UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Optimal Control

Optimal Control of a Robot Manipulator

Professor: **Giuseppe Notarstefano**

Students: **Luca Santoro**
Armando Spennato

Academic year 2021/2022

Abstract

A robot manipulator is an electronically controlled mechanism that performs different tasks by interacting with external environment. These kind of robots are extensively used in the industrial manufacturing sector. Moreover, the study of robotic manipulators mainly consists in defining the positions and orientations of the various links of which they are made up. In this project we deal with a simple 2-DOF manipulator which can be seen as an example of a non-linear system. In particular, we will deal with the design and implementation of an optimal control law.

Contents

Introduction

In the following project particular attention is given to the design and implementation of an optimal control law for a 2-DOF robot. The project consists of four main steps:

- **Task 0 - Problem Setup :**
  Discretization of the manipulator dynamics and writing the *Dynamics.m* function;
- **Task 1 - Trajectory Exploration :**
  Design of an optimal trajectory to move from one equilibrium configuration to another, using the *DDP* algorithm;
- **Task 2 - Trajectory Optimization :**
  The same robotic manipulator is used to draw a custom curve to be followed by the end effector within its workspace. Furthermore, the corresponding optimal trajectory is defined in the joint space by using the *DDP* algorithm;
- **Task 3 - Trajectory Tracking :**
  Linearization of the model around the optimal trajectory obtained in the previous point and definition of the optimal feedback controller to perform trajectory tracking, through *LQR* algorithm;
- **Task 4 - Animation:**
  Designing an animation using *Simscape Multibody*, which provides a multibody simulation environment for 3D mechanical systems such as robots, to visualize the results obtained;

Chapter 1

Task 0 - Problem Setup

In this chapter all the main mechanical parameters of our robot will be defined; in figure 1.1 it is possible to see a representation of manipulator realized using software *PTC Creo Parametric*. In particular, the principal target is to define *Matlab* functions necessary to get the dynamics of our 2-DOF robot arm.

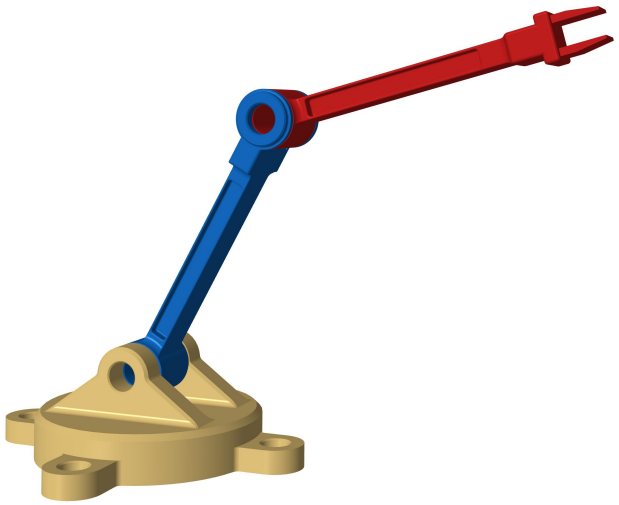

Figure 1.1: 2-DOF Robot Arm.

1.1 Matlab script *GenerateDynamics.m*

Our first aim is to describe how the system evolves in time. Therefore, to achieve this task we need to define the dynamics of the manipulator. For this purpose we create and execute a Matlab script called **GenerateDynamics.m** which allows to generate the Matlab function **Dynamics.m**. In

particular, to write this file we exploit symbolic math toolbox to build all the computations considering the state vector $\boldsymbol{x}$, the input vector $\boldsymbol{u}$, and $\boldsymbol{p}$ vector as symbolic elements. At the end of script, through matlabFunction (..,...,{..}), we generate the Dynamics.m function. Now we will see all the necessary elements we used in our script to define the dynamics of our robot arm.

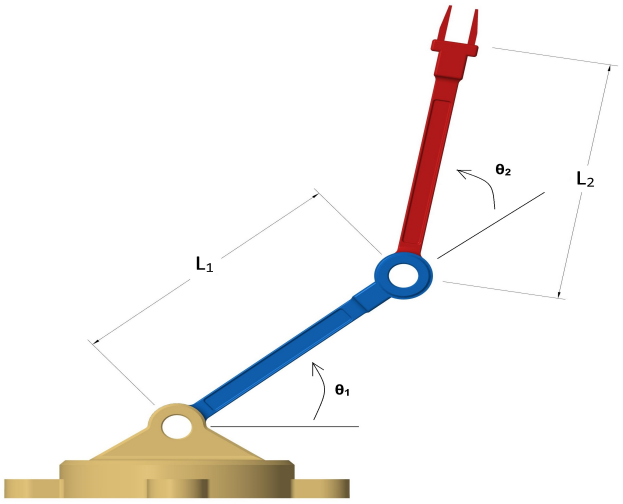

Figure 1.2: Structure of the 2-DOF Robot Arm.

Firstly, in order to describe the configuration of our manipulator, as we can see in figure 1.2, we define the joint variable vector $\boldsymbol{q}$ that collect the parameters θ_1 and θ_2, so the rotation angles of the two joints:

$$\boldsymbol{q} = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} \in \mathbb{R}^2$$

Secondly, another important step is to define the actuation of the system, more precisely the torques (inputs) to be applied to the two joints. For this reason we define the control torque vector $\boldsymbol{\tau}$ that collect our two inputs τ_1 and τ_2.

$$\boldsymbol{\tau} = \begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix} \in \mathbb{R}^2$$

In conclusion, in order to define the dynamic model we need to determine three important matrices: the inertia matrix $\boldsymbol{M(q)} \in \mathbb{R}^{2\times2}$, the matrix $\boldsymbol{C(q,\dot{q})} \in \mathbb{R}^{2\times2}$ and the matrix $\boldsymbol{g(q)} \in \mathbb{R}^2$ that represents gravitational actions on the links of our manipulator.

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

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

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

In the table 1.1 it is possible to see all the mechanical parameters of the robot that we have set in our script. In particular the masses, the inertias, the lengths and the positions of the centers of mass of the two links.

Parameters		
		Matlab
m_1	2 [Kg]	mm1
m_2	2 [Kg]	mm2
J_1	0.5 $[Kg \cdot m^2]$	JJ1
J_2	0.5 $[Kg \cdot m^2]$	JJ2
l_1	1 [m]	ll1
l_2	1 [m]	ll2
r_1	0.5 [m]	rr1
r_2	0.5 [m]	rr2

Table 1.1: Mechanical parameters of the 2-DOF Robot Arm.

Once all these aspects have been defined it is possible to write the continuous dynamic model of our robot that is given by the following equation:

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

We want to work with a discrete dynamical system because in this way we are able to collect the informations in vectors, and consequently, given the state samples and input samples at time t, we can understand the behaviour of the state sample at time t+1. In order to do that we need to use the state-space representation, for this reason we define the state vector $\boldsymbol{x}$(t) and the input vector $\boldsymbol{u}$(t) :

$$\boldsymbol{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} = \begin{bmatrix} \theta_1(t) \\ \dot{\theta}_1(t) \\ \theta_2(t) \\ \dot{\theta}_2(t) \end{bmatrix} \in \mathbb{R}^4$$

$$\boldsymbol{u}(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix} = \begin{bmatrix} \tau_1(t) \\ \tau_2(t) \end{bmatrix} \in \mathbb{R}^2$$

As we can see there are 4 states and 2 inputs.
Consequently, we can rewrite the joint variables and their derivatives as follows:

$$\boldsymbol{q}(t) := \begin{bmatrix} \theta_1(t) \\ \theta_2(t) \end{bmatrix} = \begin{bmatrix} x_1(t) \\ x_3(t) \end{bmatrix} \in \mathbb{R}^2$$

$$\dot{\boldsymbol{q}}(t) := \begin{bmatrix} \dot{\theta}_1(t) \\ \dot{\theta}_2(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ x_4(t) \end{bmatrix} \in \mathbb{R}^2$$

$$\ddot{\boldsymbol{q}}(t) := \begin{bmatrix} \ddot{\theta}_1(t) \\ \ddot{\theta}_2(t) \end{bmatrix} = \begin{bmatrix} \dot{x}_2(t) \\ \dot{x}_4(t) \end{bmatrix} \in \mathbb{R}^2$$

Exploiting the state-space representation, the dynamics of our model becomes the following:

$$\dot{x}_1(t) = x_2(t)$$

$$\dot{x}_3(t) = x_4(t)$$

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

As a last step we implement discretization that allows to obtain the discrete time dynamics. The simplest procedure to approximate a continuous dynamic system with a discrete one is the application of Forward Euler method, that is defined as follows:

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

From the previous we obtain the discretized continuous time dynamics by $\delta \in \mathbb{R}$ step size (sampling time).

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

In explicitly way :

$$\begin{bmatrix} x_{1,t+1} \\ x_{2,t+1} \\ x_{3,t+1} \\ x_{4,t+1} \end{bmatrix} = \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} + \delta \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \dot{x}_3(t) \\ \dot{x}_4(t) \end{bmatrix}$$

In other words we get the evolution in time of the discretized system:

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

The solutions of the obtained system approximate the solutions of the continuous system. The difference between the two solutions, i.e. the discretization error, is a rapidly increasing function with the step size δ. For this reason we have to select a small step size, in particular, we have chosen $\delta = 10^{-3}$. Since we have to implement the DDP algorithm we need to compute the gradient and the hessian of the system dynamics $f(x, u)$, that is a vector field:

$$f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$$

Therefore, we have a collection of scalar functions:

$$\begin{bmatrix} f_1(x,u) \\ \vdots \\ f_n(x,u) \end{bmatrix} \in \mathbb{R}^n$$

In particular, considering each single term:

$$f_i(x,u) : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R} \qquad \forall \quad i = 1, \dots, n$$

In our case: $n = 4$ and $m = 2$.
As a first step we compute the gradient $\nabla f(x, u)$ (that is the collection of the gradient of all scalar function f_i with i from 1 to n). This can be seen as the sum of two blocks $\nabla_x f(x, u)$ gradient of f with respect to x and $\nabla_u f(x, u)$ gradient of f with respect to u.

$$\nabla f(x,u) = \begin{bmatrix} \nabla_x f(x,u) \\ \nabla_u f(x,u) \end{bmatrix} \in \mathbb{R}^{(n+m)\times n}$$

$$\nabla_x f(x,u) = [\nabla_x f_1(x,u), \nabla_x f_2(x,u), \nabla_x f_3(x,u), \nabla_x f_4(x,u)] =$$
$$= \begin{bmatrix} \frac{\partial f_1(x,u)}{\partial x_1} & \frac{\partial f_2(x,u)}{\partial x_1} & \frac{\partial f_3(x,u)}{\partial x_1} & \frac{\partial f_4(x,u)}{\partial x_1} \\ \frac{\partial f_1(x,u)}{\partial x_2} & \frac{\partial f_2(x,u)}{\partial x_2} & \frac{\partial f_3(x,u)}{\partial x_2} & \frac{\partial f_4(x,u)}{\partial x_2} \\ \frac{\partial f_1(x,u)}{\partial x_3} & \frac{\partial f_2(x,u)}{\partial x_3} & \frac{\partial f_3(x,u)}{\partial x_3} & \frac{\partial f_4(x,u)}{\partial x_3} \\ \frac{\partial f_1(x,u)}{\partial x_4} & \frac{\partial f_2(x,u)}{\partial x_4} & \frac{\partial f_3(x,u)}{\partial x_4} & \frac{\partial f_4(x,u)}{\partial x_4} \end{bmatrix} \in \mathbb{R}^{n\times n}$$

$$\nabla_u f(x,u) = [\nabla_u f_1(x,u), \nabla_u f_2(x,u), \nabla_u f_3(x,u), \nabla_u f_4(x,u)] =$$
$$= \begin{bmatrix} \frac{\partial f_1(x,u)}{\partial u_1} & \frac{\partial f_2(x,u)}{\partial u_1} & \frac{\partial f_3(x,u)}{\partial u_1} & \frac{\partial f_4(x,u)}{\partial u_1} \\ \frac{\partial f_1(x,u)}{\partial u_2} & \frac{\partial f_2(x,u)}{\partial u_2} & \frac{\partial f_3(x,u)}{\partial u_2} & \frac{\partial f_4(x,u)}{\partial u_2} \end{bmatrix} \in \mathbb{R}^{m\times n}$$

The previous matrices are important because if we transpose them, they coincide with the linearization matrices. Given the dynamic (Discrete-time

system) $x_{t+1} = f(x_t, u_t)$ and a trajectory, the linearization is given by the following liner time-varying system:

$$\triangle x_{t+1} = A_t \triangle x_t + B_t \triangle u_t$$

Where A_t is state matrix of linearization and B_t is input state matrix of linearization, with $t \in \mathbb{N}_0$. They are show below:

$$A_t = \nabla_x f(x,u)^T = \begin{bmatrix} \frac{\partial f_1(x,u)}{\partial x_1} & \frac{\partial f_1(x,u)}{\partial x_2} & \frac{\partial f_1(x,u)}{\partial x_3} & \frac{\partial f_1(x,u)}{\partial x_4} \\ \frac{\partial f_2(x,u)}{\partial x_1} & \frac{\partial f_2(x,u)}{\partial x_2} & \frac{\partial f_2(x,u)}{\partial x_3} & \frac{\partial f_2(x,u)}{\partial x_4} \\ \frac{\partial f_3(x,u)}{\partial x_1} & \frac{\partial f_3(x,u)}{\partial x_2} & \frac{\partial f_3(x,u)}{\partial x_3} & \frac{\partial f_3(x,u)}{\partial x_4} \\ \frac{\partial f_4(x,u)}{\partial x_1} & \frac{\partial f_4(x,u)}{\partial x_2} & \frac{\partial f_4(x,u)}{\partial x_3} & \frac{\partial f_4(x,u)}{\partial x_4} \end{bmatrix} \in \mathbb{R}^{n \times n}$$

$$B_t = \nabla_u f(x,u)^T = \begin{bmatrix} \frac{\partial f_1(x,u)}{\partial u_1} & \frac{\partial f_1(x,u)}{\partial u_2} \\ \frac{\partial f_2(x,u)}{\partial u_1} & \frac{\partial f_2(x,u)}{\partial u_2} \\ \frac{\partial f_3(x,u)}{\partial u_1} & \frac{\partial f_3(x,u)}{\partial u_2} \\ \frac{\partial f_4(x,u)}{\partial u_1} & \frac{\partial f_4(x,u)}{\partial u_2} \end{bmatrix} \in \mathbb{R}^{n \times m}$$

