Advanced Robot Control Project Report

La Scala Giovanni Maria Francesco 241561
Di Lorenzo Andrea 239221

1 Introduction

Our final project, Project B, focuses on learning a terminal constraint set for a Model Predictive Control (MPC) formulation. The main goal is to figure out this terminal constraint set in order to make sure that the MPC we're working on stays feasible as we solve it repeatedly.

In this section we exploit the steps followed during the project:

1. Formulation the problem using CasaDi (a software library used for solving OCPs);
2. Training the Neural Network;
3. Formulation of the MPC;

The work was done at first on a single pendulum and then, once the algorithm developed has been validated, we moved the problem to a double pendulum.

2 Formulation of the problem using CasaDi

CasaDi is software framework for dynamic optimization and control in the field of computational mathematics. It provides a user-friendly environment for modeling and solving complex optimization problems, particularly within the context of optimal control and dynamic optimization applications.

As said, at first, we want to apply this a single pendulum whose dynamics is:

$$f(x,u) = \begin{cases} \dot{q} = v \\ \dot{v} = g \sin q + u \end{cases} \tag{1}$$

The problem we need to implement in our code is:

$$\begin{aligned} \text{minimize } & J = \sum_{i=0}^{N} w_v v_i^2 + w_u u_i^2 \\ \text{subject to: } & x_{k+1} = f(x_k, u_k) \quad \text{for } k = 0, \dots, N-1 \\ & u_k \in U \quad \text{for } k = 0, \dots, N-1 \\ & x_k \in X \quad \text{for } k = 0, \dots, N-1 \\ & x_0 = x_{\text{sample}} \\ & x_N = x_{N-1} \end{aligned} \tag{2}$$

This is not an important step but it helped us familiarize with CasaDi. Also, with this we recover the previous assignment, in fact we plot again the viability kernel.

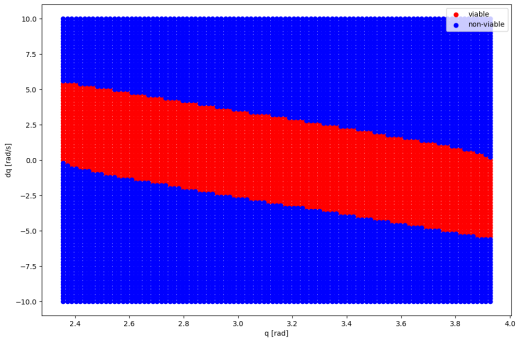

Figure 1: Viability Kernel

At this stage, we save the data collected as a DataFrame (a `.csv` file) which will make up our dataset for the following step. The datset contains the classifications of the states evaluated with the OCPs and are labled as viable (= 1) or non-viable (= 0).

3 Implementing the neural network

Now we have to train a neural network in order to classify a state either as viable or non viable[1].

In the file `neural_network.py` we parse the dataset, loaded from a `.csv` file, containing the states collected before and labeled as viable or non-viable states represented by arrays of features. After splitting the data into training (80% of dataset) and testing sets and after having normalized the features using `StandardScaler`, a neural network model is created.

The model consists of an input layer with dimensions corresponding to the feature space, followed by three hidden layers with 64, 32, and 16 neurons, each utilizing the **ReLU** activation function. The output layer, employing the `ReLU` activation function as well, consists of a single neuron to produce binary predictions.

[1] The definition of viable/non viable is to be addressed accordingly to our constraints

The script then trains the neural network model over 200 epochs using the **Adam optimizer** and **binary crossentropy loss**.

ADAM is a method for stochastic optimization, the algorithm is, computationally efficient, requires little memory, and is appropriate for situations with large data sets and parameters.

ADAM combines two stochastic gradient descent approaches, Adaptive Gradients, and Root Mean Square Propagation. Instead of using the entire data set to calculate the actual gradient, this optimization algorithm uses a randomly selected data subset to create a stochastic approximation.

