UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Optimal Control

Optimal Control of a Robot Manipulator

Professor: **Giuseppe Notarstefano**

Students: **Lorenzo Balandi**
Ludovica Ghinelli
Tommaso Prandin

Academic year 2021/2022

Abstract

Robotic manipulators are widely used in an high amount of industrial cases. They represent a very common example of non-linear system and are interesting to be studied for a lot of reasons.
This project aims at applying some optimal control techniques to a 2-DOF manipulator, which has to perform some desired tasks.

Contents

Introduction

This project aims at designing and implementing an optimal control law in order to let a 2-DOF planar manipulator manipulator perform some desired tasks.
The steps we have to consider are:

Task 0 Definition of the discretized dynamics of the manipulator;

Task 1 Design of an optimal trajectory to pass between two configurations, exploiting DDP;

Task 2 Design of a curve to be followed by the end-effector in the workspace and definition of the corresponding optimal trajectory in the jointspace, using DDP;

Task 3 Linearization of the model around the optimal trajectory and definition of the optimal feedback law to perform trajectory tracking, through LQR;

Task 4 Visualization of the result, with an animation.

Chapter 1

Task 0 - Framework Setting

In this first section, we try to setup everything we need in order to solve the problem, which basically means that we have to define all the Matlab functions that will be used in the main code.

1.1 Function `dynamics`

As a first task, it's necessary to define the dynamics of the manipulator, which describes how the system evolves. To do that, let's consider the variables needed to study the robotic structure in exam. In doing that, we simplify the notation neglecting the time dependency of the variables θ_1, θ_2, τ_1 and τ_2.

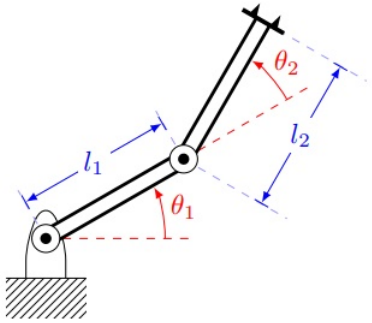

Figure 1.1: Structure of the manipulator

As can be seen from Figure 1.1, the parameters needed to describe the configuration of the manipulator are θ_1 and θ_2, which are stack in

the vector q, defined as:

$$q = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$

Another very important vector to be defined in order to study the dynamics is the vector τ, which contains the torques applied on the two joints. This means that it contains all the information regarding the actuation.

$$\tau = \begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix}$$

In order to define the dynamical model, three important matrices have to be defined. These are $M(q)$, the inertia matrix, $C(q, \dot{q})$ and $g(q)$, the matrix representing the effect of the gravity on the manipulator. These three matrices are defined as:

$$M(q) = \begin{bmatrix} m_1 r_1^2 + m_2(l_1^2 + r_2^2 + 2 l_1 r_2 cos\theta_2) + J_1 + J_2 & m_2(r_2^2 + l_1 r_2 cos\theta_2) + J_2 \\ m_2(r_2^2 + l_1 r_2 cos\theta_2) + J_2 & m_2 r_2^2 + J_2 \end{bmatrix}$$

$$C(q, \dot{q}) = -l_1 m_2 r_2 sin\theta_2 \begin{bmatrix} \dot{\theta}_2 & \dot{\theta}_1 + \dot{\theta}_2 \\ -\dot{\theta}_1 & 0 \end{bmatrix}$$

$$g(q) = \begin{bmatrix} (m_1 r_1 + m_2 l_1) g cos\theta_1 + m_2 r_2 g cos(\theta_1 + \theta_2) \\ m_2 r_2 g cos(\theta_1 + \theta_2) \end{bmatrix}$$

The variables m_1, m_2, J_1, J_2, l_1, l_2, r_1 and r_2, refer to the physical parameters of the robot, like mass, inertia, length and position of the center of mass, while g is the gravity acceleration. The values assumed by these parameters are collected in the following table:

Parameters:		
m_1	2	[Kg]
m_2	2	[Kg]
J_1	0.5	[Kg·m^2]
J_1	0.5	[Kg·m^2]
l_1	1	[m]
l_2	1	[m]
r_1	0.5	[m]
r_2	0.5	[m]

Figure 1.2: Values assumed by the parameters

In the end, it's possible to write the dynamic model of the manipulator, which turns out to be:

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

In order to write the dynamics of the system, it's necessary to consider its state-space representation, which means that we have to consider the state vector $x(t)$ and the input vector $u(t)$. They are defined as:

$$x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} = \begin{bmatrix} \theta_1(t) \\ \dot{\theta}_1(t) \\ \theta_2(t) \\ \dot{\theta}_2(t) \end{bmatrix}$$

$$u(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix} = \begin{bmatrix} \tau_1(t) \\ \tau_2(t) \end{bmatrix}$$

Our purpose is to obtain the discretized evolution of the system, which means that we have to understand how the $(t+1)^{th}$ sample of the state will behave given the t^{th} samples of the state and of the input. To do that, it's necessary to find the expression of all the 4 components of $\dot{x}(t)$, which will have to be discretized through Euler's method. Neglecting again the time dependancy, the evolution of the state derivative can be expressed as:

$$\dot{x}_1 = x_2$$

$$\dot{x}_3 = x_4$$

$$\begin{bmatrix} \dot{x}_2 \\ \dot{x}_4 \end{bmatrix} = M(x_1, x_3)^{-1} \left(\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} - C(x) \begin{bmatrix} x_2 \\ x_4 \end{bmatrix} - g(x_1, x_3) \right)$$

The discretization through Euler's method can be defined considering δ, which is the discretization time unit. Let's consider how this method is applied:

$$\dot{x}(t) = \frac{(x_{t+1} - x_t)}{\delta}$$

$$x_{t+1} = x_t + \delta \dot{x}(t)$$

In this way we have found the evolution in time of the discretized system, which corresponds to having found the function f:

$$x_{t+1} = f(x_t, u_t)$$

As the last step, we have to take into account the fact that DDP algorithm, in order to be implemented, requires both the first and the

second order derivative of the function f. Moreover, another quantity that the algorithm requires is the product between the vector p and the tensor representing the hessian of the function f. (In the DDP algorithm, p is a vector that is used to compute the control law).
It's important to highlight all these concepts because they have to be used in order to build, in our project, the Matlab function `dynamics`. This function is written considering another function, called `generatedynamics`, which builds all the computations seen so far considering x, u, and p as symbolic elements. Then, through `matlabFunction`, `generatedynamics` creates the final function, i.e. `dynamics`, which implements all the computations, but no more in a symbolic way. As a matter of facts, `dynamics` is able to find all the needed quantities in a numerical way, substituting the input values to their symbolic counterparts.

1.2 Function `stage_cost`

Another very important function to be defined is the one computing the stage cost. In the DDP algorithm, as well as in all the other optimal control algorithms, we try to minimize a function that is composed by one slice for each $t \in [0,\ T]$. The stage cost is defined as the sum of all the slices ranging from $t = 0$ to $t = T - 1$, while the final cost is the one we have when $t = T$.
In general, the cost represent how far we are from the desired behaviour of our system, which means that it's necessary to define some references x^{ref} and u^{ref}. Moreover, it's possible to define some matrices Q_t and R_t that represent the weights with which we compute the cost. The definition of this matrices allows us to give more or less importance to the distance between a certain quantity and its reference.
The stage cost is defined as:

$$\text{stage cost} = \sum_{t=0}^{T-1} l_t = \sum_{t=0}^{T-1} \left(||x_t - x_t^{ref}||_{Q_t}^2 + ||u_t - u_t^{ref}||_{R_t}^2 \right)$$

The function `stage_cost` is a very simple piece of code that, given x_t, u_t, their references and the weight matrices, computes the stage cost for the particular instant t. Moreover, this function implements the computation of the gradient and the hessian of the stage cost at time t, which will be useful in the definition of the DDP algorithm.

1.3 Function `term_cost`

As we have said, the terminal cost is the slice of the cost that we consider at time T. In order to define it, as well as we did for the stage cost, we have to define a weight matrix, named Q_T, and a reference for the state trajectory at time T, x_T^{ref}.
The terminal cost is, then:

$$\text{terminal cost} = l_T = ||x_T - x_T^{ref}||^2_{Q_T}$$

Very similarly with respect to what we did for `stage_cost`, also in this case we want our function to give as output not only the cost, but also its gradient and hessian, which will be used in the DDP implementation.

1.4 Conclusions

So far, we have introduced all the functions that will be used in our project. They are:

1. `generatedynamics`
2. `dynamics`$\big(x_t, u_t, p_t\big) = \big(x_{t+1}, \nabla_x f, \nabla_u f, \nabla^2_{xx} f \cdot p, \nabla^2_{uu} f \cdot p, \nabla^2_{xu} f \cdot p\big)$
3. `stage_cost`$\big(x_t,\ u_t,\ x^{ref},\ u^{ref},\ parameters\big) = \big(l_t,\ \nabla_x l_t,\ \nabla_u l_t,\ \nabla^2_{xx} l_t,\ \nabla^2_{uu} l_t,\ \nabla^2_{xu} l_t\big)$
4. `term_cost`$\big(x_T, x_T^{ref}, parameters\big) = \big(l_T, \nabla_x l_T, \nabla^2_{xx} l_T\big)$

Chapter 2

Task 1 - Trajectory Exploration

The purpose of this task is to define an optimal control law to design a trajectory that the robot will follow in order to pass from a particular configuration to another one, as it's shown in figure 2.1.