Another important aspect concerns the tensor products which we need for DDP algorithm implementation. More precisely we are interested in calculating the tensor product between the tensor representing the hessian of the function f and the costate vector $p \in \mathbb{R}^n$, which is an important vector used to calculate the control law. It is convenient to calculate the tensor products directly within the Dynamics.m function, which is generated with the script we are considering. In particular we have to define the following quantities:

$$\nabla^2_{xx} f \cdot p \qquad \nabla^2_{uu} f \cdot p \qquad \nabla^2_{xu} f \cdot p$$

To find the previous tensor product we first find the hessian matrices $\nabla^2_{xx} f, \nabla^2_{uu} f, \nabla^2_{xu} f$. The function f is a vector field and this implies that each hessian is a tensor, i.e a set of matrices, this means that the complexity increases as we can see below:

$$\nabla^2_{xx} f(x,u) = [\nabla_{xx} f_1(x,u), \nabla_{xx} f_2(x,u), \nabla_{xx} f_3(x,u), \nabla_{xx} f_4(x,u)] =$$

$$= \begin{bmatrix} \begin{bmatrix} \frac{\partial^2 f_1(x,u)}{\partial x_1^2} & \frac{\partial^2 f_1(x,u)}{\partial x_1 \partial x_2} & \frac{\partial^2 f_1(x,u)}{\partial x_1 \partial x_3} & \frac{\partial^2 f_1(x,u)}{\partial x_1 \partial x_4} \\ \frac{\partial^2 f_1(x,u)}{\partial x_2 \partial x_1} & \frac{\partial^2 f_1(x,u)}{\partial x_2^2} & \frac{\partial^2 f_1(x,u)}{\partial x_2 \partial x_3} & \frac{\partial^2 f_1(x,u)}{\partial x_2 \partial x_4} \\ \frac{\partial^2 f_1(x,u)}{\partial x_3 \partial x_1} & \frac{\partial^2 f_1(x,u)}{\partial x_3 \partial x_2} & \frac{\partial^2 f_1(x,u)}{\partial x_3^2} & \frac{\partial^2 f_1(x,u)}{\partial x_3 \partial x_4} \\ \frac{\partial^2 f_1(x,u)}{\partial x_4 \partial x_1} & \frac{\partial^2 f_1(x,u)}{\partial x_4 \partial x_2} & \frac{\partial^2 f_1(x,u)}{\partial x_4 \partial x_3} & \frac{\partial^2 f_1(x,u)}{\partial x_4^2} \end{bmatrix}, \begin{bmatrix} \frac{\partial^2 f_2(x,u)}{\partial x_1^2} & \dots & \dots & \dots \\ \dots & \ddots & \dots & \dots \\ \dots & \dots & \ddots & \dots \\ \dots & \dots & \dots & \ddots \end{bmatrix}, \\ \begin{bmatrix} \frac{\partial^2 f_3(x,u)}{\partial x_1^2} & \dots & \dots & \dots \\ \dots & \ddots & \dots & \dots \\ \dots & \dots & \ddots & \dots \\ \dots & \dots & \dots & \ddots \end{bmatrix}, \\ \begin{bmatrix} \frac{\partial^2 f_4(x,u)}{\partial x_1^2} & \dots & \dots & \dots \\ \dots & \ddots & \dots & \dots \\ \dots & \dots & \ddots & \dots \\ \dots & \dots & \dots & \ddots \end{bmatrix} \end{bmatrix}$$

$$\nabla^2_{uu} f(x,u) = [\nabla_{uu} f_1(x,u), \nabla_{uu} f_2(x,u), \nabla_{uu} f_3(x,u), \nabla_{uu} f_4(x,u)] =$$

$$= \begin{bmatrix} \begin{bmatrix} \frac{\partial^2 f_1(x,u)}{\partial u_1^2} & \frac{\partial^2 f_1(x,u)}{\partial u_1 \partial u_2} \\ \frac{\partial^2 f_1(x,u)}{\partial u_2 \partial u_1} & \frac{\partial^2 f_1(x,u)}{\partial u_2^2} \end{bmatrix}, \begin{bmatrix} \frac{\partial^2 f_2(x,u)}{\partial u_1^2} & \frac{\partial^2 f_2(x,u)}{\partial u_1 \partial u_2} \\ \frac{\partial^2 f_2(x,u)}{\partial u_2 \partial u_1} & \frac{\partial^2 f_2(x,u)}{\partial u_2^2} \end{bmatrix}, \begin{bmatrix} \frac{\partial^2 f_3(x,u)}{\partial u_1^2} & \frac{\partial^2 f_3(x,u)}{\partial u_1 \partial u_2} \\ \frac{\partial^2 f_3(x,u)}{\partial u_2 \partial u_1} & \frac{\partial^2 f_3(x,u)}{\partial u_2^2} \end{bmatrix}, \\ \begin{bmatrix} \frac{\partial^2 f_4(x,u)}{\partial u_1^2} & \frac{\partial^2 f_4(x,u)}{\partial u_1 \partial u_2} \\ \frac{\partial^2 f_4(x,u)}{\partial u_2 \partial u_1} & \frac{\partial^2 f_4(x,u)}{\partial u_2^2} \end{bmatrix} \end{bmatrix}$$

$$\nabla^2_{xu} f(x,u) = [\nabla_{xu} f_1(x,u), \nabla_{xu} f_2(x,u), \nabla_{xu} f_3(x,u), \nabla_{xu} f_4(x,u)] =$$

$$= \begin{bmatrix} \begin{bmatrix} \frac{\partial^2 f_1(x,u)}{\partial x_1 \partial u_1} & \frac{\partial^2 f_1(x,u)}{\partial x_1 \partial u_2} \\ \frac{\partial^2 f_1(x,u)}{\partial x_2 \partial u_1} & \frac{\partial^2 f_1(x,u)}{\partial x_2 \partial u_2} \\ \frac{\partial^2 f_1(x,u)}{\partial x_3 \partial u_1} & \frac{\partial^2 f_1(x,u)}{\partial x_3 \partial u_2} \\ \frac{\partial^2 f_1(x,u)}{\partial x_4 \partial u_1} & \frac{\partial^2 f_1(x,u)}{\partial x_4 \partial u_2} \end{bmatrix}, \begin{bmatrix} \frac{\partial^2 f_2(x,u)}{\partial x_1 \partial u_1} & \frac{\partial^2 f_2(x,u)}{\partial x_1 \partial u_2} \\ \frac{\partial^2 f_2(x,u)}{\partial x_2 \partial u_1} & \frac{\partial^2 f_2(x,u)}{\partial x_2 \partial u_2} \\ \frac{\partial^2 f_2(x,u)}{\partial x_3 \partial u_1} & \frac{\partial^2 f_2(x,u)}{\partial x_3 \partial u_2} \\ \frac{\partial^2 f_2(x,u)}{\partial x_4 \partial u_1} & \frac{\partial^2 f_2(x,u)}{\partial x_4 \partial u_2} \end{bmatrix}, \begin{bmatrix} \frac{\partial^2 f_3(x,u)}{\partial x_1 \partial u_1} & \frac{\partial^2 f_3(x,u)}{\partial x_1 \partial u_2} \\ \frac{\partial^2 f_3(x,u)}{\partial x_2 \partial u_1} & \frac{\partial^2 f_3(x,u)}{\partial x_2 \partial u_2} \\ \frac{\partial^2 f_3(x,u)}{\partial x_3 \partial u_1} & \frac{\partial^2 f_3(x,u)}{\partial x_3 \partial u_2} \\ \frac{\partial^2 f_3(x,u)}{\partial x_4 \partial u_1} & \frac{\partial^2 f_3(x,u)}{\partial x_4 \partial u_2} \end{bmatrix}, \\ \begin{bmatrix} \frac{\partial^2 f_4(x,u)}{\partial x_1 \partial u_1} & \frac{\partial^2 f_4(x,u)}{\partial x_1 \partial u_2} \\ \frac{\partial^2 f_4(x,u)}{\partial x_2 \partial u_1} & \frac{\partial^2 f_4(x,u)}{\partial x_2 \partial u_2} \\ \frac{\partial^2 f_4(x,u)}{\partial x_3 \partial u_1} & \frac{\partial^2 f_4(x,u)}{\partial x_3 \partial u_2} \\ \frac{\partial^2 f_4(x,u)}{\partial x_4 \partial u_1} & \frac{\partial^2 f_4(x,u)}{\partial x_4 \partial u_2} \end{bmatrix} \end{bmatrix}$$

To construct the tensor product we can perform a linear combination between the hessian of each component of the dynamic function and the k-th component of the vector p, in the following way:

$$\nabla^2_{xx}(x,u) \cdot p = \sum_{k=0}^{n} \nabla^2_{xx}{}^{k} p_k$$

$$\nabla^2_{uu}(x,u) \cdot p = \sum_{k=0}^{n} \nabla^2_{uu}{}^{k} p_k$$

$$\nabla^2_{xu}(x,u) \cdot p = \sum_{k=0}^{n} \nabla^2_{xu}{}^{k} p_k$$

At the end of the script, once all parameters and computations have been defined, we exploit the command *matlabFunction* $(x_{t+1}, \nabla x f, \nabla u f, \nabla^2_{xx} f \cdot p, \nabla^2_{uu} f \cdot p, \nabla^2_{xu} f \cdot p, \{x, u, p\})$ to generate the Dynamics.m function. This aspect will be better explained in the following paragraph.

1.1.1 Matlab function *Dynamics.m*

Dynamics.m is the function generated by the file **GenerateDynamics.m** described in the previous paragraph. More precisesly, the function is generated through *matlabFunction*(..,...,{..}) command. The latter generates a Matlab function which is able to implement all the computations in a non symbolic way. Practically, this means that the generated Matlab function, which is our Dynamics.m, computes all the quantities in a numerical way, substituting input values to their symbolic counterparts.

To sum up, the function takes as input three vectors $\boldsymbol{x}$, $\boldsymbol{u}$ and $\boldsymbol{p}$ and returns as output the next state x_{t+1}, the gradient of the dynamics $\nabla_x f$,$\nabla_u f$

and the tensor products $\nabla^2_{xx} f \cdot p$, $\nabla^2_{uu} f \cdot p$, $\nabla^2_{xu} f \cdot p$. These outputs will be usefull for our tasks, for example in the implementation of the DDP algorithm. In the table 1.2 it is possible to see Dynamic.m function's schematization:

Dynamics.m						
Input:	x_t	u_t	p_t			
0utput:	x_{t+1}	$\nabla_x f$	$\nabla_u f$	$\nabla^2_{xx} f \cdot p$	$\nabla^2_{uu} f \cdot p$	$\nabla^2_{xu} f \cdot p$

Table 1.2: Matlab function *Dynamics.m.*

1.2 Matlab function *Stage_Cost.m*

Optimal Control task is trajectory generation (Tracking) or trajectory exploration. In this project we face a trajectory tracking problem, so we need to define a cost function. In general, this function represents a penalty or better the cost "to be payed" for a chosen trajectory; in our case this cost corresponds to the distance from a desired trajectory. In particular we choose a standard cost function that is a quadratic function with matrices Q_t, R_t and Q_T. These are called weight matrices, with the following properties:

- $Q_t \in \mathbb{R}^{n \times n}$ Positive Semi-definite matrix
- $R_t \in \mathbb{R}^{m \times m}$ Positive Definite matrix
- $Q_T \in \mathbb{R}^{n \times n}$ Positive Semi-definite matrix

The cost function is defined as:

$$l(\mathbf{x}, \mathbf{u}) := \sum_{t=0}^{T-1} \frac{1}{2} ||x_t - x_t^{des}||^2 Q_t + \frac{1}{2} ||u_t - u_t^{des}||^2 R_t + \frac{1}{2} ||x_T - x_T^{des}||^2 Q_T \quad (1.1)$$

Where:

$$l(x, u) = \sum_{t=0}^{T-1} l_t(x_t, u_t) = \sum_{t=0}^{T-1} (\frac{1}{2} ||x_t - x_t^{des}||^2 Q_t + \frac{1}{2} ||u_t - u_t^{des}||^2 R_t) \quad (1.2)$$

$$l_T(x_T) = \frac{1}{2} ||x_T - x_T^{des}||^2 Q_T \quad (1.3)$$

We can see the cost function as the sum of two terms:

- Stage Cost $l(x, u)$:

$$l_t : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$$

- Term Cost $l_T(x_T)$:

$$l_T : \mathbb{R}^n \to \mathbb{R}$$

The Term Cost is implemented in the subsequent **Term_Cost.m** function. Whereas, through **Stage_cost.m** Matlab function we define the Stage Cost function to understand how far we are from the desired behaviour of our system. As we seen in the previous equation 1.2 the Stage Cost is defined as a weighted square norm of errors on the state and input errors. For this reason the function takes as inputs: x_t, u_t, x_t^{des},u_t^{des}; where the state-input curve (x_t^{des},u_t^{des}) with $t \in \mathbb{N}_0$, represent the desired behaviour of the system and does not satisfy the dynamics. The function is schematized in the table 1.3.