A summary of the model is reported in `Table 1`

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 2)	0
dense (Dense)	(None, 64)	192
dense_1 (Dense)	(None, 32)	2080
dense_2 (Dense)	(None, 16)	528
dense_3 (Dense)	(None, 1)	17

Table 1: Model Summary

The goodness of the trained neural network and the resulting model is showed by the confusion matrix reported in `Figure2`. In fact, as we can see, there are many True Negatives and True Positives, and few instances of False Positives and False Negatives, highlighting a low incidence of classification errors. These results indicate a good reliability in the neural network's ability to effectively discriminate between the classes of interest.

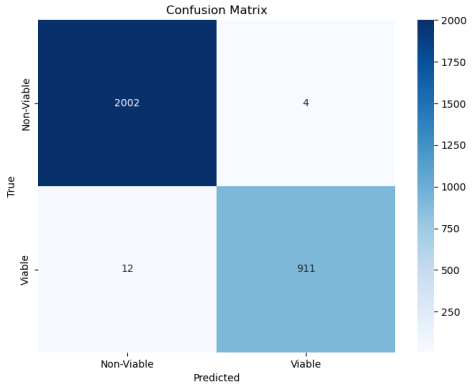

Figure 2: Confusion Matrix

Loss: 0.03285 **Accuracy**: 0.9916

4 Formulation of the MPC

As previously stated, the goal is to implement a Model Predictive Control algorithm to control the single pendulum (1).

At a theoretical level, MPC consists in solving an infinite horizon OCP using the current state as initial state. Though it is not possible to compute an infinite horizon OCP, there are some challenges we need to face.

In particular we'd want our algorithm to ensure:

- **Stability**: For a finite horizon can lead to amplification of modeling errors.
- **Feasibility**: In theory, with $N = \infty$ the problem is always feasible because predicted and actual trajectory are the same. In real life we face this problem because of the hard state constraints. To ensure a **closed loop convergence** we need to add a **terminal constraint** which leads to a reduction in the domain of feasibility;
- **Computation time**: To solve this problem we need to work on the number of time steps used to solve the OCP. To fast up the computation (and simulation) time we have used `multiprocess` library in python and run the code on multiple cores (4 in our case).

The problem we want to solve now looks like this:

$$
\begin{aligned}
\text{minimize } & J = \sum_{i=0}^{N} w_v v_i^2 + w_u u_i^2 + w_q (q_{\text{target}} - q_i)^2 \\
\text{subject to: } & x_{k+1} = f(x_k, u_k) \quad \text{for } k = 0, \dots, N-1 \\
& q_{lb} \leq q[i] \leq q_{ub} \quad \text{for } k = 0, \dots, N \\
& v_{lb} \leq v[i] \leq v_{ub} \quad \text{for } k = 0, \dots, N \\
& u_{lb} \leq u[i] \leq u_{ub} \quad \text{for } k = 0, \dots, N \\
& x_0 = x_{\text{init}} \\
& g_{\text{terminal}}(x_N) \geq 0
\end{aligned}
\tag{3}
$$

Where $f(x_k, u_k)$ is the dynamics of the single pendulum reported in (1), q_{lb} and q_{ub} are respectively the **position lower bound** and **position upper bound**; v_{lb} and v_{ub} are the **velocity lower bound** and **velocity upper bound**; u_{lb} and u_{ub} are instead the **control lower bound** and **control upper bound**. Those are all defined in the `mpc_single_pendulum_conf.py`.

The coefficients in the cost function (w_q, w_v and w_u) tell us the penalty we are giving for velocity, control and position. We chose values for them in order to penalize very little velocity and torques (control actions) while we want to push the system really close to the position limits.

4.1 Recursive feasibility

To ensure recursive feasibility, we have implemented a terminal constraint based on the neural network built previously which takes as input the normalized position and velocity at the last step of the prediction. The function `nn_to_casadi` evaluates the neural network based on the current state and constraints the output to be > 0.

