UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Optimal Control

Optimal Control of a Flexible Robotic Arm

Professor: **Giuseppe Notarstefano**

Students:
Matteo Bonucci
Filippo Samorì
Jacopo Subini

Academic year 2024/2025

Abstract

This report presents the development and implementation of the project for the Optimal Control Course at the University of Bologna. The assignment aims to control an underactuated 2-DOF robot manipulator. The report is structured such that each chapter is dedicated to the solution of a specific task.

The project demonstrates the capabilities of the Optimal Control framework in controlling complex systems, like underactuated systems or time-variant linear systems, such as the linearization of the manipulator system.

The results highlight the robustness in the presence of noise and the variety of possible approaches to the problem.

Contents

Introduction

In this project, we are tasked with designing an optimal trajectory for a flexible robotic arm. Such systems can represent robotic arms used in medical assistance or other precision applications, where flexibility must be taken into account. The flexible arm is simplified and modeled as a planar two-link robot with torque applied to the first joint.

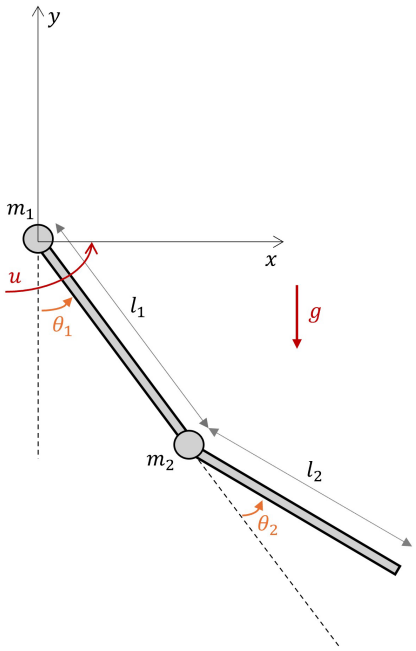


Figure 1: 2-dof manipulator

The 2-dof manipulator dynamics is reported in matrix form:

$$M(q)\ddot{q} + C(q,\dot{q}) + G(q) + F\dot{q} = \begin{bmatrix} u \\ 0 \end{bmatrix} \tag{1}$$

The configuration variable q is composed of the angles θ_1 and θ_2, which are defined as the relative angular displacement between the two links.

$$M = \begin{pmatrix} I_1 + I_2 + m_1 r_1^2 + m_2(l_1^2 + r_2^2) + 2m_2 l_1 r_2 cos(\theta_2) & I_2 + m_2 r_2^2 + m_2 l_1 r_2 cos(\theta_2) \\ I_2 + m_2 r_2^2 + m_2 l_1 r_2 cos(\theta_2) & I_2 + m_2 r_2^2 \end{pmatrix}$$

$$C = \begin{pmatrix} -m_2 l_1 r_2 \dot{\theta}_2 sin(\theta_2)(\dot{\theta}_2 + 2\dot{\theta}_1) \\ m_2 l_1 r_2 sin(\theta_2)\dot{\theta}_1 \end{pmatrix}$$

$$G = \begin{pmatrix} g(m_1 r_1 + m_2 l_1)sin(\theta_1) + g m_2 r_2 sin(\theta_1 + \theta_2) \\ g m_2 r_2 sin(\theta_1 + \theta_2) \end{pmatrix}$$

$$F = \begin{pmatrix} f_1 & 0 \\ 0 & f_2 \end{pmatrix}$$

The parameters of the robot dynamics are described in the table below:

Parameters : set 1

m_1	1
m_2	1
l_1	1
l_2	1
r_1	0.5
r_2	0.5
I_1	0.33
I_1	0.33
g	9.81
f_1	0.1
f_2	0.1

Task 0: requires writing the dynamics and obtaining the linearization of the system.

Task 1: requires defining a reference transition curve between equilibrium configurations and computing an optimal trajectory between them.

Task 2: requires defining a smooth reference curve between equilibrium configurations and computing again an optimal trajectory. It is suggested to first try tracking the reference with an LQR controller to obtain a feasible initial guess.

Task 3: requires tracking the obtained optimal trajectory with an LQR in the presence of noise.

Task 4: requires tracking the optimal trajectory obtained with an MPC solution in the presence of noise.

Task 5: requires animations for all of the previous tasks.

Task 0

Task 0 asks to discretize the dynamics, write the discrete-time state-space equations, and code the dynamics function `Dynamic.py`.

The state representation is obtained from the formula (1), defining a state vector $x = [\theta_1 \;\; \theta_2 \;\; \dot{\theta_1} \;\; \dot{\theta_2}]^T$ and writing the dynamics as follows:

$$\dot{x} = f(x,u) = \begin{bmatrix} x_3 \\ x_4 \\ M^{-1}(-G(x_1,x_2) - C(x_1,x_2,x_3,x_4) - F\begin{bmatrix} x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} u \\ 0 \end{bmatrix} \end{bmatrix} \tag{2}$$

The simulations are made numerically by approximating the continuous dynamics with a discrete one using Euler's method for numerical integration. We denote the discrete time state vector as x_t, where the subscript describes the discrete time instant.

$$x_{t+1} = x_t + \dot{x}\Delta t = x_t + f(x_t,u_t)\Delta t = f_{dis}(x_t,u_t) \tag{3}$$

The linearization matrices are obtained by computing the Jacobian of $f_d(x_t,u_t)$ with respect to x_t for matrix A_t and u_t for matrix B_t. They are time-varying if the linearization is computed about a trajectory of the system.

It is possible to compute the discrete matrices starting from the continuous-time Jacobians and obtaining A_t and B_t by discretizing.

$$\Delta x_{t+1} = (I_{n\times n} + \frac{\partial f}{\partial x}\bigg|_{x_t,u_t}\Delta t)\Delta x_t + (\frac{\partial f}{\partial u}\bigg|_{x_t,u_t}\Delta t)\Delta u_t \tag{4}$$

$$A_t = I_{n\times n} + \frac{\partial f}{\partial x}\bigg|_{x_t,u_t}\Delta t = \nabla_1 f_{dis}(x,u)^T$$

$$B_t = \frac{\partial f}{\partial u}\bigg|_{x_t,u_t}\Delta t = \nabla_2 f_{dis}(x,u)^T$$

To compute the derivatives, the symbolic Python package SymPy has been used. The package computes the continuous dynamic Jacobians directly, and then we obtain the discrete Jacobians using (4).

Python code

The discrete dynamics is implemented in Python in the file `Dynamics.py`. In the code, four main functions are defined:

- **gravity(xx)** : compute the gravity torque applied to the motor in joint one at a specific state x.

  Arguments:

  - xx : state vector.

  Return:

  - gravity : torque value.

- **dynamics(xx,uu,dt)** : Compute the future state x_{t+1} given the states x_t and the input u_t. The functions first compute the continuous dynamics $f(x, u)$ of the manipulator (1) and then discretize it using (3).

  Arguments:

  - xx : state vector at time t.
  - uu : input vector at time t.
  - dt : time of dicretization.

  Return:

  - xx_plus : computed state at time t+1.

- **linearized_dynamics_symbolic()**: Compute the symbolic Jacobians of the continuous dynamics with respect to x and u. The computations are made using the Python package Sympy. The symbolic terms are computed only once and then used by other functions to compute the numerical values by substitution of values.

  Arguments:

  - None.

  Return:

  - A_func : Symbolic jacobian respect to x.
  - B_func : Symbolic jacobian respect to u.

- **linearized_dynamics_numeric(xx,uu,A_func,B_func,dt)**: Compute the numerical value of the Jacobians of the discrete dynamics with respect to x and u. The computations are made using the continuous symbolic Jacobians computed from `linearized_dynamics_symbolic()` and then applying the formulas from (4) to compute the discrete ones.