As in the Dynamics.m function case, this function allows to compute the Gradient and the Hessian of the Stage Cost. As a result, all these things are useful in the implementation of our DDP algorithm. (little a and little b parameters that we need to compute the algorithm).

Stage_Cost.m						
Input:	x_t	u_t	x_t^{des}	u_t^{des}	parameters	
0utput:	l_t	$\nabla_x l_t$	$\nabla_u l_t$	$\nabla^2_{xx} l_t$	$\nabla^2_{uu} l_t$	$\nabla^2_{xu} l_t$

Table 1.3: Matlab function *Stage_Cost.m.*

1.3 Matlab function *Term_Cost.m*

With this function we implement the Terminal Cost that is the piece of the cost that we consider at time T. In particular, the term cost is defined as follows:

$$l_T(x_T) = \frac{1}{2}||x_T - x_T^{des}||^2 Q_T \tag{1.4}$$

Therefore, this function takes as input x_T, x_T^{des}, which is a reference for the state trajectory at instant T, and the variable parameters used to calculate the weight matrix Q_T. The function is schematized in table 1.4. We can observe that as the previous functions, also *Term_cost.m* function gives as output not only the cost, but also the Gradient and the Hessian, which will be useful when we will implement the DDP algorithm.

Term_Cost.m			
Input:	x_T	X_T^{des}	parameters
0utput:	l_T	$\nabla_x l_T$	$\nabla_{xx}^2 l_T$

Table 1.4: Matlab function *Term_Cost.m.*

Chapter 2

Task 1 - Trajectory Exploration

In this section we want to design an optimal trajectory that our robot will follow to move from one equilibrium configuration to another, as shown in figure 2.1, using the DDP algorithm; The achievement of this task is a relevant aspect in pick and place operations in order to move industrial manipulators within automated warehouses.

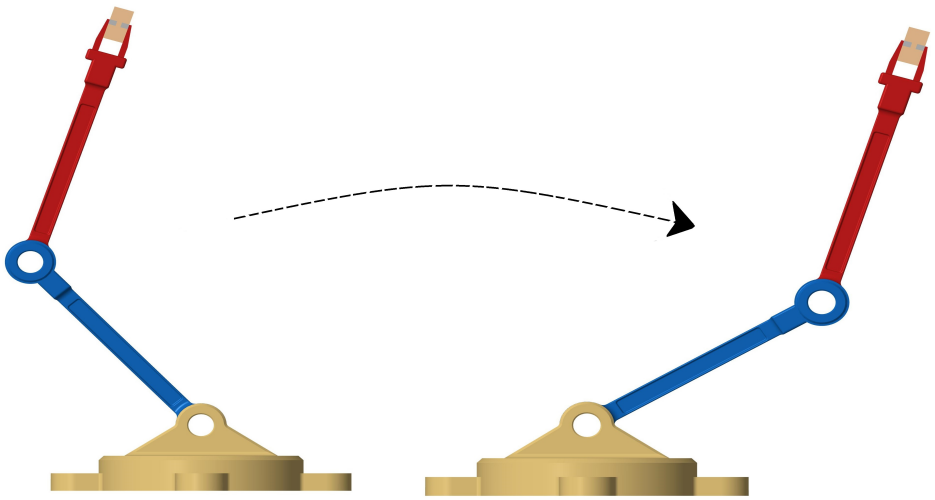

Figure 2.1: Two examples of Robot Arm 2-DOF configuration.

2.1 Definition of parameters

Talking more in details, considering Matlab script **Task_1.m**, all necessary aspects to achieve our goal are defined. As seen in the previous chapter we

have discretized the dynamics using a step size (Sampling time) $\delta = 10^{-3}$. It is important to observe that the number of the samples is given by:

$$T = \frac{t_f}{\delta} = \frac{30}{10^{-3}} = 30000$$

Where t_f is the length of time window (The final time). Therefore, we can observe that if the length of the time window increases, the number of samples increases accordingly. In general, it is useful to have longer time horizon in order to have more stable phases. The idea is to isolate the perturbations in the middle of the time horizon to let the system rest in equilibrium position for a longer time,this gives a numerical stability to the whole algorithm.

2.2 Weight matrices definition

We define the weight matrices that we will use to define our standard Cost Function. To apply DDP algorithm we need to define the cost function, as seen in the section 1.2 page 15, that is a quadratic function in which we find the weight matrices.

It is important to select the weights in order to obtain some desired behaviour (or performance), taking into account the following properties:

- Matrix **Q**: If we choose elements with high values we make the states converge to zero quickly.
- Matrix **R**: If we choose elements with high values we penalize the control.

Taking these details into account we define our matrices to track the position reference as accurately as possible. For this reason we set high values (1500) with regard to the cost related to the position and lower cost as concern velocities (100). Subsequently, We choose low cost for torques (0,005) in order to avoid numerical problems. At the end, the obtained matrices are the following:

$$\mathbf{Q} = \mathbf{Q_T} = \begin{bmatrix} 1500 & 0 & 0 & 0 \\ 0 & 100 & 0 & 0 \\ 0 & 0 & 1500 & 0 \\ 0 & 0 & 0 & 100 \end{bmatrix}$$

$$\boldsymbol{R} = \begin{bmatrix} 0,005 & 0 \\ 0 & 0,005 \end{bmatrix}$$

2.3 Definition of references

What we want is the movement of our robot between an initial configuration to a final one.

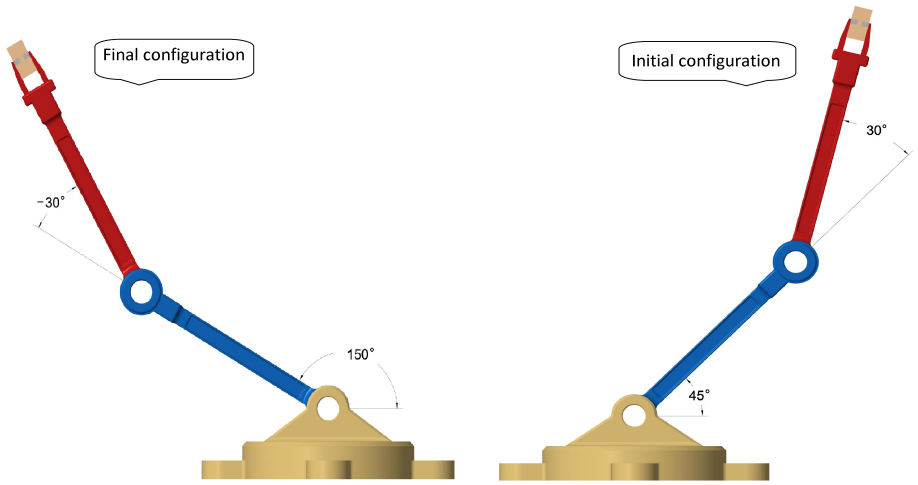

Figure 2.2: Initial and Final configuration of 2-DOF Robot Arm.

Therefore, to obtain this we have defined two configurations reference, showed in figure 2.2, and a step that connects the starting position $\boldsymbol{q}_{init}$ with the final one $\boldsymbol{q}_{final}$ as we can see below:

$$\boldsymbol{q}_{init} = \begin{bmatrix} \theta_{1,init} \\ \theta_{2,init} \end{bmatrix} = \begin{bmatrix} 45 \\ 30 \end{bmatrix} \quad [deg]$$

$$\boldsymbol{q}_{final} = \begin{bmatrix} \theta_{1,final} \\ \theta_{2,final} \end{bmatrix} = \begin{bmatrix} 150 \\ -30 \end{bmatrix} \quad [deg]$$

While, as concern the velocity reference we define a zero velocity since we want to pass from one equilibrium configuration to another with a slowly behaviour. This means that our torque reference is the torque that our manipulator need to balance the gravity.

$$\boldsymbol{x}_t^{des} = \begin{bmatrix} \theta_1^{des} \\ \dot{\theta}_1^{des} \\ \theta_2^{des} \\ \dot{\theta}_2^{des} \end{bmatrix} = \begin{bmatrix} 0,7854 \\ 0 \\ 0,5236 \\ 0 \end{bmatrix} \quad [rad] \qquad \forall t \in \left[1, \frac{T}{2}\right]$$

$$\boldsymbol{x}_t^{des} = \begin{bmatrix} \theta_1^{des} \\ \dot{\theta}_1^{des} \\ \theta_2^{des} \\ \dot{\theta}_2^{des} \end{bmatrix} = \begin{bmatrix} 2,6180 \\ 0 \\ -0,5236 \\ 0 \end{bmatrix} \quad [rad] \qquad \forall t \in \left]\frac{T}{2}, T\right]$$

In order to maintain the initial equilibrium configuration and then the final one, where $\dot{\boldsymbol{q}} = 0$ and $\ddot{\boldsymbol{q}} = 0$, our desired input $\boldsymbol{u}_t^{des}$ must balance the term $\boldsymbol{g}(\boldsymbol{q})$. As a result, $\boldsymbol{u}_t^{des}$ is:

$$\boldsymbol{u}_t^{des} = \boldsymbol{g}(q_{init}) = \begin{bmatrix} 23,3492 \\ 2,5390 \end{bmatrix} \quad [N \cdot m] \qquad \forall t \in \left[1, \frac{T}{2}\right]$$

$$\boldsymbol{u}_t^{des} = \boldsymbol{g}(q_{final}) = \begin{bmatrix} -30,3921 \\ -4,9050 \end{bmatrix} \quad [N \cdot m] \qquad \forall t \in \left]\frac{T}{2}, T\right]$$

All the references are shown below, in particular in figure 2.3 we have the desired evolution of the joint angles and in figure 2.4 we have the desired evolution of the torque of joints (input).

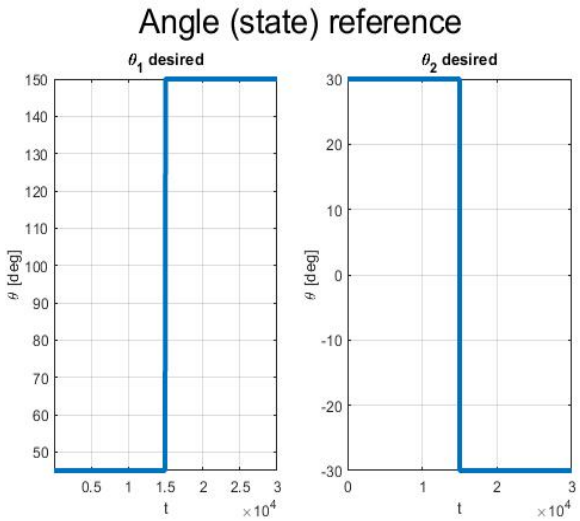

Figure 2.3: Desired evolution (step behaviour) of the angles of the joints.

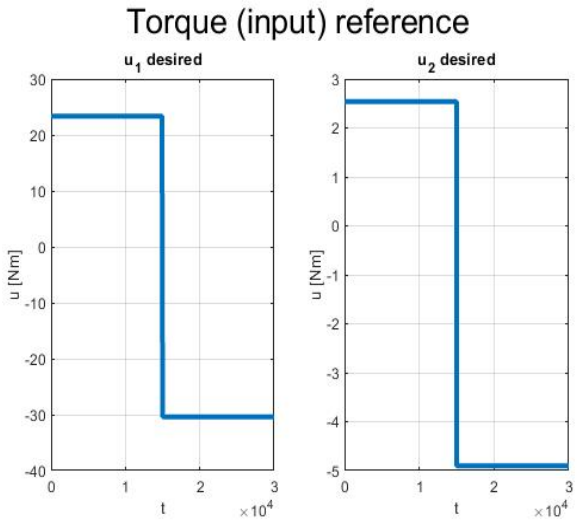

Figure 2.4: Desired evolution (step behaviour) of the torque (input) of the joints.

We want to find a trajectory of the system that is similar to the previous plots. It is important to notice that the reference is not a trajectory, but it just defines the desired behaviour.

2.4 DDP Implementation

The DDP method is an iterative procedure, for this reason, to apply it we must give a value to the initial iteration to initialize the algorithm, to do that we have chosen the Inverse Dynamics Control in order to not start from a trajectory too far from the final one. Practically, we have an initial trajectory of the system that is improved at each iteration, then this improved trajectory is applied to the input sequence in order to get a new trajectory for the next iteration. In the next paragraph we can see all the steps of our algorithm.

2.4.1 Initialization

To initialize our algorithm we exploit the *Inverse Dynamic Position Control* which is a basic control that allows to control ideally a manipulator, because we assume the exactly knowledge of the parameters of our system, but in real life we have modelling errors and disturbances, so is not possible to know exactly the parameters in real time [1]. It is based on the following two step:

1. We compensate the natural dynamics (non linear and coupled) of the

manipulator.

$$\tau = M(q)\ddot{q} + c(q,\dot{q})\dot{q} + g(q) = M(q)\ddot{q} + n(q,\dot{q}) \tag{2.1}$$

$$\tau = M(q)y + c(q,\dot{q})\dot{q} + g(q) = M(q)y + n(q,\dot{q}) \tag{2.2}$$

in this way we obtain:

$$\boxed{\ddot{q} = y} \tag{2.3}$$

That is a decoupled linear system, in figure 2.5 is possible to see the corresponding control scheme.

