SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF INGEGNERIA INFORMATICA, AUTOMATICA E GESTIONALE ANTONIO RUPERTI

UNDERACTUATED ROBOTS FINAL PROJECT

Professors:
Leonardo Lanari
Giuseppe Oriolo

Tutor:
Giulio Turrisi

Students:
Fulvio Sanguigni
Saverio Borrelli
Monica De Pucchio

Academic Year 2021/2022

Contents

1 Introduction

In this work, we aim to investigate and further extend the results obtained from the authors [1].

In the last decade, many techniques have been explored to enhance autonomous robot navigation, but in the last years, there is a profound interest in more advanced movements, with the final goal of achieving human-like performance.

In this case, we focus on the reproduction of cat-like movements, such as the ability to fall on their legs from different heights.

This particular task is of great interest for the domain of legged robots, because it allows preparing our robot to face difficult scenarios, such as the exploration of an indoor environment after an earthquake, or walking over highly complex terrain with huge steps.

There are previous works that explore the re-orientation task of a falling robot. Some of them made use of a tail to accelerate the process ([2], [3]), others adopt Reinforcement Learning ([4]). All these approaches had the assumption of zero-gravity plus the assumption of "smooth" dynamics.

In our case, we have to operate in the presence of gravity, so we have to perform rapid movements to assess the robot's position, thus losing the assumption of "smooth" dynamics of the system.

In the next sections, we will dive into the detail of our solution. Section 2 is about the modelling of the robot, where we describe its parameters, we define its dimensions and we develop a Lagrangian approach to model it. Section 3 refers to the Model Predictive Control approach we used to perform a re-orientation task for our robot. Section 4 is focused on the alternative solution for our problem, based on Neural Networks.

2 Modelling

The model proposed in [1] is used to build our mini cheetah.

The dynamics model in free-fall of the robot characterized in Figure 1 appears as:

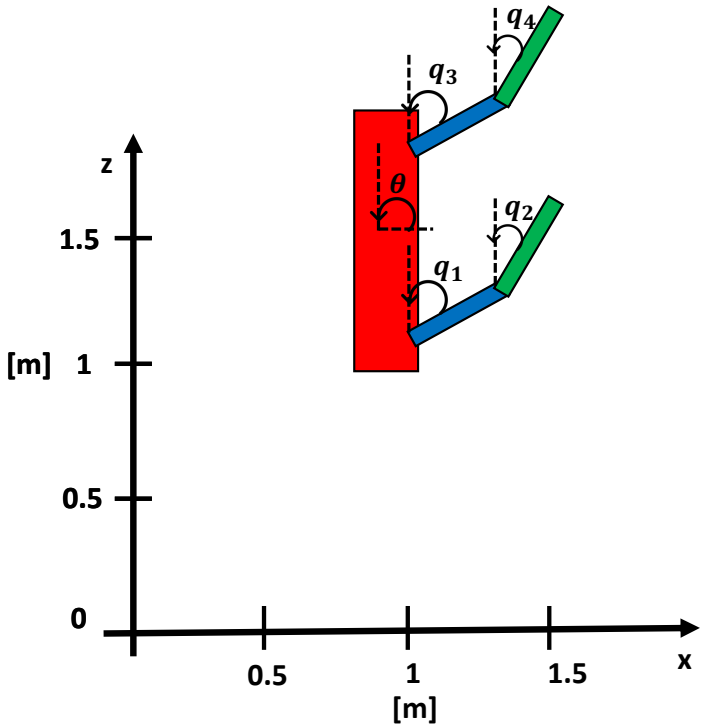


Figure 1: Free-fall Mini Cheetah

$$M \begin{bmatrix} \ddot{p} \\ \ddot{\theta} \\ \ddot{q} \end{bmatrix} + C \begin{bmatrix} \dot{p} \\ \dot{\theta} \\ \dot{q} \end{bmatrix} + g = \begin{bmatrix} 0 \\ 0 \\ \tau \end{bmatrix} \tag{1}$$

where $p \in \Re$ is the position of the center-of-mass, $\theta \in \Re$ is the orientation (pitch) of the body frame, $q \in \Re^4$ are joint angles and $\tau \in \Re^4$ are applied joint torques (control inputs). M is the positive-definite inertia matrix, C is the Coriolis matrix and g characterize the influence of gravity. This model is built considering an external reference frame. Since the position p is not important for the reorientation task in free-fall, we ignore it in most of our computations and consider the robot's state to be given by the 10 state vector

$$x = [\theta, \dot{\theta}, q^T, \dot{q}^T]^T \tag{2}$$

The input vector is equal to zero in the coordinates θ and $\dot{\theta}$, that describe the angular position ad the angular velocity of the cat's body, while it is equal to τ, the torque produced by rotors, in the coordinate that describe the robot' legs angular position and angular velocity.

$$u = [0, \tau] \tag{3}$$

with $\tau \in \Re^4$.

At this point, M, inertia matrix, C, Coriolis term, and g are unknowns.

