ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Robotics project I (MICRO-580)

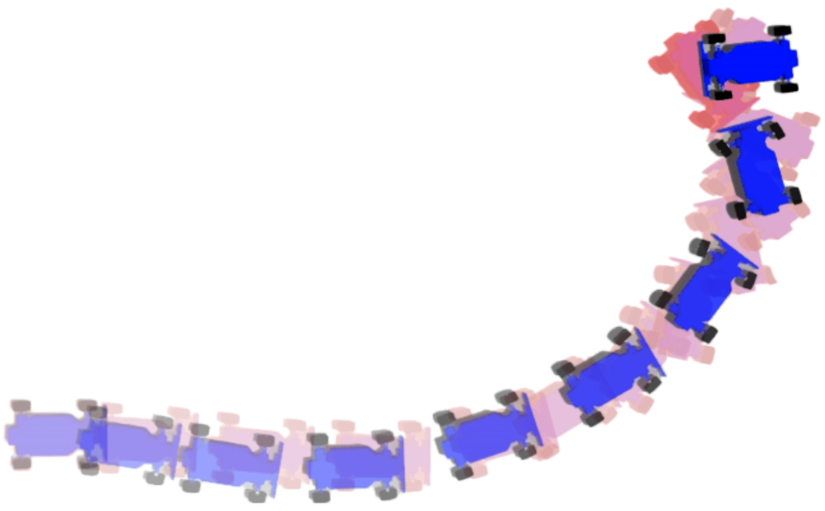
Model Predictive Control with Neural Network augmented vehicle model for dynamic drifting

BioRobotics Laboratory (BioRob)

Author:
Tom Fähndrich (289106)
(*Robotics master student*)

Supervisors:
Dr. Guillaume Bellegarda
(*Postdoctoral Research Associate*)

Prof. Auke Ijspeert
(*Full Professor*)



Fall 2023

Contents

Acronyms

DNN	Deep Neural Network
MPC	Model Predictive Control
MPPI	Model Predictive Path Integral
ms	multiple shooting
MSE	Mean Squared Error
NLP	Nonlinear Program
NMPC	Nonlinear Model Predictive Control
NN	Neural Network
NNDRM	Neural Network Dynamic Residuals Model
NN-MPC	Neural Network augmented Model Predictive Control
OCP	Optimal Control Problem
RC	Radio-Controlled
ROS	Robot Operating System
RTI	Real-Time Iteration scheme
SMPPI	Smooth Model Predictive Path Integral
std	Standard deviation

1 Introduction

Within autonomous vehicle control, managing the dynamics of vehicles during drift phases and at their limits poses a significant challenge. The intricate development of an accurate model and the inherently nonlinear nature of vehicle dynamics create substantial obstacles. This challenge is amplified within Nonlinear Model Predictive Control (NMPC) frameworks, where precise predictions and swift resolution of high-frequency optimization problems are crucial for effectiveness.

To address this challenge, a novel approach is proposed. This method aims to augment a fused kinematic-dynamic bicycle model of an RC car presented in [1] by integrating a Deep Neural Network (DNN) learned layer capturing dynamic residuals. The primary aim is twofold: improving prediction accuracy and reducing reliance on the complex vehicle dynamics model. By capturing residual behaviors through the neural network, this method aims to fine-tune NMPC for optimal performance in controlling the vehicle during drift phases.

The ultimate objective is to enhance performance in two distinct drift phases. Firstly, in feedforward control, targeting precise trajectories of dynamic parking drifts pre-generated offline. Secondly, focusing on real-time control during steady-state (donut) phases. This approach focuses on achieving greater precision and efficiency in controlling the vehicle's slip behavior, marking a significant advancement in autonomous vehicle dynamics.

Large-scale implementation of this method would firstly improve vehicle safety by offering more robust control methods in the event of slippage on slippery roads or heavy braking. Secondly, in the context of autonomous racing, efficient slip management would offer a tool for maintaining maximum velocity when cornering, and open the door to more aggressive trajectories while controlling extreme speeds.

Code: https://github.com/tomfahndrich/NN-MPC-for-autonomous-drifting

2 Problem statement

2.1 Context

This work is an extension of that introduced in the reference publication [1], which presents an optimization framework based on the kinematic-dynamic fusion of the bicycle model to generate drifting trajectories for an RC car using a Nonlinear Model Predictive Control (NMPC) algorithm. This approach proposes three different features: generate steady-state drifting maneuvers (donut), offline drifting trajectory planning, and additionally enable online tracking of the offline computed parking maneuvers (see Figure 1). The results presented by these models are very promising, making it possible to control the drift of the car under several different configurations with notable robustness. However, the model presented remains somewhat inaccurate in drift condition sand opens the door to improvements in the modelling of drifting behavior, which currently rely heavily on the fine tuning of Pacejka tire parameters [2].

We are going to start by presenting the foundations of the fused kinematic-dynamic bicycle model and the optimization framework proposed by the NMPC and the various cost functions in Section 2. We will then propose an improvement approach to the current model based on the addition of a learned dynamic residual model that should, if possible, improve prediction accuracy and controller performances.

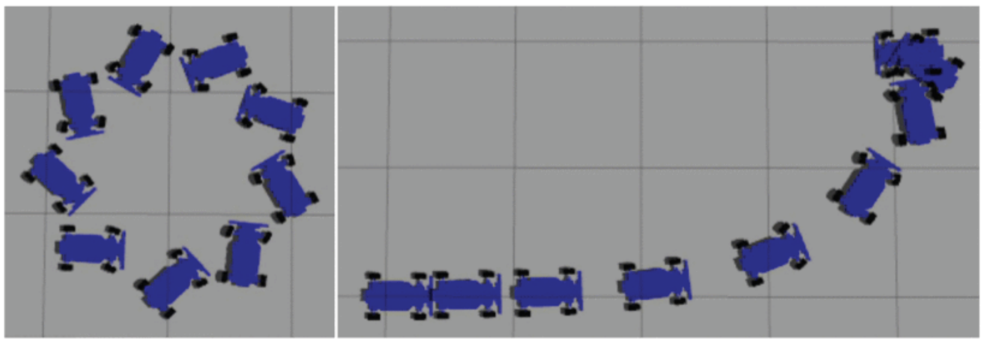


Figure 1: Drifting trajectories of the MIT RACECAR on Gazebo simulations. **Left**: Steady-state drifting. **Right**: Dynamic parking drift [1]

2.2 Dynamic vehicle model

To enable the NMPC to accurately predict the evolution of vehicle states ($\mathbf{x}$) in the future as a function of given commands ($\mathbf{u}$), and to optimize the trajectory, a kinematic and dynamic fusion of the bicycle model is proposed from the reference work. Figure 2 presents the bicycle model of mass $m = 4.78$ kg and inertia $I_z = 0.0665$ kg·m^2 with $l_R = l_F = 0.18$ m respectively the distances of the Rear and Front wheels from the center of mass. The state $\mathbf{x}$ (eq. 1) of the car consists of the body position (X, Y) [m], the yaw ϕ [rad], the body velocities (v_x, v_y) [m/s], the yaw rate r [rad/s] and steering angle δ [rad].

$$\mathbf{x} = [X, Y, \varphi, v_x, v_y, r, \delta]^T \tag{1}$$

The control inputs are presented in eq. 2 and consist of the forces applied by the wheels F_X and the derivative of the steering angle $\Delta\delta$.

$$\mathbf{u} = [F_X, \Delta\delta]^T \tag{2}$$

The full equation of motion of the dynamic $f_{\text{dyn}}(\mathbf{x}, \mathbf{u})$ and kinematic $f_{\text{kin}}(\mathbf{x}, \mathbf{u})$ models are given respectively by equation 3 and 4.

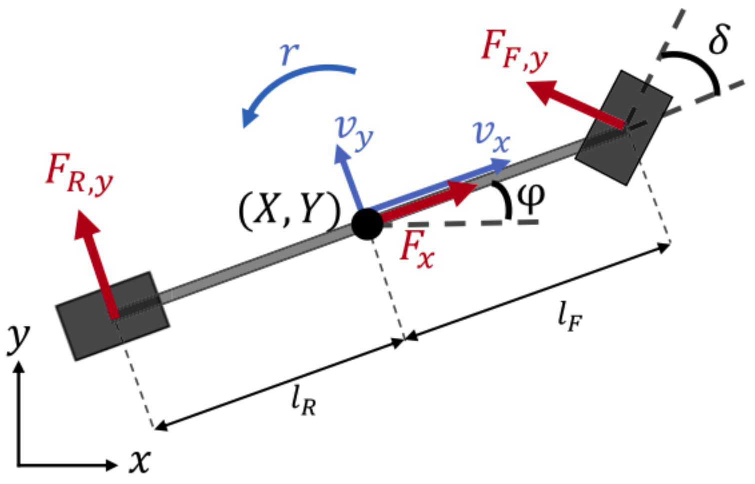


Figure 2: Dynamic bicycle model, with position vectors in black, velocities in blue, and forces in red [1]

Dynamic bicycle model:

$$f_{\text{dyn}}(\mathbf{x},\mathbf{u}) = \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\varphi} \\ \dot{v}_x \\ \dot{v}_y \\ \dot{r} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} v_x \cos\varphi - v_y \sin\varphi \\ v_x \sin\varphi + v_y \cos\varphi \\ r \\ \frac{1}{m}\left(F_x - F_{F,y}\sin\delta + m v_y r\right) \\ \frac{1}{m}\left(F_{R,y} + F_{F,y}\cos\delta - m v_x r\right) \\ \frac{1}{I_z}\left(F_{F,y} l_F \cos\delta - F_{R,y} l_R\right) \\ \Delta\delta \end{bmatrix} \tag{3}$$

Kinematic bicycle model:

$$f_{\text{kin}}(\mathbf{x},\mathbf{u}) = \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\varphi} \\ \dot{v}_x \\ \dot{v}_y \\ \dot{r} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} v_x \cos\varphi - v_y \sin\varphi \\ v_x \sin\varphi + v_y \cos\varphi \\ r \\ \frac{1}{m}F_x \\ \left(\dot{\delta} v_x + \delta \dot{v}_x\right)\frac{l_R}{l_R + l_F} \\ \left(\dot{\delta} v_x + \delta \dot{v}_x\right)\frac{1}{l_R + l_F} \\ \Delta\delta \end{bmatrix} \tag{4}$$