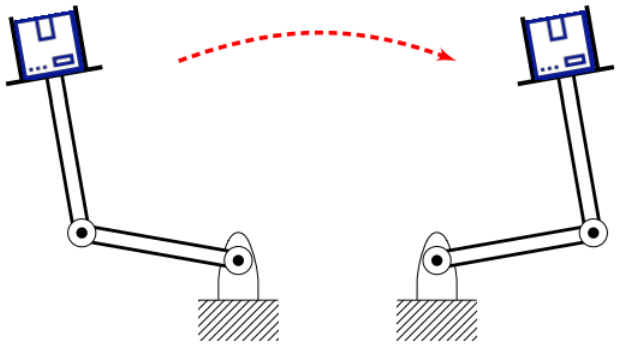

Figure 2.1: We have to pass from a first configuration to the second one

In the development of this project, we will consider as discretization unit $\delta = 10^{-3}$ and a time range of $30s$, which means that the overall number of samples will be:

$$T = \frac{30}{\delta} = 30000$$

2.1 Reference Definition

In order to perform this task, first of all we have to define the references, needed to define the cost to be minimized. The position reference is a step that goes from the initial position to the final one. We choose to start from the vertical downward condition and to span an angle as shown below:

$$q_i = \begin{bmatrix} -90 \\ 0 \end{bmatrix} [deg]$$

$$q_f = \begin{bmatrix} 70 \\ -10 \end{bmatrix} [deg]$$

In order to define the velocity reference and the input reference, we have to define how we would like this task to be achieved. We would like the robot to pass between the two configurations performing the task as slow as possible, which means that the velocity reference will be 0 and that the torque reference will be related to the torque needed to balance the gravity.
It's important to notice that the reference is not a trajectory, but it just defines the desired behaviour. The technique we will apply tries to get a trajectory as close as possible to the reference.
We are now able to define our references in terms both of x and u, which is done converting the position reference in rad:

$$x_t^{ref} = \begin{bmatrix} -1,5708 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{ when } t \, \epsilon \, \left[1, \ \frac{T}{2}\right]$$

$$x_t^{ref} = \begin{bmatrix} 1,2217 \\ 0 \\ -0,1745 \\ 0 \end{bmatrix} \text{ when } t \, \epsilon \, \left]\frac{T}{2}, \ T\right]$$

$$u_t^{ref} = g(q_i) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ when } t \, \epsilon \, \left[1, \ \frac{T}{2}\right]$$

$$u_t^{ref} = g(q_f) = \begin{bmatrix} 33,8879 \\ 4,9050 \end{bmatrix} \text{ when } t \, \epsilon \, \left]\frac{T}{2}, \ T\right]$$

In the following images it's possible to see the various references, which have the shape of a step, as already said.

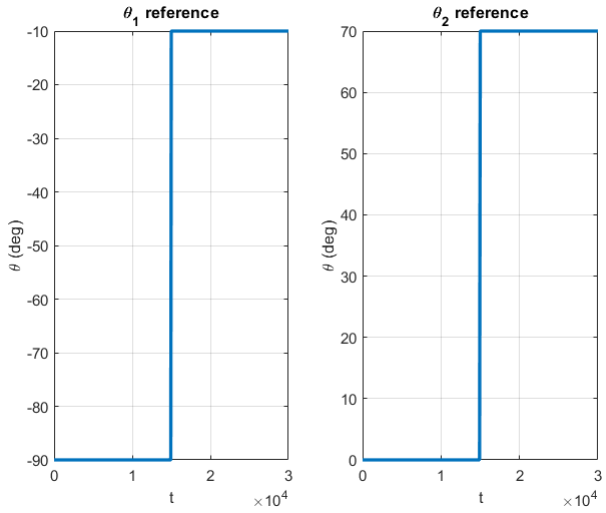

Figure 2.2: Angle reference

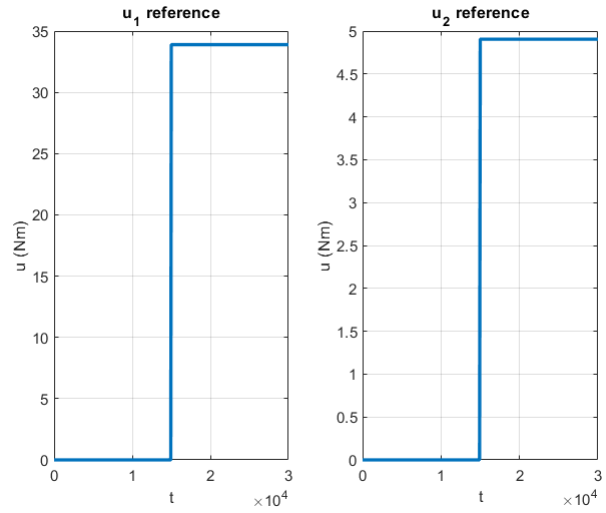

Figure 2.3: Torque reference

2.2 Cost Definition

After having defined the references, we can start applying DDP, in order to find the optimal trajectory for the manipulator. To do that, as a first step we need to define the weight matrices Q, R and Q_T. These matrices will define the cost function and the values inside them will determine the importance given to a certain quantity being far from

the reference. That's why we impose high values for the cost related to position and lower costs regarding the velocities. Concerning the torques, we use very low costs, and the reasons are that we are not so much interested in imposing a torque close to the gravity balance, and that with higher costs we get numerical issues in Matlab. The matrices chosen are:

$$Q = Q_T = \begin{bmatrix} 100 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 100 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

$$R = \begin{bmatrix} 0,0007 & 0 \\ 0 & 0,0007 \end{bmatrix}$$

2.3 Initialization

The DDP method is an iterative procedure, which means that, to apply it, it's necessary to give a value to the initial iteration. In order not to start from a trajectory too far from the final one, to define the first iteration we can use a classical controller studied in robotics. We have chosen the controller PD + gravity compensation, which defines the torque as:

$$\tau(\theta) = K_P \, (\theta^{\,ref} - \theta) + K_D \, \dot{\theta} + g(\theta)$$

This controller is implemented in a new function we have defined, called `algorithm_initialization`, which requires as input the state reference, the parameters of the problem and the task that we are trying to solve. It's necessary to specify the task because different values of K_P and K_D are exploited for the second task of the project.

$$\texttt{algorithm\_initialization}\big(x^{ref},\, parameters,\, task\big) = \big(x^{init},\, u^{init}\big)$$

The result obtained through this function are possible to be observed in the following figures, in which both θ^{init} and τ^{init} are depicted.

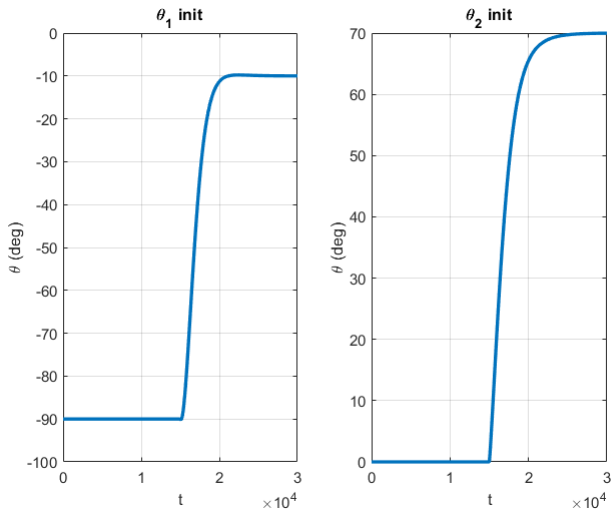

Figure 2.4: Angle initialization

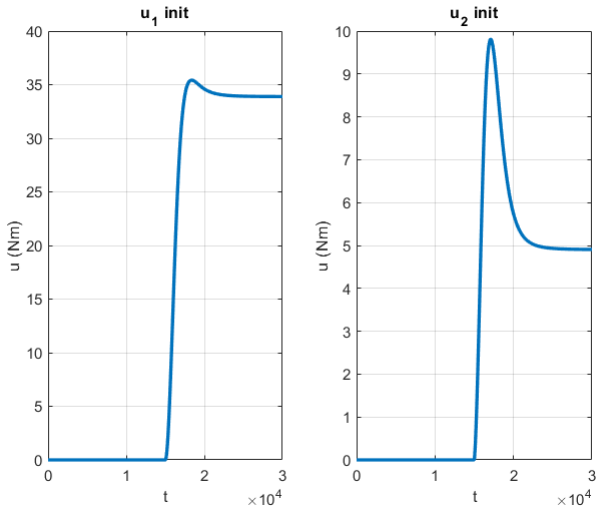

Figure 2.5: Torque initialization

This controller, even though not optimal, is a good way to initialize the DDP algorithm.

2.4 DDP Implementation

Having defined the cost matrices and the initialization, we have everything we need to apply Differential Dynamic Programming technique.

1. Compute the following quantities, which allow us to map everything in a general LQ problem:

$$q_t^k = \nabla_x l_t(x_t^k, u_t^k) \text{ with } q_T^k = \nabla l_T(x_T^k)$$

$$r_t^k = \nabla_u l_t(x_t^k, u_t^k)$$

$$Q_t^k = \nabla_{xx}^2 l_t(x_t^k, u_t^k) + \nabla_{xx}^2 f(x_t^k, u_t^k) \cdot p_{t+1}^k \text{ with } Q_T^k = \nabla^2 l_T(x_T^k)$$

$$R_t^k = \nabla_{uu}^2 l_t(x_t^k, u_t^k) + \nabla_{uu}^2 f(x_t^k, u_t^k) \cdot p_{t+1}^k$$

$$A_t^k = \nabla_x f(x_t^k, u_t^k)^T$$

$$B_t^k = \nabla_u f(x_t^k, u_t^k)^T$$

2. Compute the solution of the LQ problem related to the current k:

$$K_t^k = -(R_t^k + B_t^{k\ T} P_{t+1}^k B_t^k)^{-1} B_t^{k\ T} P_{t+1}^k A_t^k$$

$$\sigma_t^k = -(R_t^k + B_t^{k\ T} P_{t+1}^k B_t^k)^{-1} (r_t^k + B_t^{k\ T} p_{t+1}^k)$$

$$p_t^k = q_t^k + A_t^k p_{t+1}^k + K_t^{k\ T} (R_t^k + B_t^{k\ T} P_{t+1}^k B_t^k) \sigma_t^k \text{ with } p_T^k = q_T^k$$

$$P_t^k = Q_t^k + A_t^k P_{t+1}^k A_t^k - K_t^{k\ T} (R_t^k + B_t^{k\ T} P_{t+1}^k B_t^k) K_t^k \text{ with } P_T^k = Q_T^k$$

3. Apply the Armijo procedure in order to find γ^k

4. Update the state and input:

$$x_{t+1}^{k+1} = f(x_t^{k+1}, u_t^{k+1})$$

$$u_t^{k+1} = u_t^k + \gamma^k \sigma_t^k + K_t^k (x_t^{ref} - x_t^k)$$

This procedure has to be done until the descent direction becomes lower than the tolerance, which has been set as 10^{-6}.