Starting from g therm, we have considered it as the gradient of system total potential energy

$$\begin{aligned} g &= \frac{\partial U_{tot}}{\partial x} \\ U_{tot} &= U_{Body} + \sum_{i=1}^{4} U_{leg,i} \end{aligned} \tag{4}$$

While the potential energy of the body is constant and, therefore, it g contribution is null, the potential energy terms linked with legs are

$$\begin{cases} U_{leg,1} = m_1 g(h_b - \frac{l_1}{2}\sin(q1)) \\ U_{leg,2} = m_2 g(h_b - l_1\sin(q1) - \frac{l_2}{2}\sin(q2)) \\ U_{leg,3} = m_3 g(h_b - l_3\sin(q3) \\ U_{leg,4} = m_4 g(h_b - l_3\sin(q3) - \frac{l_4}{2}\sin(q4)) \end{cases} \tag{5}$$

where h_b is the body height, m_i are the leg masses and l_i are the legs length. To obtain the inertia matrix we have developed, instead, the robot kinematic energy T as

$$T_{tot} = T_{Body} + \sum_{i=1}^{4} T_{leg,i} \tag{6}$$

Considering the centre of mass position of a single joint, CoM_i for $i = 1, 2, 3, 4$, depending on the state variable, the kinematic energy for a single joint is

$$\begin{cases} T_{leg,1} = \frac{1}{2}I_1(\dot{q}_1 + \dot{\theta})^2 + \frac{1}{2}m_1(\frac{\partial CoM_1}{\partial t})^2 \\ T_{leg,2} = \frac{1}{2}I_2(\dot{q}_1 + \dot{q}_2 + \dot{\theta})^2 + \frac{1}{2}m_2(\frac{\partial CoM_2}{\partial t})^2 \\ T_{leg,3} = \frac{1}{2}I_3(\dot{q}_3 + \dot{\theta})^2 + \frac{1}{2}m_3(\frac{\partial CoM_3}{\partial t})^2 \\ T_{leg,4} = \frac{1}{2}I_4(\dot{q}_3 + \dot{q}_4 + \dot{\theta})^2 + \frac{1}{2}m_4(\frac{\partial CoM_4}{\partial t})^2 \end{cases} \tag{7}$$

where I_i, for $i = 1, 2, 3, 4$, is the joint inertia associated with its own centre of mass. The body inertia is $T_{Body} = \frac{1}{2}I_{Body}(\dot{\theta})^2$. Once we have gained the total kinematic energy of the body, we have inverted $T_{tot} = \dot{x}^T M(x)\dot{x}$ obtaining the inertia matrix $M(x)$. From that matrix, looking at the $k-$column, we can obtain Coriolis terms as

$$\begin{aligned} s_k(x) &= \frac{1}{2}(\frac{\partial M_k}{\partial x} + \frac{\partial M_k}{\partial x}^T - \frac{\partial M}{\partial x_k}) \\ C_{k,j}(x, \dot{x}) &= \sum_i (s_{k,i,j}(x))\dot{x} \end{aligned} \tag{8}$$

2.1 Parameters' values

These are the values that we chase for the parameters.
They are real values for the mini cheetah robot.

Height from the ground:	$h_b = 1.5\ [m]$
Gravity force:	$g = 9.81\ [m/s^2]$
Dimension of central body:	$h = 0.1$ and $b = 0.4\ [m]$
Mass of the central body:	$m_b = 0.5\ [kg]$
Mass of the joints:	$m = 0.05\ [kg]$
Joints' inertia:	$I = m_1 * (l^2)/12\ [kg * m^2]$

3 PD Controller

The first kind of controller applied is a PD. To develop it, we have done some simplification in the robot modeling, as suggested in the [1]. Since we are interested in the momentum balance to control our falling Mini-Cheetah, we have considered the robot reference frame. It allows us to consider the Coriolis term C null and to reduce the inertia term $M(x)$ and the gravity term $g(x)$. We have, hence, defined the position reference for x as:

$$\begin{cases} x_{1,des} = 0, \quad \dot{x}_{1,des} = 0, \quad \ddot{x}_{1,des} = 0 \\ x_{3,des} = \frac{3\pi}{4}, \quad \dot{x}_{3,des} = 0, \quad \ddot{x}_{3,des} = 0 \\ x_{5,des} = \frac{\pi}{4}, \quad \dot{x}_{5,des} = 0, \quad \ddot{x}_{5,des} = 0 \\ x_{7,des} = \frac{\pi}{4}, \quad \dot{x}_{7,des} = 0, \quad \ddot{x}_{7,des} = 0 \\ x_{9,des} = \frac{\pi}{4}, \quad \dot{x}_{9,des} = 0, \quad \ddot{x}_{9,des} = 0 \end{cases} \tag{9}$$

Applying an input/output linearization, with

$$\begin{cases} v1 = \ddot{x}_{1,des} + k_{p,1}(x_1 - x_{1,des}) + k_{d,1}(\dot{x}_1 - \dot{x}_{1,des}) \\ v2 = \ddot{x}_{2,des} + k_{p,2}(x_2 - x_{2,des}) + k_{d,2}(\dot{x}_2 - \dot{x}_{2,des}) \\ v3 = \ddot{x}_{3,des} + k_{p,3}(x_3 - x_{3,des}) + k_{d,3}(\dot{x}_3 - \dot{x}_{3,des}) \\ v4 = \ddot{x}_{4,des} + k_{p,4}(x_4 - x_{4,des}) + k_{d,4}(\dot{x}_4 - \dot{x}_{4,des}) \\ v5 = \ddot{x}_{5,des} + k_{p,5}(x_5 - x_{5,des}) + k_{d,5}(\dot{x}_5 - \dot{x}_{5,des}) \end{cases} \tag{10}$$