The lateral forces between Front/Rear tires and the ground $F_{R,y}$ and $F_{F,y}$ are computed using eq. 5. The coefficients $B_i, C_i, D_i, i \in \{F, R\}$ (see Table 1) are the parameters from the simplified Pacejka model and are determined experimentally in the reference work. Finally, α_i are the slip angles between the Front/Rear tires and the ground.

$$\alpha_R = \arctan\left(\frac{l_R r - v_y}{v_x}\right) \quad ; \quad \alpha_F = -\arctan\left(\frac{l_F r + v_y}{v_x}\right) + \delta \tag{5}$$
$$F_{i,y} = D_i \sin\left(C_i \arctan\left(B_i \alpha_i\right)\right) \text{ with } i \in \{F, R\}$$

Table 1: Identified Pacejka parameters

	B	C	D
Front	22.70402563	0.13572353	100
Rear	77.24422939	0.1075323	100

The idea behind fusing the dynamic and kinematic models is to take advantage of the strengths of each model with respect to the car's behavior, and to obtain a single model. At low speeds, the simpler kinematic version accurately models the car's response. At higher speeds, the dynamic model takes over to better model the drift phenomena. The speed-dependent linear fusion is presented by the equation 6.

$$\dot{\mathbf{x}} = f(\mathbf{x},\mathbf{u}) = \lambda f_{\text{dyn}}(\mathbf{x},\mathbf{u}) + (1-\lambda) f_{\text{kin}}(\mathbf{x},\mathbf{u}) \tag{6}$$

With the fusion sigmoid λ based on the minimum and maximum blending velocities $v_{b,min}$, $v_{b,max}$ respectively.

$$\lambda = \frac{1}{2}\left(\tanh\left(\omega\left(\left(v_x^2 + v_y^2\right) - \phi\right)\right) + 1\right) \tag{7}$$

$$\phi = v_{b,min} + 0.5\left(v_{b,max} - v_{b,min}\right) \quad \text{and} \quad \omega = \frac{2\pi}{v_{b,max} - v_{b,min}}$$

It is within this dynamic framework that the project aims to bring an improvement based on the addition of a third *learned* ($\mathcal{F}_{\text{learned}}\left(\mathbf{x}_k \ldots \mathbf{x}_{k-h}, \mathbf{u}_k\right)$) term in equation 6 based on a Neural Network capturing dynamic residuals . This addition will be discussed in Section 3.

2.3 Nonlinear Model Predictive Control

The car is controlled using a Nonlinear MPC. The system is discretized and solved using direct multiple shooting (ms) and backward Euler integration. The trajectory is generated by optimizing the cost functions presented in equations 8, 9 and 10. $\boldsymbol{x}_k$ and $\boldsymbol{u}_k$ are the full states and control inputs at time step k (with $k = 1...N$) and the constraints of the optimization problem are: the next state $\boldsymbol{x}_{k+1}$ is constrained by the discretized dynamic (eq. 11), the initial state is given by the actual state of the car (eq. 12), the states and inputs are physically bounded (eqs. 13 and 14).

2.3.1 Optimal Control Problems (OCP)

Steady-state drifting (*donut*): tracks a particular yaw rate r_{ref} and body velocity $v_{x,ref}$ at each time step k to maintain steady-state drifting.

$$J_{\text{donut}} = \min_{\boldsymbol{x}_k,\boldsymbol{u}_k;k=1\cdots N} \sum_{k=1}^{N} \alpha_{v_x}\left(v_{x,k} - v_{x,ref}\right)^2 + \alpha_r\left(r_k - r_{ref}\right)^2 + J\left(\boldsymbol{x}_N\right) \tag{8}$$

Feedforward dynamic drifting (*parking*): offline optimization to bring the car to a certain position (X_{ref}, Y_{ref}) and orientation φ_{ref}, fully stopped (0 velocities) at the end of the sequence.

$$J_{\text{park}} = \min_{\boldsymbol{x}_k,\boldsymbol{u}_k;k=1\cdots N} \left(X_N - X_{ref}\right)^2 + \left(Y_N - Y_{ref}\right)^2 + \left(\varphi_N - \varphi_{ref}\right)^2 + \left(v_{x,N}\right)^2 + \left(v_{y,N}\right)^2 + \left(r_N\right)^2 \tag{9}$$

Tracking: tracks a desired trajectory sequence $(\boldsymbol{x}_{k,ref}, \boldsymbol{u}_{k,ref})$ of length N generated by the offline optimization.

$$J_{\text{tracking}} = \min_{\boldsymbol{x}_k,\boldsymbol{u}_k;k=1\cdots N} \sum_{k=1}^{N} \left(\boldsymbol{x}_k - \boldsymbol{x}_{k,ref}\right) \boldsymbol{R} \left(\boldsymbol{x}_k - \boldsymbol{x}_{k,ref}\right)^T + J\left(\boldsymbol{x}_N\right) \tag{10}$$

These cost functions are subject to:

$$\boldsymbol{x}_{k+1} = f\left(\boldsymbol{x}_k, \boldsymbol{u}_k\right) \tag{11}$$

$$\boldsymbol{x}_0 = \boldsymbol{x}_{\text{init}} \tag{12}$$

$$\boldsymbol{x}_{\min} \leq \boldsymbol{x}_k \leq \boldsymbol{x}_{\max} \tag{13}$$

$$\boldsymbol{u}_{\min} \leq \boldsymbol{u}_k \leq \boldsymbol{u}_{\max} \tag{14}$$

During this project, mainly the feedforward parking drifting will be discussed. In fact, the principal goal of this project is to improve the algorithm's ability to accurately predict the car's behavior in order to better forecast trajectories and reduce the gap between prediction and actual car behavior.

2.3.2 Implementation

This NMPC is implemented in C++ and Python using a symbolic framework provided by CasADi [3] and IPOPT [4] to solve the NLP. Simulations are performed using ROS and Gazebo and the MIT RACECAR as model. All trajectories are run at 50 Hz and the size N of the horizon depends on the case: for example 20 steps (0.4s) for steady-state donut and 115 steps (2.3s) for offline parking trajectory generation ($\Delta t = 20$ ms). In the case of parking trajectory generation, varying the size of the horizon can also generate different car paths with more or less aggressive drift. As expected, the C++ version is more time-efficient than the Python version, reducing the time needed to solve the optimization problem by a factor of 2 or 3.

During this project, as the focus here is on offline trajectory generation, and due to constraints related to the implementation of the Neural Network in the framework, the Python implementation was preferred. In the future, and with a more in-depth real-time perspective in mind, the C++ version should be privileged.

2.3.3 NMPC results

Let's take a look at the results obtained with the reference framework to better understand the interests associated with augmenting the dynamic model.

Feedforward dynamic parking drift

Feedforward parking drift results are shown in Figures 3 and 4 for similar desired end positions ($X_{\text{ref}} = 4$ m, $Y_{ref} = 2$ m) but different orientations ($\varphi_{\text{ref}} = 3.14$ rad and $\varphi_{\text{ref}} = 1.57$ rad).
As these trajectories are generated offline and no feedback is given during the application, we observe significant errors on the position and particularly on the final orientation obtained (see Table 5). We therefore observe purely dynamic offset on these trajectories, which becomes particularly noticeable at the boundaries of the car's performance, when drifting. Finally, Figure 5 shows the trajectories predicted by the MPC compared to the actual trajectory performed by the car during the Gazebo simulation (for the states X, Y, φ). The final offset is easily observable here.

Table 2: Mean final error and std averaged over 10 feedforward dynamic parking trajectories

	X_{ref} [m]	Y_{ref} [m]	φ_{ref} [rad]
Mean error ($\varphi_{\text{ref}} = 3.14$ rad)	0.370 ± 0.112	0.227 ± 0.207	2.127 ± 0.215
Mean error ($\varphi_{\text{ref}} = 1.57$ rad)	0.723 ± 0.084	0.550 ± 0.043	2.107 ± 0.281

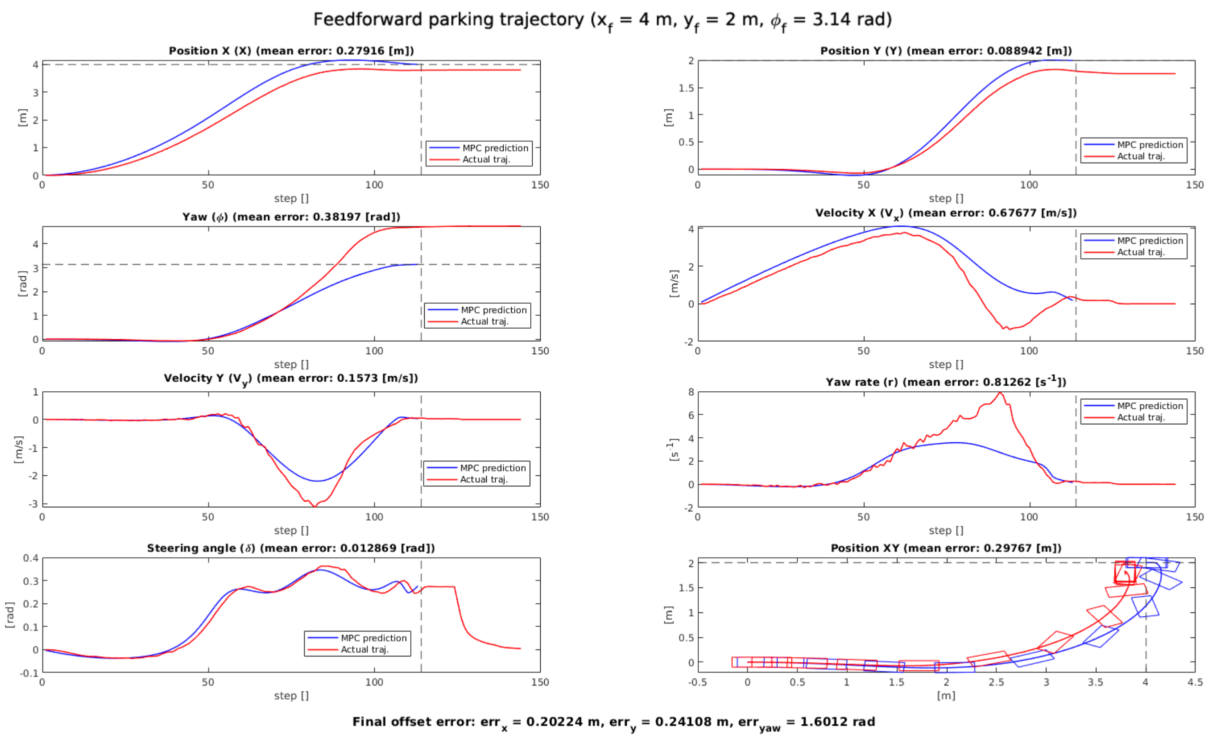


Figure 3: Feedforward dynamic parking trajectory ($X_{\text{ref}} = 4$ m, $Y_{ref} = 2$ m, $\varphi_{\text{ref}} = 3.14$ rad) comparison between the NMPC optimal prediciton (horizon $N = 115$ (2.3s)) and the actual trajectory of the car