In simpler terms, consider this as a safety measure in our system. A neural network output exceeding 0 implies a secure condition for proceeding. Conversely, an output of 0 or less signals a potential issue, triggering precautionary measures.

In the table below are the configurations used for Model Predictive Control (MPC). This includes the Time Horizon, weights for position, velocity, and torque, as well as the initial state and target state.

Time horizon	1
OCP time step	0.01
MCP step	1000
Weight in position w_q	1e2
Weight in velocity w_v	1e-1
Weight in torque w_u	1e-4
Initial State	$[\pi, 0]$
Target State	$5/4 \cdot \pi$

Table 2: Configuration_plot_**Target 1**

As we may see in `Figure3` if the terminal constraint is active , the algorithm manages to find a solution for every step in the horizon and successfully reaches the target position satisfying all the constraints.

If instead the terminal constraint is not active `Figure4`, at a certain point (around step 40) the problem becomes unfeasible meaning that a solution cannot be found unless we violate the constraints on the path.

In Figures `Figure5`, `Figure6`, and `Figure7` display the graphs related to position, velocity, and torque for this test. Moreover, an animation of this simulation is available in the GitHub repository.

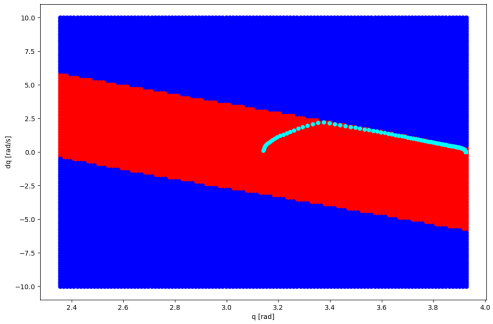

Figure 3: MPC with TC

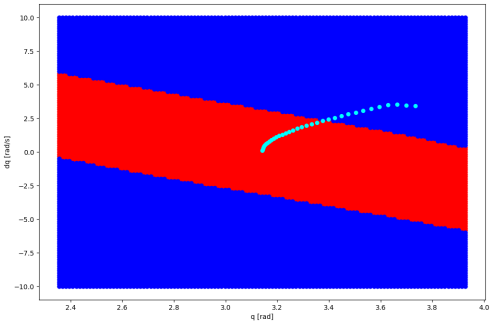

Figure 4: MPC without TC

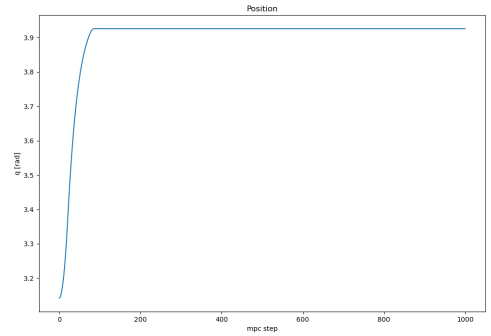

Figure 5: Position

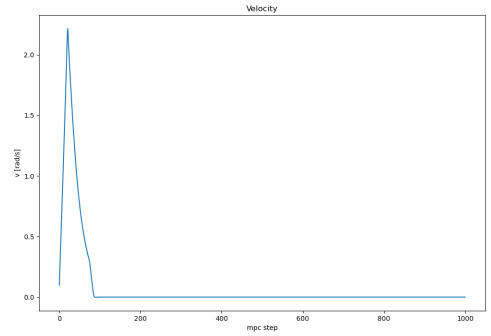

Figure 6: Velocity

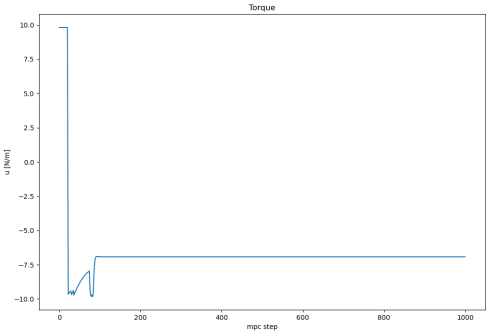

Figure 7: Torque

We conducted two additional tests. In the first one, we started from a constrained condition, with the goal of reaching a position limit. In the second test, we introduced noise to the state.