we have obtained

$$\begin{aligned} \begin{bmatrix} \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \end{bmatrix} = \begin{bmatrix} M_{2,1} & M_{2,2} & M_{2,3} & M_{2,4} & M_{2,5} \\ M_{3,1} & M_{3,2} & M_{3,3} & M_{3,4} & M_{3,5} \\ M_{4,1} & M_{4,2} & M_{4,3} & M_{4,4} & M_{4,5} \\ M_{5,1} & M_{5,2} & M_{5,3} & M_{5,4} & M_{5,5} \end{bmatrix} \begin{bmatrix} M_{1,1} \\ M_{1,2} \\ M_{1,3} \\ M_{1,4} \\ M_{1,5} \end{bmatrix}^{-1} * M_{1,1} * v_1 + c_1 + g_1 + \\ + \begin{bmatrix} M_{2,1} & M_{2,2} & M_{2,3} & M_{2,4} \\ M_{3,1} & M_{3,2} & M_{3,3} & M_{3,4} \\ M_{4,1} & M_{4,2} & M_{4,3} & M_{4,4} \\ M_{5,1} & M_{5,2} & M_{5,3} & M_{5,4} \end{bmatrix} \begin{bmatrix} v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} + \begin{bmatrix} c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} + \begin{bmatrix} g_2 \\ g_3 \\ g_4 \\ g_5 \end{bmatrix} \end{aligned} \tag{11}$$

The input matrix was $u = [0 \ \tau_2 \ \tau_3 \ \tau_4 \ \tau_5]^T$, represented in Figure 2 from which we can compute

$$\begin{bmatrix} \ddot{\theta} \\ \ddot{q}_1 \\ \ddot{q}_2 \\ \ddot{q}_3 \\ \ddot{q}_4 \end{bmatrix} = M(x)^{-1}(u - g(x) - c(x)). \tag{12}$$

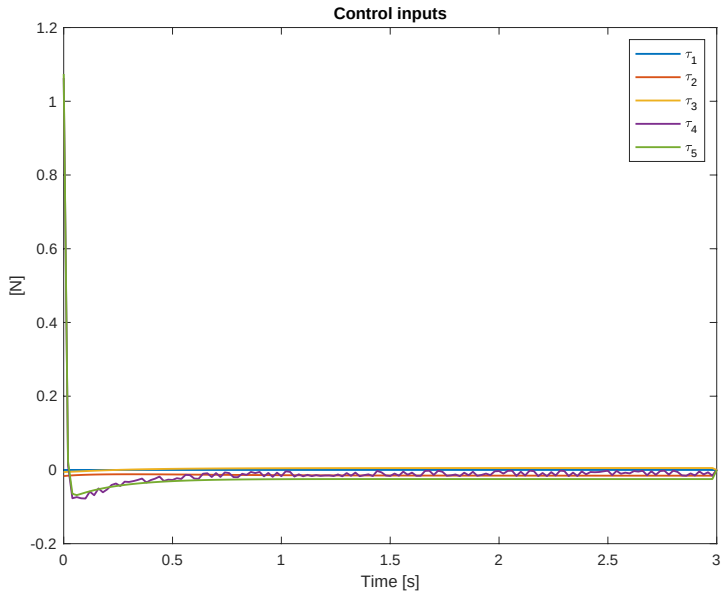


Figure 2: Control input behaviours

Integrating it with the Euler method and discretization step $T_s = 0.1$, we obtain a convergent and slow θ and $\dot{\theta}$, as you can see in Figure 3. The joints behaviour is shown in Figure 4