  Arguments:

  - xx : state vector at time t.
  - uu : input vector at time t.
  - A_func : Symbolic jacobian respect to x.

- B_func : Symbolic jacobian respect to u.
- dt : time of dicretization.

Return:

- A_dis : Numeric jacobian respect to x at time t.
- B_dis : Numeric jacobian respect to u at time t.

Task 1

Task 1 requires computing two equilibrium points for the system, defining a reference curve between them. Then, it is required to compute an optimal trajectory to move from one equilibrium to another using the Newton's-like algorithm.

In general, the equilibrium points are all the points that satisfy the equation:

$$x_e = f_{dis}(x_e, u_e) \tag{5}$$

In our case, the discrete dynamics are the one described in (3), so a condition for equilibrium becomes that:

$$x_e = x_e + f(x_e, u_e)\Delta t \longrightarrow f(x_e, u_e) = 0 \tag{6}$$

That means that the equilibrium points for the discrete dynamics are equal to those of the continuous dynamics.

Since the second joint is not controlled, the system is in equilibrium only if the second joint is in equilibrium. As we know for a simple pendulum, the equilibrium configurations are the ones where the absolute angle θ with respect to the world frame is equal to 0 or π. The world frame is defined such that the y axis is parallel to the gravity vector and it is equal to the frame of joint one. So, we have that the second link is in equilibrium for each value of q_2 such that $q_1 + q_2 = 0$ or π.

So, for the whole system, we have equilibrium when the velocities are zero, the input compensates for the gravity in the first joint (7), and the second joint satisfies the equation $q_1 + q_2 = 0$ or π.

$$u = G_1(q) \tag{7}$$

Once two equilibrium points have been chosen, it is necessary to run the Newton's-like method with an initial guess that is a valid trajectory of the system. For our purposes, in task 1, we initialize the initial guess as constant vectors in the chosen initial equilibrium configuration.

The cost function is defined as follows:

$$\begin{aligned} \ell(x,u) = \sum_{t=0}^{T-1} (x_t - x_t^{ref})^T Q_t (x_t - x_t^{ref}) + (u_t - u_t^{ref})^T R_t (u_t - u_t^{ref}) \\ + (x_T - x_T^{ref})^T Q_T (x_T - x_T^{ref}) \end{aligned} \tag{8}$$

and the unconstrained optimal control problem is stated as:

$$\begin{aligned} &\min_{x\ u} \ \ell(x,u) \\ &subj.to \quad x_{t+1} - f_d(x_t,u_t) = 0 \end{aligned} \tag{9}$$

The cost is a quadratic function of the states and the inputs. The Newton's-like method solves at each iteration a convex quadratic approximation of the whole problem, computing the descent direction as the solution of the Linear Quadratic Regulator.

The optimization is solved by considering an augmented state vector with the corresponding augmented matrices.

$$\Delta\tilde{x}_t := \begin{bmatrix} 1 \\ \Delta x_t \end{bmatrix}$$

$$q_t = \nabla_1 \ell(x_t^k, u_t^k) = Q_t(x_t^k - x_t^{ref}) \quad r_t = \nabla_2 \ell(x_t^k, u_t^k) = R_t(u_t^k - u_t^{ref})$$

$$Q_t = \nabla_{11} \ell(x_t^k, u_t^k) \quad R_t = \nabla_{22} \ell(x_t^k, u_t^k)$$

$$\tilde{Q}_t := \begin{bmatrix} 0 & q_t^T \\ q_t & Q_t \end{bmatrix} \quad \tilde{S}_t := \begin{bmatrix} r_t & S_t \end{bmatrix} \quad \tilde{R}_t := R_t \quad \tilde{A}_t := \begin{bmatrix} 1 & 0 \\ c_t & A_t \end{bmatrix} \quad \tilde{B}_t := \begin{bmatrix} 0 \\ B_t \end{bmatrix}$$

Then the associated LQR problem can be solved using the Difference Riccati Equation (12) for the matrix P and computing the optimal gain $\tilde{K}^*$ [1]. To solve the problem, we use a regularized Newton's-like method where the terms $\nabla_{11} f_d(x_t^k, u_t^k)$ and $\nabla_{22} f_d(x_t^k, u_t^k)$ are ignored. In this way, we are sure that the matrices $\tilde{Q}_t$ and $\tilde{R}_t$ are always positive (semi-[2]) definite.

$$\begin{aligned} &\min_{\Delta x \Delta u} \sum_{t=0}^{T-1} \frac{1}{2} \begin{bmatrix} \Delta\tilde{x}_t \\ \Delta u_t \end{bmatrix}^T \begin{bmatrix} \tilde{Q}_t & \tilde{S}_t^T \\ \tilde{S}_t & \tilde{R}_t \end{bmatrix} \begin{bmatrix} \Delta\tilde{x}_t \\ \Delta u_t \end{bmatrix} + \frac{1}{2} \Delta\tilde{x}_T^T \tilde{Q}_T \Delta\tilde{x}_T \\ &s.t. \quad \Delta\tilde{x}_{t+1} = \tilde{A}_t \Delta\tilde{x}_t + \tilde{B}_t \Delta\tilde{u}_t \quad t = 0, .., T-1 \end{aligned} \tag{10}$$

The Difference Riccati Equation (12) is solved by backward integration, with the initial condition $P_T = \tilde{Q}_T$.

$$\tilde{K}_t^* = -(\tilde{R}_t + \tilde{B}_t^T P_{t+1} \tilde{B}_t)^{-1} (\tilde{S}_t + \tilde{B}_t^T P_{t+1} \tilde{A}_t) \tag{11}$$

$$P_t = \tilde{Q}_t + \tilde{A}_t^T P_{t+1} \tilde{A}_t - K_t^{*T} (\tilde{R}_t + \tilde{B}_t^T P_{t+1} \tilde{B}_t) K_t^* \quad t = T-1, ..., 0 \tag{12}$$

To solve the convex quadratic problem, a shooting approach is considered. The solution is computed for Δu, and then Δx is obtained through forward integration of the linearized dynamics.

[1] The * denotes an optimal result

[2] The matrix Q_t must be semi-positive definite, the matrix R_t must be positive definite

$$\begin{aligned}\Delta\tilde{u}_t &= \tilde{K}_t^* \Delta\tilde{x}_t \\ \Delta\tilde{x}_{t+1} &= \tilde{A}_t \Delta\tilde{x}_t + \tilde{B}_t \Delta\tilde{u}_t\end{aligned} \tag{13}$$

Once the descent direction Δu^k is computed, the global solution can be updated using a robustified feedback integration of the dynamics. The step size can be chosen fixed or computed using Armijo's rule. Armijo's method is also useful to understand if the actual solution Δu^k is effectively a descent direction, and it helps to check if the linearization of the dynamics is correctly computed by checking if the line (14), parametrized by γ, is tangent to the current cost $\ell(x^k, u^k)$.

$$y(\gamma) = J(u^k) + \gamma \nabla J(u^k)^T \Delta u^k \tag{14}$$

The step size selection condition for Armijo's rule is

$$J(u^k + \gamma \Delta u^k) \geq J(u^k) + \gamma \nabla J(u^k)^T \Delta u^k \tag{15}$$

$$\begin{aligned}u_t^{k+1} &= u_t^k + K_t^k (x_t^{k+1} - x_t^k) + \gamma^k \sigma_t^k \\ x_{t+1}^{k+1} &= f_d(x_t^{k+1}, u_t^{k+1})\end{aligned} \tag{16}$$

The algorithm stops the iterations when it finds a descent direction whose squared amplitude is lower than a certain threshold, or when Armijo's method fails to determine a feasible step size.

