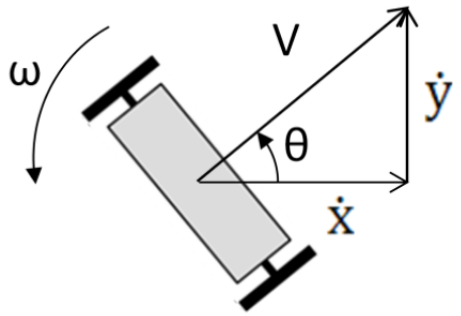

Figure 2.1: Illustration of the motion equations of a 2-wheel robot

Thus, we get to following motion function for the robot:

$$x_{k+1} = f_{robot}(x_k, u_k) = x_k + \Delta t * F_{robot}(x_k, u_k) \tag{2.2}$$

with x_k the k^{th} position of the robot, u_k the k^{th} control tuple of the robot and Δt the sampling time.

Here, the MPC algorithm must optimize the angular velocity ω and the translational velocity υ of the center of the robot so that the robot follows the path accurately.

2.3 Error/Cost function

The goal of the MPC is to minimize the cost of control inputs in order to follow the most optimally the given path. Hence, for each tested solution, we have to associate it to a cost/error value in order to get the optimal tuple $\boldsymbol{u} = (\upsilon, \omega)$.

2.3.1 Coordinates Error

When a tuple $\boldsymbol{u}$ is tested, we compute the next position of the robot given those parameters and compare it to the goal position (waypoint reference) to estimate the correctness of the tested tuple. We compare them using MSE as follows:

$$E(x_k) = \frac{(X_k - X_{ref})^2 + (Y_k - Y_{ref})^2}{2} \tag{2.3}$$

with X_k, Y_k the predicted position of the position and X_{ref}, Y_{ref} the points of the path.

2.3.2 Velocity Cost

When the robot is following the path, another important property of its driving is the smoothness: we do not want the vehicle to make abrupt movements, which could lead to hardware malfunction (hard break, too much acceleration, etc). Hence, the derivative of v and ω should be minimal so that the trajectory is as smooth as possible.

We calculate the derivative $\boldsymbol{\Delta u} = (\Delta v, \Delta\omega)$ and associate it to a cost as follow:

$$C(\Delta\mathbf{u}) = \Delta\mathbf{u}\mathbf{R}\Delta\mathbf{u}^T \tag{2.4}$$

with $\mathbf{R}$ a 2x2 matrix containing the weight of cost of each variable.

2.3.3 Overall Error

If we only sum the errors (equations 2.3 and 2.4), this would lead to a non-homogeneous expression and one error could minimize the other. Consequently, each error should be scaled with a reference error value as follows:

$$Error(x_k, \mathbf{u_k}) = q_c * E(x_k) + q_v * C(\boldsymbol{\Delta}\mathbf{u_k}) \tag{2.5}$$

$$= \mathrm{q_c} * \frac{\frac{(X - X_{ref})^2}{maxErrorX^2} + \frac{(Y - Y_{ref})^2}{maxErrorY^2}}{2} + q_v * \begin{bmatrix}\Delta v & \Delta\omega\end{bmatrix} \begin{bmatrix}0.5/(v_{max})^2 & 0.0 \\ 0.0 & 0.5/(\omega_{max})^2\end{bmatrix} \begin{bmatrix}\Delta v & \Delta\omega\end{bmatrix}^T$$

with q_c and q_v the weight of each error, $maxErrorX$ and $maxErrorY$ the references for the coordinates error and v_{max} and ω_{max} the references for the velocity cost.

2.4 Final Formulation

In the end, the final MPC problem consists of predicting N tuple u, compute the sum of corresponding errors (using N points from the path) and minimize it as much as possible. N is called the prediction horizon. The final formulation of the MPC problem is the following:

$$\begin{aligned} \min_{\mathbf{u}} \quad & \sum_{k=0}^{N} q_c * E(x_k) + q_v * C(\mathbf{u_k}) \\ s.t.\ & x(0) = x_0 \\ & x_{k+1} = f_{robot}(x_k, u_k) \\ & u_{min} \leq \mathbf{u} \leq u_{max} \\ & \Delta u_{min} \leq \boldsymbol{\Delta}\mathbf{u} \leq \Delta u_{max} \end{aligned} \tag{2.6}$$

To solve this problem, I used the SciPy library from Python. It provides a package, "*optimize*", that is able to solve such a MPC problem as well as other optimization problems.