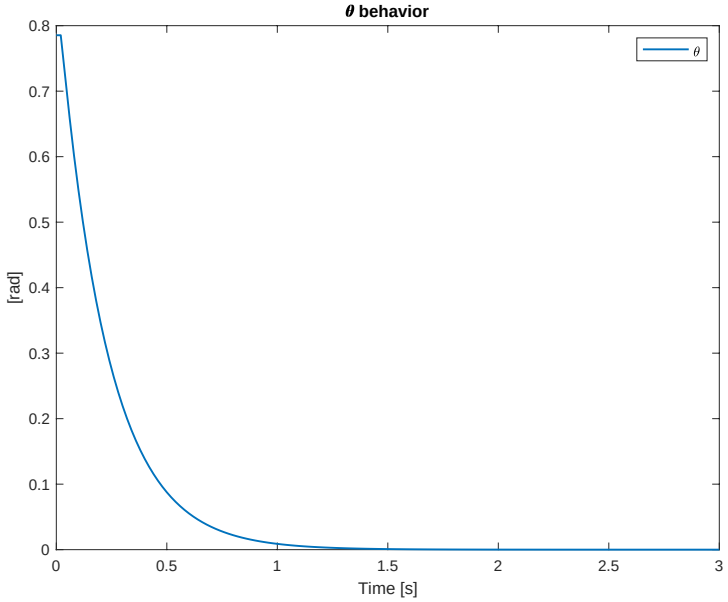


Figure 3: θ numerically integrated with respect to the time

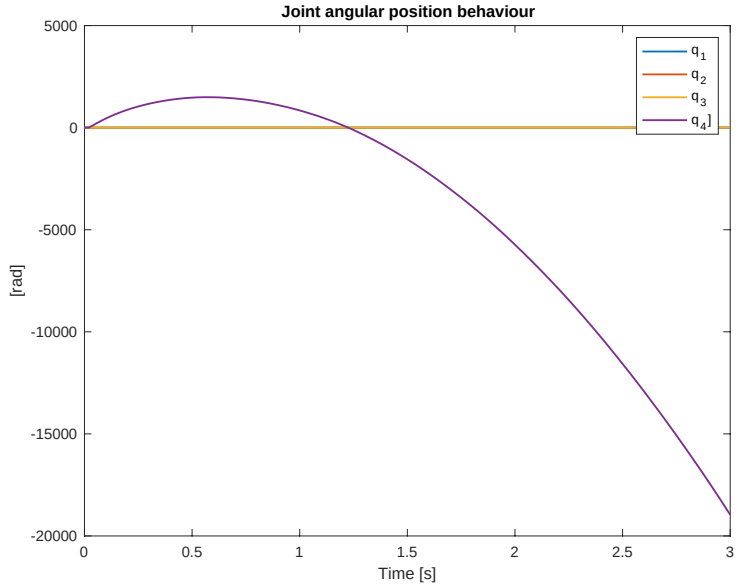


Figure 4: Joint angular position numerically integrated with respect to the time

4 MPC

4.1 Introduction on MPC

A model predictive controller uses linear plant, disturbance, and noise models to estimate the controller state and predict future plant outputs.
Using the predicted plant outputs, the controller solves a quadratic programming optimization problem to determine control moves.
In our case, considering that we have a nonlinear model, we used the nonlinear MPC.
As the traditional linear MPC, it calculates control actions at each control interval, using a combination of model-based prediction and constrained optimization.
The differences are that:
- The prediction model can be nonlinear and include time-varying parameters.
- The equality and inequality constraints can be nonlinear.
- The scalar cost function to be minimized can be a nonquadratic (linear or nonlinear) function of the decision variables.

4.2 Our implementation of nonlinear MPC

For implementing it in Matlab, we used the function: *nlmpcmove* that computes optimal control action for a nonlinear MPC controller.
At first we initialize the MPC:

```
nx = 10;     % number of states
ny = 5;      % number of outputs (in our case 4 joint angels
             % + theta)
nu = 5;      % number of inputs that correspon to the number of
             %manipulated variables
nlobj = nlmpc(nx,ny,nu);
```

And after that, we specify the sampling time, the prediction horizon and the control horizon.

```
Ts = 0.005; % sampling time
nlobj.Ts = Ts;
nlobj.PredictionHorizon = 20;
nlobj.ControlHorizon = 10;
```

We set the initial configuration of the robot.

```
x0 = [pi/2 0 pi/4 0 pi/4 0 pi/4 0 pi/4 0]';
u0 = zeros(5,1);
validateFcns(nlobj, x0, u0, [], {Ts});
```

So it has the central body at 90 degrees and all the other angles at 45 degrees.
And then we add the constraints, on the input variables and the minimum and maximum torque

```
% Constraint on input variation
nlobj.Weights.ManipulatedVariablesRate = ones(1,5)*0.1;

nlobj.MV(2).Min = 0;      % Minimum torque given [Nm]
nlobj.MV(3).Min = 0;
nlobj.MV(4).Min = 0;
nlobj.MV(5).Min = 0;
nlobj.MV(2).Max = 6;       % Maximum torque given [Nm]
nlobj.MV(3).Max = 6;
nlobj.MV(4).Max = 6;
nlobj.MV(5).Max = 6;
```

We specify the output reference value (final configuration of the robot)

```
yref = [0 pi/2 pi/2 pi/2 pi/2];
```

So it has to land with the body at 0 degrees all the other angles at 90 degrees.
We create the nonlinear MPC object, specifying the sampling time, and we specify also the options of the solver

```
nloptions = nlmpcmoveopt;
nloptions.Parameters = {Ts};
% Solver Constraints
nlobj.Optimization.SolverOptions.Algorithm = 'sqp';
nlobj.Optimization.SolverOptions.MaxIterations = 1300;
nlobj.Optimization.SolverOptions.MaxFunctionEvaluations = 1500;
nlobj.Optimization.SolverOptions.FiniteDifferenceType = 'central';
nlobj.Optimization.SolverOptions.OptimalityTolerance = 1e-3;
```

And this

```
Duration =3;
xHistory = x;
tic
for ct = 1:(Duration/Ts)
    % Compute optimal control moves
    [mv,nloptions] = nlmpcmove(nlobj,x,mv,yref,[],nloptions);
    % Implement first optimal control move
    x=x+Ts*model01(x, mv);
    % Generate sensor data
    y = x([1 3 5 7 9]);
    % Save plant states
    xHistory = [xHistory x];
```

```
13 end
14 toc
```

4.3 Results

These are the results that we obtain with the MPC controller.
We can see the behaviours of the θ angle, and the other angles (q_1, q_2, q_3, q_4).