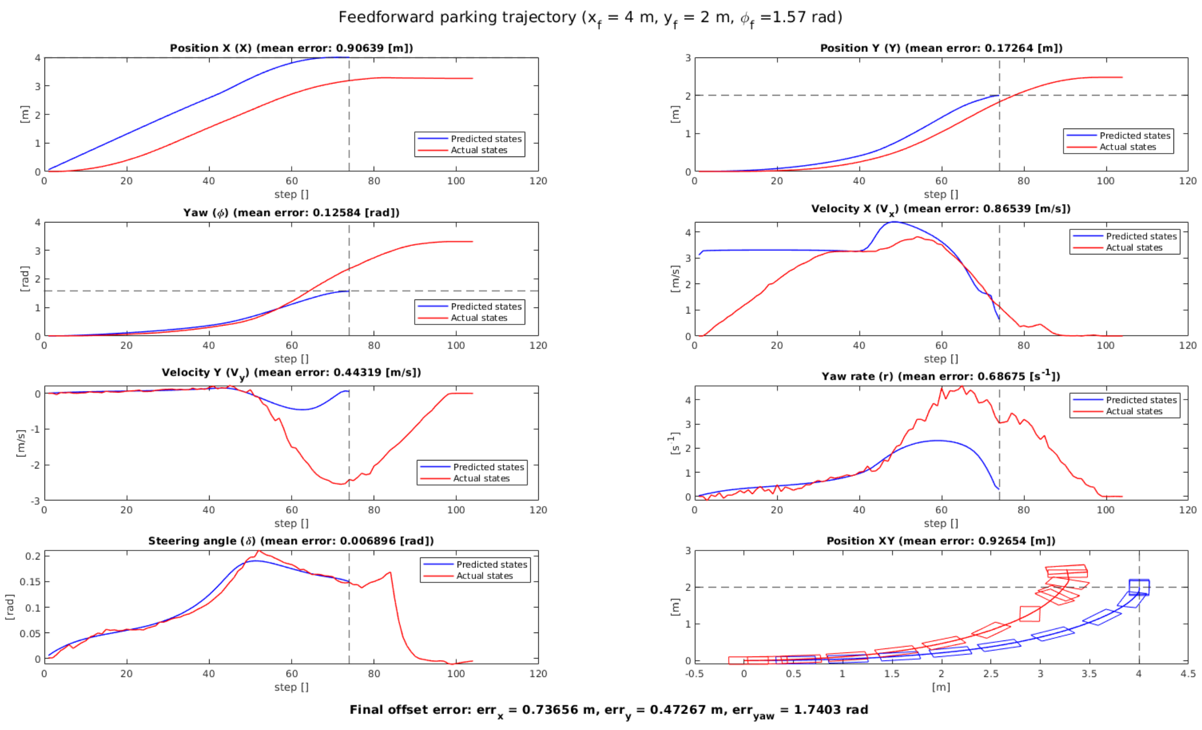
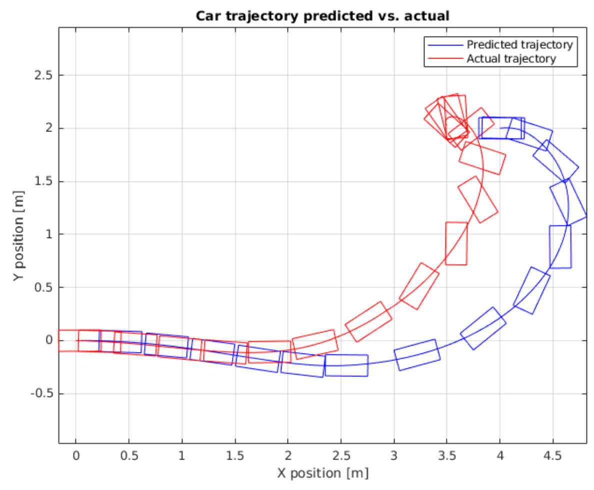
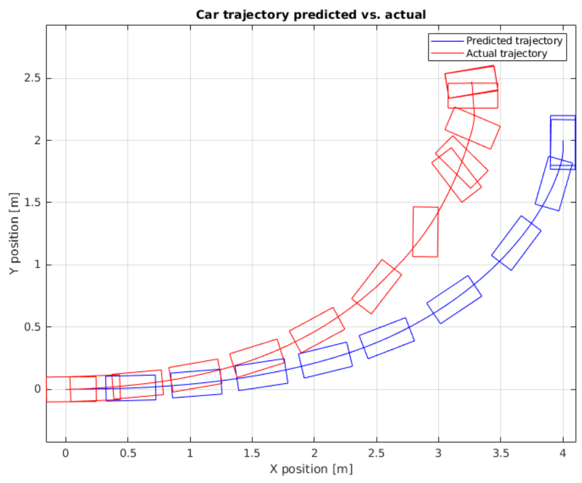


Figure 4: Feedforward dynamic parking trajectory ($X_{\text{ref}} = 4$ m, $Y_{ref} = 2$ m, $\varphi_{\text{ref}} = 1.57$ rad) comparison between the NMPC optimal prediciton (horizon $N = 75$ (1.5s)) and the actual trajectory of the car.



($X_{\text{ref}} = 4$ m, $Y_{ref} = 2$ m, $\varphi_{\text{ref}} = 3.14$ rad)



($X_{\text{ref}} = 4$ m, $Y_{ref} = 2$ m, $\varphi_{\text{ref}} = 1.57$ rad)

Figure 5: Feedforward drift parking trajectories comparison between the NMPC optimal prediciton and the actual trajectory of the car for different final desired orientations.

Steady-state donut drifting

As the focus of the project was not on the online tracking applications of the NMPC, we will briefly present the kind of trajectories obtained for steady-state donut drifting phases. The results showed in Figure 6 are obtained by tracking the following reference states : $v_{x,\text{ ref}} = 1.5$ m/s and $r_{\text{ref}} = 3.5$ rad/s with an horizon of N = 20 (0.4 s) and a measured solving time of approximately 30 ms.

We can observe that although the car is fully capable of stabilizing its skidding by keeping it in a steady-state regime around the desired speeds, a noticeable offset between the one step prediction of the NMPC and the simulation is indeed present on the states of interest (v_x and r). The optimization process is not fully able of predicting the state of the car in the very near future, as the fused bicycle model is somewhat limited in this extreme configuration. Augmenting the dynamic model to make it more accurate near this regime could therefore increase the tracking performance obtained.

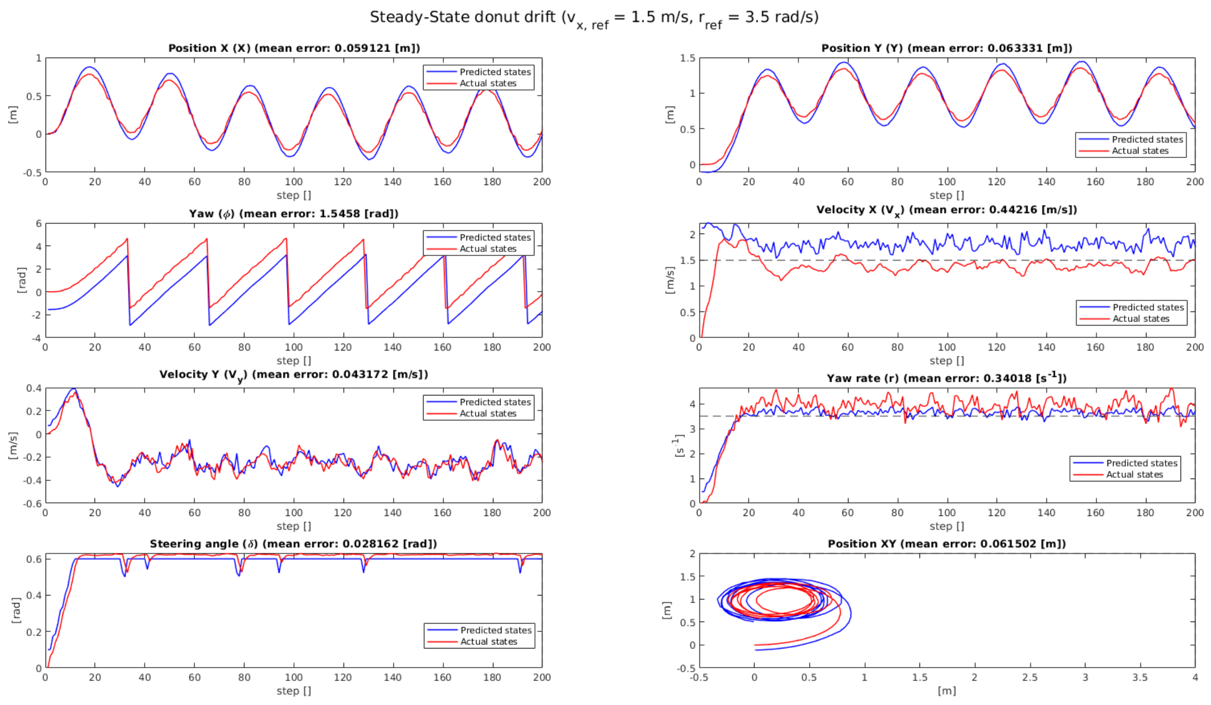


Figure 6: Steady-state (donut) drifting ($v_{x,\text{ ref}} = 1.5$, $r_{\text{ref}} = 3.5$) comparison between the NMPC control algorithm (horizon $N = 15$ (0.4 s)) and the actual trajectory of the car

3 Neural Network Dynamic Residuals Model (NNDRM)

The fused Kinematic-Dynamic bicycle model presented in the section 2.2 and derived by the equation 6 is an approximation of the real car model. This approximation is quite accurate in describing the car's behavior in most cases, but becomes relatively inaccurate in extreme maneuvers such as drifting. The general idea of this approach is to be able to capture the discrepancy between the one-time step prediction of the simplified model and the real behavior of the car in a Deep Neural Network (DNN), so as to be able to correct it.