In our case, we have chosen the following equilibrium point:

$$\begin{aligned}x_{init} &= [0, 0, 0, 0] \\ x_{fin} &= [90, -90, 0, 0]\end{aligned}$$

These points are easy to reach because they are stable. A step reference transition has been chosen (Fig 2). The simulation time is set to 10 seconds, and the discretization time is 0.01 seconds.

The weights in the cost have been selected experimentally, and a good compromise resulted in:

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

$$R_t = 10$$

The algorithm requires 8 iterations to converge to a solution. As shown in the descent direction figure (8b), the value of the norm falls below 10^{-6}, which is the value chosen for the threshold.

The Armijo parameters have been selected using the values from the course slides:

- $\beta = 0.7$

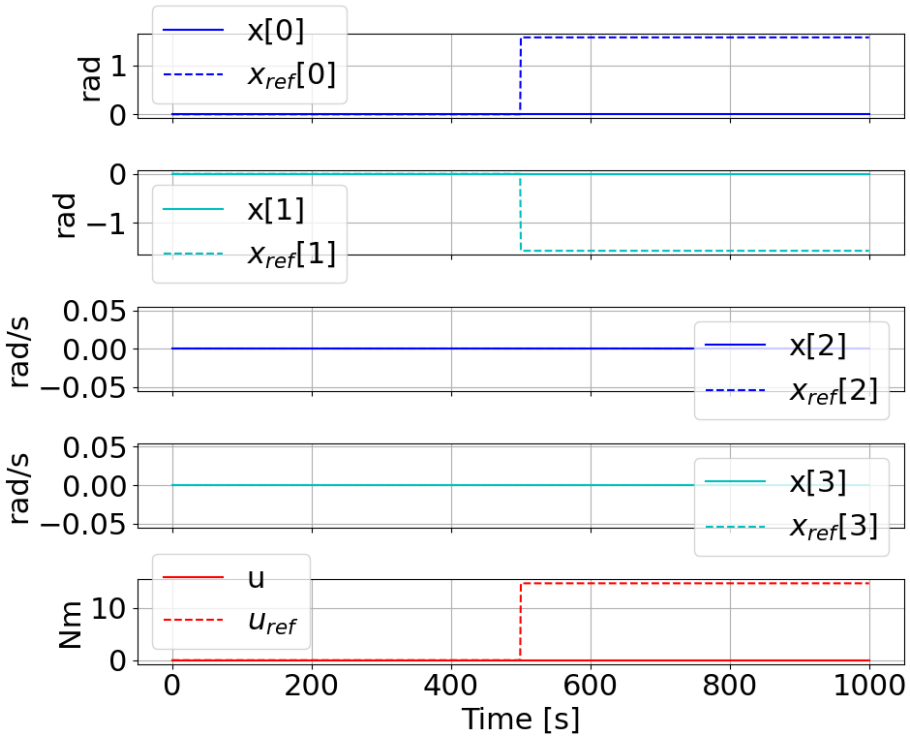


Figure 2: Reference and initial guess

- $c = 0.5$

Python code

The reference trajectory is defined in the Python file reference_trajectory.py.

The Newton's method solver is defined in the Python file solver.py.

- **step(xx_init, xx_fin, tf, dt)** Compute a step curve between xx_init and xx_fin. The input is calculated as the gravity compensation along the resulting state reference curve.

  Arguments:

  - xx_init : Initial state vector in degrees.
  - xx_fin : Final state vector in degrees.
  - tf : Final time in seconds.
  - dt : Time of dicretization in seconds.

  Return:

  - xx_ref : Reference curve between the two states from 0 to tf, in radiants.
  - uu_ref : Input reference curve.

- **step_comp(points, tf, dt)** : Compute a composition of step curves between state configuration points. The input is calculated as the gravity compensation along the resulting