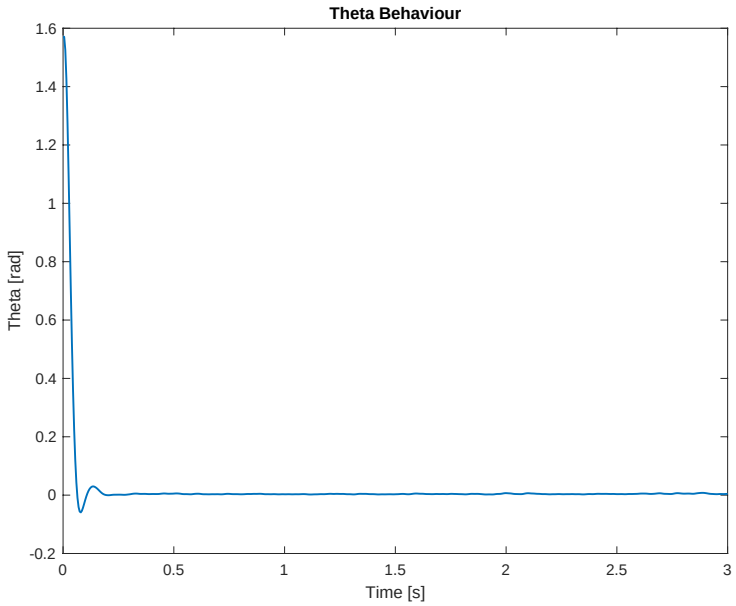


Figure 5: θ behaviour with MPC

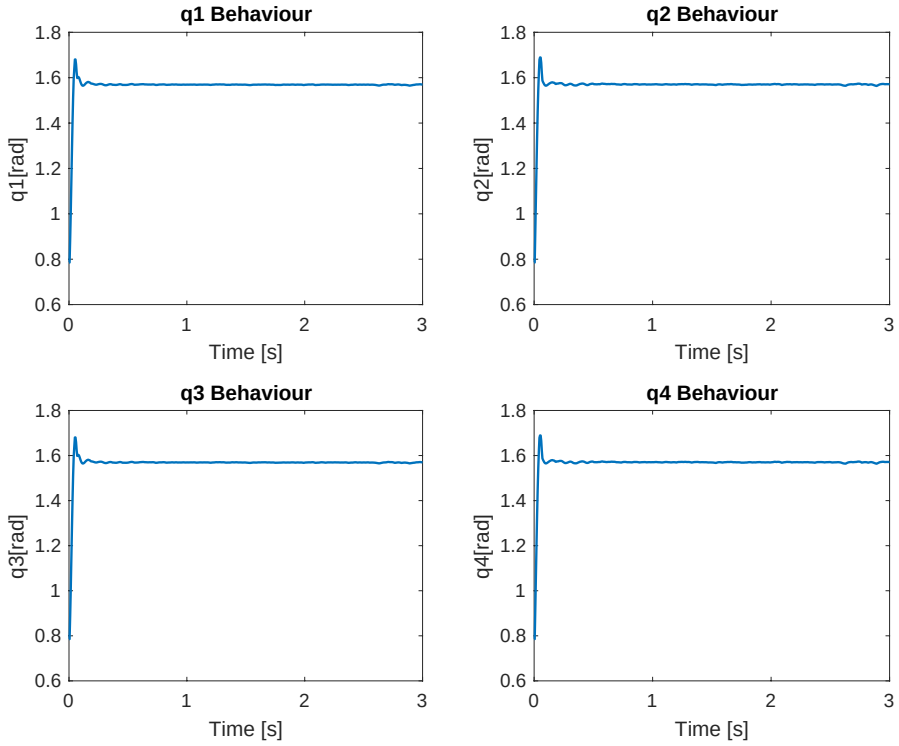


Figure 6: q_1, q_2, q_3, q_4 behaviours with the MPC

So we can see that the MPC works good because it reaches a steady-state in approximately 0.2 seconds.
The problem with this implementation is that the time needed for every iteration is quite big.
Later we will see that with the MPC Warm Start, there's a significant reduction of the computation time for every iteration.

4.4 Solution of Tracking Problem

Another possible solution to increase the speed in the optimization problem is computing the solution off-line and solving a tracking problem. In the latter, the reference is the optimal solution: we aim to follow the optimal solution, using a PD controller that reduces the error between the reference and the output.

Those results are available in Figure 7and Figure8: using a very high derivative coefficient K_d, we obtain a similar rise time of the reference. However, given a 0.2 seconds convergence time of the MPC solution, the convergence time of the tracking problem solution will be slower with respect to the MPC solution.