Instead of using a neural network to re-estimate the Pacejka model parameters at each time step in order to update the dynamics according to the current and past states as presented by [5] or [6], our approach is to directly use the DNN to correct the velocities (eqs. 15 and 16) and assume the Pacejka model parameters to be fixed (Table 1).

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \underbrace{f(\mathbf{x}_k, \mathbf{u}_k)}_{=\dot{\mathbf{x}}_k} \Delta t + \underbrace{\mathcal{F}_{\text{learned}}\left(\mathbf{x}_k \ldots \mathbf{x}_{k-h}, \mathbf{u}_k\right)}_{\text{NN correction}} \tag{15}$$

With:

$$\mathcal{F}_{\text{learned}}\left(\mathbf{x}_k \ldots \mathbf{x}_{k-h}, \mathbf{u}_k\right) = \begin{bmatrix} 0 & 0 & 0 & \Delta v_{x,k+1} & \Delta v_{y,k+1} & \Delta r_{k+1} & 0 \end{bmatrix}^T \tag{16}$$

By correcting velocities exclusively, position and yaw rate states are indirectly corrected, as they are derived entirely from them. The correction made to the one-time step prediction is therefore dependent on the car's current state, as well as a small state history and the current input. In this way, the network can be trained to identify precisely how wrong its next prediction will be, based on the fused bicycle model prediction and the recent states.

But first of all, in order to make accurate corrections, this data-driven model needs to be trained on the most exhaustive set of data possible, exploring a wide range of different car drifting configurations.

3.1 Data generation

The training data consists of dynamic drifting and donut trajectories collected in simulation. All trajectories are different and obtained by varying optimization parameters to achieve different behaviours (horizon length, final tracked position, final orientation, speeds and yaw rate, etc.). This Dataset $\mathcal{D}_{\text{act}}$ represents a total of **71** different drift trajectories and N = **7084** data points.

$$\mathcal{D}_{\text{act}} = \left[\begin{array}{cccccc} \mathbf{x}_{act,0}, & \mathbf{x}_{act,1}, & \mathbf{x}_{act,2}, & \dots & \mathbf{x}_{act,N-1}, & \mathbf{x}_{act,N} \end{array}\right] \quad \text{with } \mathbf{x}_{act,i} \in \mathbb{R}^7$$

From each recorded data point, we can determine the next predicted state using the fused kinematic-dynamic bicycle model (eq. 6) to obtain the corresponding predicted dataset $\mathcal{D}_{\text{pred}}$ as shown in Figure 7.

$$\mathbf{x}_{pred,i+1} = \mathbf{x}_{act,i} + f\left(\mathbf{x}_{act,i}, \mathbf{u}_{act,i}\right)\Delta t \quad \forall i \in [0, N[\tag{17}$$

With Δt = 20 ms exactly as in simulation. The model-based dataset is the following:

$$\mathcal{D}_{\text{pred}} = \left[\begin{array}{ccccc} \mathbf{x}_{pred,1}, & \mathbf{x}_{pred,2}, & \dots & \mathbf{x}_{pred,N-1}, & \mathbf{x}_{pred,N} \end{array}\right] \quad \text{with } \mathbf{x}_{pred,i} \in \mathbb{R}^7$$

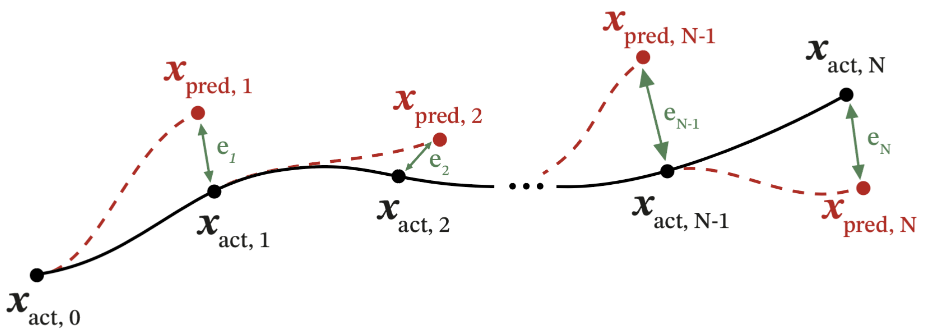


Figure 7: Scheme of the mismatch between actual states recorded in simulation ($\mathbf{x}_{act,i}$) and states predicted by the fused kinematic-dynamic model ($\mathbf{x}_{pred,i}$) for $i = 1...N$.

The mistach between each point of the two datasets represents the prediction error $\mathbf{e}_i$ of the physic-based model with respect to the car's actual behavior, and can be used as a training set for the DNN to correct it. Moreover, as explained at the beggining of this Section, we are only interested in correcting the offset on the velocity states subset. The reduced states training data can therefore be rewritten as follows:

$$\mathbf{x}_{\text{act},i} = \left[\begin{array}{cccc} v_{x,\text{act},i} & v_{y,\text{act},i} & r_{\text{act},i} & \delta_{\text{act},i} \end{array}\right]^T \quad \forall i \in [0, N[\tag{18}$$

$$\mathbf{x}_{\text{pred},i} = \left[\begin{array}{cccc} v_{x,\text{pred},i} & v_{y,\text{pred},i} & r_{\text{pred},i} & \delta_{\text{pred},i} \end{array}\right]^T \quad \forall i \in [0, N[\tag{19}$$

3.2 Deep Neural Network

The fully-connected Deep Neural Network used is designed to be as simple as possible, in order to minimize the complexity addition to the dynamics and avoid adding a useless and excessively non-linear component. The idea is to be able, if possible, to integrate this network as it stands into the dynamics without increasing the solving time of the optimization too much, and maintain the real-time character of the NMPC. A diagram of the DNN is shown in Figure 8. The latter consists of an input layer containing the reduced actual state ($\mathbf{x}_k = [v_{x,k}, v_{y,k}, r_k, \delta_k]$) and input commands ($\mathbf{u}_k = [F_{x,k}, \Delta\delta_k]$), as well as a $h = 2$ time-steps reduced state historic $\mathbf{x}_{(k-1)}$ and $\mathbf{x}_{(k-2)}$.
The inputs are then propagated forward through m hidden layers of activation function $a_i = \text{act}(z_i)$ with z_i the input of each activation and a_i the output ($i = 1...m$). The network output layer are corrections to the next state velocities predicted by the physics-based model and derived from the equations 20-24 with W_i and b_i are respectively the weights and biases of layer i.

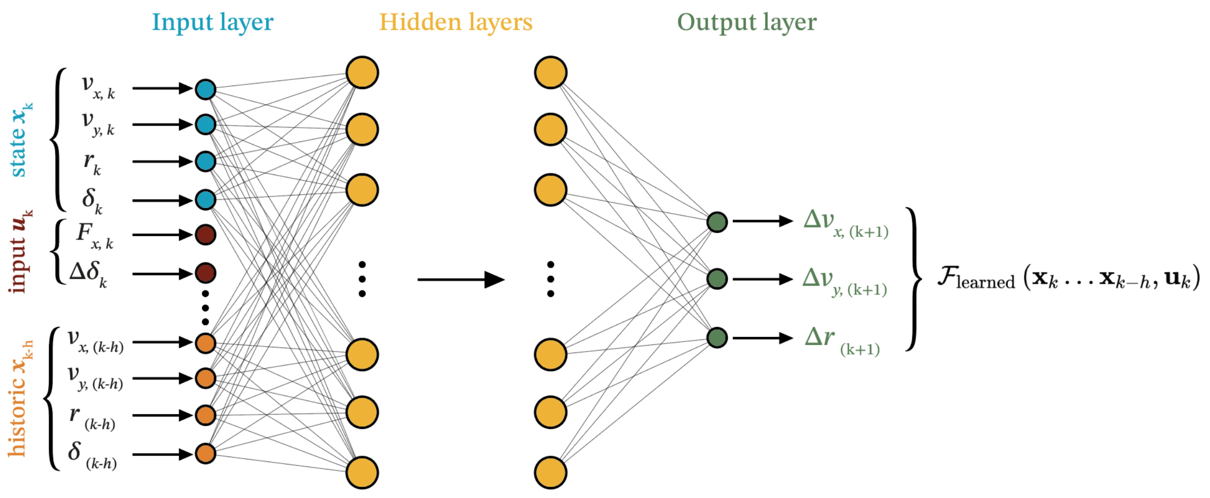


Figure 8: Diagram of the Deep Neural Network dynamic residual model used to correct velocity predicition offsets $\mathcal{F}_{\text{learned}}(\mathbf{x}_k \dots \mathbf{x}_{k-h}, \mathbf{u}_k)$

$$z_1 = W_1^T [\mathbf{x}_k, \dots, \mathbf{x}_{k-h}, \mathbf{u}_k] + b_1 \tag{20}$$

$$a_1 = \text{act}(z_1) \tag{21}$$

$$\dots$$

$$z_m = W_m^T a_{m-1} + b_m \tag{22}$$

$$a_m = \text{act}(z_m) \tag{23}$$

$$\begin{bmatrix} \Delta v_{x,(k+1)} \\ \Delta v_{y,(k+1)} \\ \Delta r_{(k+1)} \end{bmatrix} = W_{\text{out}}^T a_m + b_{\text{out}} \tag{24}$$

Different configurations for the number of hidden layers and activation functions were tested. The results observed were that, as suggested by [7] and [8], often 2-3 or even a single hidden layer of 32 neurons is sufficient to capture a large proportion of the dynamic residuals. Adding more layers does not significantly improve learning but increases significantly the complexity of the model. Regarding activation functions (act(z_i)), 3 variants are tested: `ReLU`, `Softplus` which is a smooth approximation of `ReLU` as suggested in [7], and `Sigmoid` which also has a smooth character and presented by [9]. In view of the difficulties of integrating the Neural Network Dynamic Residuals Model (NNDRM) into the optimization process due to its non-smooth behaviour (which will be discussed later) it is wise to prefer the smoothest possible activations.