Time horizon	1
OCP time step	0.01
MCP step	1000
Weight in position w_q	1e2
Weight in velocity w_v	1e-1
Weight in torque w_u	1e-4
Initial State	$[3/4 \cdot \pi, 0]$
Target State	$5/4 \cdot \pi$

Table 3: Configuration_plot_**Target 2**

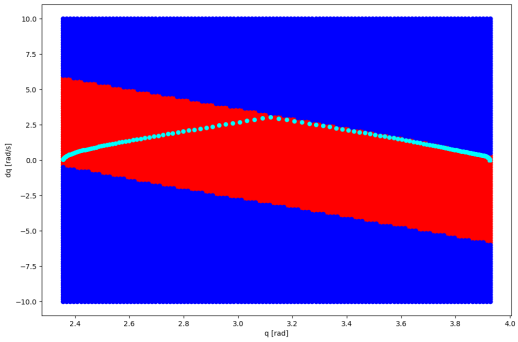

Figure 8: MPC with TC

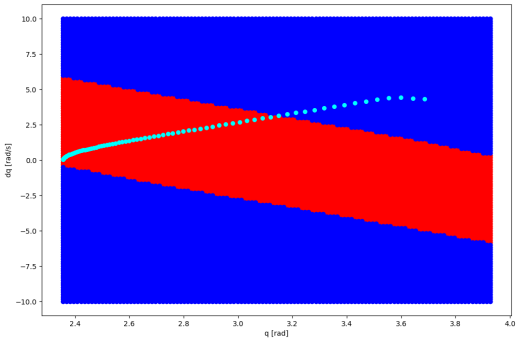

Figure 9: MPC without TC

As previously mentioned, in `Figure8`, with the terminal constraint active, the algorithm manages to find a solution for each step in the horizon and successfully reaches the target position while satisfying all constraints.

However, if the terminal constraint is not active, as shown in `Figure9`, at a certain point (around step 40), the problem becomes infeasible. This implies that a solution cannot be found unless constraints on the path are violated.

Here instead, noise was introduced to the state with mean and standard deviation values as indicated in `Table 4`.

Time horizon	1
OCP time step	0.01
MCP step	1000
Weight in position w_q	1e2
Weight in velocity w_v	1e-1
Weight in torque w_u	1e-4
Initial State	$[7/8 \cdot \pi, 0]$
Target State	π
Noise Mean	0.01
Noise Standard Deviation	0.01

Table 4: Configuration_plot_**Target 3**

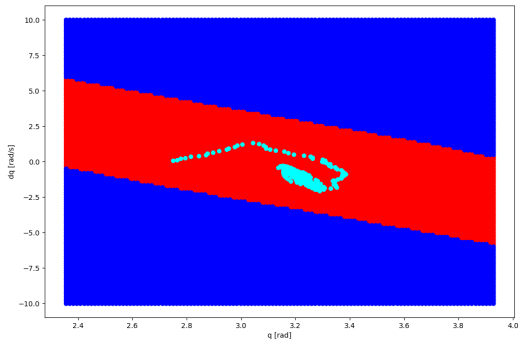

Figure 10: MPC with noise

In the presence of noise, the controller was able to reach the target state, provided that the simulation started from a safer initial condition and that the initial error was substantial. This suggests that, despite the presence of disturbances, acceptable performance can be achieved if the simulation begins from a favorable initial state and if the initial error is effectively managed.

It is redundant to mention that in the absence of a terminal constraint, the controller would be unable to reach the desired state.

Through these tests, it is evident that the concept of recursive feasibility holds consistently. In other words, the system's ability to generate consistent and feasible solutions recursively is consistently confirmed across various scenarios and conditions.