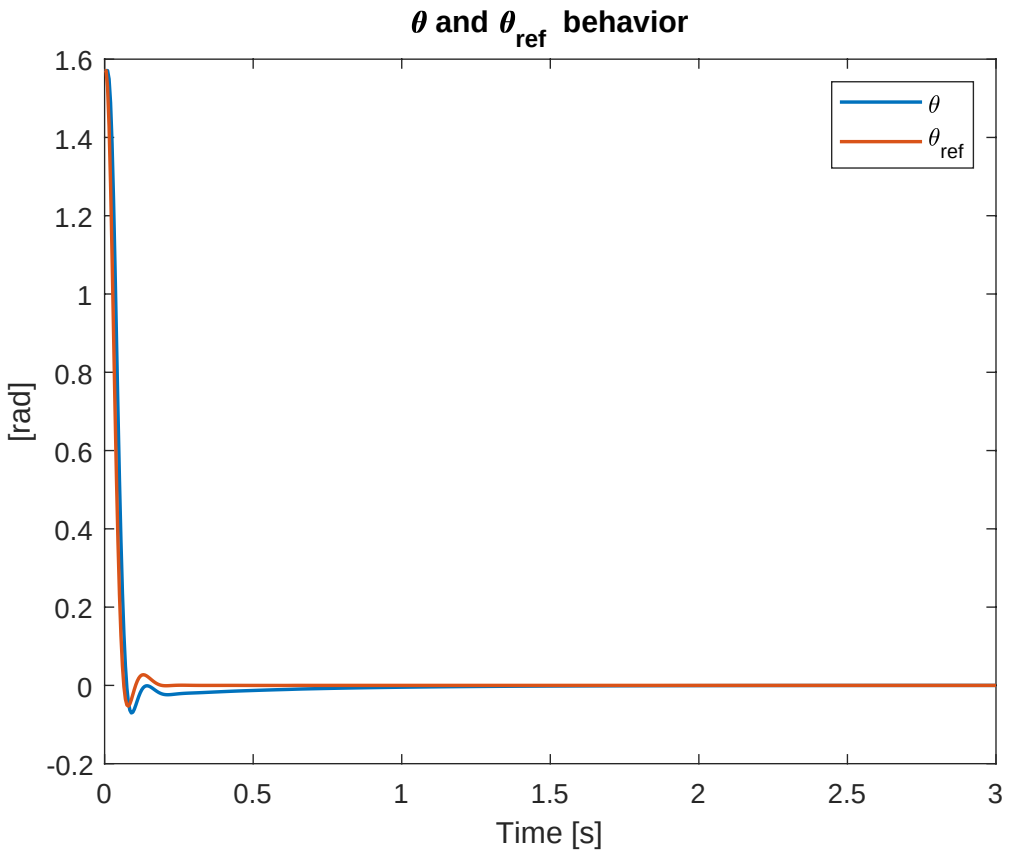


Figure 7: Comparison between the reference (blue plot) and the output (orange plot)

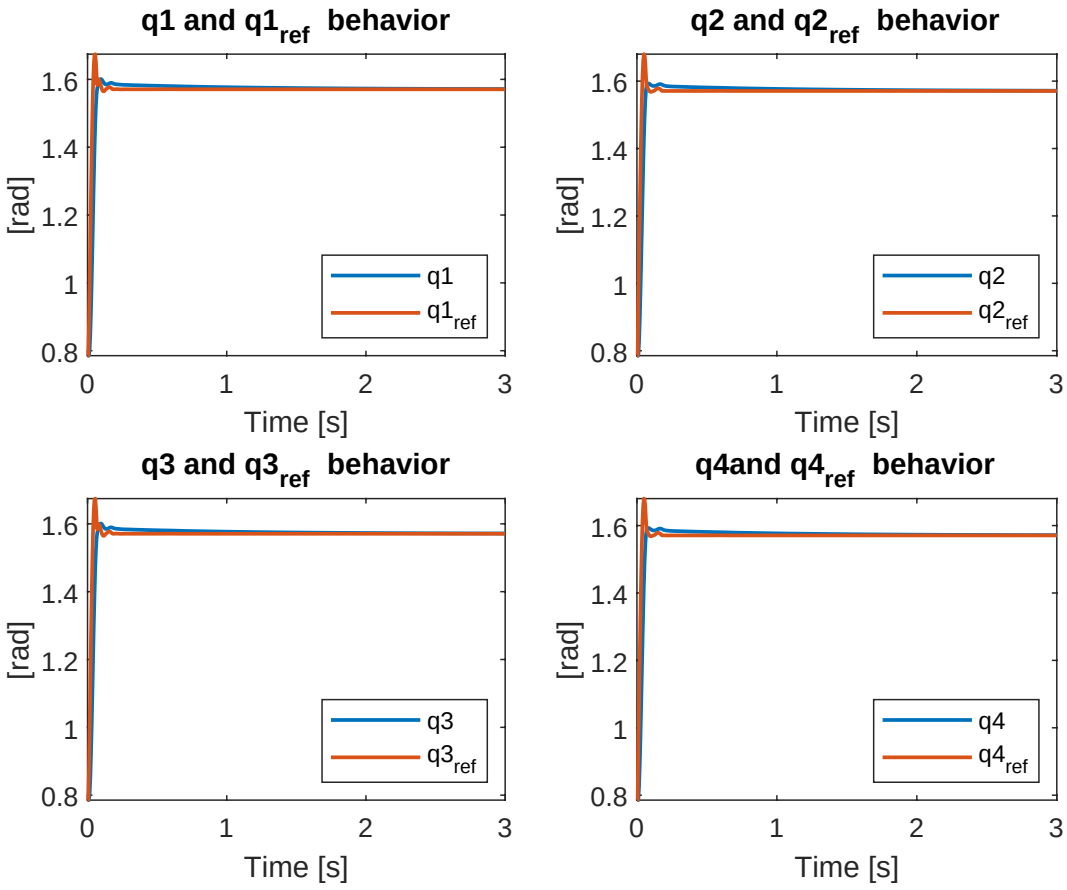


Figure 8: Comparison between the reference (blue plot) and the output (orange plot)

5 Neural Network

Model Predictive Control has proven to be a reliable method to obtain the convergence to our desired trajectory, but it has a drawback:
Given a time of 1 sec for the fall of the robot, our MPC computation can take several minutes, so it's unfeasible for an online application. In this section, we will discuss a novel approach to tackling this problem.
The intuition behind the use of a neural network is to:
1) Train the neural network to return the optimized trajectories
2) Use the pre-trained neural network on the actual task
In this way, we eliminate the time for the training of the network (which has been done offline) but we preserve good adaptability to a new situation. Now we will dive into the details to explain it.
We build a neural network that takes as input the initial state $\mathbf{x}(0)$ and outputs the entire state trajectory, namely:

$$\mathbf{x}(t), \mathbf{u}(t) = \Psi_{\mathbf{W}}^{\mathbf{Reflex}}(x(0)) \tag{13}$$