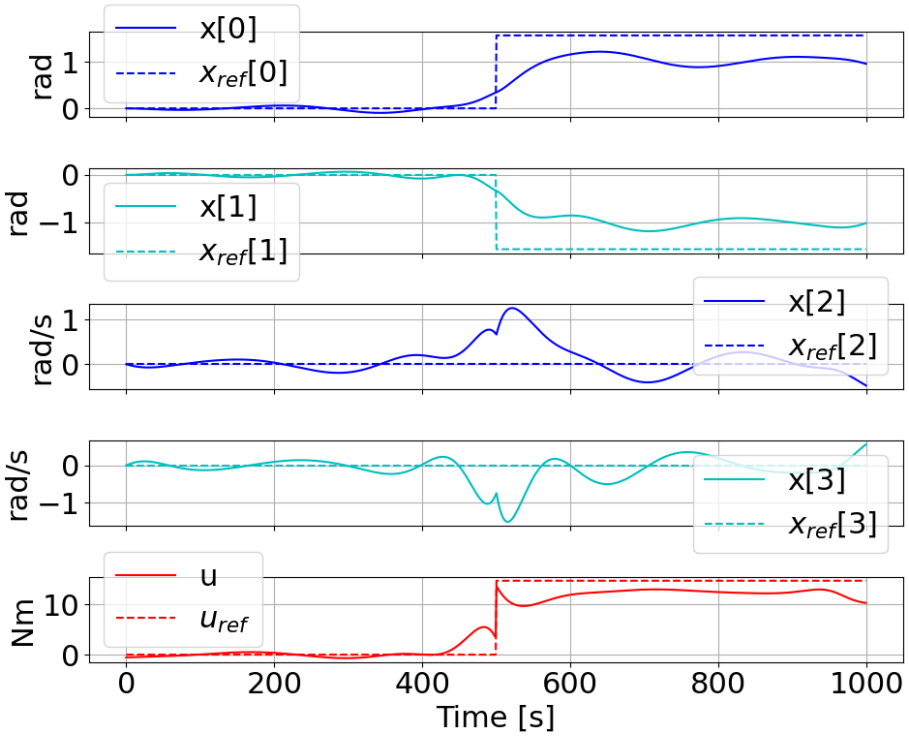


Figure 3: First iteration

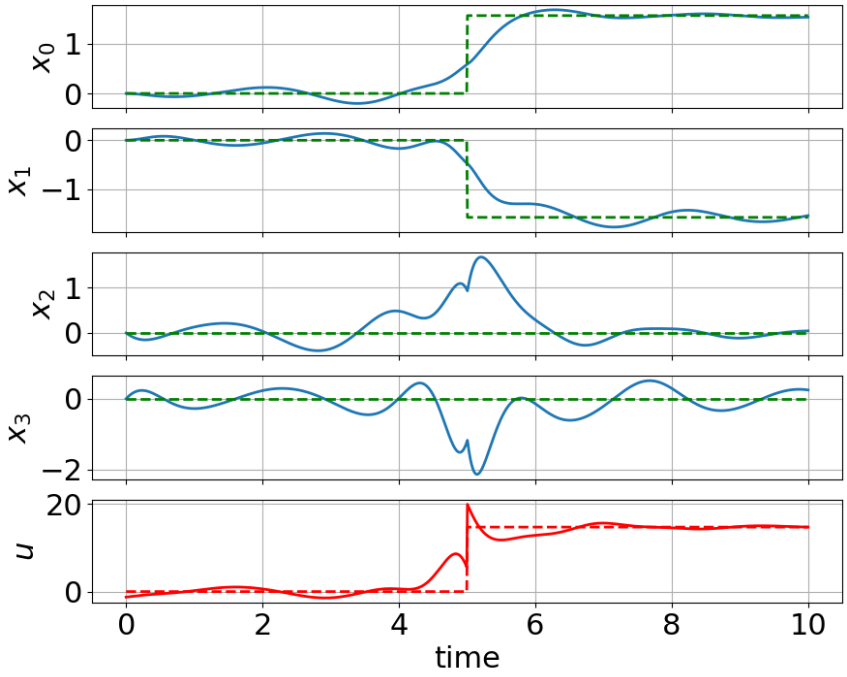


Figure 4: Final result

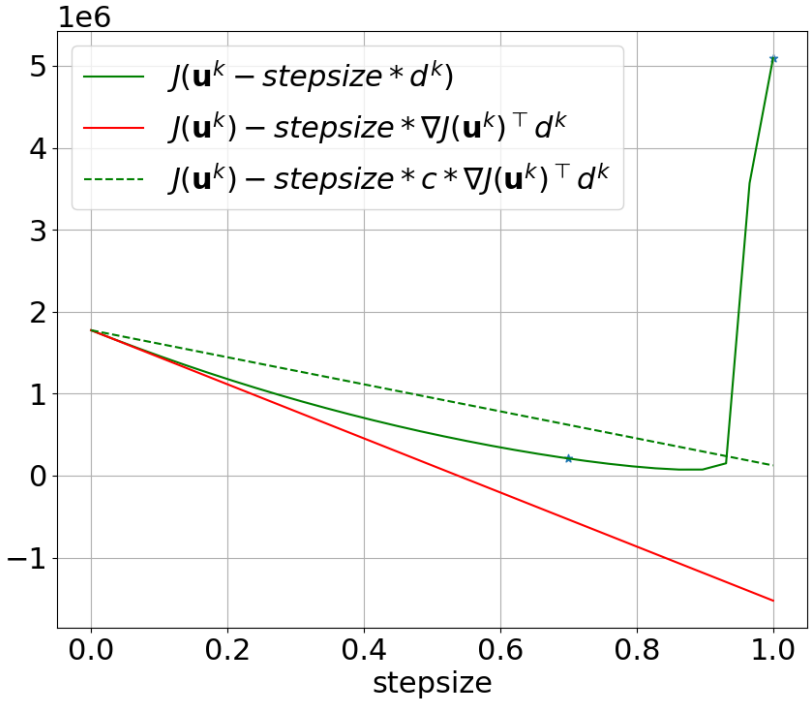


Figure 5: Backtracking line search plot at the first iteration

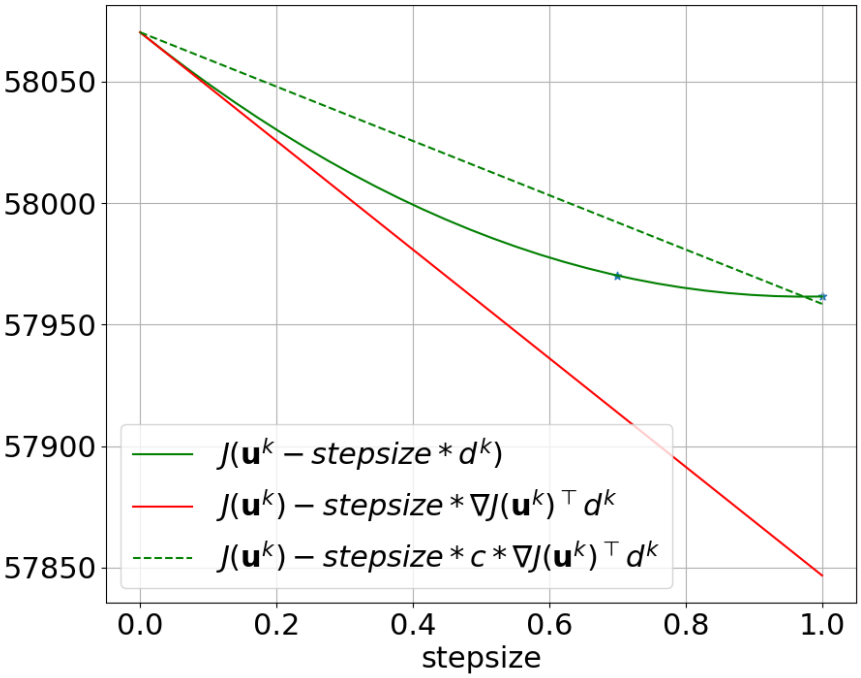


Figure 6: Backtracking line search plot at the third iteration

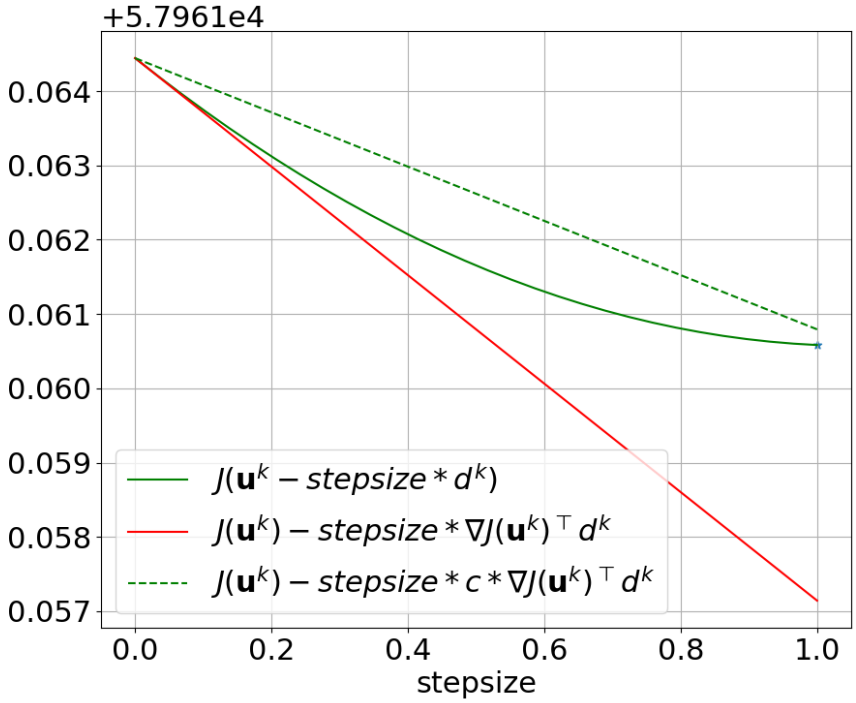
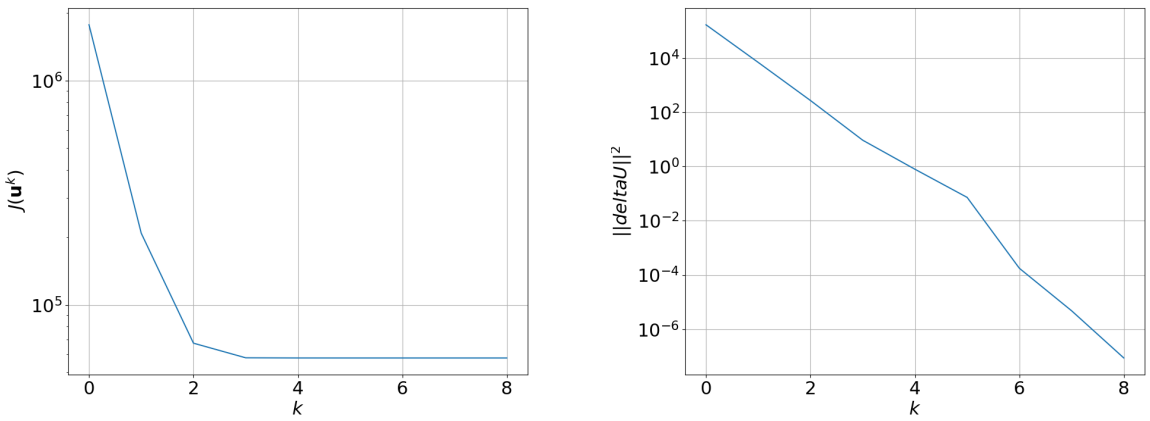


Figure 7: Backtracking line search plot at the sixth iteration



(a) Cost

(b) Descent direction

Figure 8

state reference curve.

