Assignment 03_A: Pendulums

Ruben Malacarne (mat. 239934)
University of Trento

The objective of this project concerns the control of Single and Double Pendulums, specifically by employing an Optimal Control Problem (OCP) to generate the dataset for a neural network and utilizing Model Predictive Control (MPC) to generate trajectories from an initial state for the inverted pendulum, both single and double.

The project was developed starting with the implementation of the double pendulum and was divided into four main tasks:

- Build of the Optimal Control Problem (`ocp_double_pendulum`),
- Implementation of the MPC (`mpc_double_pendulum`),
- Development of the neural network, including reading values from the OCP and integrating them into the MPC.
- Tuning the parameter to obtain good results.

Note: Between the double and single pendulums, the parameters N and M do not change much because a good result can be seen without increasing the gap.

The decision to begin with the double pendulum implementation was based on the ability to reuse concepts previously explored during laboratory lessons and to immediately integrate a URDF model. This approach allowed for visualization of the pendulum model and access to joint and link values, which were essential for computing the pendulum dynamics and defining the structure of the initial points.

Subsequently, the single pendulum implementation followed a similar process. The same code base was reused by appropriately copying and modifying the URDF of the double pendulum to implement functions specific to a single joint and link.

For further details regarding the code structure, a `README` file is provided, which explains how the code is organized and offers instructions for its correct execution. Additionally, below is an illustration of the code structure:

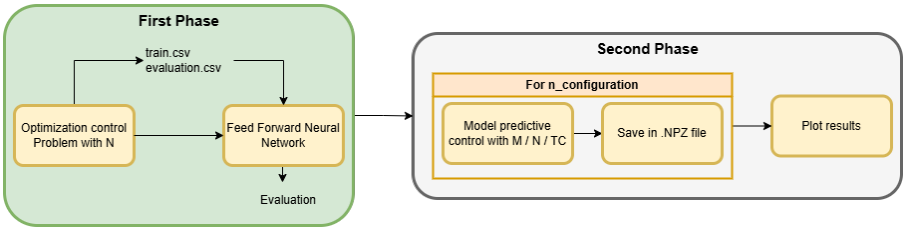

Figure 1: Code Structure

First phase implements the step to compute the model of Neural Network, to use inside like as terminal cost in MPC formulation, that is a second phase.
The Dimension of the pendulum is the following:

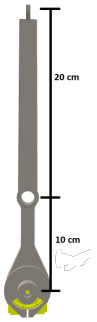

Figure 2: Length of Double pendulum L1:10cm / L2:20cm

1 Optimal control problem implementation

As mentioned previously, the implementation follows the same principle for both the single and double pendulum models. It is based on the "inverse multi-body dynamics" modeling approach introduced during lectures, which enables the use of the same code regardless of the number of joints by appropriately adapting the system dynamics. The model is directly obtained from the URDF provided by the `example_robot_data` library, which is translated into dynamic functions using `KinDynComputations`. In both models, the generation of states is performed either randomly or using a "uniform grid" approach. This involves randomly generating positions and velocities to ensure a dataset that is as comprehensive as possible.

The dynamic function of the system under consideration is expressed as:

$$\tau = M(q)\ddot{q} + h(q, \dot{q}),$$

The optimization problem formulations considered for the single and double pendulum are as follows formulation:

$$\text{Minimize:} \quad J = \sum_{i=0}^{N-1} \left(w_p \|q_i - q_{\text{des}}\|^2 + w_v \|\dot{q}_i - \dot{q}_{\text{des}}\|^2 + w_a \|u_i\|^2 \right)$$

Subject to:

$$x_{i+1} = x_i + \Delta t \cdot f(x_i, u_i), \quad f(x_i, u_i) = \begin{bmatrix} \dot{q}_i \\ u_i \end{bmatrix}, \quad x_i = \begin{bmatrix} q_i \\ \dot{q}_i \end{bmatrix}, \quad \forall i = 0, \ldots, N-1,$$

$$x_0 = \begin{bmatrix} q_0 \\ \dot{q}_0 \end{bmatrix}, \quad x_0 = \text{param\_x\_init},$$

$$\tau_{\min} \leq \tau_i \leq \tau_{\max}, \quad \tau_i = M(q_i)u_i + h(q_i, \dot{q}_i).$$

We can notice that: 1. The first constraint is about dynamic of the system, define the state of the system on time $i+1$ (x_{i+1}), define the state x_i, the control u_i,and step Δt. this guarantees that the trajectory is calculated using real dynamic system. 2. set the initial condition 3. limit of tourque.

Once the optimizer is executed, the results are stored in a CSV file, storing the state (q and dq) and the final cost associated with that state. The OCP can be solved by considering either the horizon $N(= 100)$ or $M(= 5 or 6)$, and it is also possible to combine them to visualize the potential results. The CasADi solver considered is as follows:

```
opts = {
    "ipopt.print_level":    0,
    "ipopt.max_iter":       config.max_iter_opts,
    "print_time":           0,
    "ipopt.tol":            1e-3,
    "ipopt.constr_viol_tol":1e-6
}
```

Note: in MPC process I have used different type of CasADi solver because using Neural Network it's necessary use the "hessian_approximation" to compute the value approximation function.