2.5 Results

Once we have finished the cycle, which takes 7 iteration, we can look at the results.

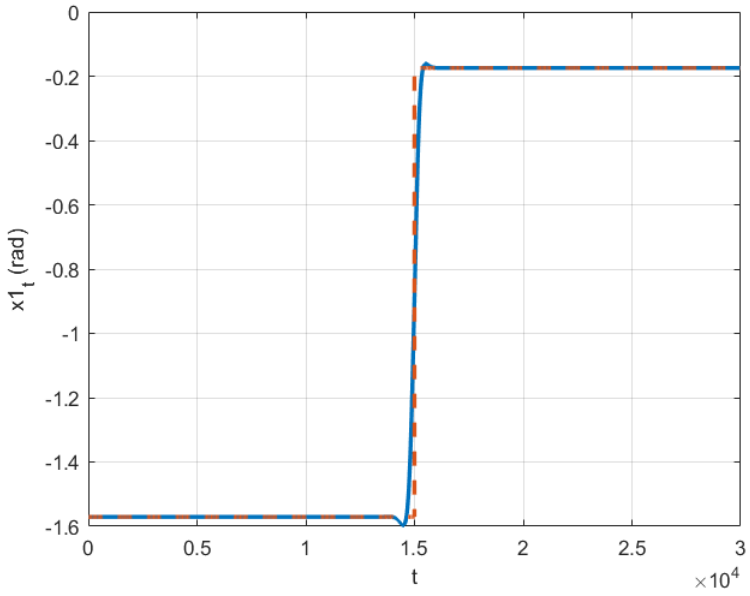

Figure 2.6: Evolution of the angle of the first joint (in blue) compared to the reference (in dashed orange)

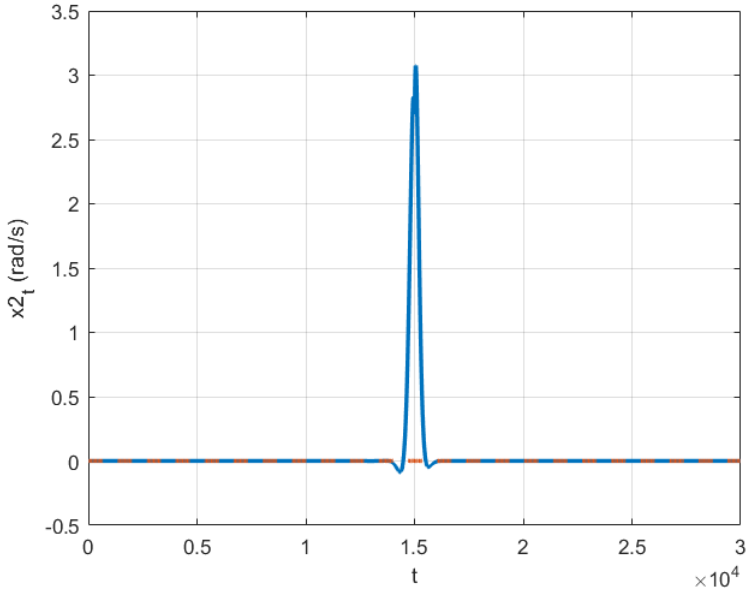

Figure 2.7: Evolution of the velocity of the first joint (in blue) compared to the reference (in dashed orange)

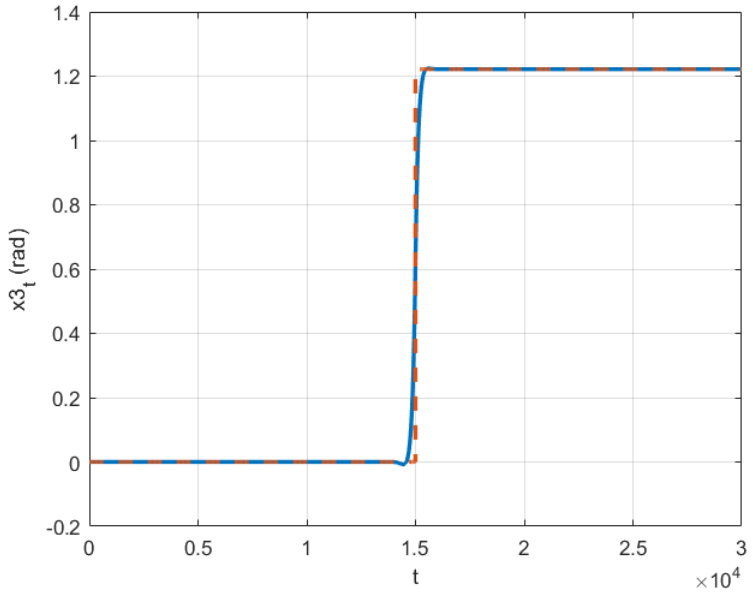

Figure 2.8: Evolution of the angle of the second joint (in blue) compared to the reference (in dashed orange)

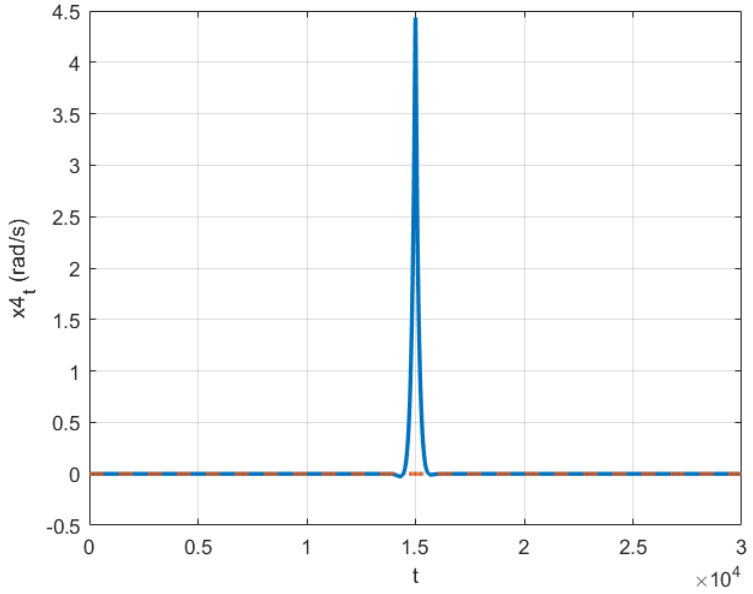

Figure 2.9: Evolution of the velocity of the second joint (in blue) compared to the reference (in dashed orange)

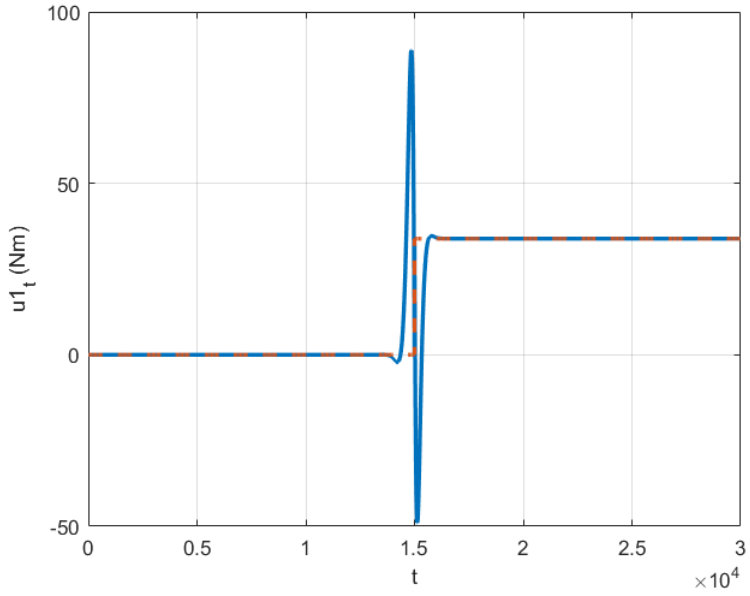

Figure 2.10: Evolution of the torque of the first joint (in blue) compared to the reference (in dashed orange)

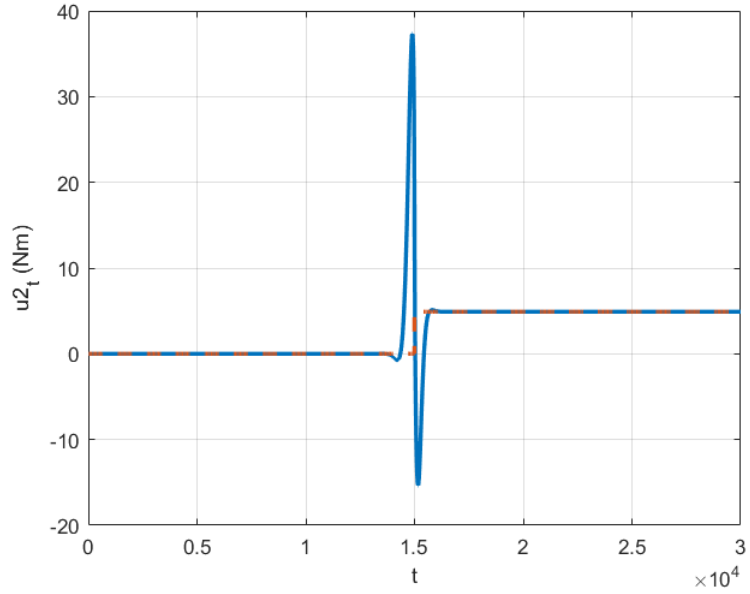

Figure 2.11: Evolution of the torque of the second joint (in blue) compared to the reference (in dashed orange)

Chapter 3

Task 2 - Wall Decoration

The purpose of the second task is to apply DDP in order to define the optimal trajectory to be followed by the joint angles and by the torque, in order to draw a precise shape with the end-effector of the manipulator.