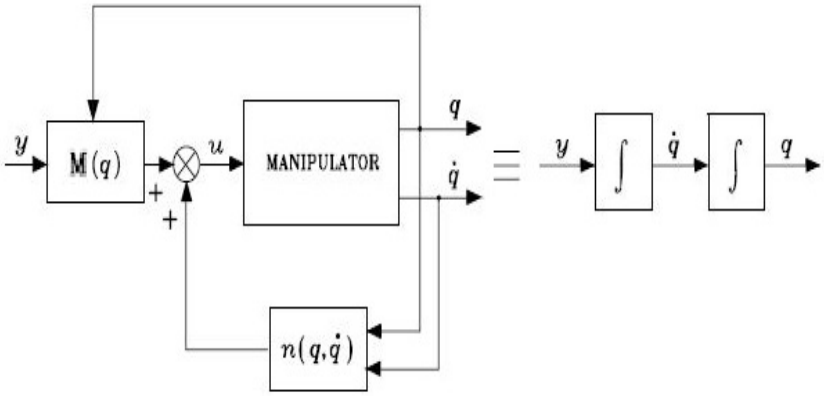

Figure 2.5: Exact linearization performed by inverse dynamics control.

2. We impose our desired dynamics since we have to linearize our system. We choose y in order to stabilize the whole system, as:

$$y = \ddot{q}_d + K_p\tilde{q} + K_D\dot{\tilde{q}} \tag{2.4}$$

Substituting into 2.3 we obtain the following homogeneous second-order differential equation, that describe an asymptotically stable system:

$$\ddot{\tilde{q}} + K_p\tilde{q} + K_D\dot{\tilde{q}} = 0 \tag{2.5}$$

We impose $\ddot{q}_d = 0$ since we are not interested in acceleration control. At the end the final control action is:

$$\boxed{\tau(\theta) = M(q)(K_p\tilde{q} + K_D\dot{\tilde{q}}) + C(q,\dot{q})\dot{q} + g(q)} \tag{2.6}$$

In figure 2.6 is possible to see the final control scheme.

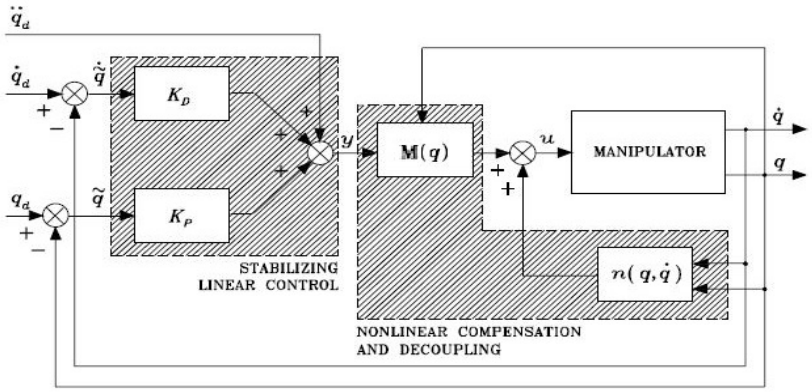

Figure 2.6: Block scheme of joint space inverse dynamics control.

The previous control is implemented with the Matlab function **controller.m**, that is schematized in the table 2.1.

Controller.m				
Input:	x_{des}	u_{des}	parameters	Task
Output:	x_{init}	u_{init}		

Table 2.1: Matlab function *Controller.m.*

It can be see that the function also accepts the task number as input since we need to chose different gain matrices for task 1 and task 2. In particular, to perform the tuning of our controller we used the table of figure 2.7 trying different gains values and evaluating the results of the Matlab plots each time in order to obtain the most suitable solution.

Parameter Increase	Rise time	Overshoot	Settling Time	Steady-state error
Kp ↑	↓	↑	Small Change	↓
Kd ↑	Small Change	↓	↓	Small Change

Figure 2.7: PD gain tuning table.

The first set of values is:

$$K_p = \begin{bmatrix} 80 & 0 \\ 0 & 80 \end{bmatrix}$$

$$K_D = \begin{bmatrix} 20 & 0 \\ 0 & 20 \end{bmatrix}$$

With these first values we have the following situation:

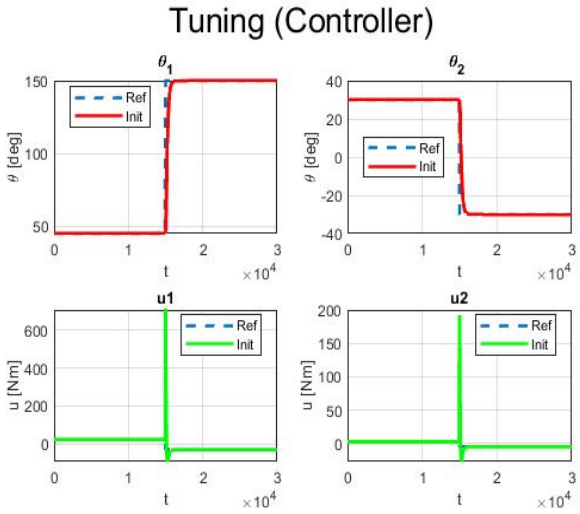

Figure 2.8: Initialization signals, first attempt.

Then, by decreasing the position gain K_p and the velocity gain K_D we consider a second set of values that are:

$$K_p = \begin{bmatrix} 25 & 0 \\ 0 & 25 \end{bmatrix}$$

$$K_D = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$$

It is possible to see in figure 2.9 that the input initialization presents an overshoot lower than the previous attempt, but in this way the angles reference tracking is less accurate:

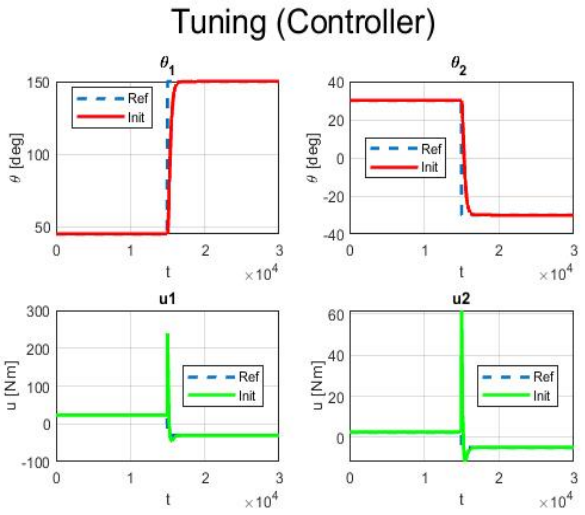

Figure 2.9: Initialization signals, final attempt.

2.4.2 Implementation

To implement the DDP algorithm we follow these points:

1. As first step we compute all quantities, which we need to map everything in a general LQ problem:

$$q_t^k = \nabla_x l_t(x_t^k, u_t^k)$$
$$q_T^k = \nabla l_T(x_T^k)$$
$$r_t^k = \nabla_u l_t((x_t^k, u_t^k))$$
$$Q_t^k = \nabla_{xx}^2 l_t(x_t^k, u_t^k) + \nabla_{xx}^2 f(x_t^k, u_t^k) \cdot p_{t+1}^k$$
$$Q_T^k = \nabla^2 l_T(x_T^k)$$
$$R_t^k - \nabla_{uu}^2 l_t(x_t^k, u_t^k) + \nabla_{uu}^2 f(x_t^k, u_t^k) \cdot p_{t+1}^k$$

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

   The previous quantities are calculated $\forall t \in [0; T-1]$.

2. Now, once all the previous quantities are available we can solve the LQ optimization problem for the current K:

The Feedback Gain:

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

The Feedforward Input Term:

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

These two terms are required to improve the trajectory (the input sequence and therefore the state trajectory). At this point we can calculate the following updates:

$$p_t^k = q_t^k + A_t^k p_{t+1}^k + {K_t^k}^T (R_t^K + {B_t^k}^T P_{t+1}^k B_t^k)^{-1} \sigma_t^k$$

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

using the following inizialization:

$$p_T^k = q_T^k$$

$$P_T^k = Q_T^k$$

3. We perform the forward simulation for $t = 0, ..., T-1$, practically we improve K while we improve x (state sequence) and u (input sequence). This third step is performed by running a "cloosed-loop system":

$$u_t^{k+1} = u_t^k + \gamma^k \sigma_t^k + K_t^k (x_t^{k+1} - x_t^k)$$
$$x_{t+1}^{k+1} = f(x_t^{k+1}, u_t^{k+1})$$

with:

$$x_0^{k+1} = x_{init} \quad given$$

As we can see we use the Armijo's rule whose idea is to use a step-size γ^k to modulate the feedforward term σ_t^k. The step-size can be constant or selected at each iteration, so we chose gamma_fix = 0.8 in the first case while in the second case we used Armijo. Moreover, we select as descent not all the perturbation applied to the input but only the norm of the feedforward term $||\sigma_t^k||$.

2.5 Results and considerations of Task 1

All the results obtained with the DDP implementation are shown below:

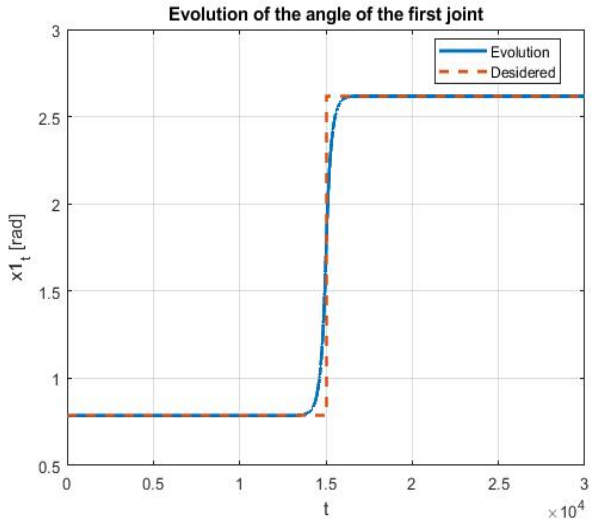

Figure 2.10: Evolution of the angle of the first joint.

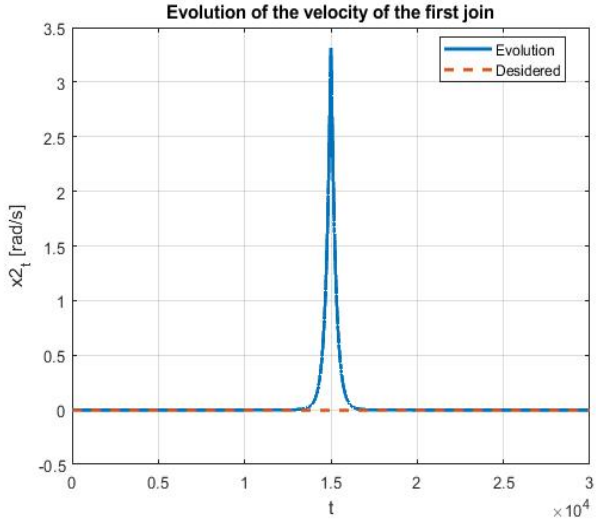

Figure 2.11: Evolution of the velocity of the first joint.

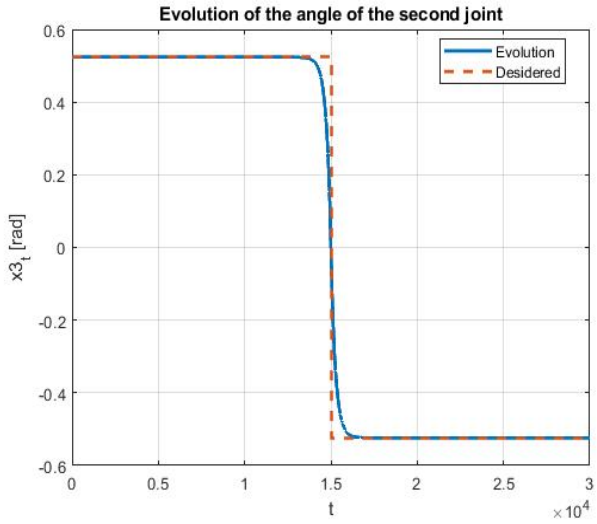

Figure 2.12: Evolution of the angle of the second joint.

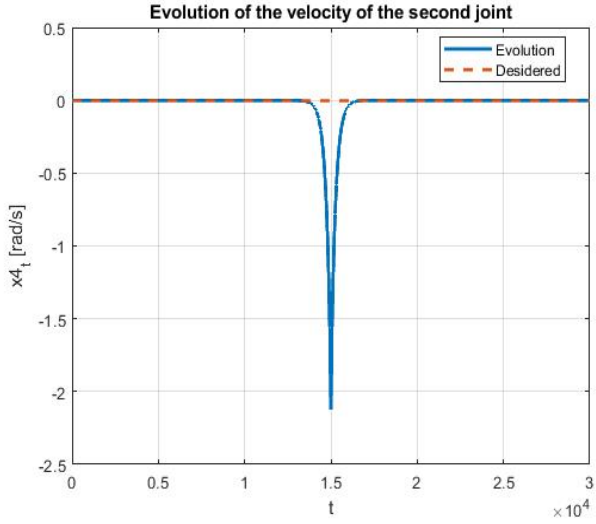

Figure 2.13: Evolution of the velocity of the second joint.

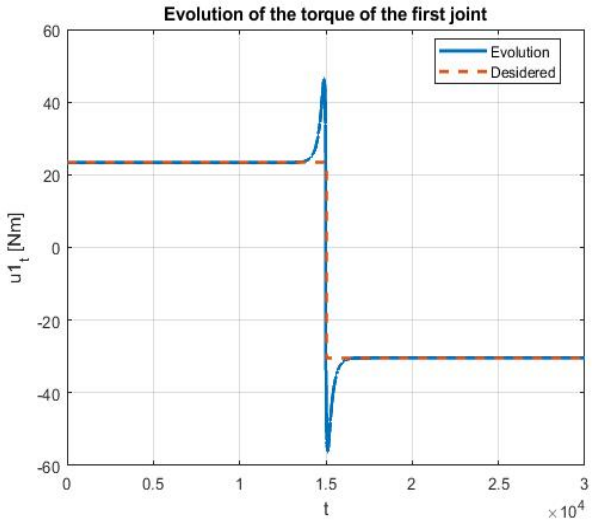

Figure 2.14: Evolution of the torque of the first joint.