2.5 Simulation

To test the module, I create several test paths:

- A straight line
- A line with sharp line using the sinus function
- A long-turn line using the exponential function
- A example of racing track manually defined

Figure 2.2, 2.3, 2.4 and 2.5 shows the results of simulations on those track.

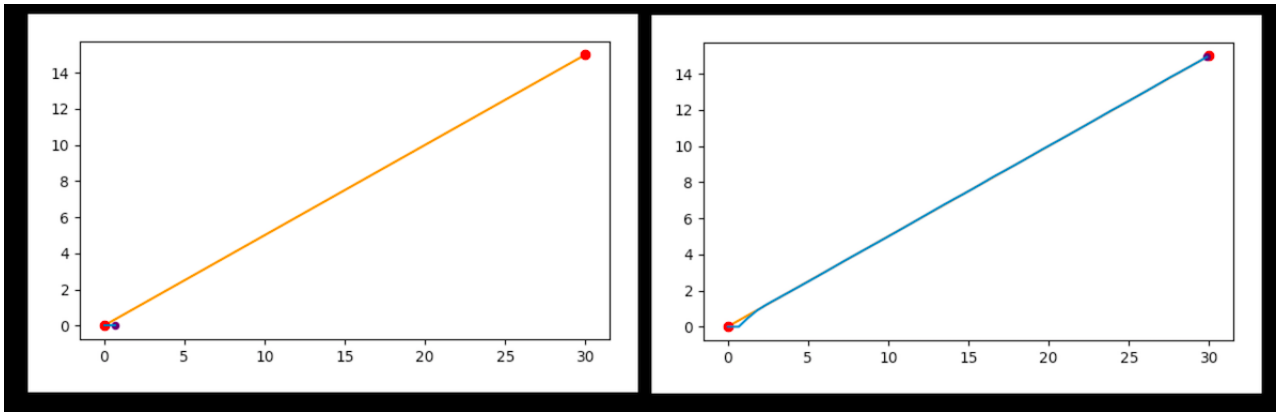

Figure 2.2: Simulation on a straight line

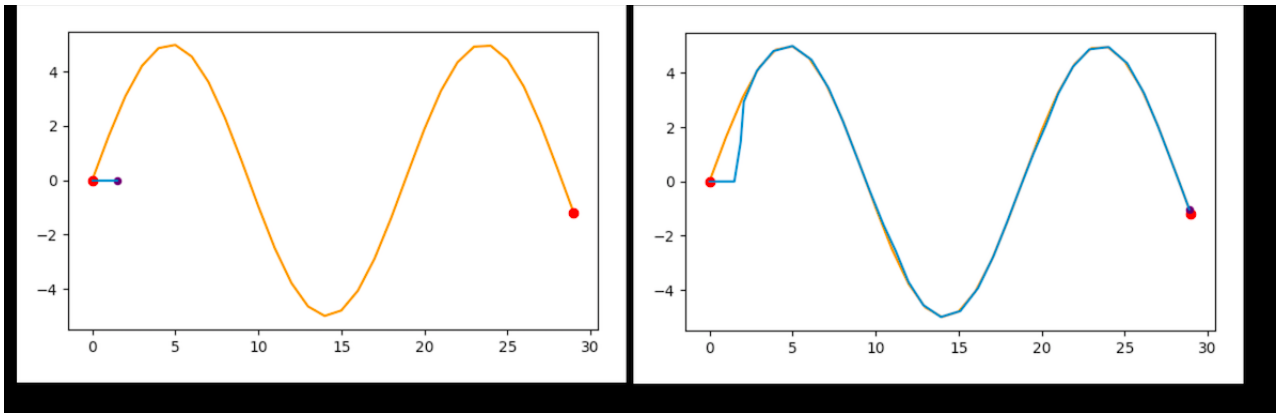

Figure 2.3: Simulation on a sharp-turn line