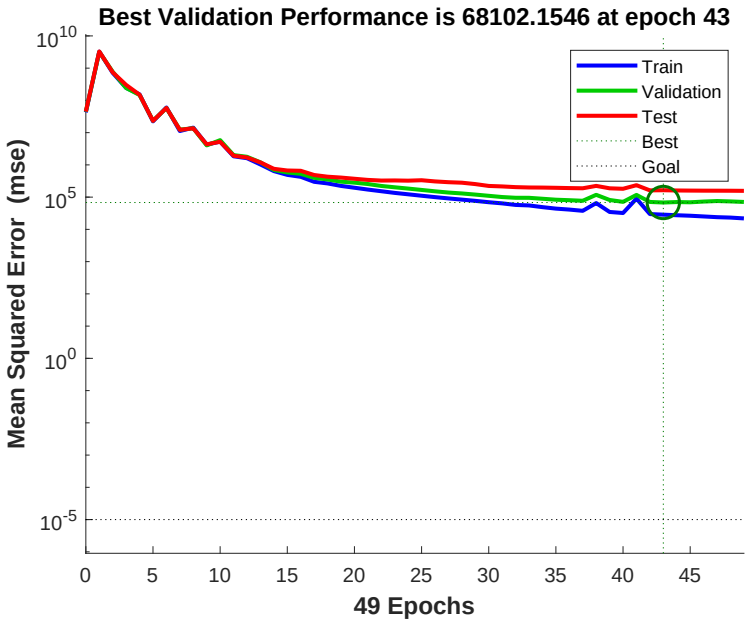
where $\mathbf{W}$ is the locally optimal weight of the network, and we call it a Reflex network because in this scenario it re-adjusts the robot orientation during the fall without knowing its actual dynamics and starting only from an initial state, imitating the reflexes of a falling feline.

5.1 Training

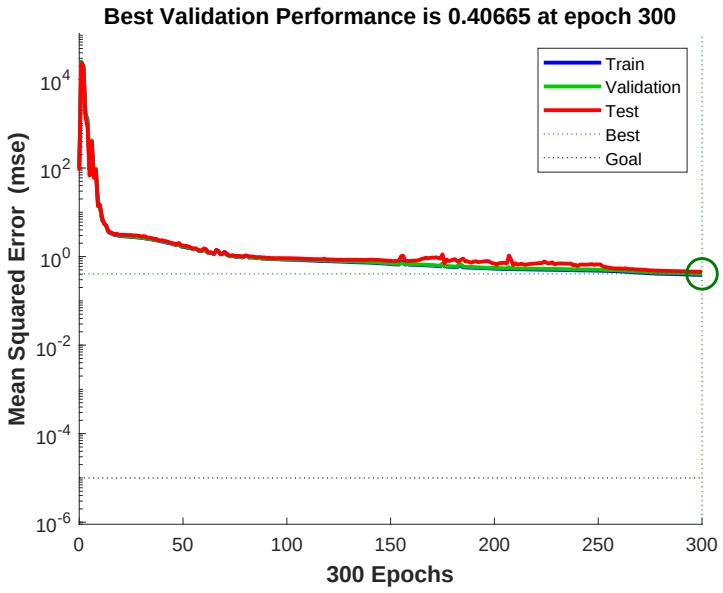
The neural network needs to be fed both with the inputs ($\mathbf{x}(0)$) and with a dataset to be used for the supervised regression task.
We generate this dataset running MPC 1000 times, with the initial state selected randomly in a given window of values, namely: $30° < \theta < 155°$ and $0° < q_i < 90°$, i=1...4
Then, we decided to modify the neural network proposed in the file, which equipped the neural network with 2 hidden layers of 512 units. In particular, we realized a feedforward network with three hidden layers with 50, 100, and 512 layer sizes and ReLu output functions.

The motivation behind this choice is twofold:
first, a deeper neural network loses a bit in the generality of results, but returns more accurate results for the specific task we are handling;
second, we reduced the size of the first layer to avoid a dispersive propagation of the information across the successive layers.
Moreover, this increased the interpretability of the model thanks to the limited number of weights connecting the input to the first layer.

Finally, we set a regularization parameter $\lambda = 0.6$ which prevented overfitting.
As you can see in the figure below, the model outperforms the previous one by checking the Mean Squared Error (MSE).