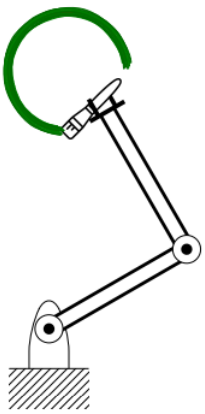

Figure 3.1: Wall Decoration

3.1 Cost Definition

The first thing to do is the definition of the parameters related to the dynamics and to the simulation. Being the system the same, the dynamics parameters are the same of the previous task, as well as the temporal window.
The weights that we use to define the cost function, in contrast, are not the same as before. Regarding the matrix R the reasoning is very similar, while for Q and Q_T there are some differences in terms of the

weights given to the angles. In fact, here we increase the weights, because an high precision in the reference tracking is needed, due to the fact that we would like to precisely draw the required shape.
The weighting matrices we choose are:

$$Q = Q_T = \begin{bmatrix} 1000 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

$$R = \begin{bmatrix} 0,0006 & 0 \\ 0 & 0,0006 \end{bmatrix}$$

With these weights, as said, we expect a good tracking of the reference angles and a relatively high input. As mentioned in the previous task, the weights of the input are small to avoid numerical problems, however this is not a problem if our main interest is in the precise tracking of the reference signal using how much input is needed.

3.2 Shape Definition

We chose to draw on the wall a circumference of radius $0.1m$ and centered in $(1; 1)$.

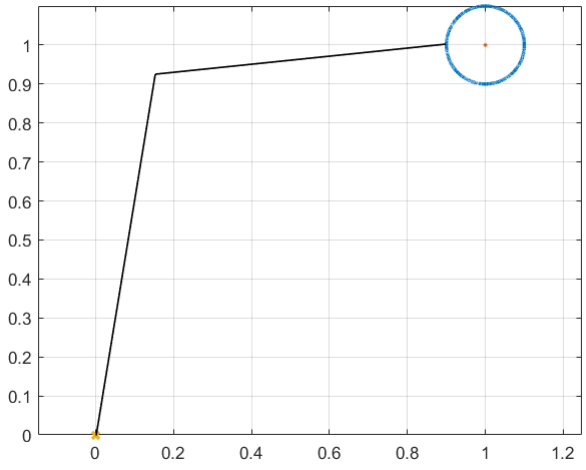

Figure 3.2: Circumference to be drawn (in blue) and manipulator arms (in black)

For each point of the circumference we solved the inverse kinematics for the 2-DOF manipulator, in order to get the angles in jointspace corresponding to those points in the workspace. It is also necessary to check that the points we want to touch are inside the reachable workspace. If not so, an error is displayed, but this is not our case. The inverse kinematics provides us of some important information that will be used in the definition of the reference.

3.3 Reference Definition

After having defined the shape we want to draw, it is necessary to define how to reach the first point of the circumference. Our complete reference trajectory is as follows:

1. from $t = 0s$ to $t = 1s$ the robot is at rest, which is the stable equilibrium in the downward configuration;

2. from $t = 1s$ to $t = 6s$ the reference has to pass from the stable equilibrium to the first component of the vectors found with the inverse kinematics, i.e. $(1, 1;\ 1)$. This is done with a fifth-order polynomial trajectory, which allows us to consider continuity in terms of position, velocity and acceleration. This choice leads to smooth profiles in terms both of position, velocity and acceleration: This is positive because prevents the final trajectories from having unwanted peaks in correspondence of the discontinuities, which would be present with other choices, like a ramp;

3. from $t = 6s$ to $t = 30s$ the reference is the one found with the inverse kinematics, which allows us to track the circumference.

The velocity references are always 0, while the input references are obtained from the balance of the gravity term only (quasi-static trajectory).

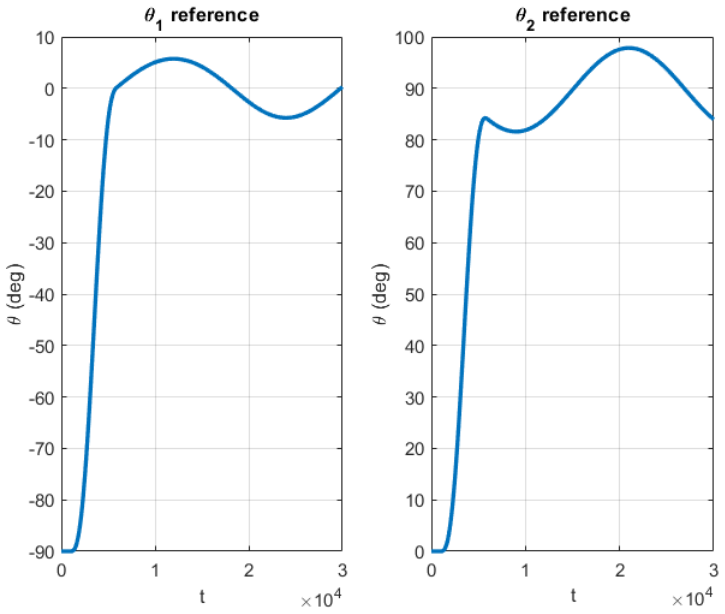

Figure 3.3: Angle reference

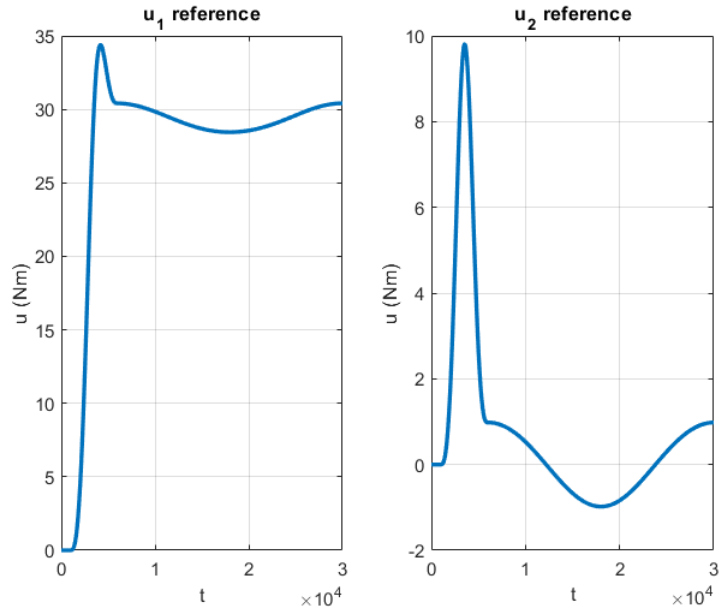

Figure 3.4: Torque reference

3.4 Initialization

In order to implement the DDP algorithm we have to give a value to the first iteration. To perform this operation we use, as in task 1, the PD + gravity controller implemented in `algorithm_initialization`. In this way we obtain an initialization which is very similar to the reference, which means that the DDP algorithm will not need a great effort to get the final result from this initialization. This may make the difference from a successful application of the DDP and a failure.
The initial trajectories we have imposed can be seen in the following figures.

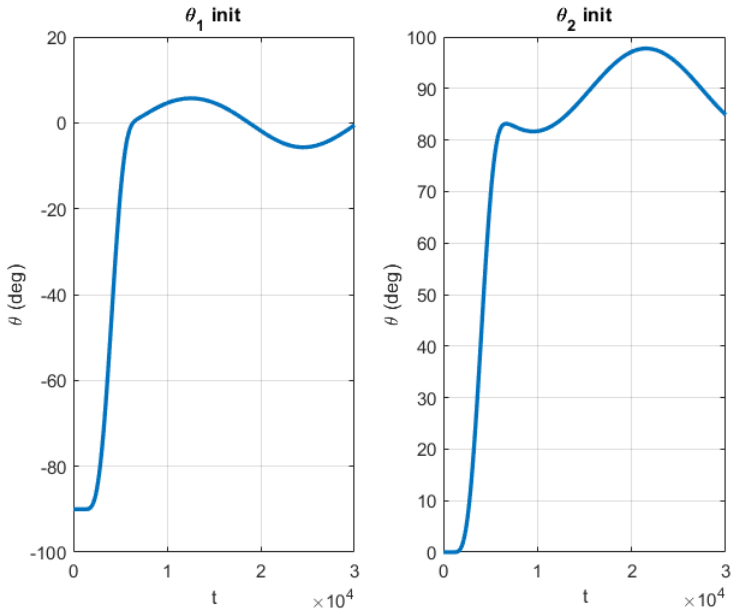

Figure 3.5: Angle initialization

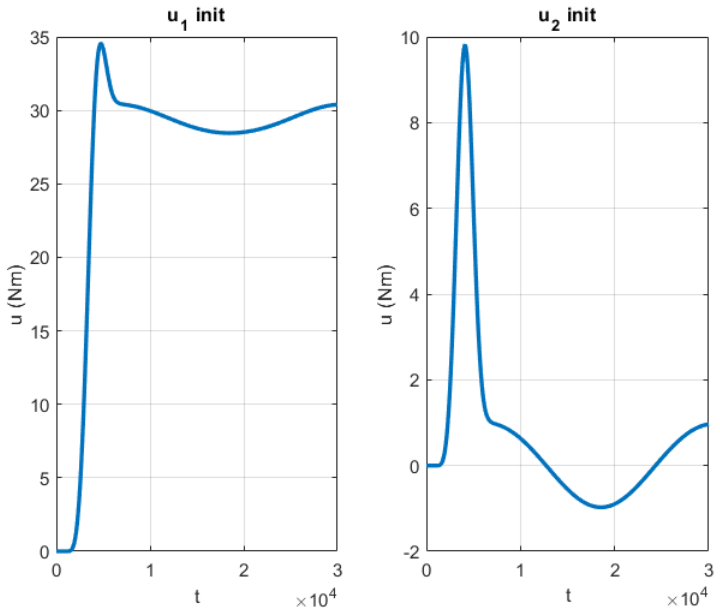

Figure 3.6: Torque initialization

3.5 DDP Implementation

At this point we have defined the cost matrices and an initialization, which is all we need to apply DDP. The implementation follows the same steps described in section 2.4. The only difference is the reference, which in this task is a little bit more elaborated than the one in the previous task, as described in section 3.3.