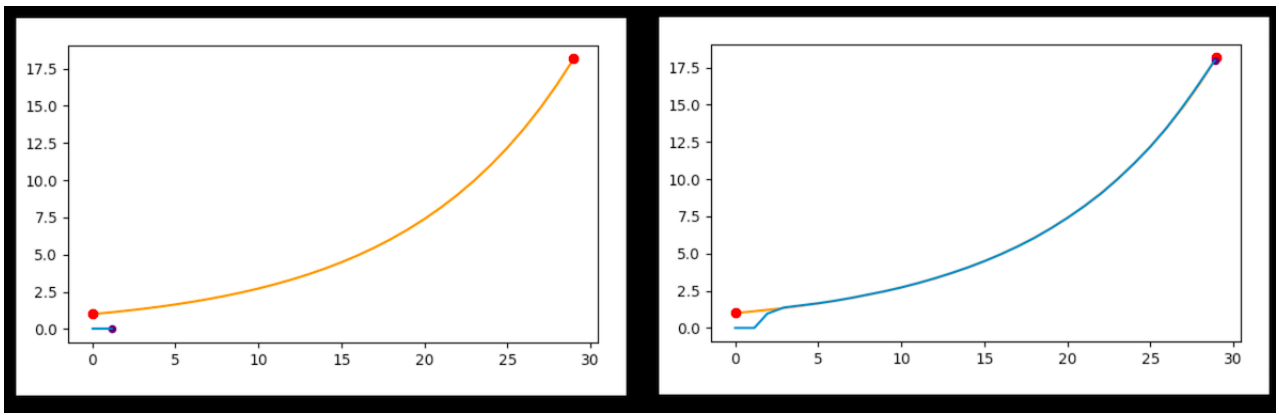

Figure 2.4: Simulation on a long-turn line

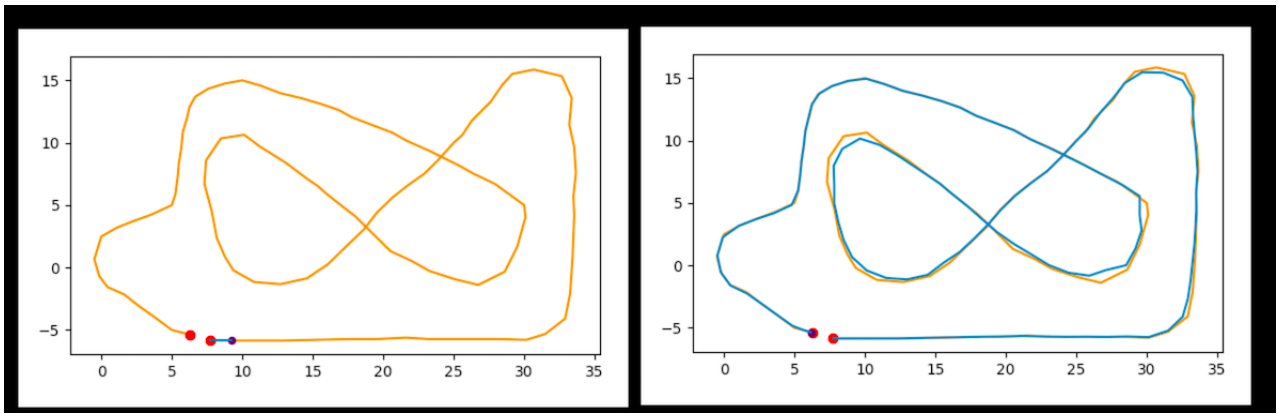

Figure 2.5: Simulation on a track

With the 3 first figures, we can see that if we ask the robot to follow a simple line with no complicated patterns, it will follow it very accurately. On the other side, on more complicated tracks like Figure 6, we can see that the MPC will compute optimal controls that take in account the rest of the track resulting in the robot taking sharper turns.