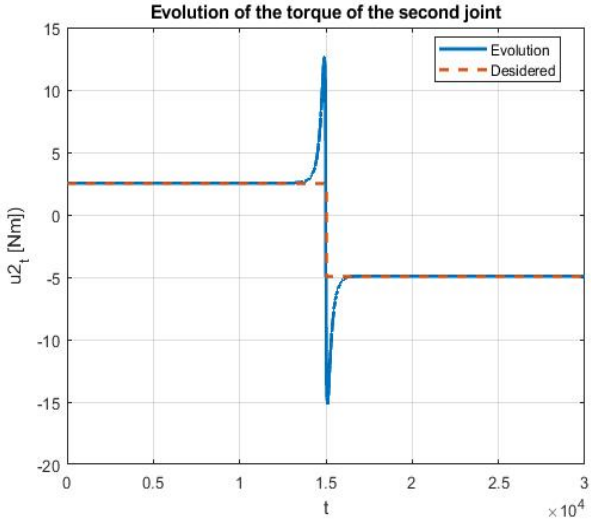

Figure 2.15: Evolution of the torque of the second joint.

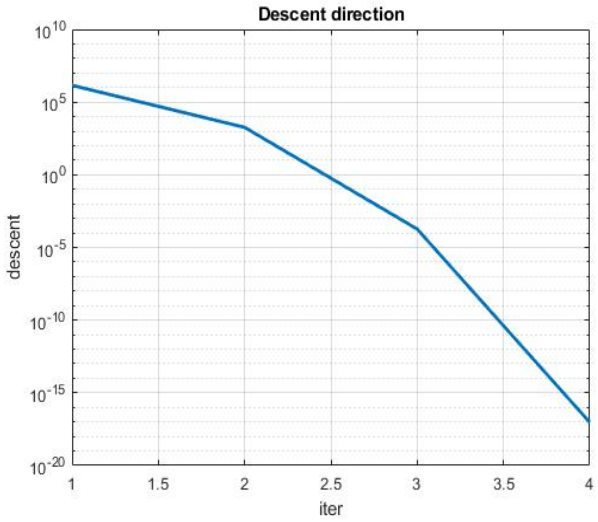

Figure 2.16: Descent direction plot.

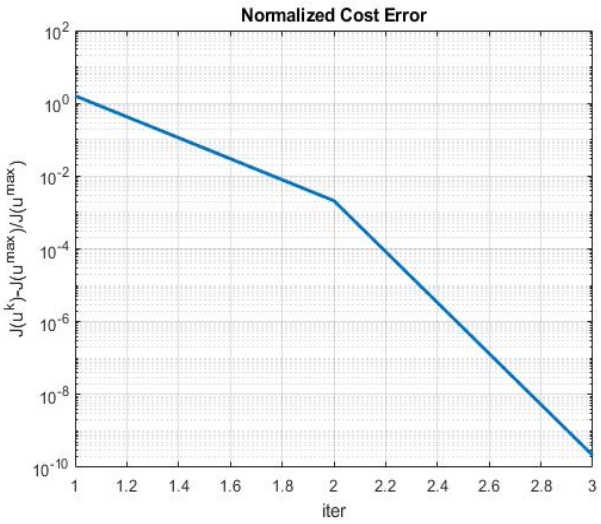

Figure 2.17: Evolution of the normalized cost error.

Chapter 3

Task 2 - Trajectory Optimization

In the second task, implemented in the Matlab script **Task_2.m**, we want to use the same robotic manipulator to draw a desired curve that has to be followed by the end-effector within the workspace. In particular, we define in the joint space a trajectory that we want to follow and we exploit again the DDP algorithm to obtain the optimal trajectory. In the figure 3.1. we can see the curve that we want to obtain:

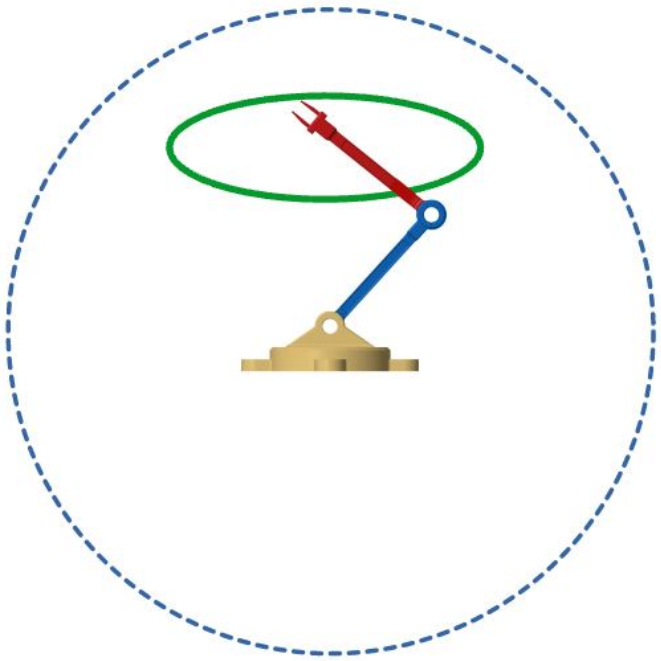

Figure 3.1: Shape definition.

3.1 Weights and parameters definition

Since we deal with the same system as first step we define again the dynamics parameters. In this case we chose higher costs for the state with respect to the previous case since we want to track the reference with high precision in order to obtain a precise sketch of the shape we want to obtain. As concern the inputs we chose low costs to avoid numerical issues.

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

$$\boldsymbol{R} = \begin{bmatrix} 0,0005 & 0 \\ 0 & 0,0005 \end{bmatrix}$$

3.2 Shape definition of the trajectory

Before defining the shape to draw we have implemented a piece of code that allows us to understand all points that the end-effector can reach within the workspace.The results are shown in figure 3.2. The reachable area depends on the length of the two links and relative angles θ_1,θ_2.

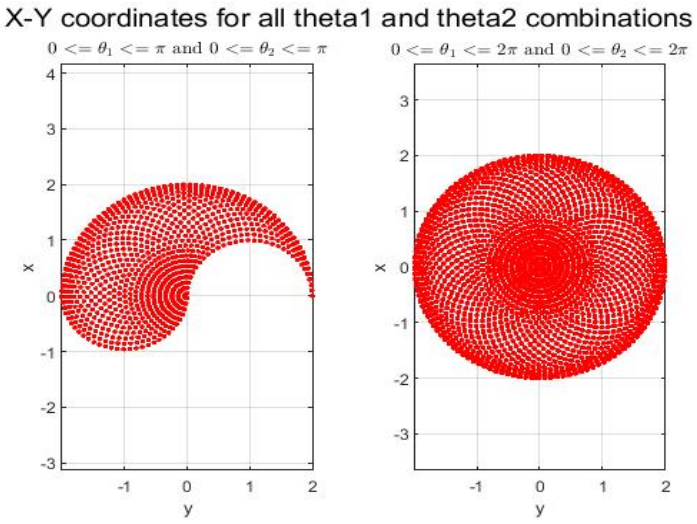

Figure 3.2: X,Y coordinates for all combination of θ_1 and θ_2.

Due to the length of the two links we have that the work area is a circle with a radius of 2 m around the origin. In choosing the shape of the curve to draw we must respect the previous constraint. We choose as curve an ellipse, that has the following equation:

$$\frac{(x^2-\alpha)^2}{a^2}+\frac{(y^2-\beta)^2}{b^2}=1 \tag{3.1}$$

That is the equation of a translated ellipse, where α and β are the coordinates of the center of the ellipse. We define:

$$\alpha = 0\ [m]; \quad \beta = 1,5\ [m]; \quad a = 1\ [m]; \quad b = 0,3\ [m];$$

Where a and b are respectively the half of the greater length and the less one. In conclusion, the chosen curve is shown in the figure 3.3.

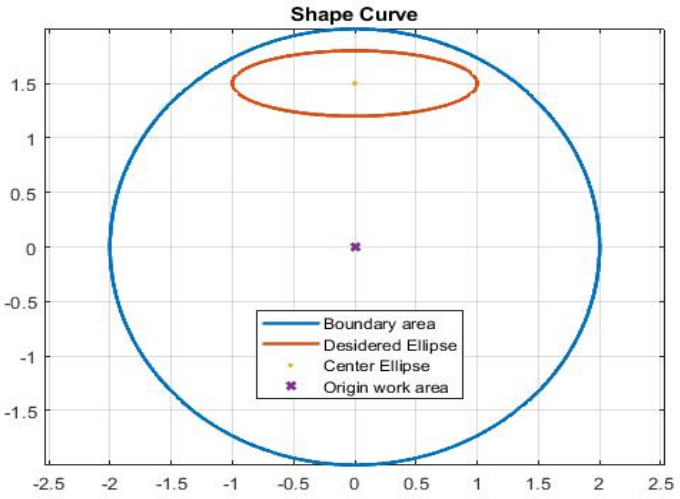

Figure 3.3: Ellipse chosen as a trajectory.

3.2.1 Inverse kinematic

For each point of the ellipse we solved the inverse kinematics for the 2-DOF manipulator, in order to get the angles in joint space corresponding to those points in the workspace. It is important that the points we want to touch are inside the reachable workspace. The inverse kinematics allows us to obtain some important information that will be used in the definition of the reference, as specified in the next paragraph. In particular we perform the inverse kinematic solution based on the trigonometry, referring to the figure 3.4 we can write all the equations.

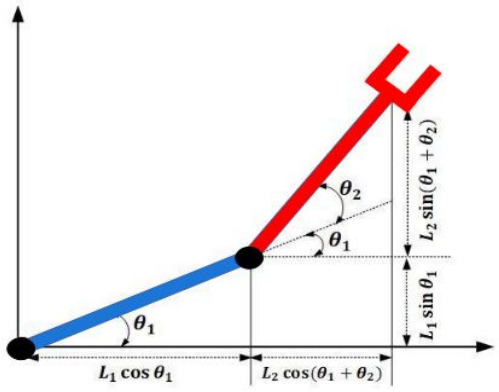

Figure 3.4: The Inverse Kinematics Solutions Based on Trigonometry.

The forward kinematic equations are:

$$\begin{aligned} P_x &= l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) \\ P_y &= l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2) \end{aligned} \tag{3.2}$$

Where l_1 and l_2 are length of first and second links respectively. The θ_1 and θ_2 as joint angles of the 2-DOF planar robot arm can be calculated by applying the equations of geometric solution approach. In particular by applying the cosines rule that θ_2 is specified by:

$$\cos\theta_2 = \frac{P_x^2 + P_y^2 - l_1^2 - l_2^2}{2 l_1 l_2} \tag{3.3}$$

Furthermore, the $\sin\theta_2$ is:

$$\sin\theta_2 = \mp\sqrt{1 - \cos{\theta_2}^2} \tag{3.4}$$

Using the geometric approach, elbow-up and elbow-down solutions are retrieved by selecting both the positive and negative signs in the equations [3]. Based on the configuration of a two-link robot arm, three possible solutions to the inverse kinematics problem essentially are identified. In the first case, precisely one solution might be identified in the situation where the robot is completely stretched out to attain the point. Whereas in the second case, two solutions might be captured if a given (P_x, P_y) coordinate point is within the reach of the robot, while in the third case, no solutions might be identified in the condition where a given (P_x, P_y) coordinate point is out of robot reach. Accordingly, both probable solution to θ_2 are identified by:

$$\theta_2 = \text{atan2}\left(\frac{\mp\sqrt{1 - \cos{\theta_2}^2}}{\cos\theta_2}\right) \tag{3.5}$$

Subsequently, θ_1 is determined by:

$$\theta_1 = \text{atan2}(\frac{P_y}{P_x}) - \tan^{-1}(\frac{l_2 \sin\theta_2}{l_1 + l_2 \cos\theta_2}) \tag{3.6}$$

Below is the code that allows you to implement inverse kinematics:

```
%% Inverse kinematic

teta1 = zeros(size(x_ellipse)); % Vectors of the same dimension
teta2 = zeros(size(x_ellipse)); % of the number of points of the shape

for i=1:size(x_ellipse,1)
    x = x_ellipse(i); % Define the point for which
    y = y_ellipse(i); % we require the IK solution

    % IK solution:
    cost2 = (x^2+y^2-l1^2-l2^2)/(2*l1*l2);
    sint2 = sqrt(1-cost2^2);

    if (imag(sint2)~=0) % check if point is outside workspace
      disp("IK ERROR"); % (-1<= cost2 <=1)
    end

    teta2(i) = atan2(sint2,cost2);

    teta1(i) = atan2(y,x)-atan2((l1*sint2),(l1+l2*cost2));
    %slide pag 123/144

% we have the same result with:
%     sint1 = ((l1+l2*cost2)*y-l2*sint2*x)/(x^2+y^2);
%     cost1 = ((l1+l2*cost2)*x+l2*sint2*y)/(x^2+y^2);
%
%     teta1(i) = atan2(sint1,cost1);

end
```

3.3 Definition of references

In addition to defining the curve we want to draw, it is important to define how the robot will reach the first point of the curve (ellipse), in this way we obtain a complete reference trajectory. In particular as shown in figure 3.5, our reference trajectory is the following:

1. from instant $t = 0$ $[s]$ to instant $t = 2$ $[s]$ the reference is fixed, the robot remains in the starting position $\theta_1 = 45$ $[deg]$ and $\theta_2 = 30$ $[deg]$.

2. from instant $t = 2$ $[s]$ to instant $t = 6$ $[s]$ the reference is defined by a fifth-order polynomial described by the equation 3.7 in order to

obtain smooth profiles, which prevent the final trajectory from having unwanted peaks in correspondence of the discontinuities [2].

$$q(t) = q_0 + a_1(t-t_0) + a_2(t-t_0)^2 + a_3(t-t_0)^3 + a_4(t-t_0)^4 + a_5(t-t_0)^5 \quad (3.7)$$

3. from instant $t = 6$ $[s]$ to instant $t = 30$ $[s]$ the reference become the output of the inverse kinematic, in order to track the ellipse.