Finally, for each of these tests, there are available plots depicting position, velocity, and torque, along with animations of the simulations. You can access these visualizations directly on the associated GitHub repository.

5 Double Pendulum

5.1 Problem formulation

For what concerns the double pendulum, the implementation steps we took are basically the same. We just need to modify the codes accordingly to the new dynamics and based on the fact t hat the states we are interested in analyzing is now a 4-Dimensional states ($x \in \mathbb{R}^4$) while the control input vecotr is 2-Dimensional ($u \in \mathbb{R}^2$):

$$x = \begin{bmatrix} q_1 & v_1 & q_2 & v_2 \end{bmatrix} \qquad u = \begin{bmatrix} u_1 & u_2 \end{bmatrix}$$

We start again by generating the dataset labeling the states as viable or non viable[2] by solving a series of OCPs for the double pendulum whose dynamics $f(x, u)$ is reported in the file `double_pendulum_dynamics`.

The Optimal Control Problems to solve have the following formulation:

Minimize:

$$J = \sum_{i=0}^{N} \left(w_{v_1} v_1[i]^2 + w_{v_2} v_2[i]^2 + w_{u_1} u_1[i]^2 + w_{u_2} u_2[i]^2 \right)$$

Initial state constraints:

$$q_1[i+1] = f(q_1[i], v_1[i], q_2[i], v_2[i], u_1[i], u_2[i])$$
$$v_1[i+1] = f(q_1[i], v_1[i], q_2[i], v_2[i], u_1[i], u_2[i])$$
$$q_2[i+1] = f(q_1[i], v_1[i], q_2[i], v_2[i], u_1[i], u_2[i])$$
$$v_2[i+1] = f(q_1[i], v_1[i], q_2[i], v_2[i], u_1[i], u_2[i])$$

Initial state constraints:

$$q_1[0] = q_{1,\text{init}}, \quad v_1[0] = v_{1,\text{init}}$$
$$q_2[0] = q_{2,\text{init}}, \quad v_2[0] = v_{2,\text{init}}$$

State constraints:

$$q_{1,\min} \leq q_1[i] \leq q_{1,\max}, \quad q_{2,\min} \leq q_2[i] \leq q_{2,\max}$$
$$v_{1,\min} \leq v_1[i] \leq v_{1,\max}, \quad v_{2,\min} \leq v_2[i] \leq v_{2,\max}$$

Control constraints:

$$u_{1,\min} \leq u_1[i] \leq u_{1,\max}, \quad u_{2,\min} \leq u_2[i] \leq u_{2,\max} \tag{4}$$

It is worth mentioning that the computation time increases exponentially with the number of states.

The challenge was to compute the dataset for the new system. To minimize the computation time we have built a grid where the state array is represented by each joint position and velocities, having fixed q_2, v_2 and increasing the number of samples for q_1, v_1 gradually until we got to `n_pos_q1` $= 11$ and `n_vel_v1` $= 11$. Here we had to fix one of the two to 21 and increase again the other and then repeat the procedure switching the values[3]. Also at every run we would discard from the simulation all the points that have been evaluated at the previous run in order to avoid computations that have been already done.

5.2 Neural Network

For what concerns the neural network, the reasonment at the basis is the same, we just need to put the input layer dimension to 4.

Again we built the neural network with three hidden layers while the output dimension is 1 since we want to perform a boolean classification of the states stored in the DataFrame file (`data_double.csv`).

In the following we evaluate the goodness of the model obtained via the confusion matrix.

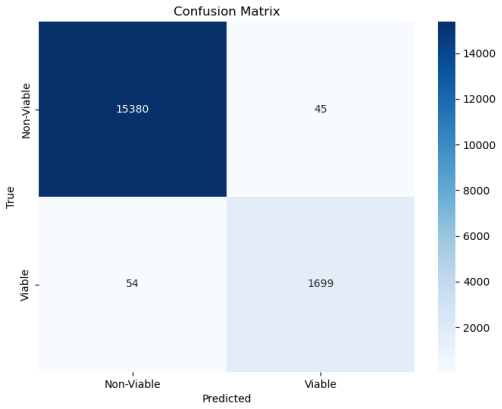

Figure 11: Confusion Matrix

The idea under the implementation of the neural network is that, due to the fact we had a very coarse and limited dataset we could use 95% of the dataset to train the neural network and we have also incremented the number of epochs to 300.