3.6 Results

The application of the DDP, after 4 iterations, leads to the following results.

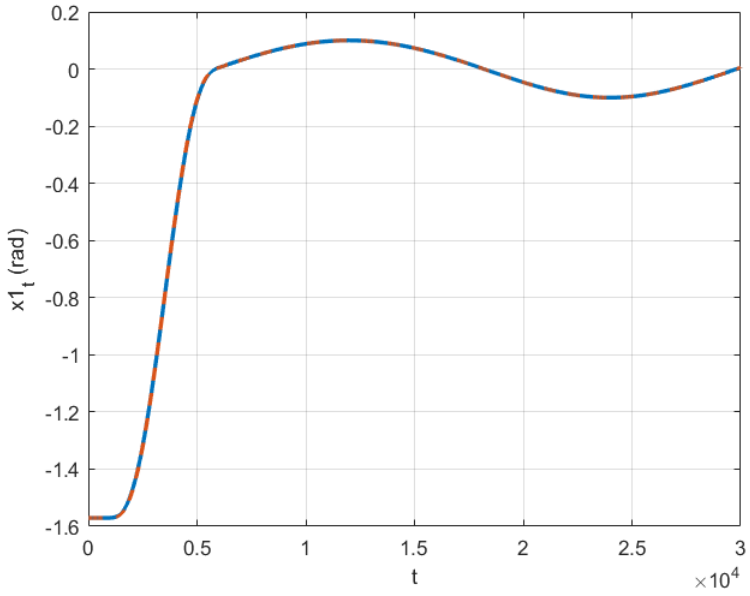

Figure 3.7: Evolution of the angle of the first joint (in blue) compared to the reference (in dashed orange)

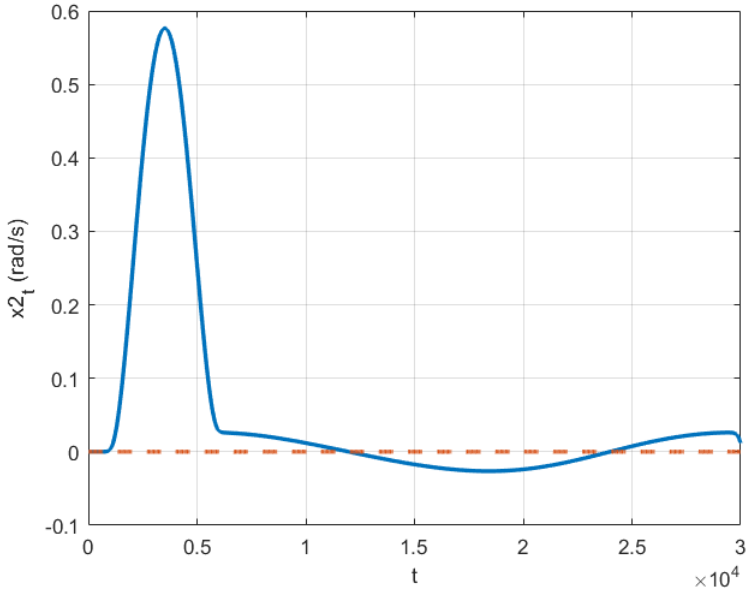

Figure 3.8: Evolution of the velocity of the first joint (in blue) compared to the reference (in dashed orange)

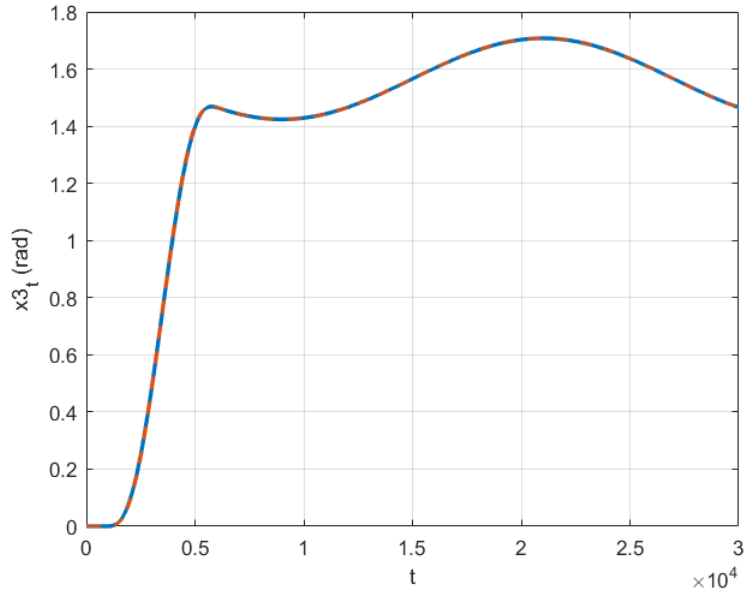

Figure 3.9: Evolution of the angle of the second joint (in blue) compared to the reference (in dashed orange)

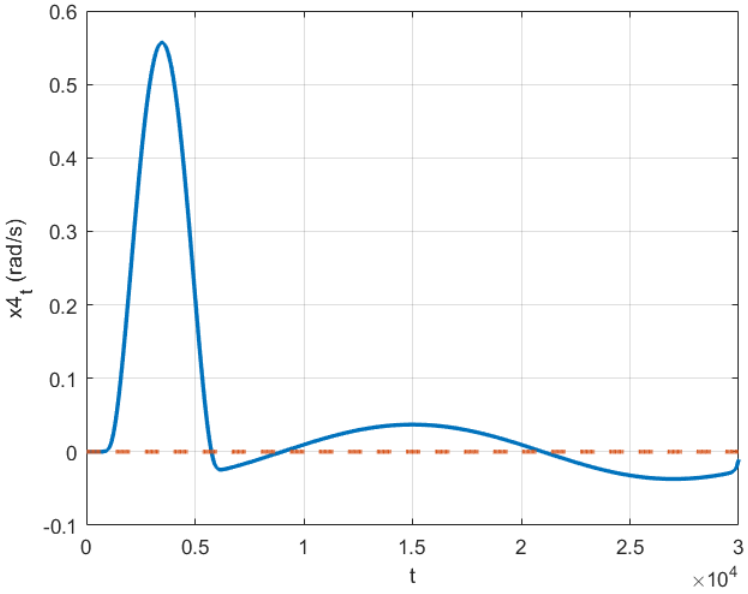

Figure 3.10: Evolution of the velocity of the second joint (in blue) compared to the reference (in dashed orange)

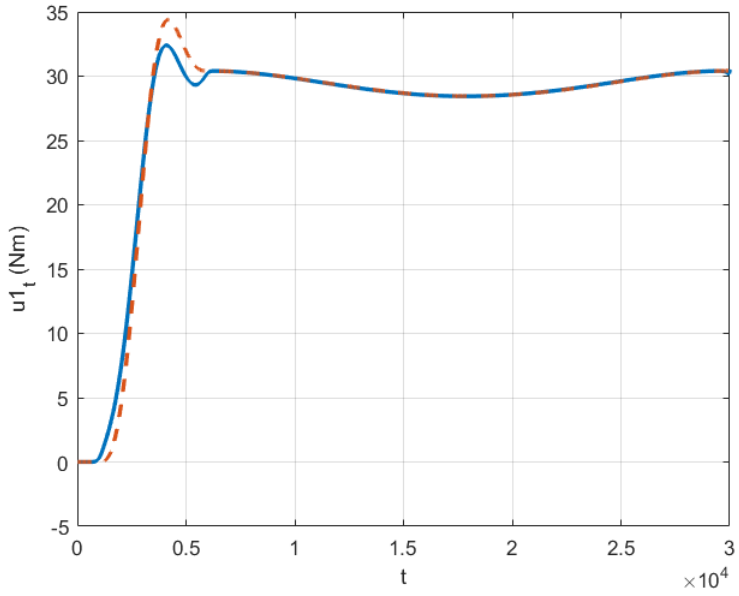

Figure 3.11: Evolution of the torque of the first joint (in blue) compared to the reference (in dashed orange)

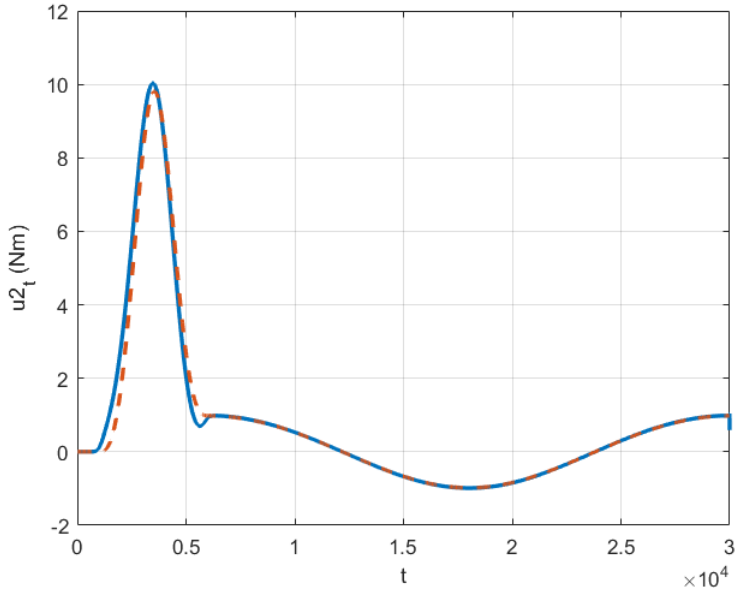

Figure 3.12: Evolution of the torque of the second joint (in blue) compared to the reference (in dashed orange)

Chapter 4

Task 3 - Trajectory Tracking

The purpose of the third task is to exploit the results obtained in the second task and design a controller able to track the optimal trajectory (x_t^*, u_t^*).