3.2.1 Training

The training procedure for the NNDRM is carried out on Python 3 using the PyTorch toolbox [10] and the optimizer Adam [11]. The collection of data $\mathcal{D}_{\text{act}}$ and $\mathcal{D}_{\text{pred}}$ explained in 3.1 are divided into 80% training and 20% testing. The actual reduced states and input commands with a 2 time step historic h_k is forwarded through the DNN (with $\mathbf{x}_{\text{act},\ k}$ defined as in eq. 18).

$$h_k = [\mathbf{x}_{\text{act},\ k}, \ldots, \mathbf{x}_{\text{act},\ k-2}, \mathbf{u}_{\text{act},\ k}] \tag{25}$$

The next fused kinematic-dynamic bicycle model predicted state $\mathbf{x}_{\text{pred},\ k+1}$ is therefore corrected with the output of the DNN.

$$\hat{\mathbf{x}}_{\text{pred},\ k+1} = \mathbf{x}_{\text{pred},\ k+1} + \underbrace{\mathcal{F}_{\text{learned}}\left(h_k\right)}_{\text{NN output}} \tag{26}$$

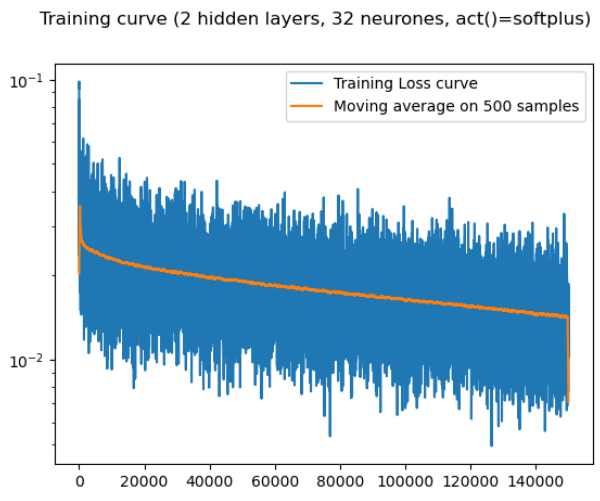
Finally, the optimizer minimizes the mismatch between the next corrected prediction $\hat{\mathbf{x}}_{\text{pred},\ k+1}$ and the next actual state $\mathbf{x}_{\text{act},\ k+1}$ for the states of interest $(v_x,\ v_y,\ r)$ described by the Mean Squared Error (MSE) loss function in equation 27.

$$\mathcal{L}(h_k) = \frac{1}{3}\sum_{i=1}^{3}\left(\mathbf{x}^{(i)}_{\text{act},\ k+1} - \hat{\mathbf{x}}^{(i)}_{\text{pred},\ k+1}\right)^2 \tag{27}$$

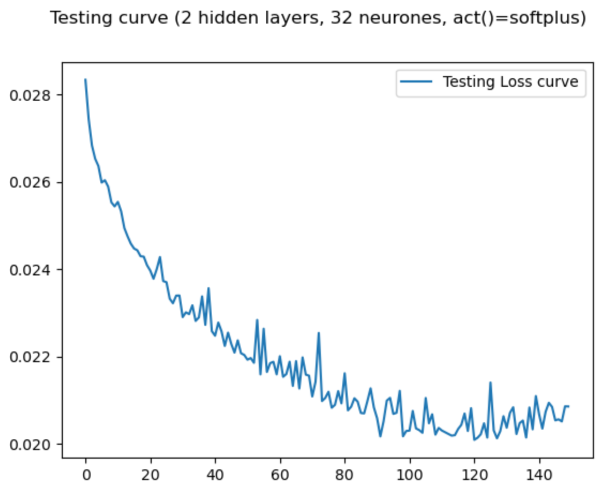
Avoiding overfitting of training data is a crucial element in ensuring that the NNDRM will be able to adapt to new data when exploited. Even if the sensitivity of training quality to these parameters remains very limited, the best parameters observed during training are shown in the Table 3. The training and testing loss curves obtained are presented in Figure 9.

Table 3: Best training parameters

Hidden layers	neurones	epochs	batch size	learning rate	activation
2	32	15'000	200	$1e^{-4}$	`softplus`



a) Training loss function



b) Testing loss function

Figure 9: Loss functions evolution during training of the DNN

Figure 10 shows an actual trajectory of the car used for NNDRM training, the next-step prediction based on the fused bicycle model at each point, and the prediction corrected by the

NNDRM once trained. We can see that the Neural Network does indeed correct some of the model's predictive error, particularly at the end of the trajectory, when the car drifts. This brings us closer to the actual trajectory, resulting in more realistic dynamics. Table 4 summarizes the error with and without trained NNDRM correction for the states concerned on all training data. It can be seen that the NN tends to correct the offset between the prediction and the actual trajectory, although the standard deviation remains significant. In fact, we observe that it seems to be counter-productive to further correct this error, which would result in an overfitting of the training data. A tradeoff must be chosen.

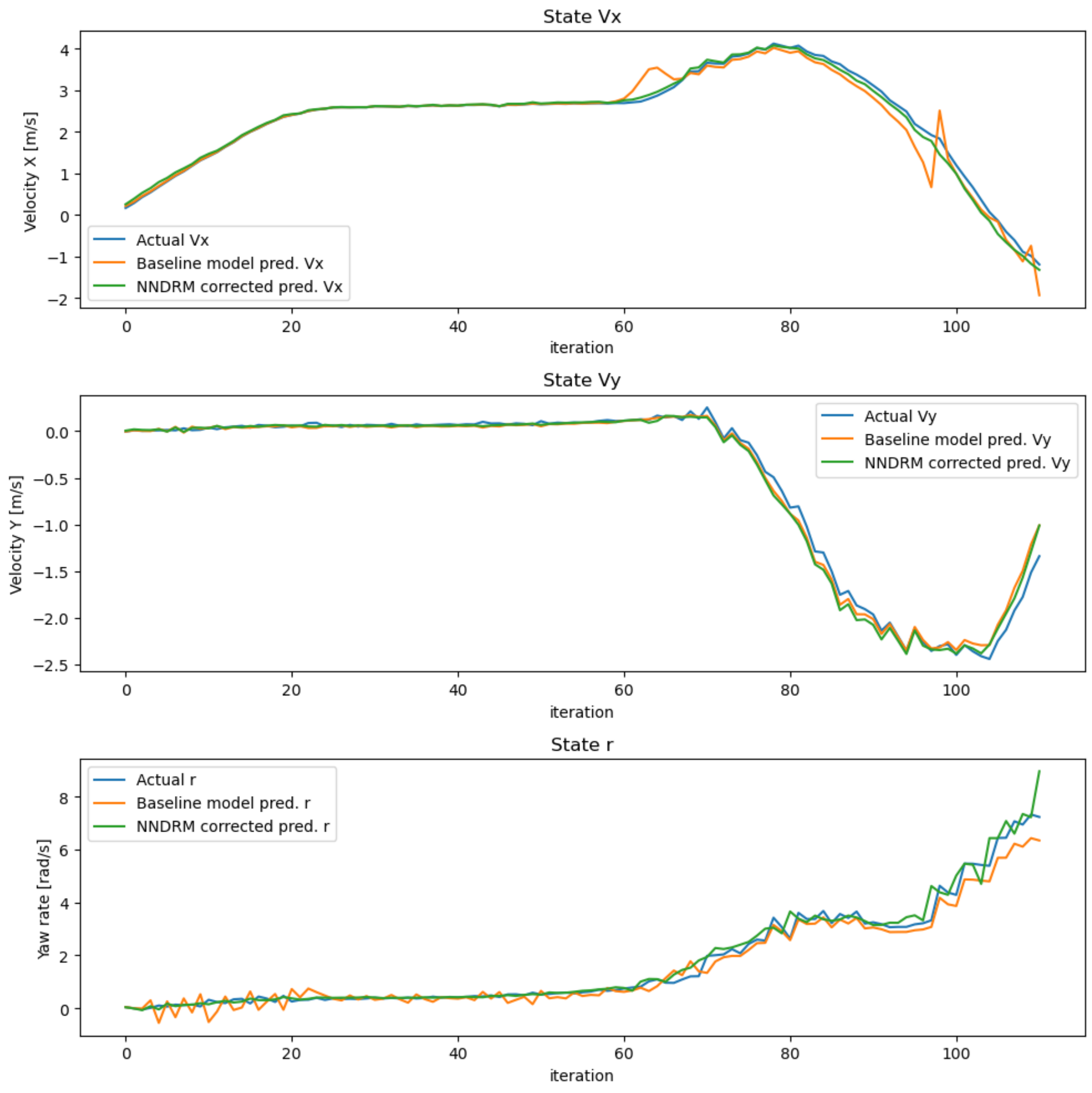


Figure 10: Comparison between an actual training trajectory, the baseline fused kinematic-dynamic bicycle model next step prediction before and after NNDRM correction for the states of interest (v_x, v_y, r)

Table 4: Mean offset and std comparison between the baseline fused kinematic-dynamic bicycle model next step prediction before and after NNDRM correction over all training trajectories for the states of interest (v_x, v_y, r)

	err. v_x [m/s]	err. v_y [m/s]	err. r [rad/s]
Baseline fused bicycle model	0.087 ± 0.174	0.029 ± 0.035	0.229 ± 0.248
after NNDRM correction	$\mathbf{0.031} \pm 0.035$	$\mathbf{0.026} \pm 0.032$	$\mathbf{0.085} \pm 0.142$

3.2.2 Implementation of the NNDRM in the MPC framework

Implementing a PyTorch-learned model within the CasADi symbolic framework required by the IPOPT solver to optimize the trajectory on the horizon is no easy task. This is achieved thanks to the `L4CasADi` toolbox [12] [13] , which enables this seamless integration. In addition, this toolbox offers computationally light locally validated model approximation up to second order to accelerate resolution time and increase real-time capabilities. These can potentially be used in the future. Here's an example of use without approximation:

```python
import torch
import l4casadi as l4c

model = DynamicsNN_torch(...) # Create PyTorch model
model.load_state_dict(torch.load(model_name)) # Load pre-trained weights
# Convert to CasADi symbolic
learned_dyn_model = l4c.L4CasADi(model, model_expects_batch_dim=True)
```

In addition to this, several modifications to the Python optimization framework presented by the reference work in `car_opt.py` had to be made in order to integrate a historic of states and to evaluate the correction determined by the NNDRM at each point of the optimization horizon within the symbolic dynamics function.