2 Model predictive control implementation

The implementation is based on an optimized formulation that combines system dynamics, control constraints, and an optional terminal cost (classical, with neural network-based, or hybrid). The based script for the OCP system is reused, with the addition of a *warm start* and the terminal cost and other stuff for MPC.

Note: In the case of the terminal cost calculated using the neural network, it is integrated to predict the terminal cost based on the last state $X[-1]$.

Regarding the simulation, it can be visualized using Pinocchio, it is possible to choose between "ideal" or the "Pinocchio" simulation type (based on evaluations and tests, trajectories, and results remain unchanged between the two). Once the simulation is executed, the results are saved in a `.npz` file to facilitate future access and analysis.

The formulation implementing for the model predictive control is the follwing:

$$J = \sum_{i=0}^{N-1} \left[w_p \|q_i - q_{\text{des}}\|^2 + w_v \|\dot{q}_i\|^2 + w_a \|u_i\|^2 \right] \qquad + w_{\text{final}} \left[\|q_N - q_{\text{des}}\|^2 + \|\dot{q}_N\|^2 \right] + w_{\text{value\_nn}} \cdot \text{NN}(x_N)$$

Note: the constraint are the same as, the terminal constraint on velocities is added:

$$\text{Subject to:} \quad dq_{\text{final}} = \begin{bmatrix} dq_1 \\ dq_2 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}$$

I have notice also that the *warm start* helps to reduce computation times. This is due to the fact that it uses the value from the previous iteration k. However, it still maintains the same number of iterations required to reach the goal without introducing any additional drawbacks in the code. This has led to a significant improvement while preserving the total number of iterations, and testing the code without attempt a lot of time.

3 Neural Network implementation

The neural network implemented is trained on the data generated by the OCP.
The key points are data handling, network architecture, training process, integration with CasADi for MPC (using L4Casadi library), and evaluation metrics.
To improve numerical stability during training, the optimal cost ($J_{x_{\text{init}}}$) is approximated using its scaled logarithm. Additionally, the model generates a symbolic representation as output, enabling seamless integration into MPC problems.
The data preprocessing steps include:
- Computing the logarithm of the cost.
- To prevent the network from "working" on too large log-cost values, additional scaling is performed [-1 and 1] using MinMaxScaler. In this way, the network receives as input and outputs smaller numerical values, which improves convergence especially when using activations tanh.

Architecture

The network is a Feedforward Neural Network (FNN) with the following configuration (the network for the single pendulum is identical, except that it is calibrated for a single joint):

- **Input Layer**: Accepts a system state $[q_1, q_2, \dot{q}_1, \dot{q}_2]$ of size 4.
- **Hidden Layer**: Contains 64 neurons with `Tanh` as the activation function, ensuring continuity and differentiability, which are necessary for optimization.
- **Output Layer**: Produces a single scalar value representing the scaled logarithm of the terminal cost.

Note: While building the code I tried to use the `RELU` activation funcionts instead of `Tanh`, but after a series of attempts, Tanh proved to perform better.

During training, it's used Adam optimizer and an early stopping to prevents overfitting, splitting in batch the dates. The loss is on the scaled log cost, which is then transformed back to the original scale during evaluation.

To check the neural network it's built the evaluation. To do that I running again the OCP script, to genrate (always in random way), other initial points, but only 100 points respects 1100 during training part.

As before, the data is analyzed by appropriately scaling it, and the terminal cost is transformed using the logarithm to ensure a more stable scale, as was done during the training phase. Subsequently, the model is set to evaluation mode to assess its performance. The real and predicted values are reported in the cost space, allowing the calculation of the three main metrics: MSE, RMSE, MAE Finally, a plot is generated to verify and compare the predicted values with the actual values.

To summarize, the follow of NeuralNetwork class is: reads data from a CSV, scales the log of the costs, and uses a feedforward network. The training process uses MSE loss on the scaled log costs. The CasADi function is created to deploy the trained model in the OCP solver, allowing the MPC to use the NN's predicted terminal cost. Invert the scaling and exponentiate to get the actual cost prediction.

4 Parameter and hyperparameter analized

The key components of the simulation, to improve the performance, is setting the parameters considered for each type of pendulum, (this part is inside of `conf_XXX_pendulum`, the configuration file). The fixed parameter are: N_step, M_step, max_iter_opts, and tau (tourque) During the training part I have adjusted the learning rate value, now is equal to 0.01, and also the size of the hidden layer, inicreasing the size I have notice an increasing overlapping between predicting point and real point. Instead in MPC process, I setting the weight like position, velocity, acceleration and weights for teminal cost, standard or with Neural Network. During the tunning I have notice that the position weight is fundamental to mantain the position and reach the goal. But if I set the value too high, the pendulum may not move.

5 Results obtained

Regarding the obtained results, they have been saved in an NPZ file, which allows for easy reloading in Python, as this format can be read without extensive reconfiguration.
To run this part it's possible see the `main_test.py`.

Initially, I set three types of the initial state for pendulums to see possible interesting differences. After that, it's create a cycle for to testing 5 types of MPC process(2 more than planned):

- Standard case with M iterations (no TC).
- Case with $M + N$ iterations (no TC).
- Case with M iterations and NN as TC.
- Case with M iterations with classical TC.
- Case with M iterations with hybrid TC, which combines both the cost predicted by the network and the classical cost.

Note: For further clarifications, refer to the folders `Images` containing all the results and plots.