4.1 Cost Definition

The first thing to do, as always, is the definition of the weights to be used to define the cost function. In this task, we would like to precisely follow the reference trajectory, which means that high costs can be assigned to the first and third component of the state.

$$Q = \begin{bmatrix} 1000 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

It's important to say that these costs actually play a role only if there are some changes between the case in which we computed the optimal trajectories and the case in which we track them. In fact, if neither the parameters nor the initial conditions change, we could use any weight and we would always obtain a perfect tracking.

4.2 LQR Implementation

The Linear Quadratic Regulator relies on the linearization of the dynamics of the system about the optimal trajectory, which we have found in task 2. The linearization is performed exploiting the function

`dynamics`, due to the fact that two of the output provided are exactly the gradient of the dynamics with respect to x and u.

$$A_t^* = \nabla_x f(x_t^*, u_t^*)^T$$

$$B_t^* = \nabla_u f(x_t^*, u_t^*)^T$$

After having found the linearization matrices for all the time instants, we can find the matrix Q_T, which is the solution of the algebraic Riccati equation determined by A_t^*, B_t^*, Q and R. To find this matrix, we exploit the Matlab function `idare`, which returns the solution of the equation needed.
Now, we can proceed considering the loops that have to be performed for every iteration:

1. Backward iteration to find P:

   $$P_t = Q + {A_t^*}^T P_{t+1} A_t^* - {A_t^*}^T P_{t+1} B_t^* (R + {B_t^*}^T P_{t+1} B_t^*)^{-1} {B_t^*}^T P_{t+1} A_t^*$$

   Initialized as $P_T = Q_T$

2. Forward iteration to find K:

   $$K_t = -(R + {B_t^*}^T P_{t+1} B_t^*)^{-1} {B_t^*}^T P_{t+1} A_t^*$$

3. Input and state update:

   $$u_t = K_t x_t$$

   $$x_{t+1} = f(x_t, u_t)$$

   Initialized in x_0, which is given

4.3 Results with expected initial conditions

After having computed the trajectory (x, u) wit x_0 corresponding to the downward equilibrium, it's possible to visualize the results. As already said in section 4.1, the tracking is perfect even with weights all equal to 1, because we are in the exact same scenario in which we built the optimal trajectory.

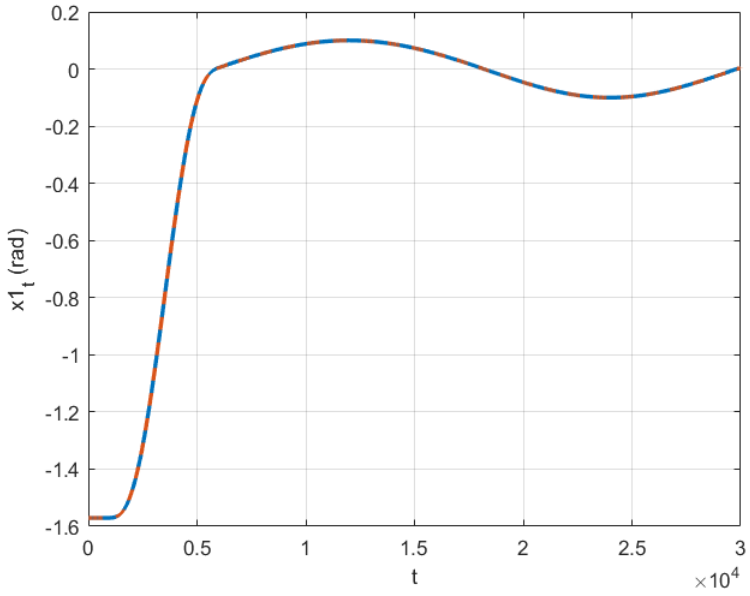

Figure 4.1: Evolution of the angle of the first joint (in blue) compared to the optimal trajectory (in dashed orange)

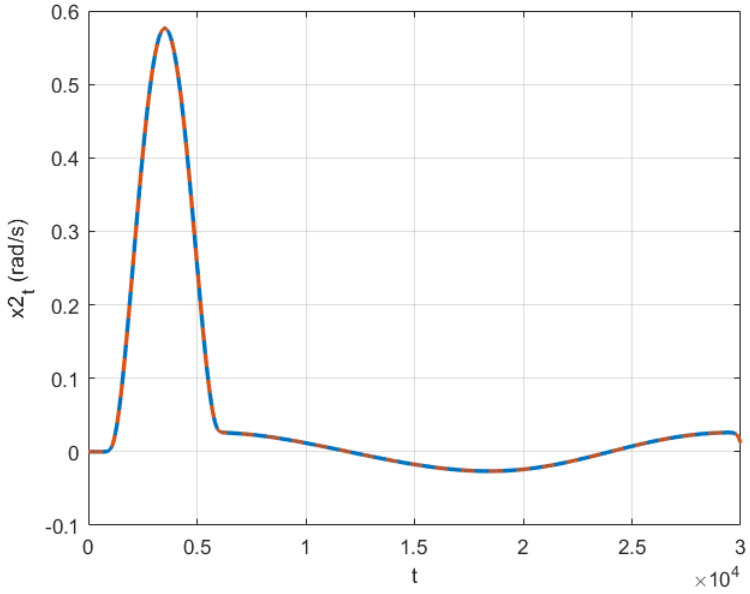

Figure 4.2: Evolution of the velocity of the first joint (in blue) compared to the optimal trajectory (in dashed orange)

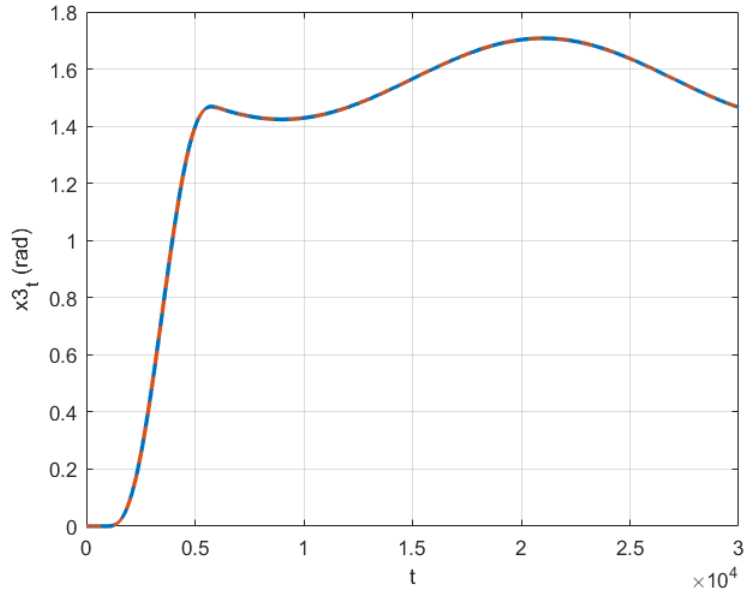

Figure 4.3: Evolution of the angle of the second joint (in blue) compared to the optimal trajectory (in dashed orange)

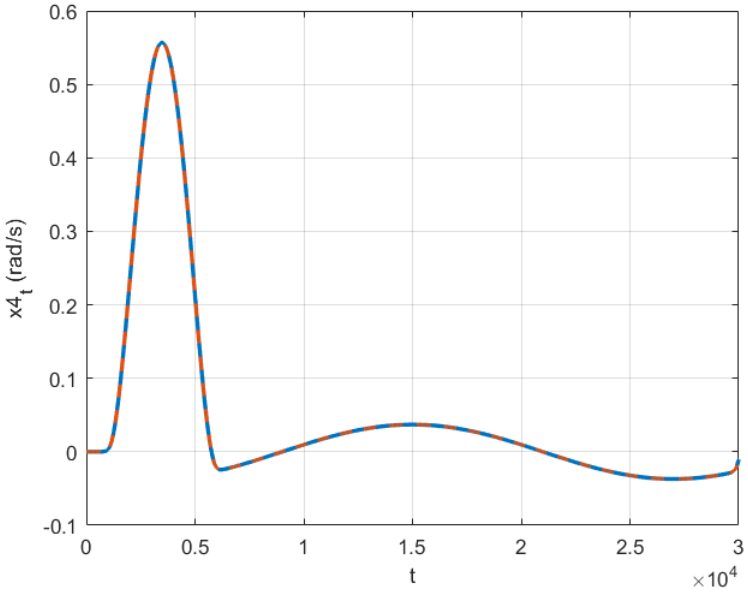

Figure 4.4: Evolution of the velocity of the second joint (in blue) compared to the optimal trajectory (in dashed orange)

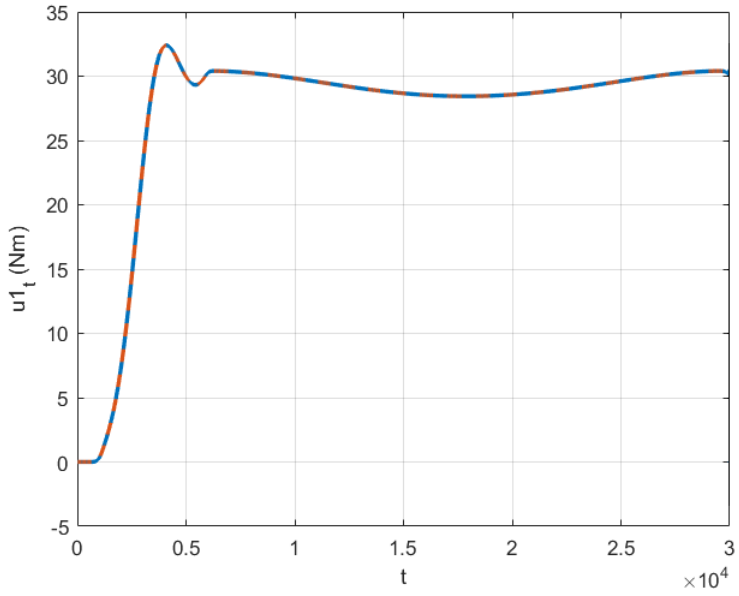

Figure 4.5: Evolution of the torque of the first joint (in blue) compared to the optimal trajectory (in dashed orange)