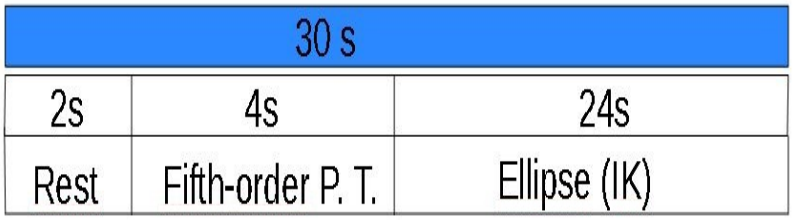

Figure 3.5: Reference signal composition.

The velocity references are always 0, while the input references are obtained from the balance of the gravity term only. In the figures below it is possible to observe the plots of the reference trajectories:

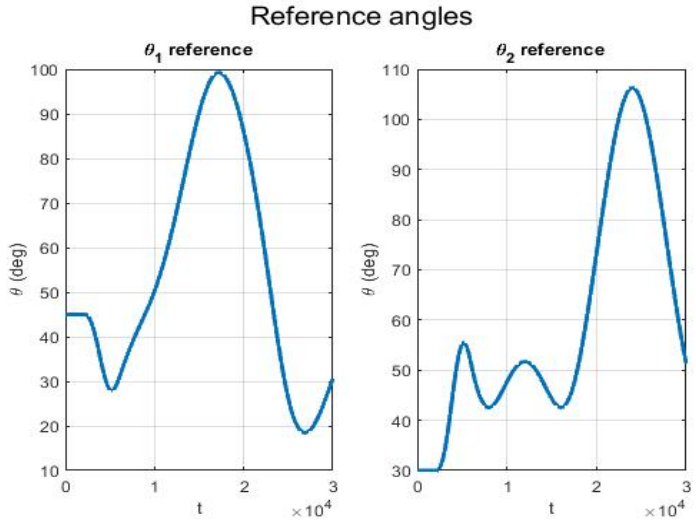

Figure 3.6: Evolution of angles reference signals.

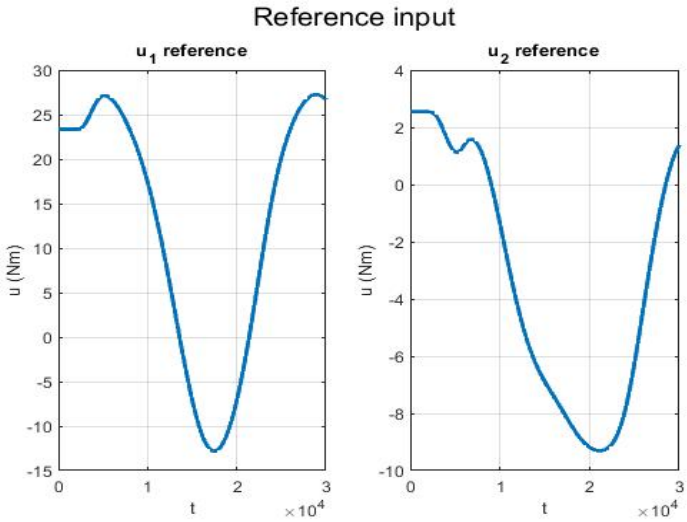

Figure 3.7: Evolution of input (Torque) reference signals.

3.4 DDP Implementation

When we use the DDP algorithm, we have to give a value to the first iteration, so as in Task 1, we perform the initialization with the function *controller.m* that allow us to get an initialization that is close to the reference, so, this means that our DDP algorithm will be more efficient. In particular,to define our initialization we need to impose a reference trajectory and, subsequently, implement our algorithm following the same steps explained in the section 2.4.2.

3.4.1 Initialization

As in task 1, we need to define the position gains k_p and the velocity gains k_D in order to make the tuning of our controller. Also in this case we adopted the same method that we have used in task 1; exploiting the table shown in figure 2.7 we made several attempts and tried different combinations of gain values until we obtained the most acceptable solution from the Matlab plots evaluations. Practically, after some attempts we have considered two different values for K_p and K_D in order to compare the graphs and see differences. The first set of values is:

$$K_p = \begin{bmatrix} 45 & 0 \\ 0 & 45 \end{bmatrix}$$

$$K_D = \begin{bmatrix} 25 & 0 \\ 0 & 25 \end{bmatrix}$$

With these first values we have the following situation:

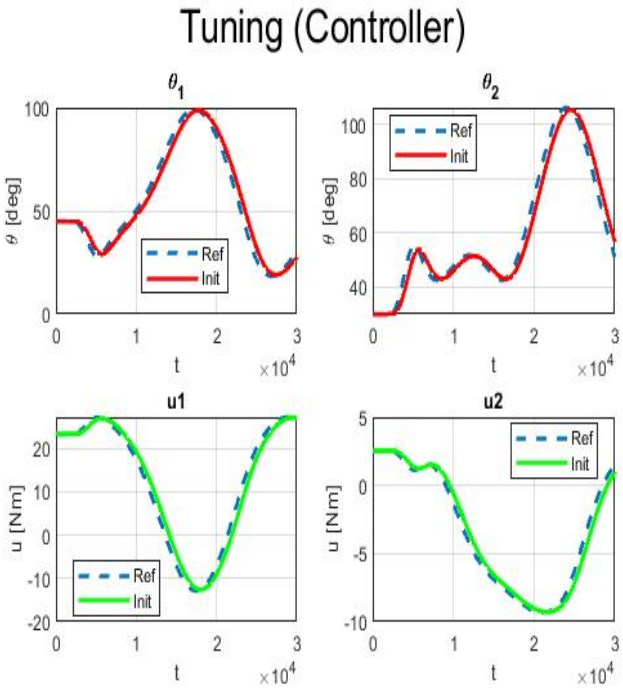

Figure 3.8: Initialization signals, first attempt.

Then, by increasing the position gain K_p and reducing the velocity gain K_D using the second set of values that are:

$$K_p = \begin{bmatrix} 80 & 0 \\ 0 & 80 \end{bmatrix}$$

$$K_D = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$$

We are able to follow better the references, as we can see in figure 3.9:

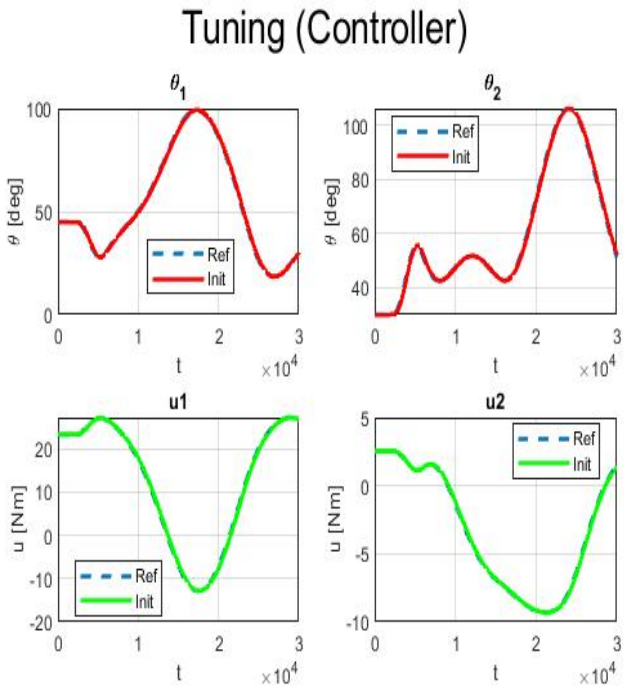

Figure 3.9: Initialization signals, final attempt.

In conclusion, in the figure 3.10 it is possible to see the initial trajectories we have imposed:

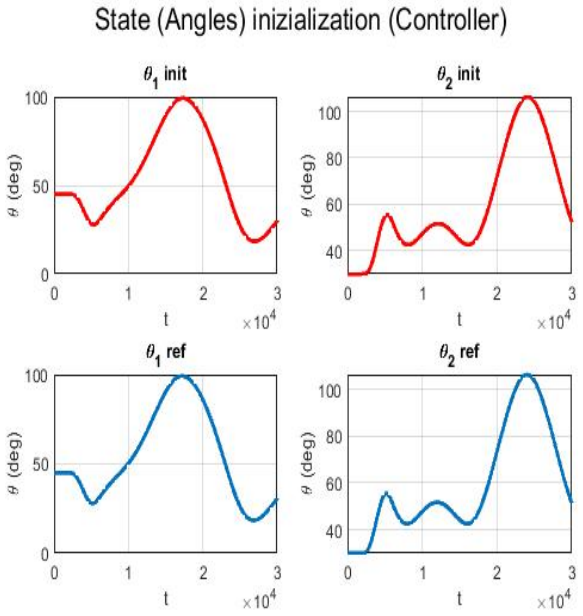

Figure 3.10: Initialization signals.

3.5 Results and consideration Task 2

All the results obtained with the DDP implementation are shown below:

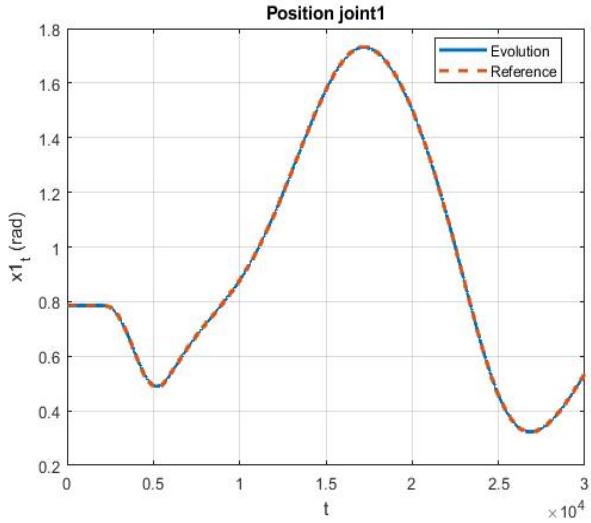

Figure 3.11: Evolution and reference of the position of the first joint.

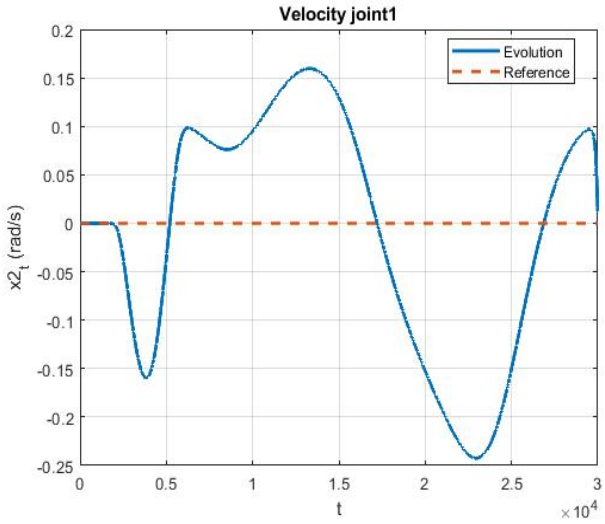

Figure 3.12: Evolution and reference of the velocity of the first joint.

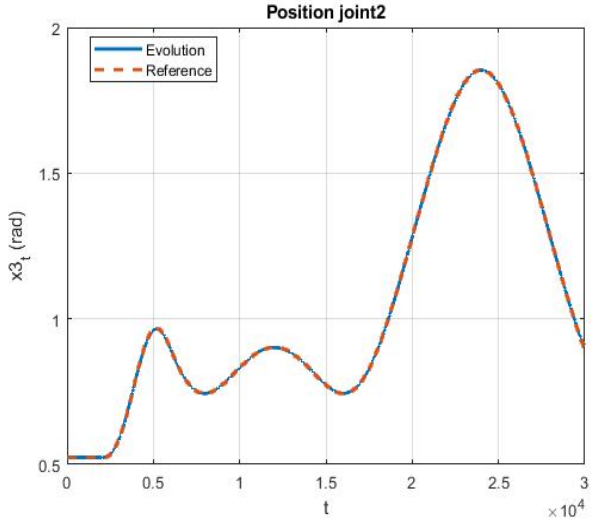

Figure 3.13: Evolution and reference of the position of the second joint.

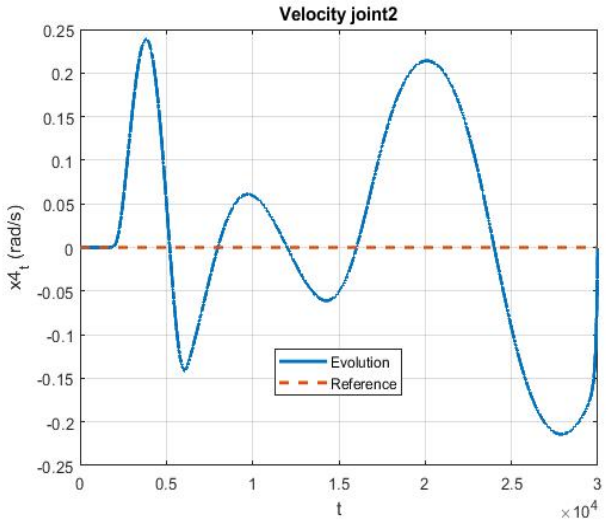

Figure 3.14: Evolution and reference of the velocity of the second joint.