Finally, one of the major problems encountered is to boost the convergence ability of the optimization process including the NNDRM, which adds considerable complexity to the dynamic model. Indeed, the non-smooth and data-driven nature of the learned model slows down the optimization significantly, and increases the number of iterations required for convergence in the best-case scenario. For most of the network configurations tested, convergence is very random and by no means guaranteed. The results obtained tend to be highly erratic and unsmooth.

One of the solutions presented to reduce the impact of this problem is to implement a "**warm start**" in the optimization sequence. The first iteration of optimization already corresponds to a past optimal solution and not to random states. This feature increases the likelihood of convergence and reduces the time required for the process to converge, by forcing it to start in a state region close to the optimal one and to those in which the NNDRM has been trained.

4 Results and Discussion

4.1 Offline trajectory optimization

We can now evaluate the optimal trajectories generated by the Neural Network augmented MPC (NN-MPC) including the learned residual of the fused kinematic-dynamic bicycle model presented in Section 3 and trained with the parameters summarized in Table 3. Figures 11 and 12 present the results obtained with the NN-MPC compared to the baseline MPC algorithm for offline optimized dynamic parking trajectories. The reference final position/orientation of the car ($X_{\text{ref}} = 4$ m, $Y_{ref} = 2$ m, $\varphi_{\text{ref}} = 3.14$ rad) and horizon length ($N = 115$ (2.3s)) is similar to the one presented in Section 2.3.3.

We observe that adding NNDRM to the MPC framework tends to significantly improve the accuracy of the optimal trajectory generated. Indeed, in the NN-MPC framework and in comparison with a reference trajectory of the baseline MPC, the optimization seems to take much better account of the car's behavior during the highly dynamic skidding phases at the end of the trajectory. This results in a smaller offset between predicted and actual states throughout the trajectory, and a significantly reduced final error on the reference position/orientation. The addition of the learned component to the dynamics seems to offer a deeper understanding of the dynamic model and enables more accurate prediction of future states, even over a large horizon and without feedback.

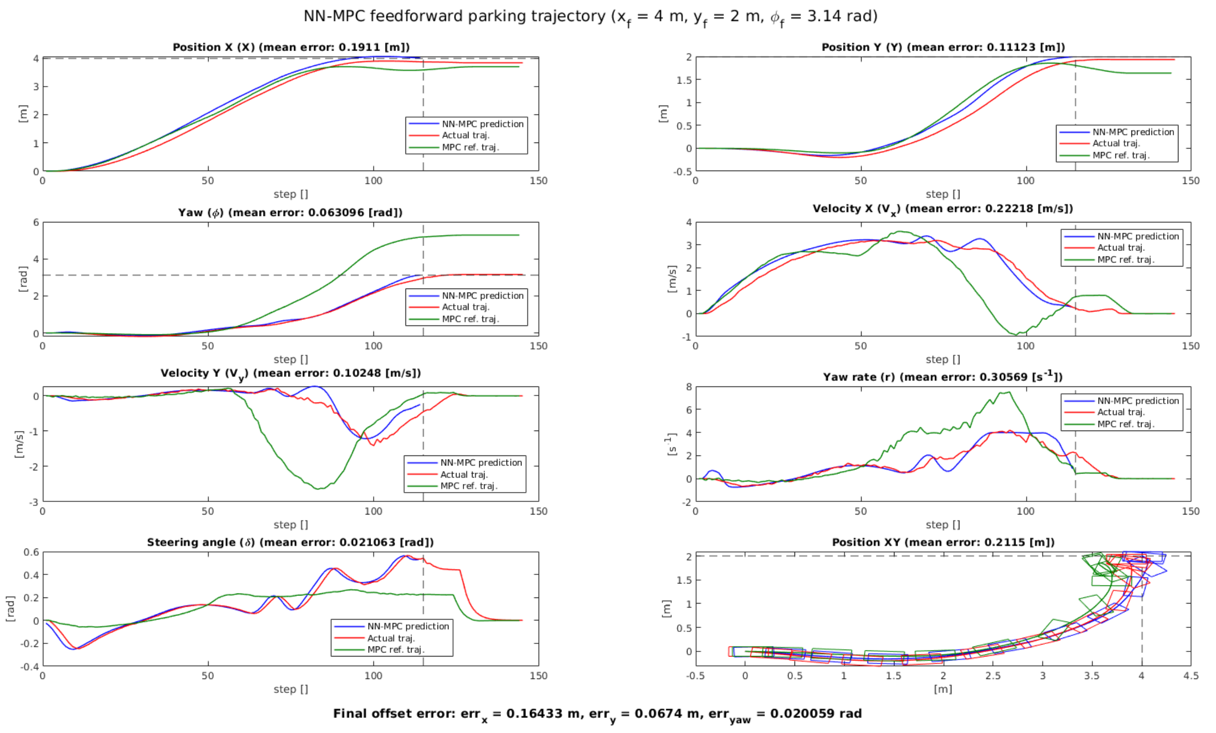


Figure 11: Feedforward dynamic parking trajectory ($X_{\text{ref}} = 4$ m, $Y_{ref} = 2$ m, $\varphi_{\text{ref}} = 3.14$ rad) comparison between the NN-MPC prediciton (horizon $N = 115$ (2.3s)), the actual trajectory of the car and an actual reference trajectory resulting from the baseline MPC

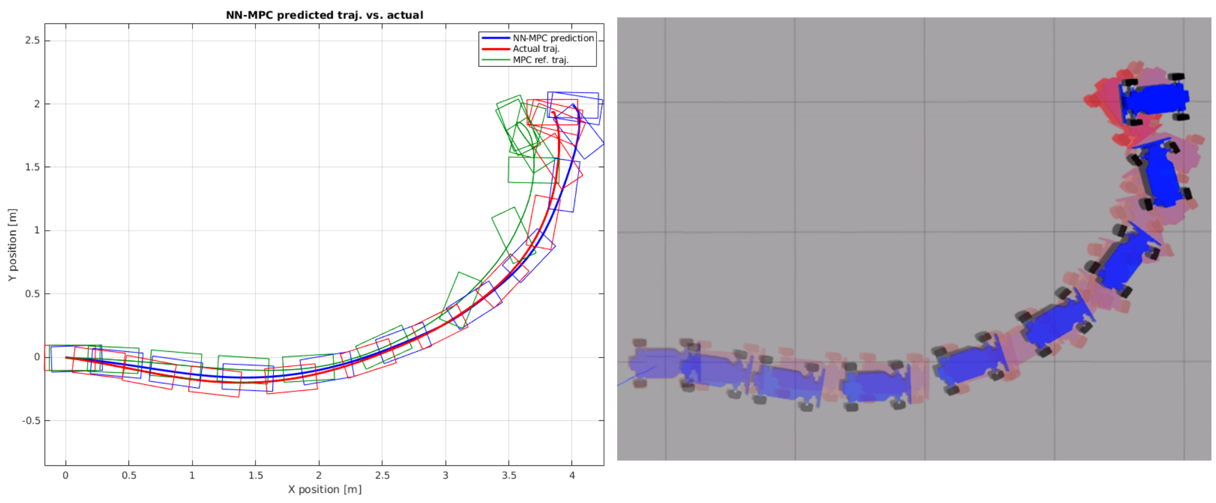


Figure 12: Feedforward dynamic parking trajectory ($X_{\text{ref}} = 4$ m, $Y_{ref} = 2$ m, $\varphi_{\text{ref}} = 3.14$ rad) **Left:** comparison between the NN-MPC prediciton (horizon $N = 115$ (2.3s)), the actual trajectory of the car and an actual reference trajectory resulting from the baseline MPC. **Right:** NN-MPC Gazebo simulation trajectory superimposed with a reference trajectory resulting from the baseline MPC (light red)

In order to better assess the contribution of NNDRM to the quality of the trajectories generated, a comparison of the results obtained between the baseline MPC and the NN-MPC on 10 optimal open-loop parking trajectories is carried out ($X_{\text{ref}} = 4$ m, $Y_{ref} = 2$ m, $\varphi_{\text{ref}} = 3.14$ rad, $N = 115$). The average offset of each state as well as the final error on position and orientation are measured. Table 5 summarizes the mean final errors with std obtained with respect to the desired position and orientation. Similarly to the results shown in Figures 11 and 12, the NN-MPC tends to reduce the final error on all desired states, and particularly on final orientation offset, reducing it by 87%. The latter, which was highly unpredictable in the context of the MPC baseline, now becomes much more accurate thanks to the augmented dynamics that better model transient drifting behavior.

Table 5: Mean final error and std comparison between the baseline MPC and the NN-MPC over 10 feedforward dynamic parking trajectories

	err. X_{ref} [m]	err. Y_{ref} [m]	err. φ_{ref} [rad]
Baseline MPC	0.370 ± 0.112	0.227 ± 0.207	2.127 ± 0.215
NN-MPC	$\mathbf{0.275} \pm 0.112$	$\mathbf{0.158} \pm 0.132$	$\mathbf{0.272} \pm 0.164$
Error reduction	26 %	30 %	87 %

With regard to the average state offsets between optimal prediction and actual simulation response for each trajectory point, the results obtained are shown in Figure 13. Here as well, the framework provided by the NN-MPC compared to the MPC baseline brings a significant correction to almost every state offset. The prediction is closer to reality and models transient drifting trajectories more accurately.