Also, the gap happening at the beginning of each track can be explained by the initial state's values: the robot is always set with $X_0 = 0$, $Y_0 = 0$ and $\theta_0 = 0$. Accordingly, it first has to turn to get the correct trajectory, which result in a gap.

Since this is a simulation, we have *fake ideal* conditions. Real-life conditions add more complexity to the system (real-test footage is available on the git repository).

2.6 Parameters' influence

The previous simulations have all been made using the same set of parameters. Changing part of it directly impacts the accuracy of the algorithm or its computation time.

- Computation time increases significantly with the prediction horizon N: the further we want to predict, the longer it takes to optimize. Figure 2.6 summarize this idea: I run the MPC algorithm on the circuit of Figure 2.5 using different values for the prediction horizon.

Prediction Horizon	**Mean computation time per iteration (in seconds)**
5	0.006
7	0.022
10	0.037
20	0.109

Figure 2.6: Relation between computation time and prediction horizon

- As said in 2.5, the prediction horizon also impacts the resulting control that will be output by the algorithm. From the test, the main difference was that it takes sharper turn because the algorithm "understands" that the robot will lately have to change its direction by a lot and taking a sharper turn will result in a slightly bigger coordinates errors (robot won't follow the exact path) but it will take less time to turn. Figure 2.7 shows the described effect.

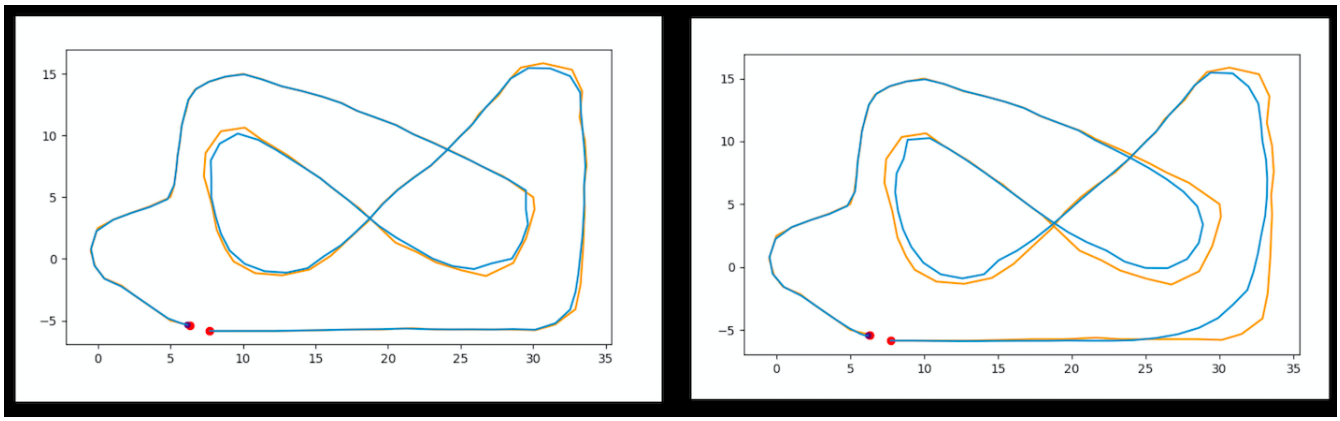

Figure 2.7: Track simulation with an prediction horizon of 5 (left) and 10 (right)

- The accuracy also depends on error weights and on the scale values of those errors.

2.7 Robot Model summary

Using the previously described model and the stated error formula, I've implemented a Control module and integrated it in an overall framework in order to have a working solution that performs well on the Loomo robot.

3 Adaptation for the EPFL Racing Team

Once the first structure of the module was done, the second goal of the project was to adapt for a Driverless car in the EPFL Racing Team.