A summary of the model is reported in `Table 5`

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 4)	0
dense (Dense)	(None, 128)	640
dense_1 (Dense)	(None, 64)	8256
dense_2 (Dense)	(None, 32)	2080
dense_3 (Dense)	(None, 1)	33

Table 5: Model Summary

[2] viable_states $= 1$, no_viable_states $= 0$

[3] Because of time we could only do this fixing `n_vel_v1` $= 21$ and `n_pos_q1` $= [6, 11]$ and for `n_vel_v1` $= 6$ having fixed `n_pos_q1` $= 21$. The last simulation would have been a grid sampled uniformly.

5.3 Recursive Feasibility

To guarantee recursive feasibility, we have incorporated a terminal constraint leveraging the previously constructed neural network.

This constraint takes the normalized position and velocity at the last step of the prediction as input. The function `nn_to_casadi` assesses the neural network based on the current state and enforces the output to be > 0.

In simpler terms, view this as a safety precaution within our system. A neural network output exceeding 0 indicates a secure condition for progression, while an output of 0 or less signals a potential issue, prompting precautionary measures.

The configurations utilized for Model Predictive Control (MPC) are outlined in the table below. This encompasses the Time Horizon, weights assigned to position, velocity, and torque, as well as the initial state and target state.

Time horizon	1
OCP time step	0.01
MCP step	200
Weight in position w_q	1e2
Weight in velocity w_v	1e-1
Weight in torque w_u	1e-4
Initial State	$[3/4 \cdot \pi,\ 0,\ 3/4 \cdot \pi,\ 5]$
Target State q1 & q2	$5/4 \cdot \pi, 3/4 \cdot \pi$

Table 6: Configuration_plot_**Target 1**

The weights we have assigned to the position, velocity and torques are the same for both joints for the seek of simplicity and the same can be said about the constraints of the second joint. We assumed that it could move in $\left[\frac{3}{4}\pi, \frac{5}{4}\pi\right]$, with velocity bounded between $[-10, 10]$ and the control action to stay in $[-g, g]$.

In the detailed depiction provided in `Figure12`, it is evident that when the terminal constraint is active, the algorithm successfully determines a solution for each step within the temporal horizon. This leads to the successful achievement of the target position while adhering to all constraints.

Positions of link 2 are represented in red, while positions of link 1 are indicated in blue, clearly demonstrating accurate goal attainment.

Conversely, if the terminal constraint is not active, the problem becomes infeasible. This implies that a solution cannot be found unless constraints along the path are violated.

Figures `Figure13`, `Figure14`, and `Figure15` present graphs related to position, velocity, and torque for this specific test. For further dynamic insights, an animation of this simulation is available in the GitHub repository.