Subsequently, after the computation times for MPC, is build up another phase to represents the results in some plots.
Note:

- An attempt was made to analyze the predicted and actual trajectories during MPC. However, due to the system's chaotic nature—resulting from multiple overlapping lines caused by the trajectory being recomputed at each MPC step—it was deemed more appropriate to perform an analysis and comparison of different simulations instead.

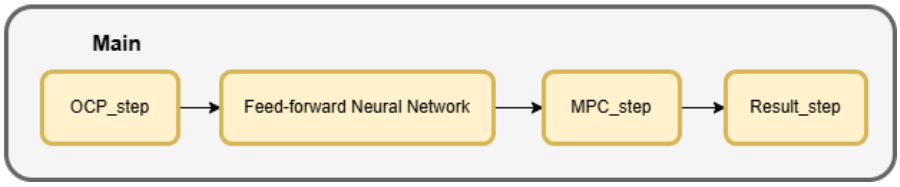

Figure 3: Flow of the code

5.1 Double_Pendulum result

Starting from the double pendulum system, in terms of the trajectories followed by the joints to reach the equilibrium point (goal position), it was observed that the case with only N time horizon, has the highest oscillations corresponds to the simulation because that does not integrate the Terminal cost and termianl constraint. This results in "over-oscillations," caused by the pendulum's inability to predict the target point sufficiently in advance.
In contrast, when using a time horizon equal to the sum of N and M, there is a significant attenuation of the over-elongation, allowing the system to better stabilize and reach equilibrium.
In the case of the standard Terminal Cost and the case using the Neural Network (NN), I observe that the system comes to a gradual stop without sudden displacements.
In all cases, the number of iterations required is less than 90, except for the first trajectory.

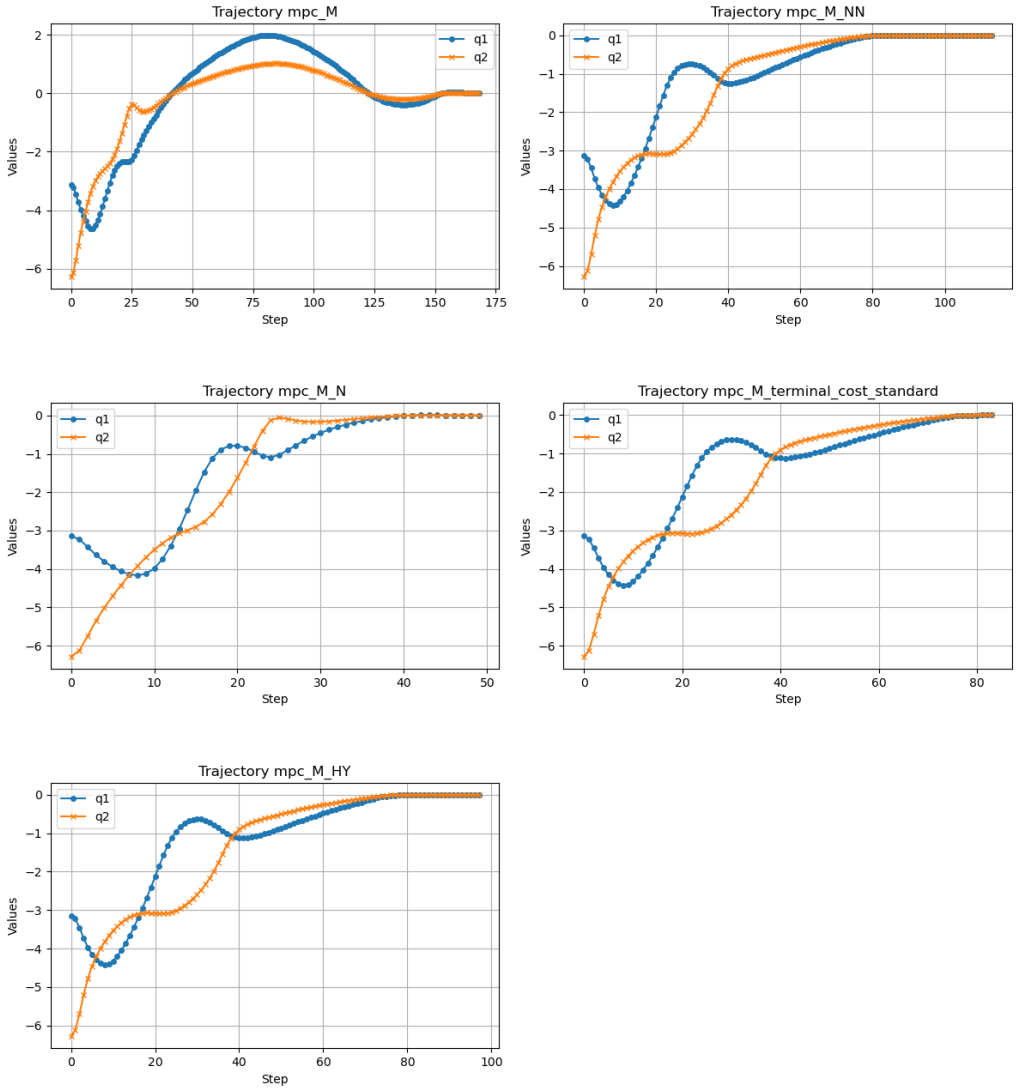

Figure 4: trajectory results

Regarding the "dance"(the movement of the pendulum) performed by the system, exhibits a less chaotic motion when Terminal Constraints are applied. (see folder `double_pendulum` and select a `Image` to see the results).