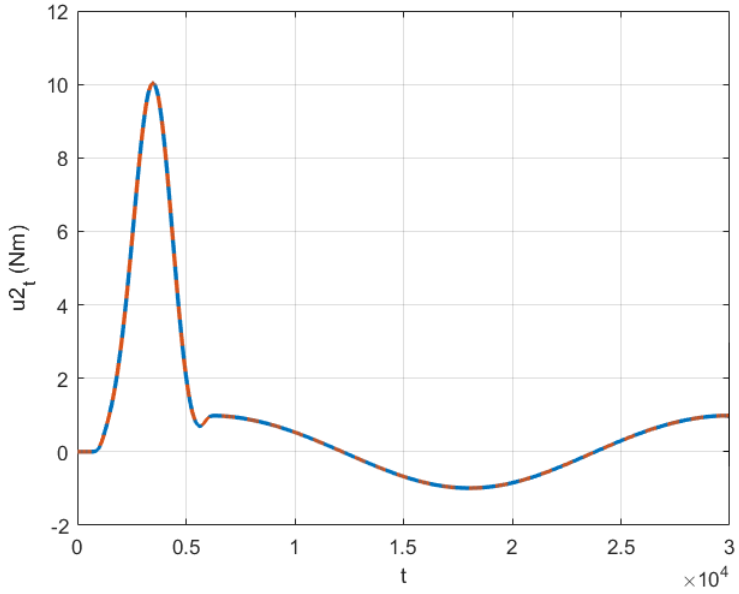

Figure 4.6: Evolution of the torque of the second joint (in blue) compared to the optimal trajectory (in dashed orange)

4.4 Results with different initial conditions

In order to see whether our controller works also in conditions different from the expected ones, we can impose x_0 to be different with respect to the downward equilibrium position. If so, we are asking our controller to recover from an unexpected initialization. From the following plots it's possible to see that, even though the final trajectory is far from the optimal one at the beginning, the joint angles are able to precisely follow the trajectory needed to draw the circle, which happens between $t = 6s$ and $t = 30s$

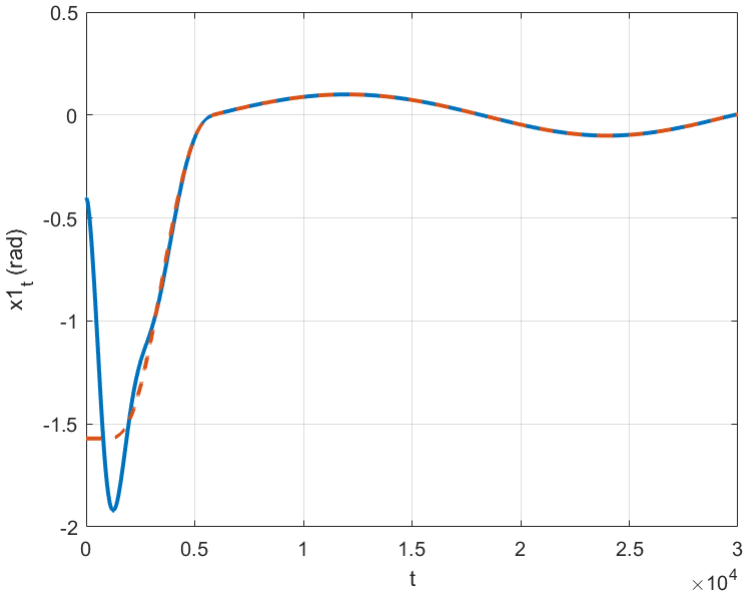

Figure 4.7: Evolution of the angle of the first joint (in blue) compared to the optimal trajectory (in dashed orange)

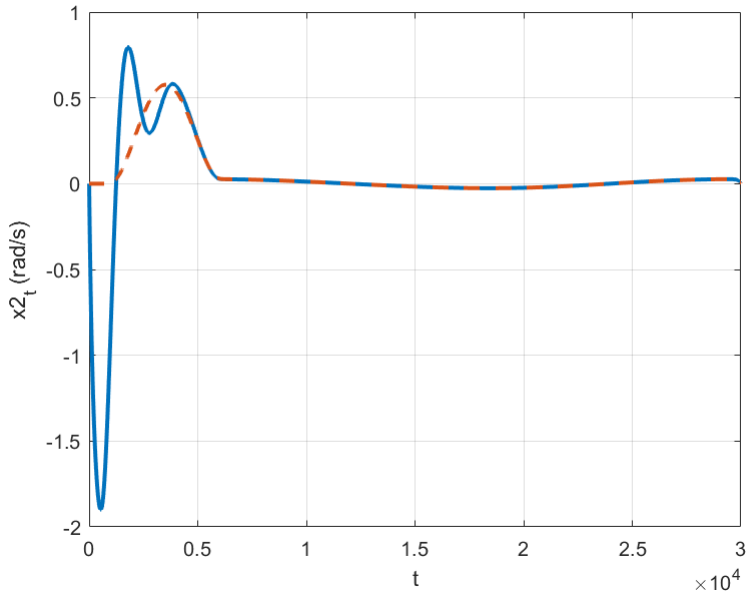

Figure 4.8: Evolution of the velocity of the first joint (in blue) compared to the optimal trajectory (in dashed orange)

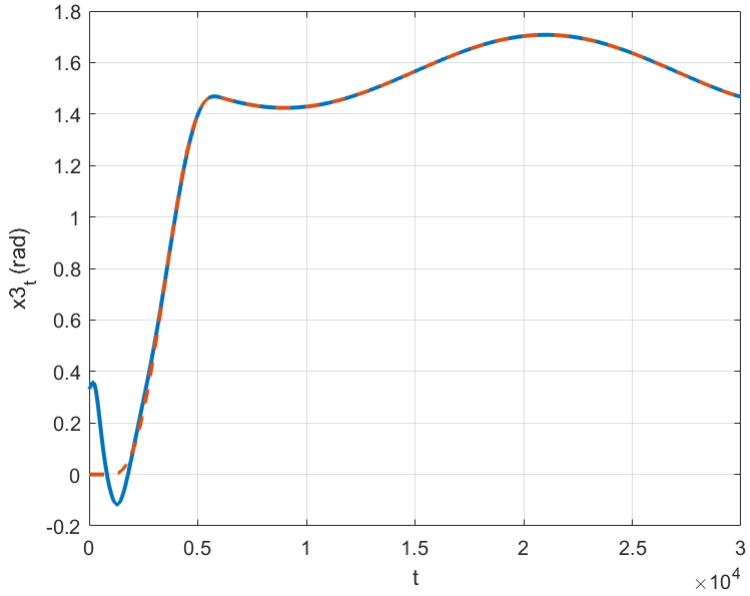

Figure 4.9: Evolution of the angle of the second joint (in blue) compared to the optimal trajectory (in dashed orange)

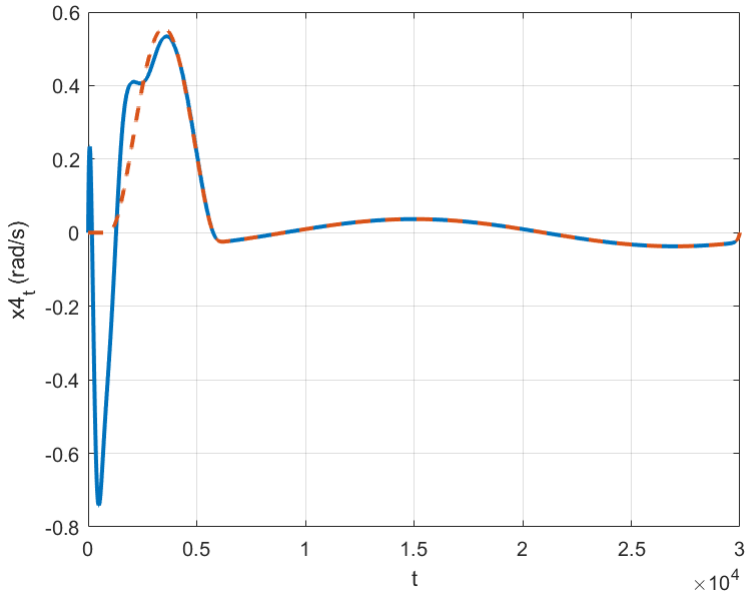

Figure 4.10: Evolution of the velocity of the second joint (in blue) compared to the optimal trajectory (in dashed orange)

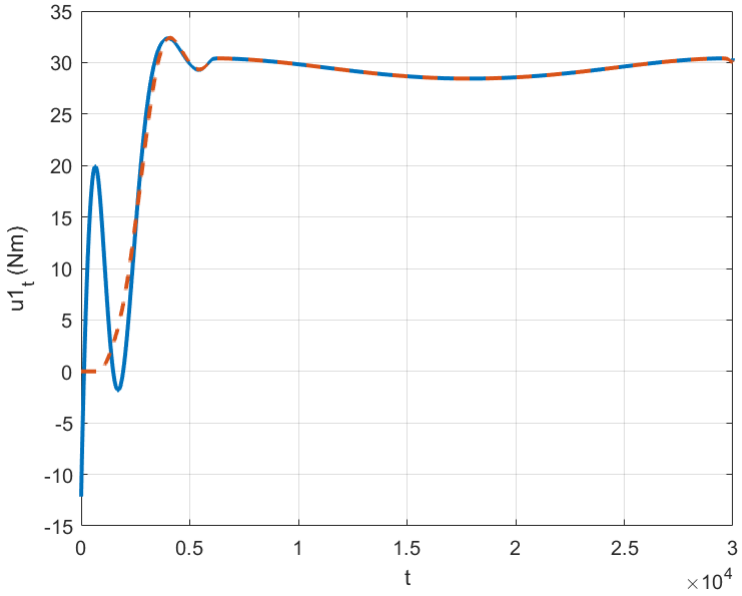

Figure 4.11: Evolution of the torque of the first joint (in blue) compared to the optimal trajectory (in dashed orange)