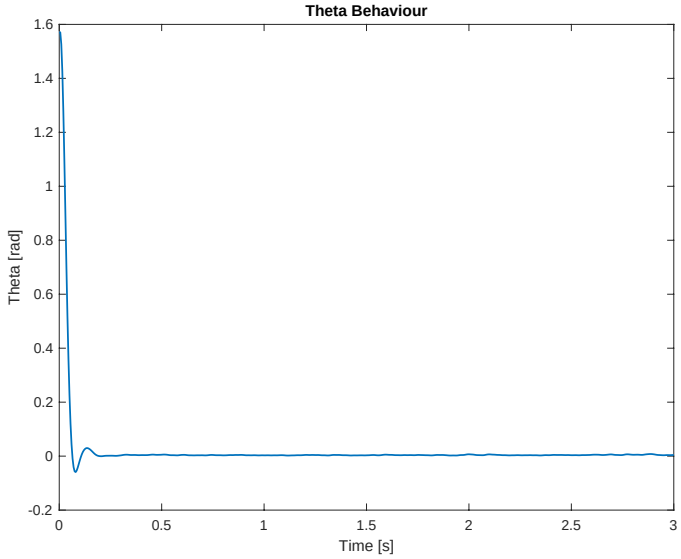
(a) mse on the reference model



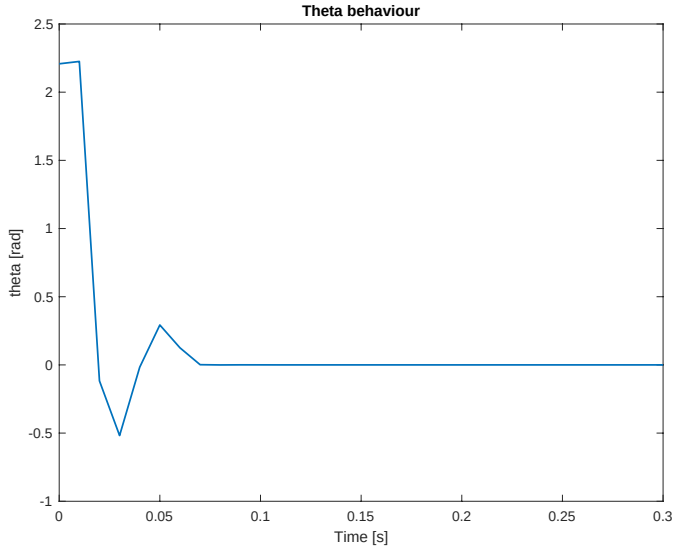
(b) mse on our model

5.2 MPC Warm-up

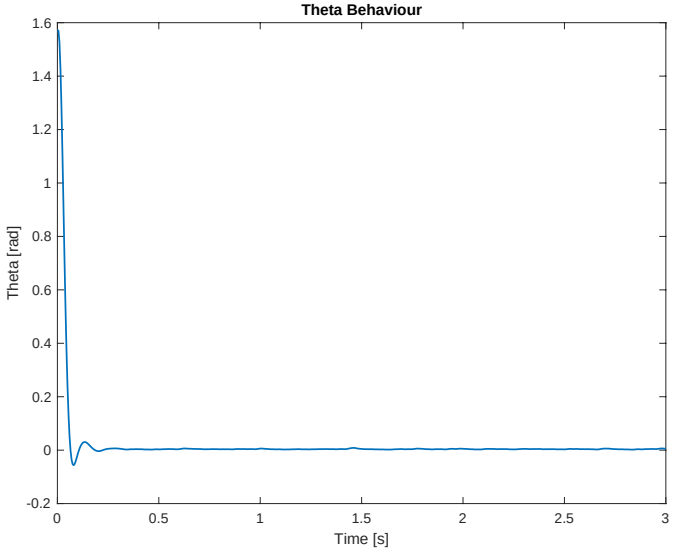
In this part, we cover another alternative to pure MPC; in fact, there is a smooth and nice way to speed up its computation kickstarting the MPC machine with the trajectories generated by our neural network.
The benefits of this approach are straightforward: if we focus on the computational time, it takes several minutes to MPC (4 seconds for every computation) for a complete generation of the desired trajectory.
With a warmed-up MPC, the computation takes approximately 200 seconds(0.3 seconds for every computation), thus getting closer to a real-time application. We show in the figure below the quantitative difference between the two approaches.



(a) The trajectory of θ generated by MPC-only



(b) θ trajectory generated by the Neural Network (subsampled)



(c) θ trajectory generated with warm-started MPC

6 Conclusions and Future Work

We have explored the problem of a quadruped robot falling from a given height.

We investigated several approaches to take this challenge, mainly involving Model Predictive Control. Then, we decided to use a pre-trained neural network to obtain a real-time re-orientation of the robot.

In the above-presented situation, there are still some interesting tricks to be done, such as improving the warming up of the MPC.

We believe that this topic has a great interest in the future because it opens to new scenarios in indoor navigation as well as in robot navigation on remote planets. We hope that this work will serve as a reference to investigate further some other "unexpected" situations, the effect of strong winds, rapid changes of illumination or a moving object suddenly hitting our robot.

References

[1] Vince Kurtz He Li Patrick M. Wensing and Hai Lin. *Mini Cheetah, the Falling Cat: A Case Study in Machine Learning and Trajectory Optimization for Robot Acrobatics*. 2021.

[2] Aaron M. Johnson, Thomas Libby, Evan Chang-Siu, Masayoshi Tomizuka, Robert J. Full, and Daniel E. Koditschek. Tail assisted dynamic self righting. 2012.

[3] Garrett Wenger, Avik De, and Daniel E. Koditschek. Frontal plane stabilization and hopping with a 2dof tail. *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 567–573, 2016.

[4] N. Rudin, Hendrik Kolvenbach, Vassilios Tsounis, and Marco Hutter. Cat-like jumping and landing of legged robots in low gravity using deep reinforcement learning. *IEEE Transactions on Robotics*, 38:317–328, 2022.