Regarding the execution time and the number of required iterations, without modifying the weights, it was observed that the MPC implementing M and $Neural\_Netrowk$ has the time horizon performs much better than other, but the lower number of iterations is the MPC with M and N this is because MPC with the neural network tends to have a running cost that is not very close to zero, even though it has reached the equilibrium state. Maybe due to the presence of some approximations in the type of OPTS used. In this phase, the system tends to perform a higher number of iterations than necessary, even after reaching stability.

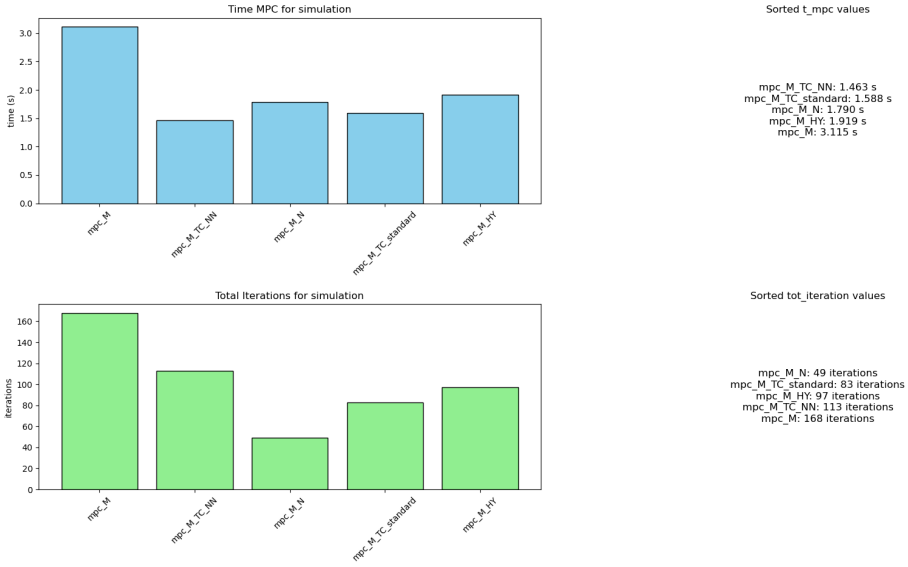

Figure 5: Time and iteration of MPC

It's possible notice that ther is a good compromise about using MPC with classic TC, infact have the similar result like MPC with NN but have more iteration and spend more time.

The dynamic analysis reveals several variations in acceleration. In particular, the acceleration is generally very high during the initial iterations due to the large error. This effect gradually decreases as the pendulum reaches the goal.

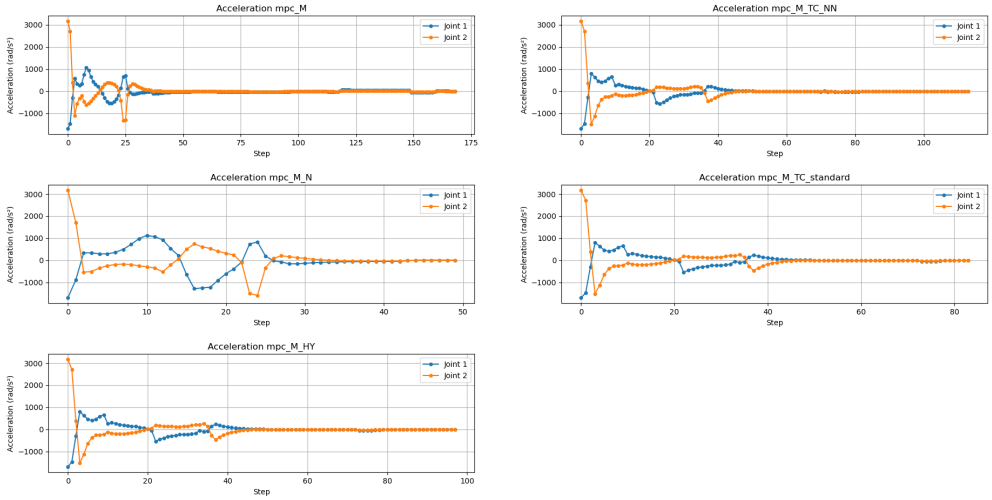

Figure 6: joint_acceleration

5.2 Evaluation Double_Pendulum

The evaluation part return 3 values: MSE, RMSE MAE and the visualization about, True vs. predicted costs (sorted and unsorted) and Error regions (over/underestimation). MSE gives the average squared error, which is useful but in squared units. RMSE is the root of MSE, bringing it back to the original unit, making it more interpretable. MAE measures the average absolute error, which is less sensitive to outliers than MSE/RMSE. The first plot shows true vs. predicted costs in the order of the samples, which helps see if predictions follow the true values across the dataset. However, since the samples are ordered by their index, it might be random. The second plot sorts the true costs and aligns the predictions accordingly. This helps visualize if the model's predictions maintain the correct order and how they deviate when the true costs are sorted. The fill between the lines highlights over and underestimations, which is crucial because in control problems, underestimating costs might be riskier than overestimating.