Arguments:

- points : List of state vectors in degrees.
- tf : Final time in seconds.
- dt : Time of dicretization in seconds.

Return:

- xx_ref : State reference curve in radiants.
- uu_ref : Input reference curve.

- **newton_solver(xx_ref, uu_ref, xx_init, uu_init, dyn, tf, dt, max_iters, ...)** : Compute the optimal trajectory using the Newton's-like method. The Armijo's method is also implemented for the selection of the step size.

  Arguments:

  - xx_ref : Reference state curve.
  - uu_ref : Reference input curve.
  - xx_init : Initial guess state trajectory.
  - uu_init : Initial guess input trajectory.
  - dyn : dynamic package imported from Dinamics.py.
  - tf : Final simulation time, in seconds.
  - dt : Time of dicretization in seconds.
  - max_iters : Maximum number of iteration of the algorithm.
  - term_cond = 1e-1 : Is the terminal condition for the squared magnitude of the discent direction.
  - fixed_stepsize = 1e-1 : Step size value if Armijio is not enabled.
  - armijo = False : Armijo enable variable.
  - dynamic_plot = False : Plot while executing.
  - visu_descent_plot = False : Armijio plot.
  - armijo_maxiters = 20 : Maximum number of iteration for the stepsize search.
  - cc = 0.5 : Armijo paramiter.
  - beta = 0.7 : rate of reduction of the stepsize at each armijo iteration.
  - stepsize_0 = 1 : Initial Armijo stepsize.

  Return:

  - xx_star : Optimal State Trajectory.
  - uu_star : Optimal Input Trajectory.
  - JJ : Vector of acquired costs.
  - descent : Squared value of the magnitude of the descent direction.
  - max_iter : Number of effective iterations.

Task 2

Task 2 requires computing a smooth state-input curve. Then, it is required to compute the optimal trajectory as in Task 1.

We have designed the trajectory as a smooth transition between equilibrium points.

Since we have to impose the initial and final configurations and velocities for each joint, we need at least a third-order polynomial curve to transit from one equilibrium point to another.

For the first joint q_1, we impose the initial and final angles, both with zero velocity. The corresponding curve is defined as:

$$\begin{aligned} q_1(t) &= a_0 + a_1 t + a_2 t^2 + a_3 t^3 \\ \dot{q_1}(t) &= a_1 + 2a_2 t + 3a_3 t^2 \\ a_0 &= q_{1i} \\ a_1 &= 0 \\ a_2 &= 3(q_{1f} - q_{1i})/(t_f^2) \\ a_3 &= -2(q_{1f} - q_{1i})/(t_f^3) \end{aligned} \tag{17}$$

For the second joint q_2, we compute it as

$$\begin{aligned} q_2(t) &= z - q_1(t) \quad z = 0 \vee \pi \\ \dot{q}_2(t) &= -\dot{q}_1(t) \end{aligned} \tag{18}$$

The value of z is chosen depending on the configuration we want to maintain:

- $z = \pi$, the second link is pointing upwards.
- $z = 0$, the second link is pointing downwards.

The input is computed as the gravity compensation along the smooth curve.

Once the reference has been computed, to improve the initial guess used to start the Newton's method algorithm, we try to track the reference with an LQR feedback computed on the linearization about it.

Since the trajectory is a sequence of equilibria, we get a solution for the LQR only if the motions are quite slow. The reason is that if we try to track a quasi-static trajectory, since the second joint is not actuated, it is difficult for the control law to track the reference for fast motions.

Moreover, since the reference is not a true trajectory, the linearized dynamics is of the form:

$$\begin{aligned} \Delta x_{t+1} &= A_t \Delta x_t + B_t \Delta u_t + c_t \\ c_t &= f(x_t^{ref}, u_t^{ref}) - x_{t+1}^{ref} \end{aligned} \tag{19}$$

where the c_t term is not zero.

In this case, we have chosen the same equilibrium points as before:

$$\begin{aligned} x_{init} &= [0, 0, 0, 0] \\ x_{fin} &= [90, -90, 0, 0] \end{aligned}$$

A smooth reference transition has been chosen (Fig 9), and as it is possible to see, it is much easier to track than a step. The time of simulation is selected as 10 seconds, and the discretization time is 0.01 seconds.

The weights in the cost have been selected as before:

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

$$R_t = 10$$

The algorithm takes 3 iterations to converge to a solution. As can be seen from the descent direction figure (14b), the value of the norm goes below 10^{-6}, which is the value we chose for the threshold.

Python code

The smooth reference trajectory has been defined in the python file reference_trajectory.py. In the code several functions are defined:

- **poly3(xx_init, xx_fin, tf, dt)** :Compute a smooth curve between xx_init and xx_fin connecting the two points with polynomial functions of order three. The input is computed as the gravity compensation along the obtained state reference curve. The initial and final configurations should be equilibrium points, and the reference curves are computed according to (17) and (18). The computation of q_2 depends on the given initial configuration. If the sum $q_1 + q_2$ at the initial configuration equals 0, the equation for the upward arm is used. Conversely, if it equals π, the equation for the downward arm is used.

  Arguments:

  - xx_init : Initial state vector in degrees.
  - xx_fin : Final state vector in degrees.
  - tf : Final time in seconds.