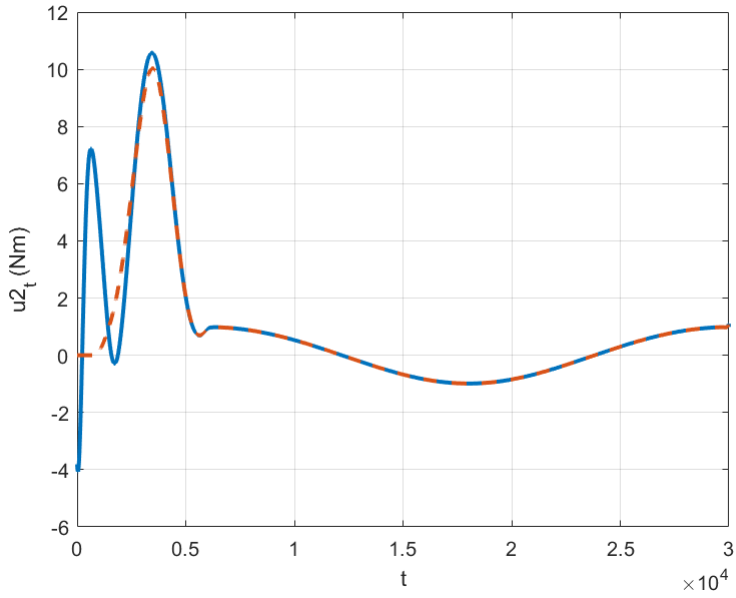

Figure 4.12: Evolution of the torque of the second joint (in blue) compared to the optimal trajectory (in dashed orange)

Chapter 5

Task 4 - Animation

After having found the trajectories that our manipulator will have to follow in order to perform the desired trajectories, it's necessary to implement them in a software devoted to the creation of an animation. This is very useful in order to visualize in practice the results produced by the algorithms we applied.

5.1 Simscape Multibody Model

The software we decided to use is Simscape Multibody, which is a Matlab package that allows to simulate mechanical system composed by more than one element interacting with each other. A perfect example of multibody system is the manipulator that we have to study, which is composed by two links and a fixed base.
In order to perform the visualization of a particular simulation in this environment, first of all it's necessary to define the model of the multibody system, i.e. our 2-DOF manipulator. To do that, after having installed Simscape Multibody we type `smnew` on the Command Window to open a new document, and we start building the proper model. The final model is depicted in the image below:

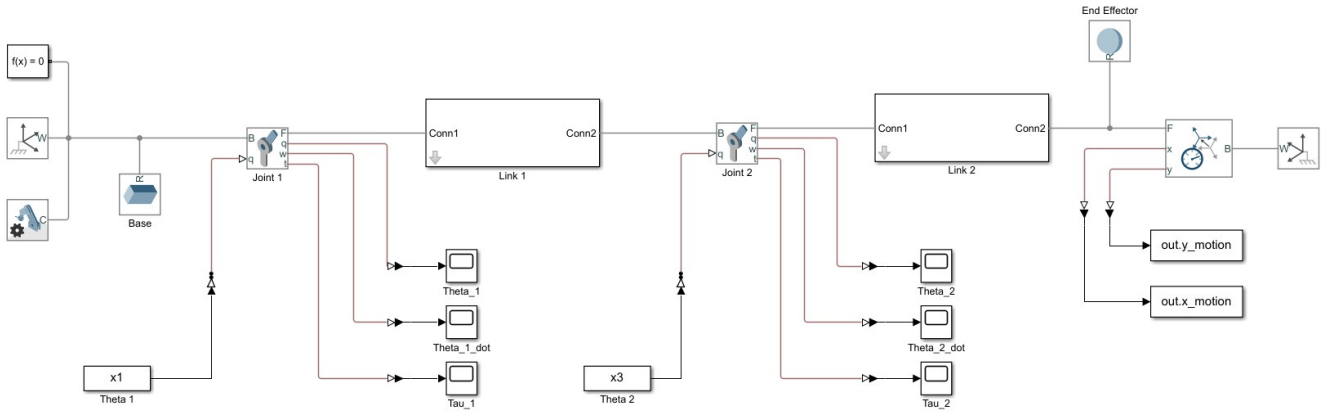

Figure 5.1: Overall manipulator model

The scheme simply represents a chain composed by four rigid bodies, i.e. the two links, the end-effector and the fixed base. The first three rigid bodies are linked by two rotational joints, which allow them to rotate one with respect to the other.
The blocks Link 1 and Link 2 have the same internal structure, which is depicted in figure 5.2.

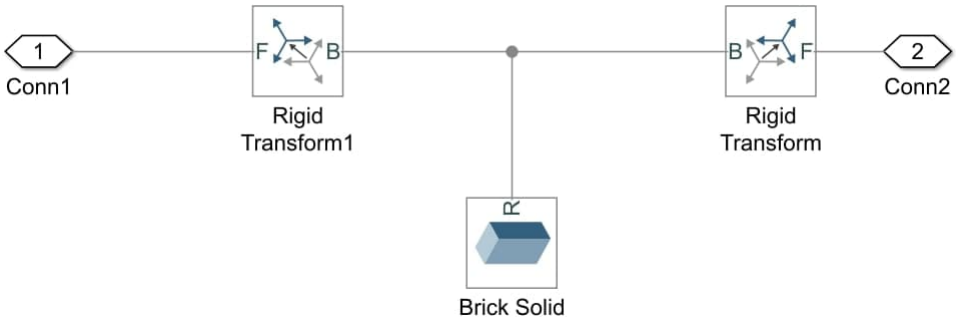

Figure 5.2: Link internal structure

It may be useful to briefly define what all the blocks are used for:

Solver Configuration: it's a default block defining the simulation settings;

World Frame: it's a default block defining the axes with respect to which everything is referenced;

Mechanism Configuration: it's a default block that allows us to define g;

Brick Solid: defines a rigid body shaped as a parallelepiped, we use it to define the base and the links;

Spherical Solid: defines a a rigid body shaped as a sphere, we use it to define the end-effector;

Rigid Transform: defines the transformation between two reference frames;

Scope: plots a variable;

Revolute Joint: allows only a single DOF, which corresponds to a rotation about the z axis;

From Workspace: receives a variable from the Workspace, we use it to transfer the input trajectories to be followed;

To Workspace: passes a variable to the Workspace, we use it to transfer the end-effector motion to be plotted;

Transform Sensor: allows us to capture the motion of the end-effector in the plane;

Simulink-PS Converter: transforms a variable from Simulink domain (Workspace variables, scope variables...) to the world of the physical system (joint angles, link coordinates...);

PS-Simulink Converter: the opposite of the previous block.

Running the model, the visual result is shown in the image below. It's possible to see that there is the blue cube representing the base, two black sticks representing the links and a red sphere representing the end-effector, which motion has to be the desired one.

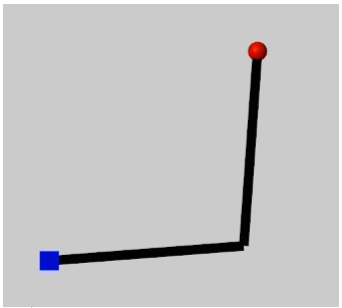

Figure 5.3: Manipulator generated from the Simscape Multibody model

5.2 Visualization of the Results

We have now everything we need to implement the final animation, which is done defining, in the Matlab Workspace, the proper input trajectory to be applied to the system. This is done defining in Matlab two matrices representing in one column the values that the two joint angles have to assume for every time instant, and in the other column the time instants.
Considering as a reference the results described in paragraph 4.3, we can visualize the motion of the manipulator. This is done considering the sensing of the motion of the end-effector, which can be plotted and observed in the figure below.

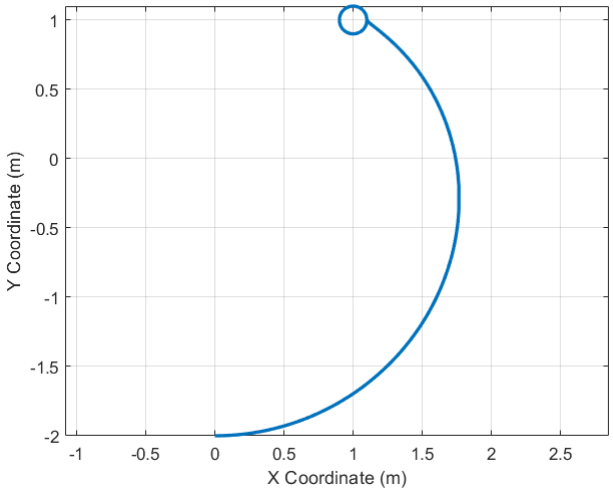

Figure 5.4: End-effector motion, we can see that it starts from the vertical downward position, reaches the desired position and draw the circle

Conclusions

During the development of this project, we have solved all the tasks that were initially required to be analysed.

Task 1: In the first task we managed to use DDP to design the optimal trajectory to follow the step reference. We did it giving high importance to the accuracy of the position and caring less about velocity and input torque;

Task 2: In the second task we succeeded in using DDP in order to design the optimal trajectory that allows the robot to start from a downward condition and to then draw the circumference. We did this avoiding peaks in the profiles of position and velocity and trying to be precise in the tracking of the position;

Task 3: In the third task we were able to design, through LQR method, a controller that allows the manipulator to follow the optimal trajectory designed in the previous task. This can be done also in case of a different initialization, which makes our controller robust;

Task 4: In the fourth and last task we defined the model of the manipulator in Simscape Multibody and used it to generate a visualization of the controller built in the third task. This allows us to check that the manipulator actually follows the desired trajectory in the workspace.

Bibliography

1. Notes and slides of the Optimal Control course of Professor Giuseppe Notarstefano and Professor Lorenzo Sforni at University of Bologna;
2. Notes and slides of the Industrial Robotics course of Professor Claudio Melchiorri at University of Bologna.