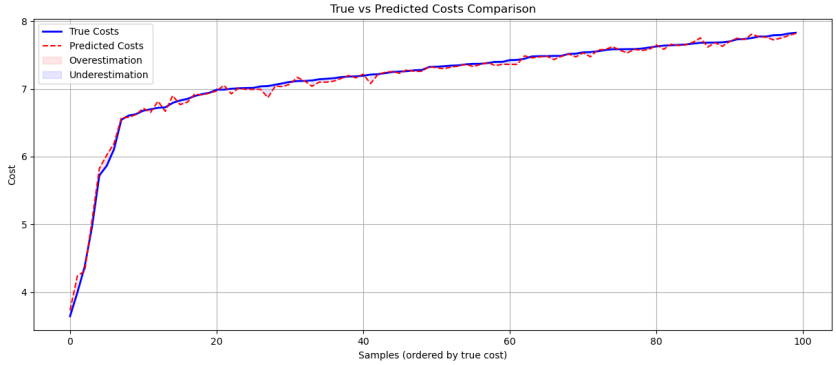

Figure 7: Enter Caption

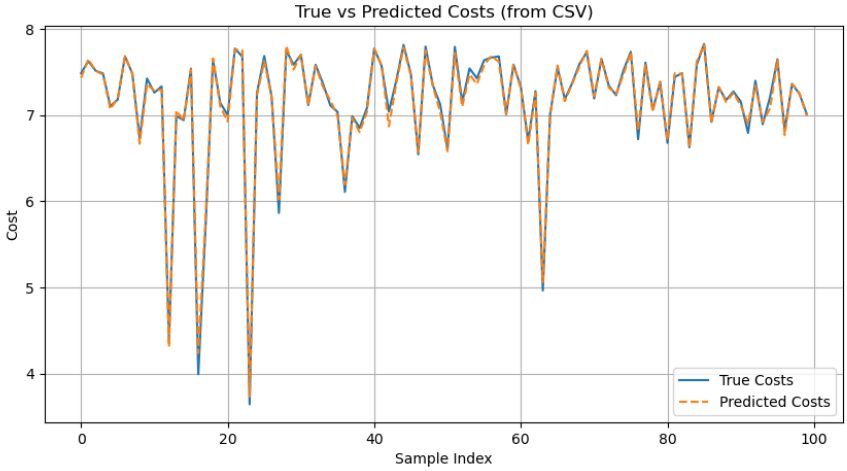

Figure 8: Enter Caption

and the evaluation result are: mean squared error (MSE): 0.0031 root mean squared error (RMSE): 0.0558 mean absolute error (MAE): 0.0396

Note: the time taken to train the neural network is not very long, about 4 or 5 seconds.

5.3 Single_Pendulum

Inside of single pendulum the comutation velocity are too fast then before, like the number of iteration is decrese, this aspect is due to the fact that I have a single-joint system and a single link. For the trajectory of the pendulum, for all the system suffer by an overshooting in the other direction, In the case of pendulum motion, all systems exhibit a slight overshoot or rebound in the opposite direction. However, this value tends to decrease with the use of the TC, both with and without a neural network.

Moreover, all four iterations tend to stabilize after 25 iterations, with stabilization occurring around the 17th iteration when using the neural network.

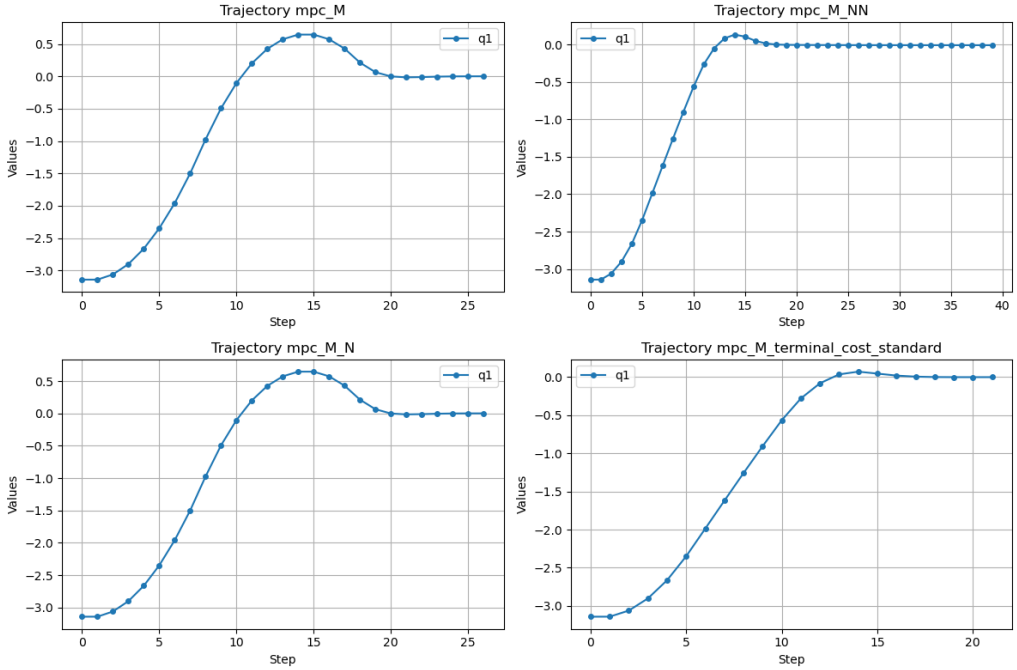

Figure 9: plot trajectory

Regarding execution time and the number of required iterations, I observe that the computational time is significantly faster compared to the double pendulum. This indicates that a less complex approach, such as using a simpler terminal cost or a longer time horizon instead of a neural network, can achieve stability in a shorter time.