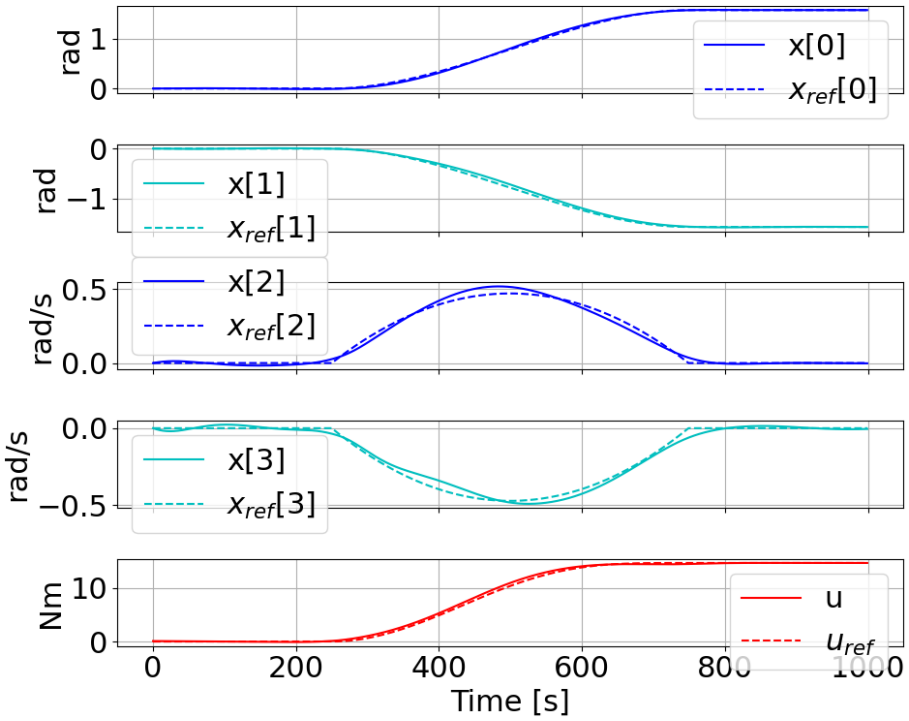


Figure 9: Reference and initial guess

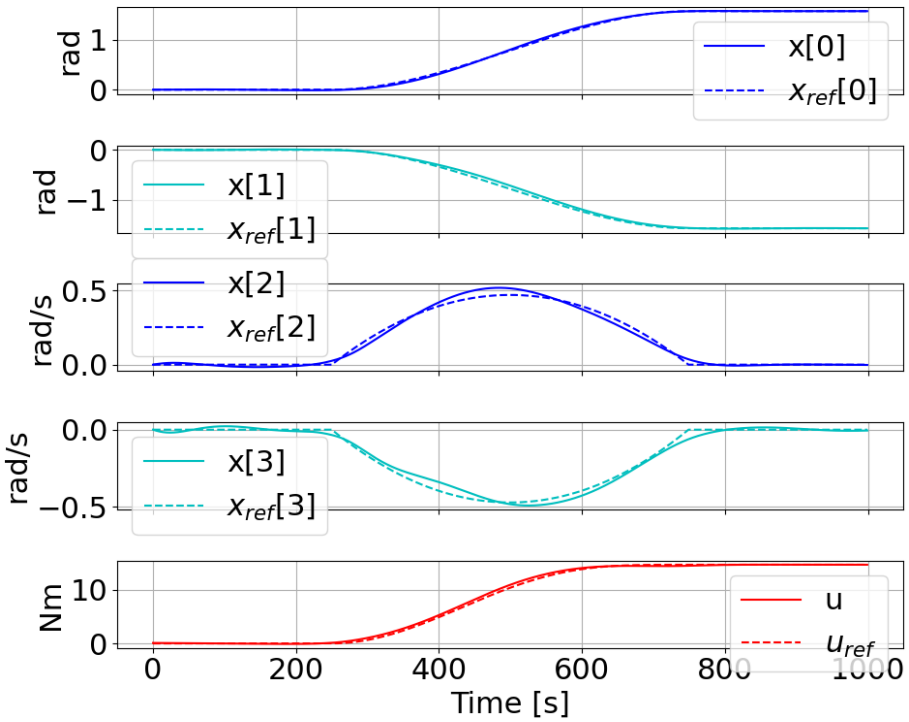


Figure 10: First iteration

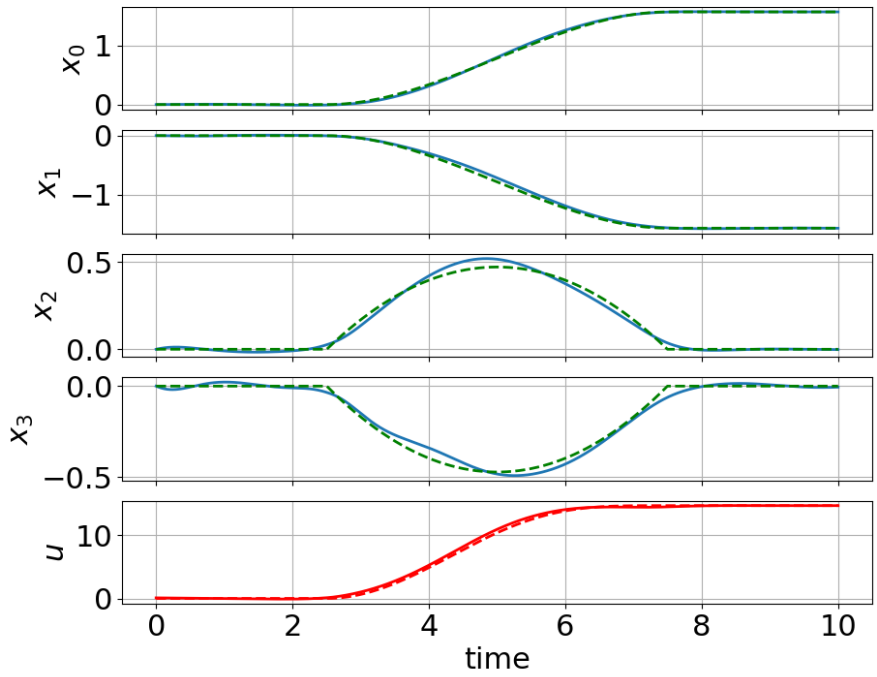


Figure 11: Final result

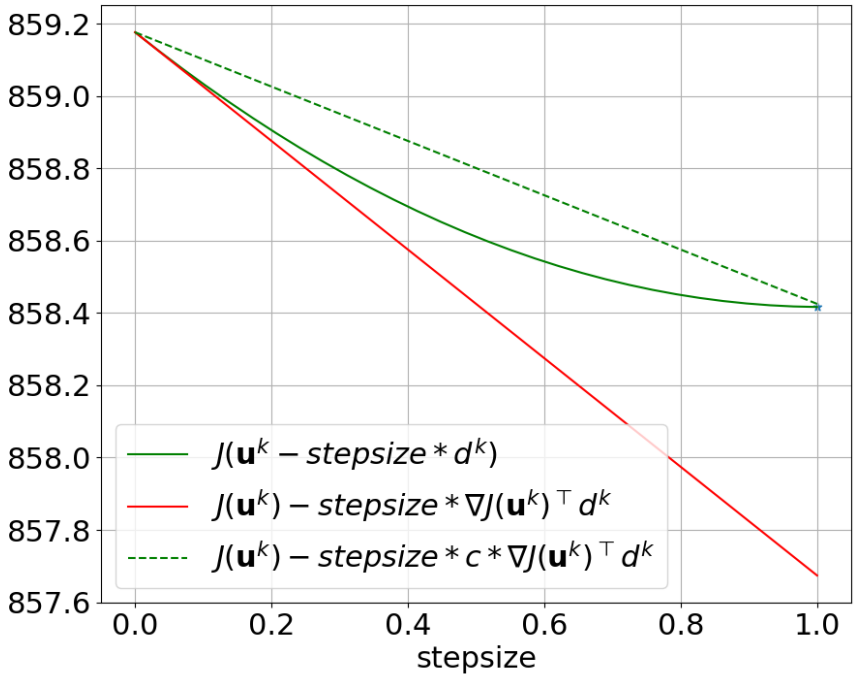


Figure 12: Backtracking line search plot at the first iteration

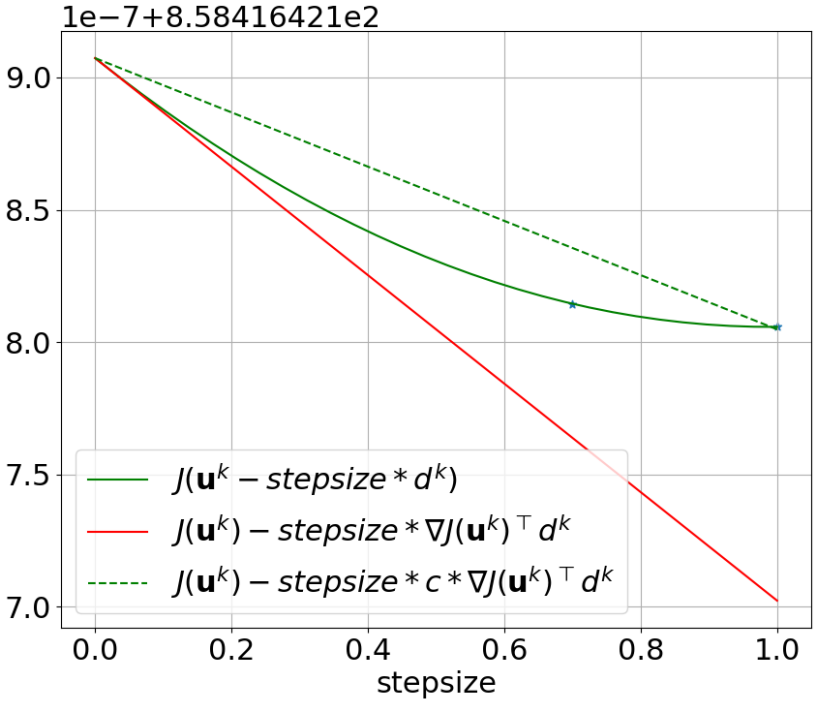
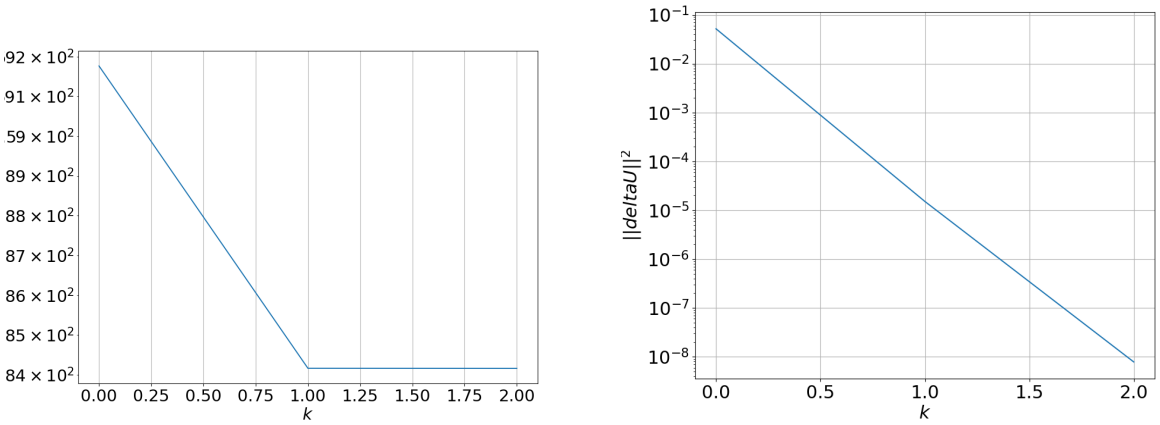


Figure 13: Backtracking line search plot at the final iteration



(a) Cost

(b) Descent direction

Figure 14

  - dt : Time of dicretization in seconds.

  Return:

  - xx_ref : Reference curve between the two states from 0 to tf, in radiants.
  - uu_ref : Input reference curve.

- **smooth_comp(points, tf, dt)** : Compute a composition of smooth curve between state configuration points. The input is computed as the gravity compensation along the obtained state reference curve.

  Arguments:

  - points : List of state vectors in degrees.
  - tf : Final time in seconds.
  - dt : Time of dicretization in second.

  Return:

  - xx_ref : State reference curve in radiants.
  - uu_ref : Input reference curve.

- **initial_guess(xx_ref, uu_ref, tf, dt)** : Compute a state-input trajectory tracking the system about a smooth reference curve using an LQR solution implemented in the **LQR_solver()** (see Task 3 for more details).

  Arguments:

  - xx_ref : Reference state vector.
  - uu_ref : Reference input vector.
  - tf : Final time in seconds.
  - dt : Time of dicretization in seconds.

  Return:

  - xx_r : State trajectory.
  - uu_r : Input trajectory.

Task 3

Task 3 requires linearizing the dynamics about the generated optimal trajectory (x^{star}, u^{star}) and using the LQR algorithm to define the optimal feedback controller to track the reference trajectory.

To compute the optimal gain for the LQR regulator, we have to solve the Linear Quadratic Optimization problem stated as follows:

$$\begin{aligned} &\min_{\Delta x \Delta u} \sum_{t=0}^{T-1} \Delta x_t^T Q^{reg} \Delta x_t + \Delta u_t^T R^{reg} \Delta u_t + \Delta x_T^T Q_T^{reg} \Delta x_T \\ &subj.to \quad \Delta x_{t+1} = A_t \Delta x_t + B_t \Delta u_t \\ &\qquad\qquad \Delta x_0 = 0 \end{aligned} \tag{20}$$

The cost is already a quadratic function of the displacement states and inputs with respect to the nominal trajectory.

The LQR problem can be solved using the Difference Riccati Equation, as in (12), for the matrix P_t and computing the optimal gain K_t^*.

The task assignment requires considering an initial perturbed condition different from 0. We have selected an initial perturbation of $\delta = [0.1, 0.1, 0, 0]^T$. In addition, we have applied noise inside the control loop.

- n_a: It is the actuation noise. It is a normally distributed random variable with mean value 0 and variance 0.01.
- n_m: It is the measurement noise. It is a normally distributed random variable with mean value 0 and variance 0.001.

The control law applies a feedforward action, which is the optimal input obtained with Newton's method, and then uses LQR to stabilize the system.

$$\begin{aligned} u_t &= K_t^*(x_{t+1} - x_t^{star} + n_m) + u_t^{star} + n_a \\ x_{t+1} &= f_d(x_t, u_t) \end{aligned} \tag{21}$$

The weights in the regulator cost have been selected as:

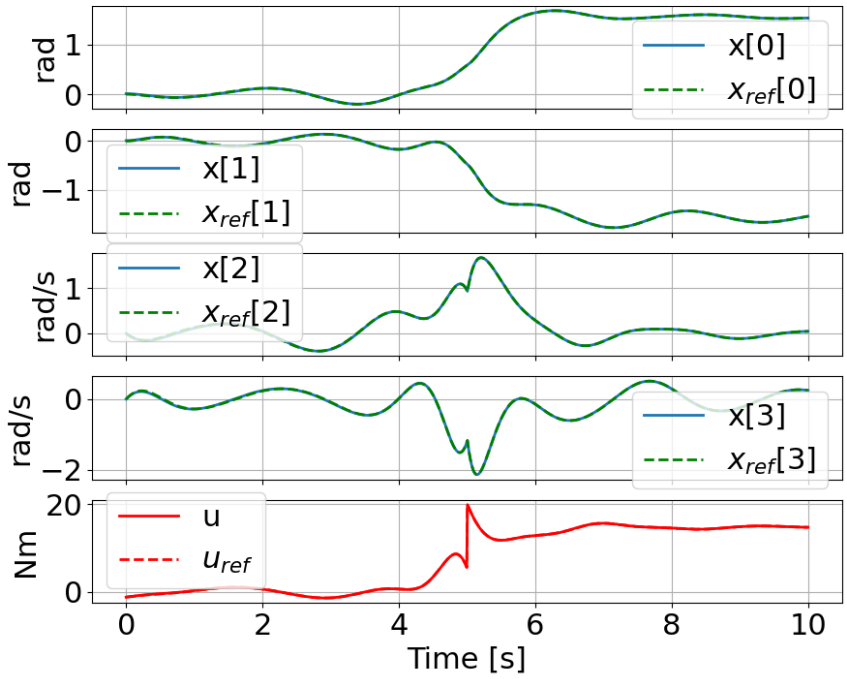


Figure 15: Reference Tracking with LQR

$$Q_t^{reg} = \begin{bmatrix} 1000 & 0 & 0 & 0 \\ 0 & 1000 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix} = Q_T^{reg}$$

$$R_t^{reg} = 10$$

As we can see from the plots, the system follows the reference, and the control loop is robust with respect to noise and a perturbed initial condition.

Python code

The LQR solver has been defined in the python file solver.py.