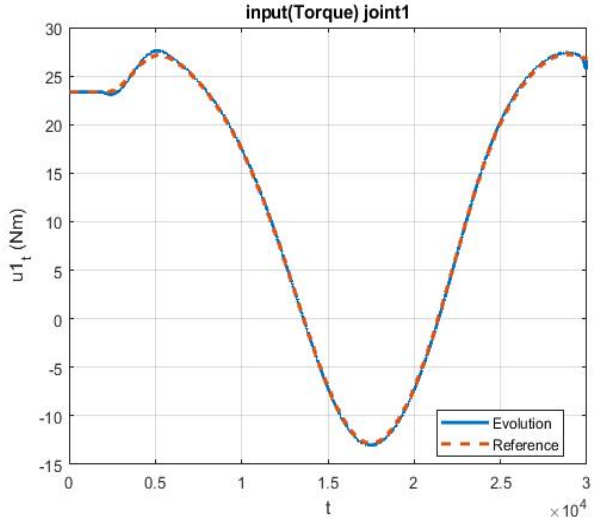

Figure 3.15: Evolution and reference of the input (torque) of the first joint.

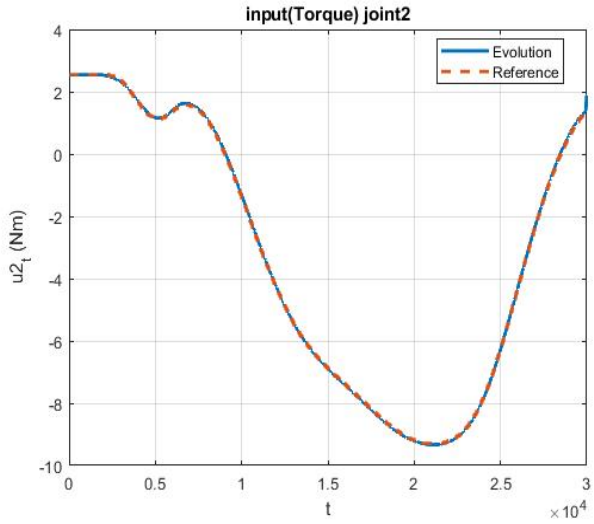

Figure 3.16: Evolution and reference of the input (torque) of the second joint.

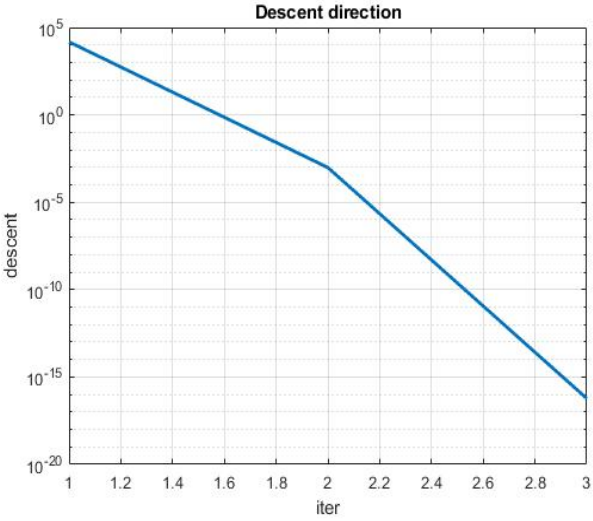

Figure 3.17: Descent direction plot.

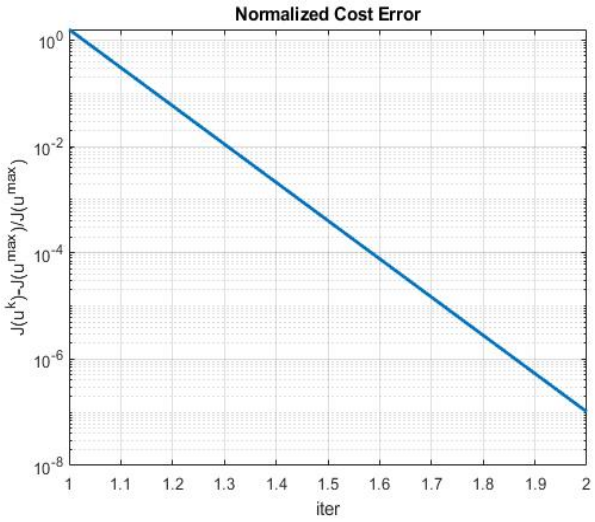

Figure 3.18: Evolution of the normalized cost error.

Chapter 4

Task 3 - Trajectory Tracking

In the third task, implented in Matlab script **Task_3**, we want to linearize the robot dynamics about the optimal trajectory implemented in the previous task. To do this we use the LQR algorithm in order to obtain an optimal feedback controller that allow us to track the reference trajectory. Practically what we have to do is to solve a LQ probem.

4.1 Cost Definition

As in the previous cases we set again the weights matrices in order to define the cost function. Since now we want to follow precisely the reference trajectory we chose for the state the same costs of task 2 and we increase the input costs. In other words we set the matrices in order to track precisely the trajectory even if there are changes in the initialization and the parameters.

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

$$\boldsymbol{R} = \begin{bmatrix} 0,5 & 0 \\ 0 & 0,5 \end{bmatrix}$$

4.2 LQR Implementation

The linear quadratic regulator depend on the linearization of the dynamics of the system about the optimal trajectory we have computed before. To obtain this linearization we use the matlab function **Dynamics.m** which present among its outputs the gradient of the dynamics with respect to the state x and the input u. Consequently, this allow us to find the linearization

matrices A_t^* and B_t^* for each time instants. These matrices are obtained as follow:

$$A_t^* = \nabla_x f(x_t^*, u_t^*)^T \tag{4.1}$$

$$B_t^* = \nabla_u f(x_t^*, u_t^*)^T \tag{4.2}$$

Once we have chosen all the costs for the we start to compute our algorithm that is performed at each iteration k.

1. We exploit the Riccati equation to compute P_t that is obtained by backward iteration

$$P_t = Q + A_t^{*T} P_{t+1} A_t^* - A_t^* P_{t+1} B_t^* (R + B_t^{*T} P_{t+1} B_t^*)^{-1} B_t^{*T} P_{t+1} A_t^* \tag{4.3}$$

   where we initialize with $P_T = Q_T$.

2. We compute the gain matrix K_t^* with a forward iteration:

$$K_t = -(R + B_t^{*T} P_{t+1} B_t^*)^{-1} B_t^{*T} P_{t+1} A_t^* \tag{4.4}$$

Then we update the input and the state getting the optimal control law for the linear quadratic problem that is:

$$u_t = K_t x_t \tag{4.5}$$

$$x_{t+1} = f(x_t, u_t) = A_t x_t + B_t u_t \tag{4.6}$$

Inizialized in x_0, that will be defined in the next section.

4.3 Result with expected initial conditions

Now we can see all results that we obtain initializing the algorithm with:

$$x_0 = \begin{bmatrix} 45° & 0 & 30° & 0 \end{bmatrix}$$

In other words our manipulator starts its motion from an initial position where $\theta_1 = 45$ $[deg]$ and $\theta_2 = 30$ $[deg]$. The results obtained with these initial conditions are shown below:

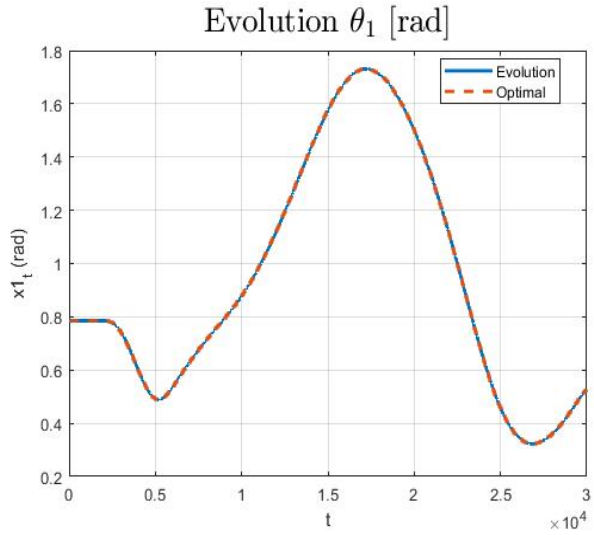

Figure 4.1: Evolution θ_1 compared with the optimal one.

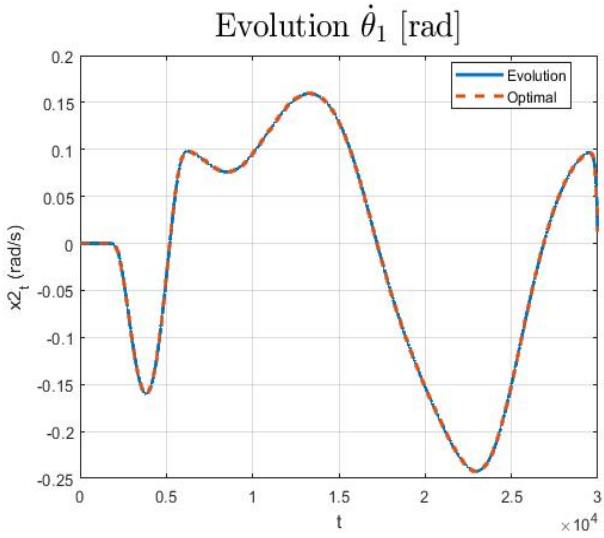

Figure 4.2: Evolution $\dot{\theta}_1$ compared with the optimal one.

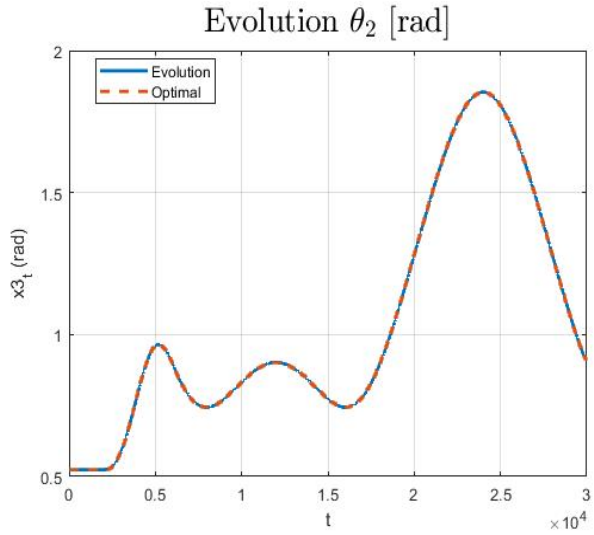

Figure 4.3: Evolution θ_2 compared with the optimal one.

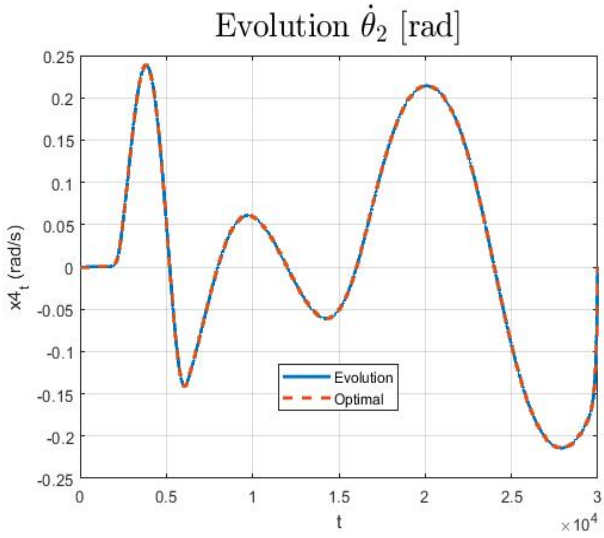

Figure 4.4: Evolution $\dot{\theta}_2$ compared with the optimal one.

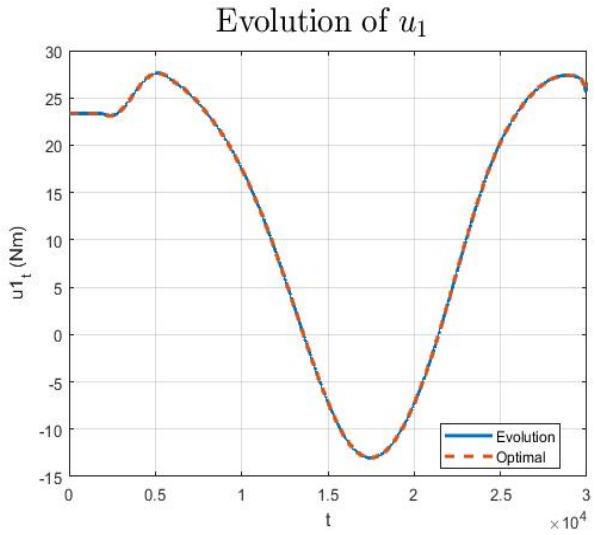

Figure 4.5: Evolution of u_1 compared with the optimal one.

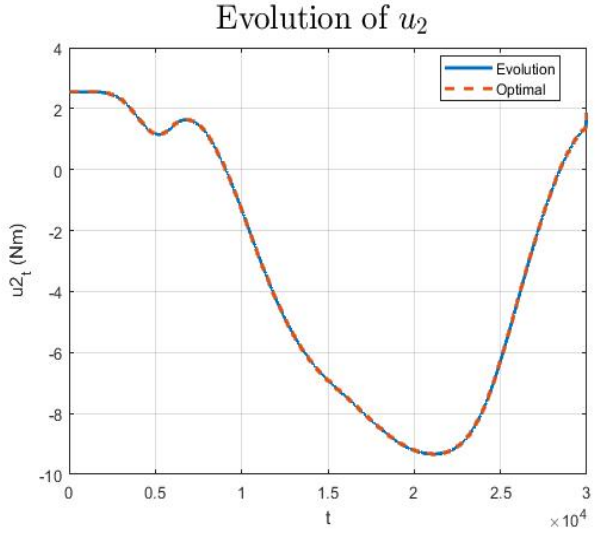

Figure 4.6: Evolution of u_2 compared with the optimal one.