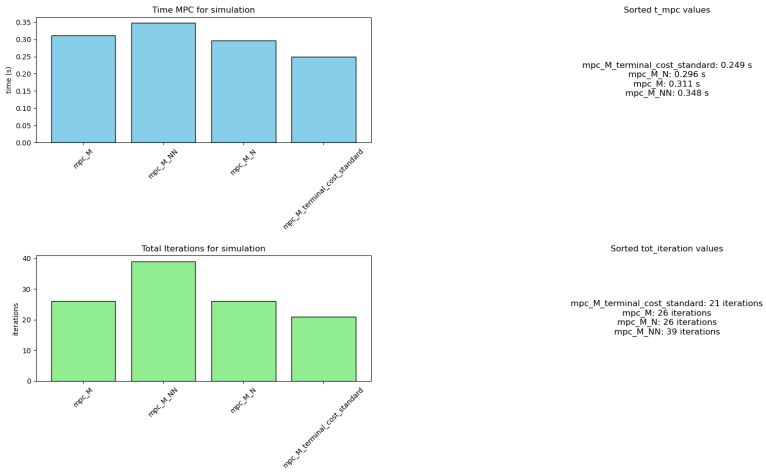

Figure 10: Enter Caption

From a dynamic perspective, the accelerations are still high, though lower than those of the double pendulum. However, they appear to be less smooth.

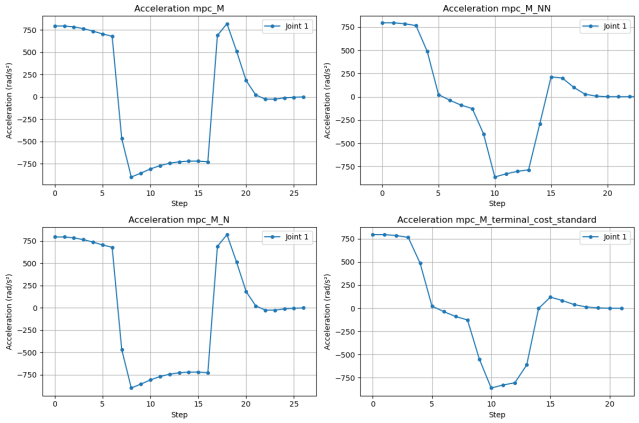

Figure 11: Enter Caption

This can be made smoother (still with peaks but less abrupt) by increasing the weight on acceleration. In fact, by doing so, I obtain:

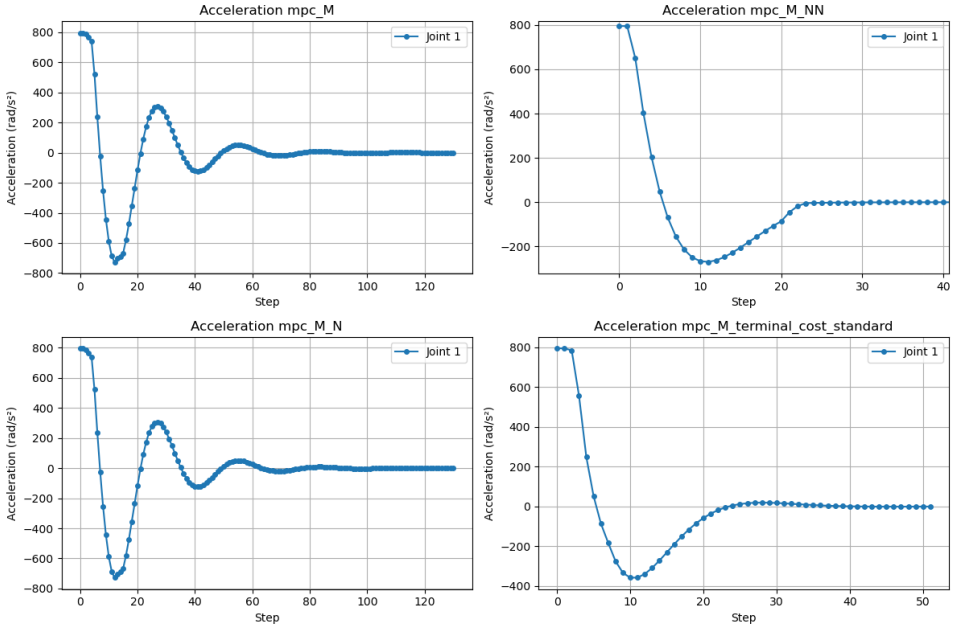

Figure 12: Enter Caption

5.4 Evaluation Single_Pendulum

Regarding the evaluation results of the single pendulum, the solutions are the same as those obtained for the double pendulum.

6 Conlusion and curiosity

In conclusion, the single pendulum can be considered a simplified instance of the double pendulum, leading to very similar outcomes in practice. The adopted approach combines the Direct Method with collocation, allowing both states and control inputs to vary at each iteration. Another fact that can help to compute the OCP steps, is that to implementing the Multy-process, in that way it's possible to decrease the time, and reach better performance. A promising direction for further enhancement involves a more extensive use of neural networks. A more in-depth study could help achieve the assignment's objective with shorter execution times and a less chaotic, more precises "dance."