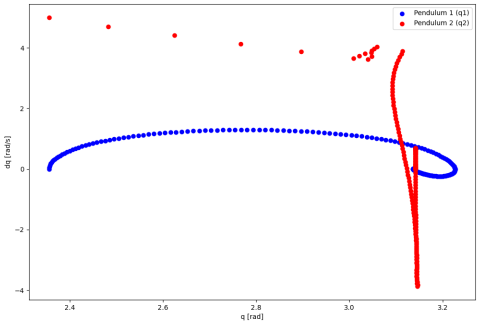

Figure 12: MPC with TC

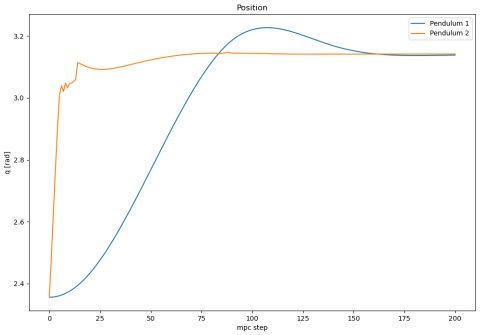

Figure 13: Position

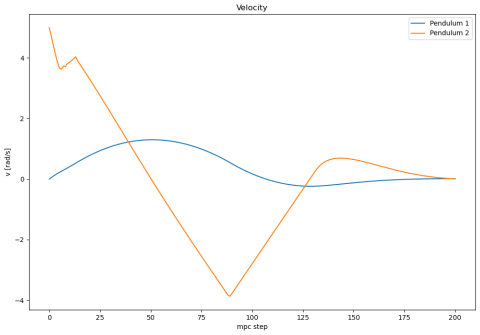

Figure 14: Velocity

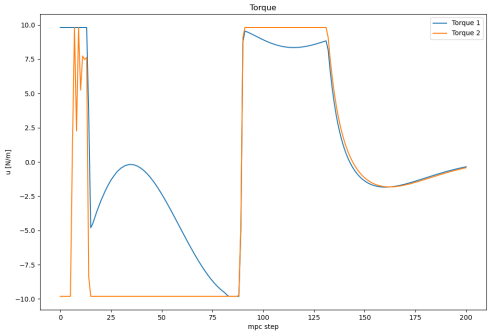

Figure 15: Torque

We conducted additional tests. In the first one, we started from a "safe" initial state, demonstrating how we are able to reach the target even without the use of the terminal constraint.

Time horizon	1
OCP time step	0.01
MCP step	200
Weight in position w_q	1e2
Weight in velocity w_v	1e-1
Weight in torque w_u	1e-4
Initial State	$[\pi, 0, \pi, 0]$
Target State	$3/4 \cdot \pi, 5/4 \cdot \pi$

Table 7: Configuration_plot_**Target 2**

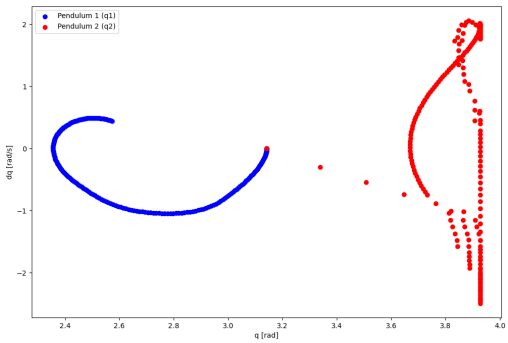

Figure 16: MPC with TC

Finally, for each of these tests, there are available plots depicting position, velocity, and torque, along with animations of the simulations. You can access these visualizations directly on the associated GitHub repository.

5.4 Conclusion

In the evolution of the Model Predictive Controller (MPC) from a single to a double pendulum, incorporating a neural network for model simulation and fine-tuning various parameters, we conducted a series of tests. At this stage, we encountered some challenges in demonstrating recursive feasibility, as we were able to reach all desired points even without the use of the terminal constraint.

Possible Explanations:

- **Adequate Initial Constraints**: The validity of the initial constraints may render the further reinforcement through the terminal constraint unnecessary. Despite efforts to explore "boundary" zones, the system's complexity, stemming from the high number of variables, made it challenging to identify points that were not excessively "safe."
- **MPC Parameter Configuration**: The specific configuration of MPC parameters, such as the time horizon and weights assigned to the neural network, can influence the role of the terminal constraint. Optimization attempts, such as reducing the time horizon and increasing the weight of the network, did not yield significant impacts. However, excessively small time horizons did not lead to convergence.
- **Training Data Quantity**: Despite the high accuracy of the neural network (99.5%), there might be room for more comprehensive training with a broader variety of data. This could enable capturing the intrinsic complexity of the system more fully.