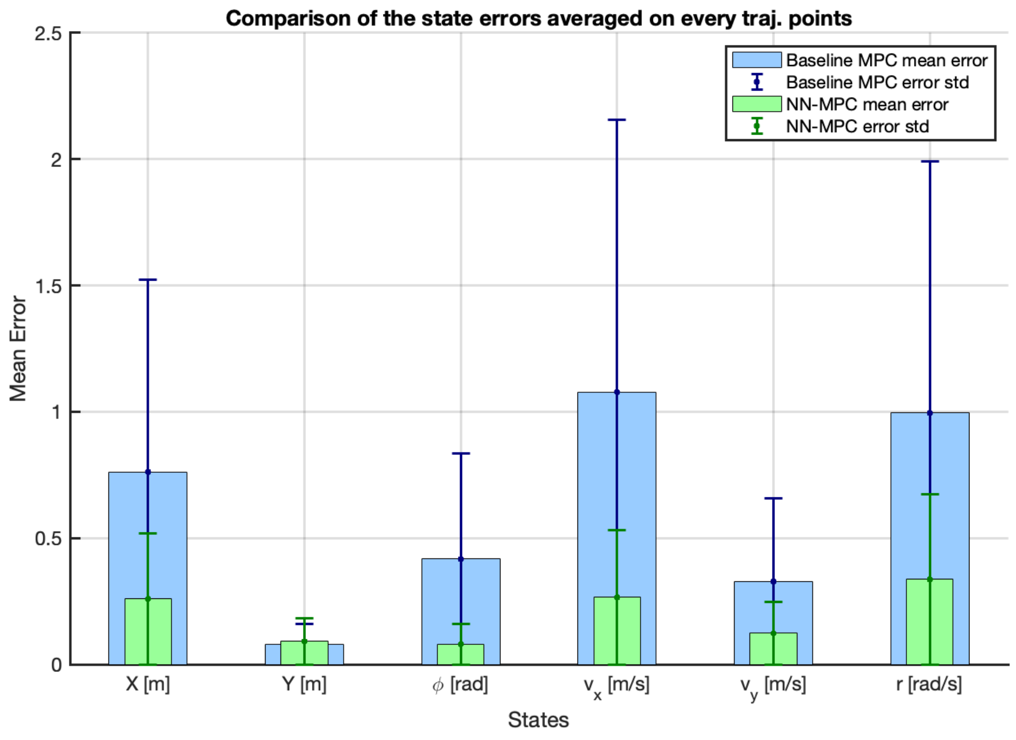


Figure 13: Average state offsets between optimal prediction and actual trajectory followed by the car on every trajectory points with standard deviation for the baseline MPC vs. NN-MPC

4.2 Limitations

Unfortunately, some limitations have to be pointed out. The main one is the difficulty of integrating NNDRM and, more generally, a NN data-driven model smoothly into the optimization process. Indeed, achieving convergence of the optimization process within the augmented dynamics framework is one of the greatest challenges encountered during this project. Several of the elements already discussed (smooth activation functions, small NN, warm starting, avoidance of overfitting, Dropout, etc.) greatly improve the convergence performances of the problem, but the latter remains very fragile and unpredictable. Most of the time, results similar to those shown in Figure 14 (Appendix A) are obtained after a significant number of iterations, even when the problem seems to have converged. The stochastic and non-smooth nature of NNDRM tends to violate several constraints during optimization, generating a noisy, non-smooth and far-from-optimal solution. Almost half of the project was spent solving this problem, mainly by testing a large number of DNN configurations (number of hidden layers, training data, activation functions, etc.), which was partially resolved by implementing the state history as input to the NNDRM and choosing `softplus` as activation.

In the current configuration, the optimization process is capable of generating smooth optimal transient parking drift trajectories like those shown in Figure 11 in around 300-500 iterations, but still struggles to be truly flexible and adaptable and sometimes doesn't converge to an optimal solution while generating certain drift trajectories. A solution to this could probably be to train NNDRM on a much larger range of drift trajectories in order to sample a wider spectrum of dynamics maneuvers and make the model more agile and robust. In addition to this, modifying the way the optimization process is carried out to better deal with the augmented dynamics in order to increase convergence (as discussed in Section 5.1) could be interesting.

The second limitation, which derives from the first, is the noticeable slowdown in the optimization process when NNDRM is integrated. Currently, this prevents the NN-MPC from being used for realtime applications. Indeed, the opmization process is in no way capable of ensuring convergence towards an optimal solution within the 20-50 ms required by vehicle physics. The application of NN-MPC is therefore currently limited to off-line trajectory planning, so that the augmented model can be use to predict optimal trajectories more accurately without the contribution of feedback.

5 Suggested improvements

5.1 Smooth Model Predictive Path Integral implementation

One of the first significant improvements that could be made to the current model is to smooth the optimal actions computed by the NN-MPC without adding a pure smoothing algorithm. This could increase the optimization's convergence performances and increase the robustness of the model. The general idea would be to use Model Predictive Path Integral (MPPI) method presented in [14] and adapted with smoothing capacities by [15] to generate smooth control sequences (Smooth Model Predictive Path Integral (SMPPI)).

MPPI is a sampling-based control approach used for solving non-convex problems that generates a sequence of control actions by sampling a set of potential trajectories and selecting actions that minimize a cost function. It samples multples trajectories based on perturbed control inputs and choose the optimal one through an optimization process. However, MPPI is subject to generate chattering in control actions du to its stochastic nature. The interest of SMPPI is to limit chattering by implementing an input-lifting strategy which consider derivative of the actions as control inputs to smooth the command signals. This method preserve the probabilistic approach of the MPPI while incorporating an intrinsic smoother character. In addition, selecting the best trajectory by evaluating multiple sampled trajectories, each derived by perturbation addition, one could naturally mitigate the non-smooth and non-linear character of NNDRM and prevent the optimization from diverging. Finally, SMPPI smooth actions without any additional smoothing algorithms. Indeed, it might be appealing to simply add a smoothing algorithm to the output of the optimization process. However, because the optimization procedure and this external smoothing mechanism are completely decoupled, the smoothed sequence may unintentionally lose its optimality.

5.2 Real-time application

A second major improvement to this project is enabling the NN-MPC to be used for real-time applications such as trajectory tracking and steady-state drift (Section 2.3.1). Indeed, this is where the NMPC framework really comes into its own, as the system can adapt in real-time and compensate for any discrepancies in order to be as accurate as possible. The first step in achieving real-time capability would be to optimize the NNDRM as discussed above, so that it doesn't destabilize the dynamics and offers optimal smooth action sequences after implementing a SMPPI if necessary. The convergence of the optimization process needs to be fast and straightforward, so training NNDRM on a larger dataset to make the augmented dynamics more agile and robust would probably be necessary. Training the model on driving data from a user in the simulator, as proposed by [16], rather than on a discrete number of trajectories, would be an interesting idea.

However, even with an integrated NNDRM as small as the one presented in Section 3.2, the model still seems to be sensibly heavier and slows down optimization by the addition of a consequent layer of nonlinearity in the dynamics. An interesting solution presented by [13] and integrated directly into the `L4CasADi` toolbox on Python is the local approximation of $\mathcal{F}_{\text{learned}}\left(\mathbf{x}_k \ldots \mathbf{x}_{k-h}, \mathbf{u}_k\right)$ by assuming it to be accurate over the entire input space of states and controls present in the training dataset. " *This way, to speed up the Quadratic Progam generation, the computationally heavy globally valid data-driven dynamics equation can be replaced with a computationally light locally valid approximation up to second order around the current iterate.* "

$$\mathcal{F}^{*}_{\text{learned}}\left(\mathbf{x}, \mathbf{u}\right) \approx \overline{\mathcal{F}}^{i}_{\text{learned}} + \mathbf{J}^{i}_{\text{learned},k}\begin{bmatrix}\mathbf{x}-\mathbf{x}^{i}_{k}\\ \mathbf{u}-\mathbf{u}^{i}_{k}\end{bmatrix} + \frac{1}{2}\begin{bmatrix}\mathbf{x}-\mathbf{x}^{i}_{k}\\ \mathbf{u}-\mathbf{u}^{i}_{k}\end{bmatrix}^{\top}\mathbf{H}^{i}_{\text{learned},k}\begin{bmatrix}\mathbf{x}-\mathbf{x}^{i}_{k}\\ \mathbf{u}-\mathbf{u}^{i}_{k}\end{bmatrix} \tag{28}$$

With $\mathbf{J}^{i}_{\text{learned},k}$ and $\mathbf{H}^{i}_{\text{learned},k}$ the Jacobian and Hessian differentiations of the learned model. Here is an example of how this toolbox functionality can be used :

```python
import torch
import l4casadi as l4c
import casadi as cs
import numpy as np

model = DynamicsNN_torch(...) # Create PyTorch model
model.load_state_dict(torch.load(model_name)) # Load pre-trained weights
# Convert to CasADi symbolic
learned_dyn_model = l4c.realtime.RealTimeL4CasADi(model, approximation_order=2)
x_sym = cs.MX.sym('x', 2, 1)
y_sym = l4c_model(x_sym)

casadi_func = cs.Function('approx',
                          [x_sym, learned_dyn_model.get_sym_params()],
                          [y_sym])

x = np.ones([1, size_in])
```

```
casadi_param = learned_dyn_model.get_params(x)
casadi_out = casadi_func(x.transpose((-2, -1)), casadi_param)
```

(See more complete examples on the toolbox's GitHub repository) However, this implementation in the current framework seems to be relatively significant and complex, but could be an interesting way of speeding up solving time, and can be combined, as explained in the paper, with a real-time-iteration scheme (RTI) [17].