The primary challenges encountered during this project were the neural network implementation using L4Casadi and the additional development required for the single-pendulum formulation, which prolonged overall progress. However, this aspect improve the hendling of the code, as the mathematical formulation was significantly simplified, allowing room for more complex implementations.

To verify this and out of personal curiosity, for exploratory purposes, a triple pendulum was also implemented by adding a third link and joint analogous to the second ($L_3 = 20$ cm). Unfortunately, due to time

constraints, the analysis was only partial, as the triple pendulum with the neural network is not yet fully functional. Further improvements and analysis for the triple pendulum can be found in the following repository: **https://github.com/RubenMalacarne/Assignment_3_orc**.

To verify the MPC behavior, a longer time horizon was ultimately employed. This improve better solution quality at the expense of higher computational effort. It is worth noting that the three different MPC configurations (M-step horizon alone, M+N-step horizon, and M-step horizon with terminal cost) produced broadly similar end results, even if with evident differences in the number of solver iterations and total computation times for each configuration. In any case, this implementation was very interesting and educational in that it gave the opportunity to test whether MPC even without NN works with a more complex system.

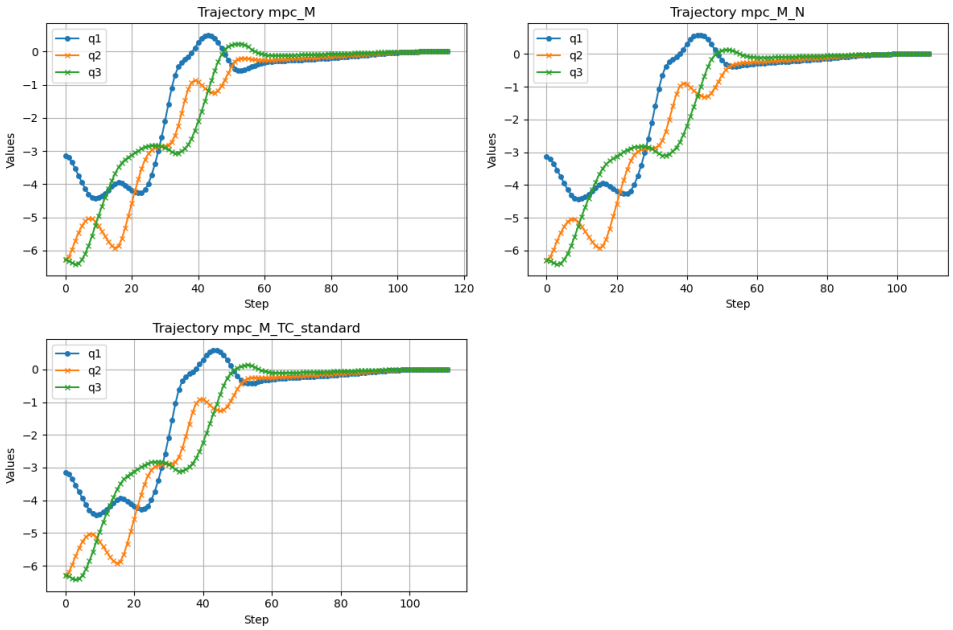

Figure 13: plot_all_trajectories

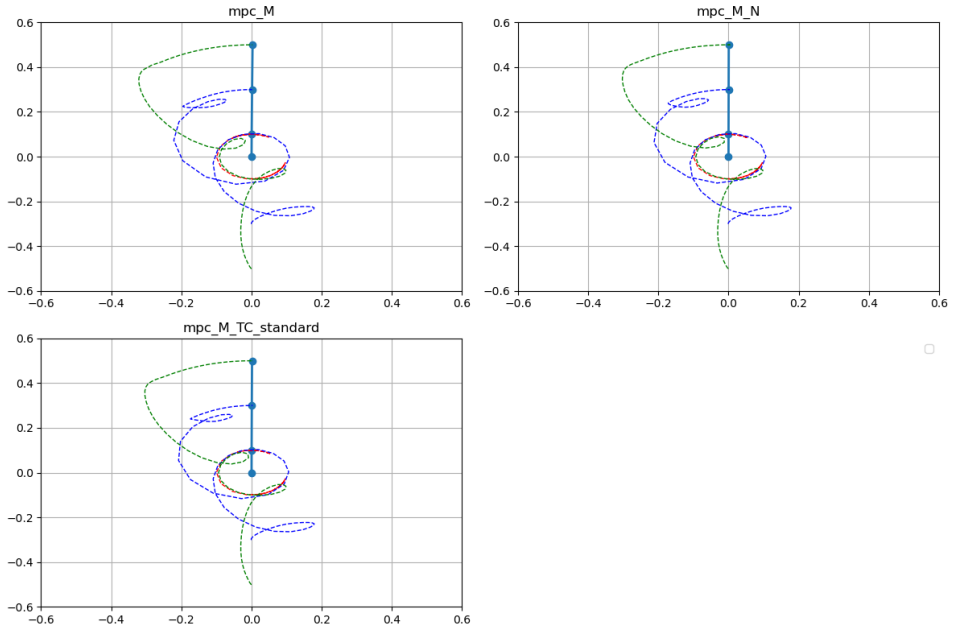

Figure 14: Movement of each pendulum

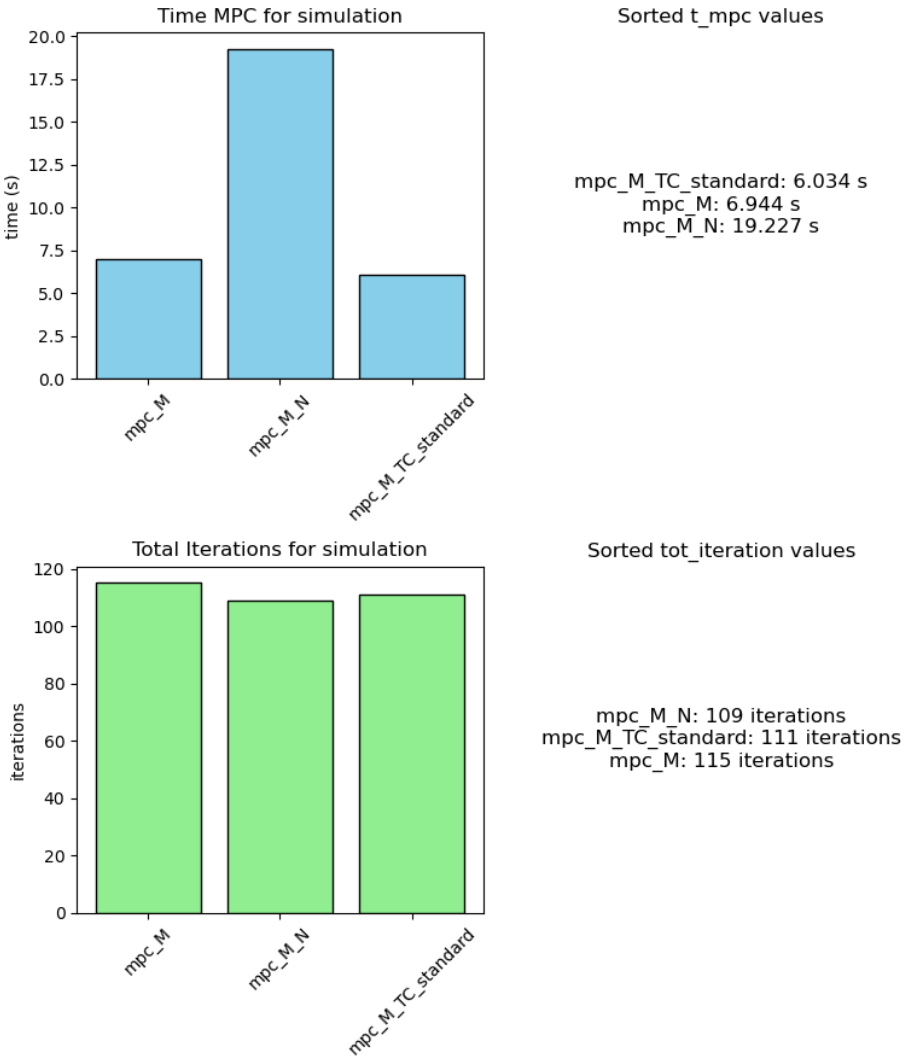

Figure 15: Time and Iteration

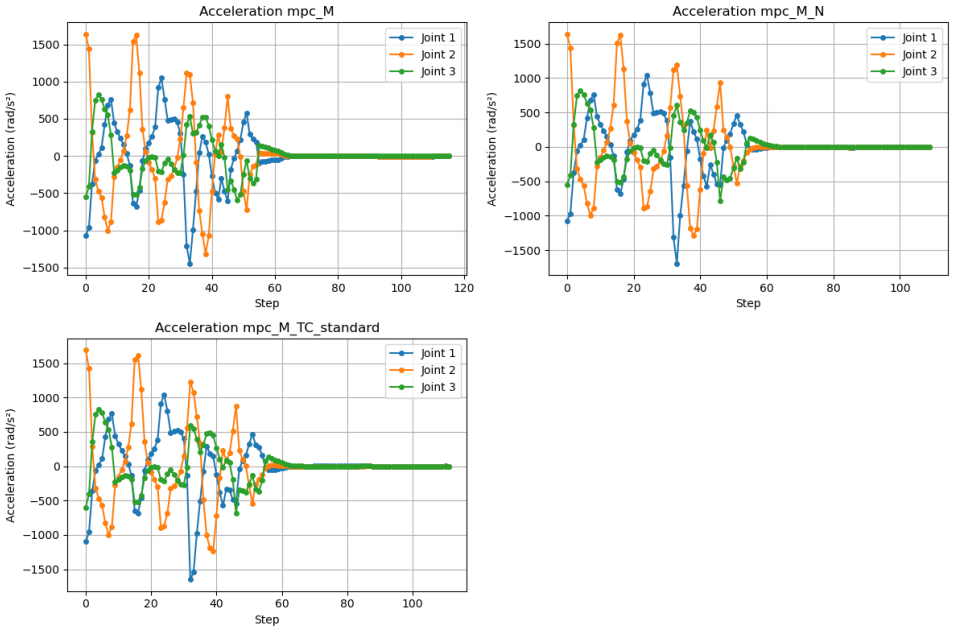

Figure 16: Joint accelerations