4.4 Result with different initial conditions

To check if our controller can work even if there are different conditions from the conditions set before. We can change the initialization of our algorithm choosing a different x_0. In this way we can see if our controller is able to track the trajectory, even if we start far from the optimal trajectory. For example, by choosing:

$$x_0 = \begin{bmatrix} 0° & 0 & 20° & 0 \end{bmatrix}$$

The result obtained modifying the initial conditions are shown below:

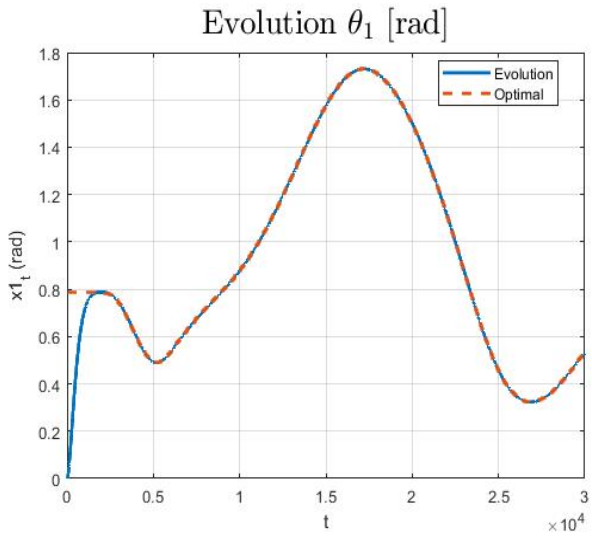

Figure 4.7: Evolution θ_1 compared with the optimal one.

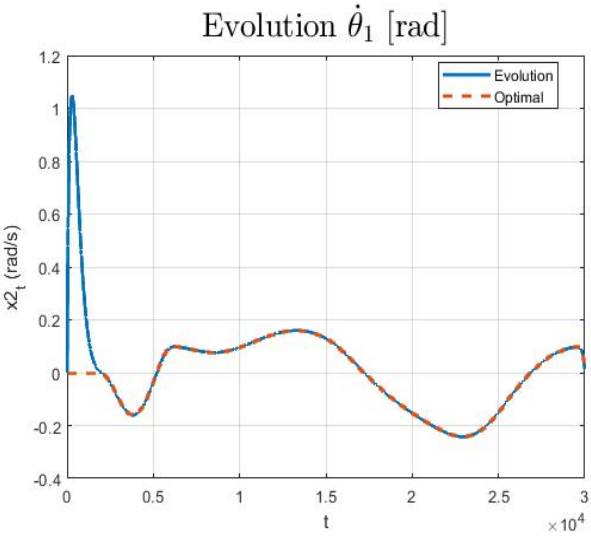

Figure 4.8: Evolution $\dot{\theta}_1$ compared with the optimal one.

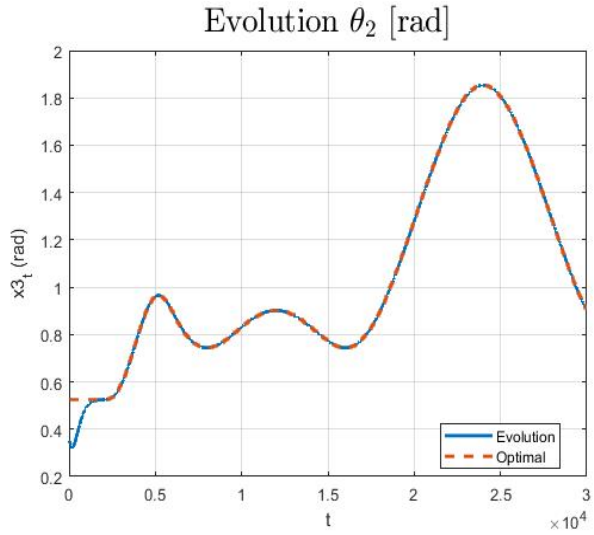

Figure 4.9: Evolution θ_2 compared with the optimal one.

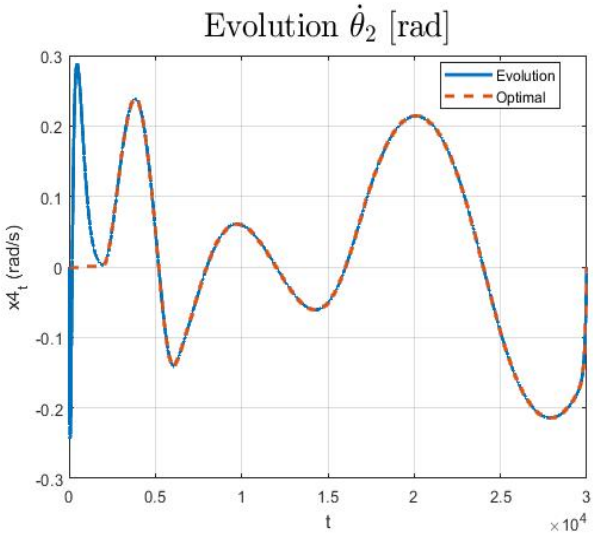

Figure 4.10: Evolution $\dot{\theta}_2$ compared with the optimal one.

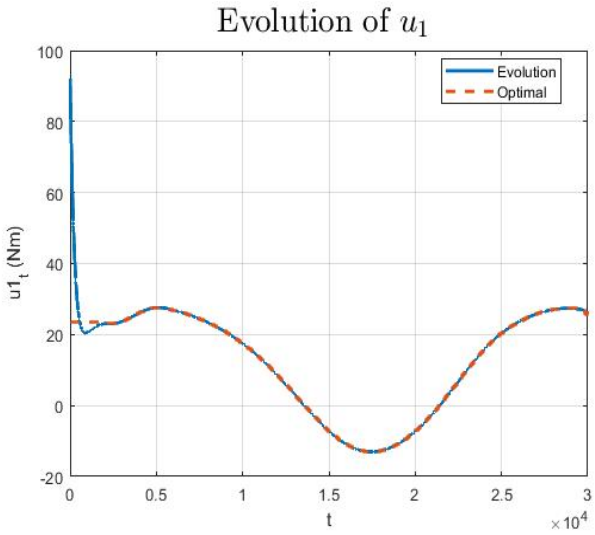

Figure 4.11: Evolution of u_1 compared with the optimal one.

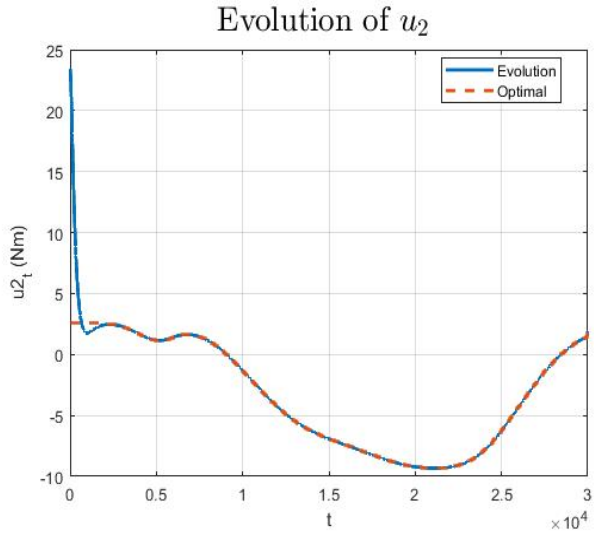

Figure 4.12: Evolution of u_2 compared with the optimal one.

Even starting from a point not close to the starting point of the optimal trajectory, the algorithm allows to recover the optimal behavior after a few moments. This means that our algorithm is efficient.

Chapter 5

Task 4 - Animation

In this chapter we perform the simulation of our manipulator. To design the 3D model of the robot we use PTC Creo Parametric 8.0.3.0 software, while to move the 3D model in the space we use Simscape Multybody. In particular with the last one we connect PTC with Matlab, where we have implemented all the tasks.

5.1 Simscape Multibody

Simscape Multibody provides a multibody simulation environment for 3D mechanical systems, such as robots. Whit this Matlab extension we import CAD geometries, masses, inertias, joints, constraints, and 3D geometries from PTC creo into our model. The final model is presented in figure 5.1.

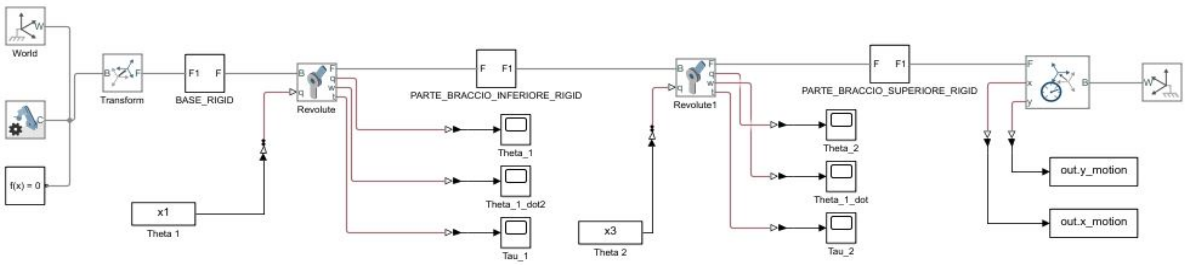

Figure 5.1: Simulink scheme.

Running the model we obtain the result represented in figure 5.2. We have a base that is fixed and two links that can move. We consider as the trajectory of the End-effector the curve drawn in the plane by the frame fixed with the base of the "bucket".

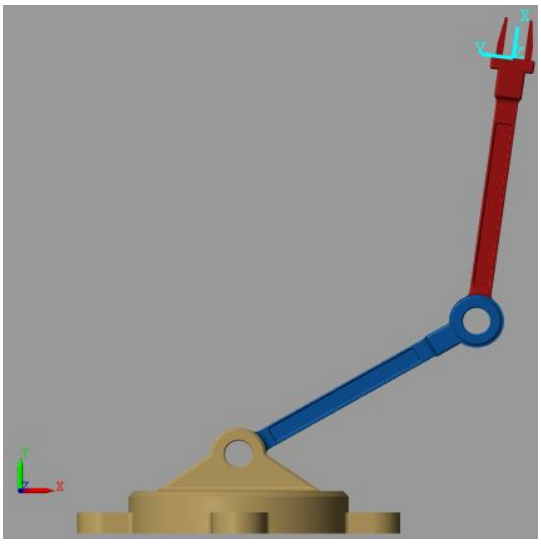

Figure 5.2: 2-DOF manipulator generated with Simscape Multibody.

5.2 Result

In the model described in figure 5.1 we use a transform Sensor block to find the position of the bucket reference frame with respect the world reference frame. We obtain as output the position coordinates directly to the workspace, and then plot these coordinates using common MATLAB commands. The result is shown in figure 5.3.

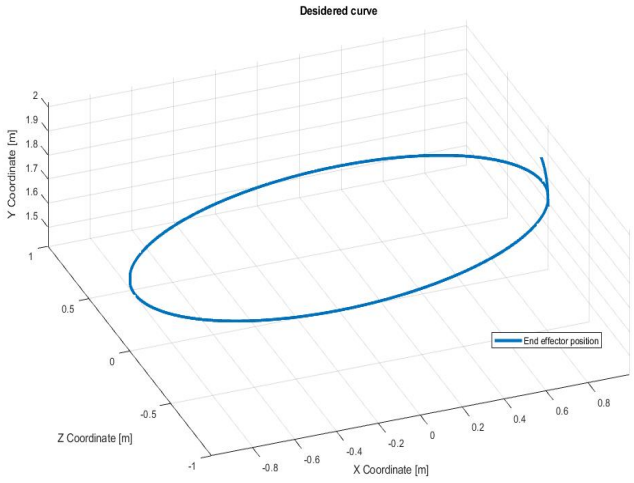

Figure 5.3: Motion of the manipulator.

The End-effector movement is the desired one, starting from the starting point where $\theta_1 = 45\ [deg]$ and $\theta_2 = 30\ [deg]$ it moves to the lower position, the position from which the robot begins to draw the ellipse.

Conclusions

With the realization of this project we solved different tasks thanks to which we learnt how to design an optimal control for a 2-DOF manipulator.

- In the first task it was implemented a DDP algorithm to design an optimal trajectoy that allows to follow a step reference. From this task we learnt how manage this algorithm and how tune the weight matrices. Overall, it was possible to understood how the manipulator can move from an initial configuration to a final one.
- In the second task we exploited again the DDP to design an optimal trajectory to allow the manipulator to draw an ellipse defined within the workspace. In particular, we seen how exploit the inverse kinematics in order to define a shape in workspace and consequently map the values from workspace to joint space. Successively, it was possible to implement a polynomial trajectory in the joint space.
- In the third task we used the LQR method, so we designed this controller that allows the manipulator to follow the optimal trajectory designed in task 2. In this part we checked also if the controller is able to achieve the target with different initial conditions. By changing these it was resulted that at the beginning there are some trajectory tracking problems, but later these are recovered by the controller.
- In the fourth task it was performed a simulation of the manipulator. It was designed a 3D model of the manipulator through the software PTC Creo Parametric 8.0.3.0. Eploiting Simscape Multibody it was possible connect PTC with matlab in order to move our robot. In this way it was possible to clearly see if the robot followed the desired trajectory drawing an ellipse in the workspace.

Bibliography

[1] Luigi Villani Bruno Siciliano, Lorenzo Sciavicco. *Robotics*. Springer, 2012.

[2] Claudio Melchiorri Luigi Biagiotti. *Trajectory Planning for Automatic Machines and Robots*. Springer, 2008.

[3] Firas Raheem, Ahmed Sadiq, and Noor F. Abbas. Robot arm free cartesian space analysis for heuristic path planning enhancement. *International Journal of Mechanical & Mechatronics Engineering*, 19:29–42, 02 2019.