3.1 EPFL Racing Team and Driverless section

EPFL Racing Team is an EPFL student association competing in the Formula Student. It is an international competition where students from universities of all over the world build racing cars and compete between each other in different types of events. The EPFL Racing Team has now been competing for several years and this year, they created an driverless section in order to compete in the driverless competition of Formula Student in 2022. Hence, the hardware/software development is spread over at least one year. During this first semester, the main objective of the driverless team (which I'm part of) was to get a first working structure to be able to start testing during the second semester.

3.2 Car model

The main change to be made to the previous model is the motion equation: a racing car is more complex than a 2-wheel robot and it should be well modeled to get accurate results using the MPC algorithm. Since this is the first year of EPFL RT's driverless section, we didn't have previous EPFL work to improve and had to start from scratch. Hence, my main source to build the car model was a report published by Academic Motorsports Club Zurich, AMZ[1].

[1]Report available on the git repository: see appendix A

To start, I chose to only implement a kinematic model. This model is mainly used for low velocities ($v \simeq 4$ m/s and below) and will be a good start point to see how the module performs and to make benchmarks.

This model is described by the following equation:

$$F_{car}(\mathbf{x},\ \mathbf{u}) = \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\varphi} \\ \dot{v_x} \\ \dot{v_y} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} v_x cos\varphi - v_y sin\varphi \\ v_x sin\varphi - v_y cos\varphi \\ r \\ \frac{F_x}{m} \\ (\dot{\delta} v_x + \delta \dot{v_x}) \frac{l_R}{l_R + l_F} \\ (\dot{\delta} v_x + \delta \dot{v_x}) \frac{1}{l_R + l_F} \end{bmatrix} \tag{3.1}$$

where the state of the model $x = [X, Y, \varphi, v_x, v_y, r]^T$ consists of X,Y the position of the car, φ its heading in global coordinate system, v_x, v_y the longitudinal and lateral velocities, and r the yaw rate. The control inputs u $= [\delta, D]^T$ represents the steering angle φ and driving command $D \in [-1, 1]$. This vector is what will be optimize by the MPC algorithm.

Also, as said in the AMZ report, F_x describes the longitudinal force acting on the car depending on the applied driver command:

$$F_x = C_m D - C_{r0} - C_{r2} v_x^2 \tag{3.2}$$

which consists of a motor model, $C_m D$, rolling resistance, C_{r0}, and drag, $C_{r2} v_x^2$ with C_m, C_{r0}, C_{r2} identified from experiments.

3.3 Formulation for the car model

Because of the low velocities, I decided to keep the same error function as the robot model. Hence, the formulation of the problem is the same as for the robot model with several differences: as said before, we use different motion equations and the bounds of the control input are changed to fit the constraints of the model and the car.

The final problem is as follows:

$$\begin{aligned} \min_{\mathbf{u}} \quad & \sum_{k=0}^{N} q_c * E(x_k) + q_v * C(\mathbf{u_k}) \\ s.t. \ & x(0) = x_0 \\ & x_{k+1} = f_{car}(x_k, u_k) \\ & u_{min} \leq \mathbf{u} \leq u_{max} \\ & \Delta u_{min} \leq \mathbf{\Delta u} \leq \Delta u_{max} \end{aligned} \tag{3.3}$$

The same tools as for the robot model are used to solve this problem.

3.4 Simulation of car model

I simulated the car model on the same tracks as the robot. The following figures illustrate the results.