6 Conclusion

The integration of Neural Networks into the MPC framework represents a step towards refining offline trajectory planning and online tracking in robotics and more specifically for autonomous dynamic drifting. By embedding a Neural Network Dynamic Residuals Model (NNDRM) with traditional Model Predictive Control, the project augments the vehicle's model accuracy. This augmentation enhances the vehicle's predictive control, especially in planning complex dynamic drifting maneuvers. The NNDRM's ability to capture and correct residual behaviors in vehicle dynamics has resulted in more accurate offline trajectory generation, showcasing the model's potential in understanding and adapting to high-dynamic scenarios more effectively.

However, several adjustments are still required. The integration of NNDRM, while promising, introduces complexity and unpredictability in the optimization process, highlighting a trade-off between model accuracy and computational tracking. The fast convergence of the optimization, crucial for real-time application, is currently constrained, learning the project's applicability towards offline planning rather than real-time execution.

Future improvements are directed towards enhancing the robustness and computational efficiency of the NN-augmented MPC. Strategies such as Smooth Model Predictive Path Integral control and expanded training datasets are considered to address the current limitations. The adoption of locally valid approximations and real-time iteration schemes may further enable the model to be applicable in real-time scenarios, expanding its utility.

The implications of this project and, more generally, the combination of machine learning and optimal control show promising advancements in autonomous vehicle safety, and performance. Particularly in autonomous racing or advanced driver-assistance systems, mastering vehicle dynamics at the edge of control can lead to safer, more efficient maneuvering. While challenges remain, the project lays down an interesting framework for future work and practical applications on the real F1TENTH [18] and aspires to bring sophisticated control in the most demanding dynamics of vehicle operation to a wider, more efficient, and safer reality.

I'd like to thank Guillaume for his precious help throughout the project. I really enjoyed working with the MPC famework presented in the reference paper and getting familiar with its details, even if it took me quite some time to get completely familiar with it. Finally, this project made me more aware of how difficult it is to implement Deep Learning models for concrete applications, and of the underlying limitations, but also of the opportunities this opens up in the field of control applied to robotics.

References

[1] Guillaume Bellegarda and Quan Nguyen. Dynamic vehicle drifting with nonlinear mpc and a fused kinematic-dynamic bicycle model. *IEEE Control Systems Letters*, 6:1958–1963, 2022. doi:10.1109/LCSYS.2021.3136142.

[2] H. Pacejka. *Tire and Vehicle Dynamics*. 01 2012. doi:10.1016/C2010-0-68548-8.

[3] CasADi. Casadi, 2023. URL: https://web.casadi.org.

[4] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming 106(1), pp. 25-57*, 2006.

[5] John Chrosniak, Jingyun Ning, and Madhur Behl. Deep dynamics: Vehicle dynamics modeling with a physics-informed neural network for autonomous racing, 2023. arXiv:2312.04374.

[6] Taekyung Kim, Hojin Lee, and Wonsuk Lee. Physics embedded neural network vehicle model and applications in risk-aware autonomous driving using latent features, 2022. arXiv:2207.07920.

[7] Nathan A. Spielberg, Matthew Brown, and J. Christian Gerdes. Neural network model predictive motion control applied to automated driving with unknown friction. *IEEE Transactions on Control Systems Technology*, 30(5):1934–1945, 2022. doi:10.1109/TCST.2021.3130225.

[8] Nathan A. Spielberg, Matthew Brown, Nitin R. Kapania, John C. Kegelman, and J. Christian Gerdes. Neural network vehicle models for high-performance automated driving. *Science Robotics*, 4, 2019. URL: https://api.semanticscholar.org/CorpusID:89616974.

[9] Morgan T. Gillespie, Charles M. Best, Eric C. Townsend, David Wingate, and Marc D. Killpack. Learning nonlinear dynamic models of soft robots for model predictive control with neural networks. *2018 IEEE International Conference on Soft Robotics (RoboSoft)*, pages 39–45, 2018. URL: https://api.semanticscholar.org/CorpusID:49656646.

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. arXiv:1912.01703.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. arXiv:1412.6980.

[12] Tim Salzmann, Jon Arrizabalaga, Joel Andersson, Marco Pavone, and Markus Ryll. Learning for casadi: Data-driven models in numerical optimization. 2023. arXiv:2312.05873.

[13] Tim Salzmann, Elia Kaufmann, Jon Arrizabalaga, Marco Pavone, Davide Scaramuzza, and Markus Ryll. Real-time neural-mpc: Deep learning model predictive control for quadrotors and agile robotic platforms. *IEEE Robotics and Automation Letters*, 2023. doi:10.1109/LRA.2023.3246839.

[14] Grady Williams, Andrew Aldrich, and Evangelos A. Theodorou. Model Predictive Path Integral Control: From Theory to Parallel Computation. *Journal of Guidance Control Dynamics*, 40(2):344–357, February 2017. doi:10.2514/1.G001921.

[15] Taekyung Kim, Gyuhyun Park, Kiho Kwak, Jihwan Bae, and Wonsuk Lee. Smooth model predictive path integral control without smoothing. *IEEE Robotics and Automation Letters*, 7(4):10406–10413, October 2022. URL: http://dx.doi.org/10.1109/LRA.2022.3192800, doi:10.1109/lra.2022.3192800.

[16] Mohammad Rokonuzzaman, Navid Mohajer, Saeid Nahavandi, and Shady Mohamed. Model predictive control with learned vehicle dynamics for autonomous vehicle path tracking. *IEEE Access*, 9:128233–128249, 2021. doi:10.1109/ACCESS.2021.3112560.

[17] Moritz Diehl, Hans Bock, Johannes Schlöder, Rolf Findeisen, Zoltan Nagy, and Frank Allgöwer. Real-time optimization and nonlinear model predictive control of processes governed by differential-algebraic equations. *Journal of Process Control*, 12:577–585, 06 2002. doi:10.1016/S0959-1524(01)00023-3.

[18] Matthew O'Kelly, Hongrui Zheng, Dhruv Karthik, and Rahul Mangharam. F1tenth: An open-source evaluation environment for continuous control and reinforcement learning. In Hugo Jair Escalante and Raia Hadsell, editors, *Proceedings of the NeurIPS 2019 Competition and Demonstration Track*, volume 123 of *Proceedings of Machine Learning Research*, pages 77–89. PMLR, 08–14 Dec 2020. URL: https://proceedings.mlr.press/v123/o-kelly20a.html.

A Appendix

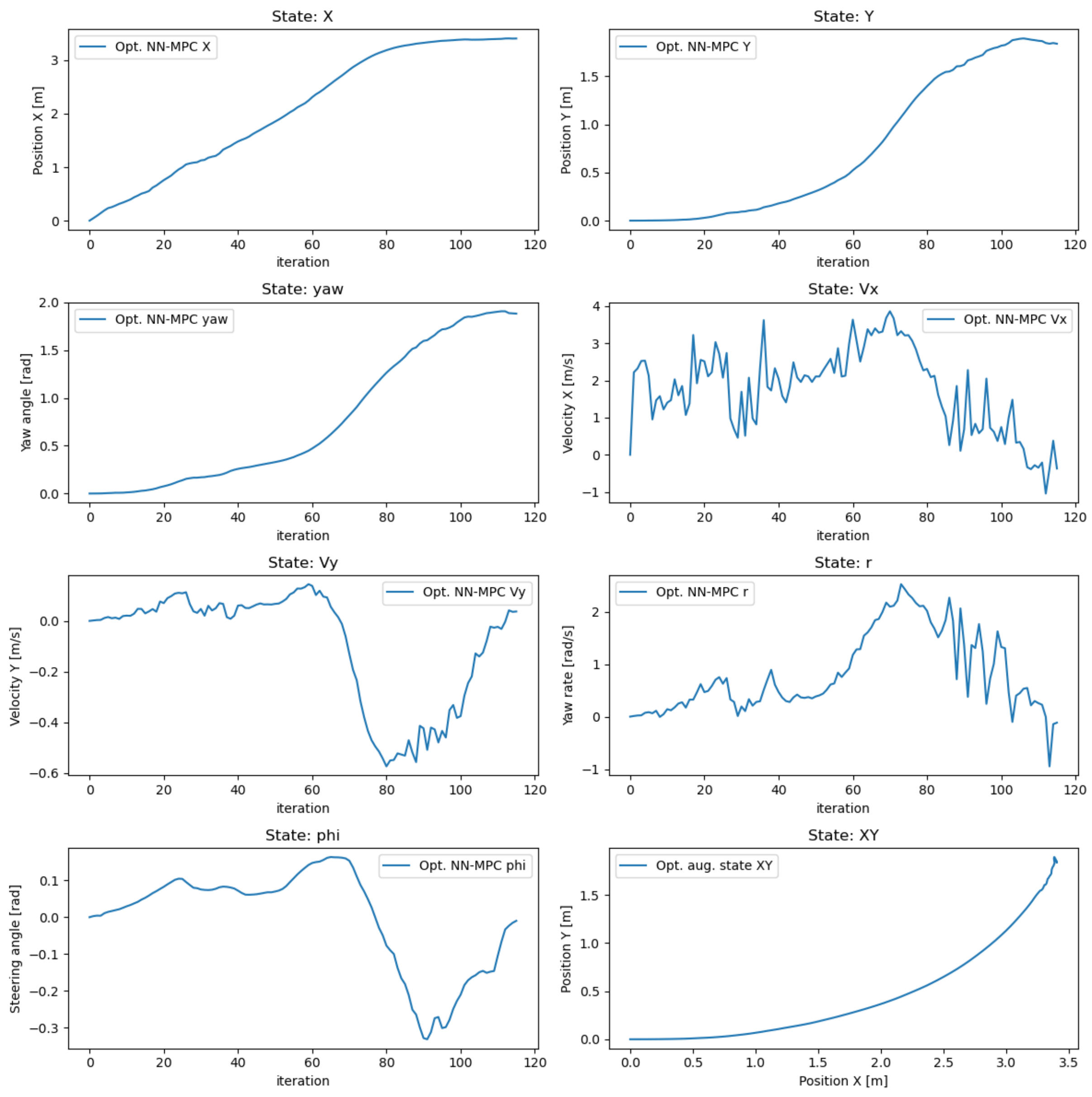


Figure 14: Bad pseudo-optimal trajectory generated by the NN-MPC