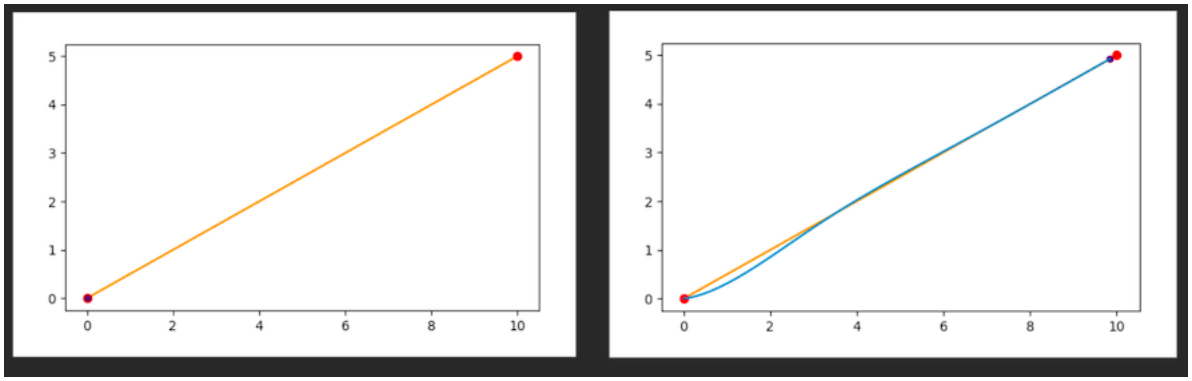

Figure 3.1: Simulation on a straight line

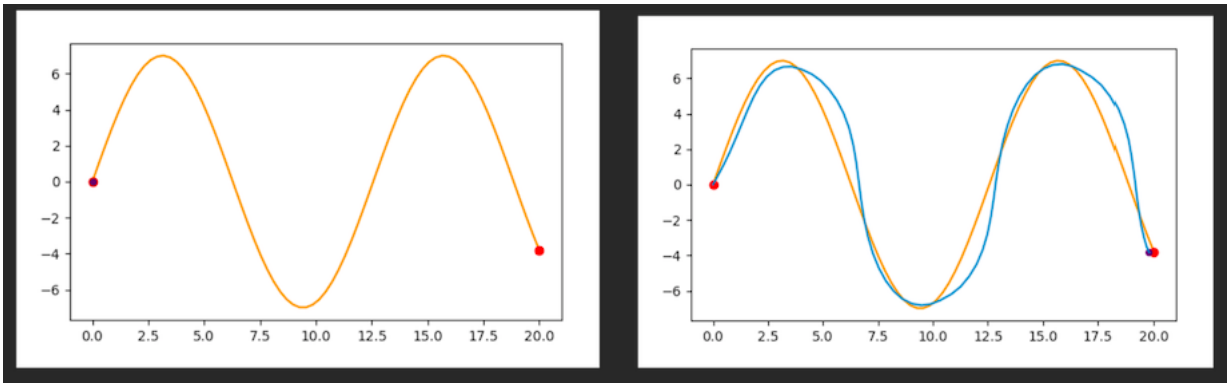

Figure 3.2: Simulation on a sharp-turn line

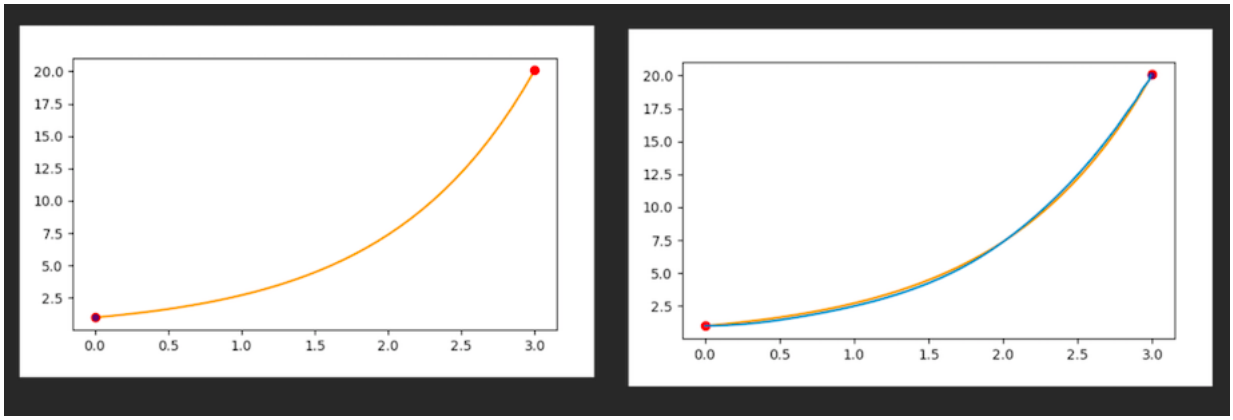

Figure 3.3: Simulation on a long-turn line

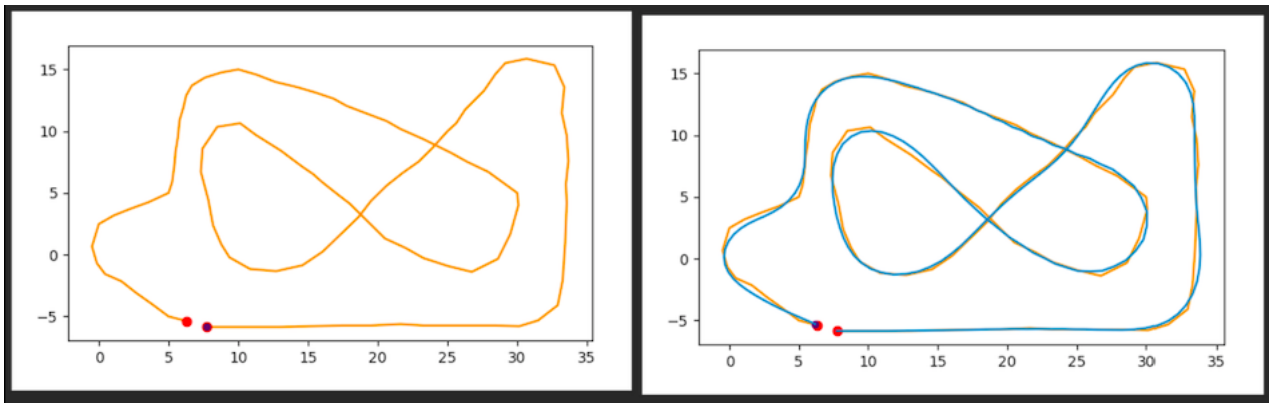

Figure 3.4: Simulation on a track

As one can see, the car's trajectory is smoother with curves and longer turns than the robot. This is because of model I chose: it doesn't allow turns as sharp as the robot that can rotate itself for example.

This semester, testing on a real car wasn't possible for several reasons: it was first scheduled to be done next semester. Also, every module should be done and fully simulated before we can start tests on a racing car.

3.5 Next semester objectives

As said before, the EPFL RT driverless project is spread over a year. The work among this structure isn't done yet: the car physics defined as now are limited. Thus, I have few objectives for the next semester:

- Expand the current car model to integrate a dynamic model beside the kinematic model used for now. These models will be weighted as defined in the AMZ report.
- Start tests on a racing car in order to refine the model: get more precise parameter values to get a model more accurate to reality.
- Switch to another programming language (C++ among others) if this algorithm is not as efficient as needed during tests.

4 Conclusion

After the implementation of was done, the results were really encouraging. Seeing the robot move accordingly to the command we wanted was very motivating. To work on a project with concrete aspects like this one was very interesting and stimulant.

The hardest part of it was to get started. I had lots of difficulties to understand what a MPC problem was, how it is defined and how it is resolved. The implementation was also demanding: I had problems choosing the tools to work with and tested different libraries before selecting the current one I'm using, thanks to others students' advice.

I would like to thank Carlos Conejo for his support and contribution to my project as well as Professor Alahi for allowing me to work on this project.

This semester was a bit unusual because of the lockdown and online courses and I am thankful to EPFL for their understanding during this period of home school.

A File Structure

GitHub repository: `https://github.com/eloigrndl/control_module`

A.1 visulation.py

This is the main file to run to simulate the implemented Control module. Once launched, it will ask on the terminal which model to run on which track.

A.2 Source files

A.2.1 EPFL_RT_car_model folder

It contains all files related to the control module of the EPFL Racing Team as well an IP wrapper to be able to exchange with other modules (and the needed library for this wrapper).

A.2.2 Loomo_robot_model folder

It contains the source file of the control module for the robot.

A.3 Test Footage folder

This folder contains small footage of tests done on the Loomo robot.

A.4 Tools folder

It consists of a file with helpers functions for the *visualuation.py* file and circuit definitions.