- **LQR_solver(xx_star, uu_star, dyn, A_f, B_f, QQt, RRt, tf, dt)** : Compute the optimal gain matrix K_t^* for each time instant.

  Arguments:

  - xx_star : Optimal state reference trajectory.
  - uu_star : Optimal input reference trajectory.
  - dyn : dynamic package imported from Dinamics.py.
  - A_f : Symbolic rappresentation of the discretized state jacobian.

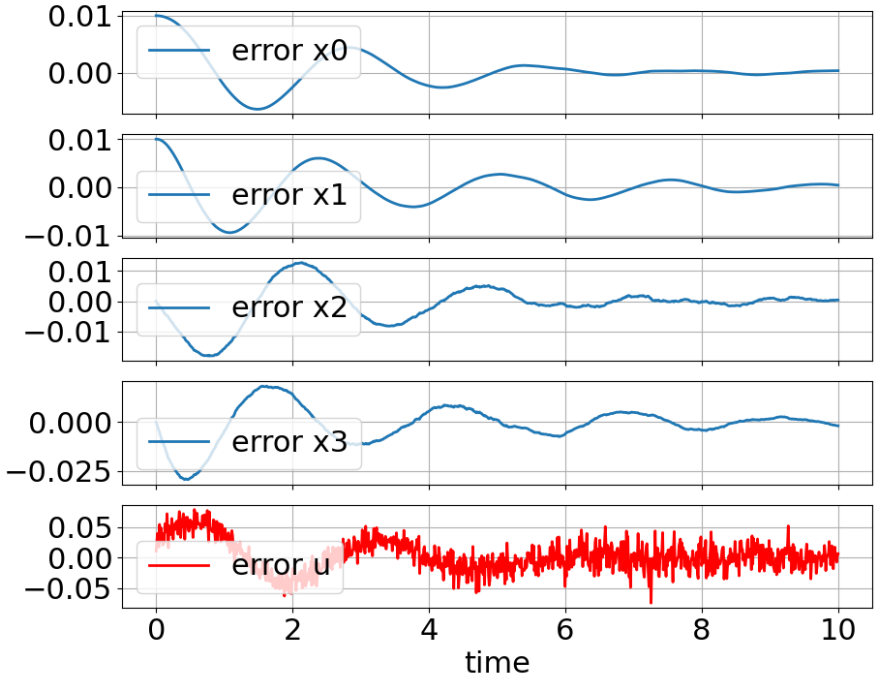


Figure 16: Tracking error

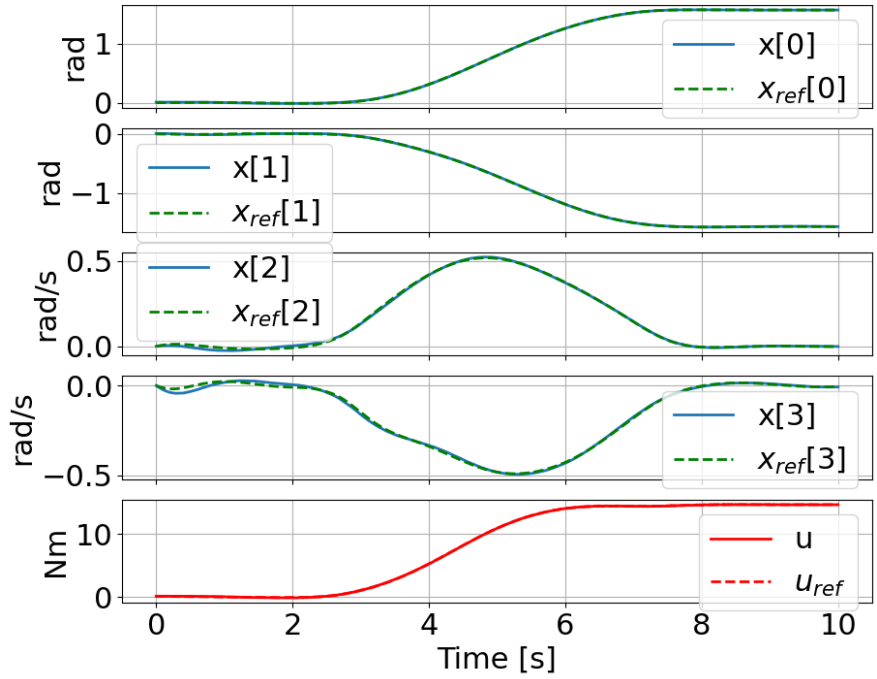


Figure 17: Smooth reference Tracking with LQR

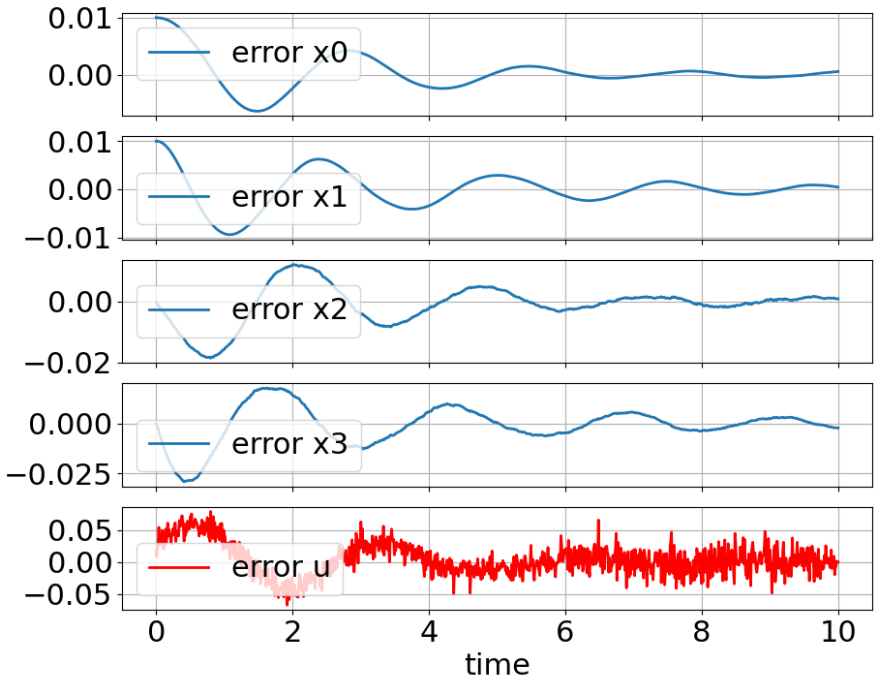


Figure 18: Smooth reference tracking error

- B_f : Symbolic rappresentation of the discretized input jacobian.
- QQt : Weight matrix for the states.
- RRt: Weight matrix for the inputs.
- tf : Final simulation time, in seconds.
- dt : Time of dicretization in seconds.

Return:

- KK_t : Optimal feedback gain matrix for each time instant.

Task 4

Task 4 requires linearizing the dynamics about the generated optimal trajectory (x^{star}, u^{star}) and exploiting the MPC algorithm to define a robust control law, able to track the reference trajectory.

Model Predictive Control is a technique that at each time iteration solves the whole optimal control problem for a fixed time window and applies only the first optimal input. After the first iteration, it moves the time window forward and starts computing the next optimal input. For this reason, Model Predictive Control works in an infinite time horizon.

How the optimal control problem is solved at each iteration depends on what cost and dynamics are present. This allows managing constraints directly with the certainty that they will be satisfied.

Since, in this case, constraints are not present, the cost is quadratic and the dynamics are linearized, we can solve the MPC problem by exploiting an LQR solution each time (22).

The task assignment requires considering an initial perturbed condition different from 0. In addition, we have applied noise inside the control loop like in Task 3.

The control law applies a feedforward action that is the optimal input obtained with Newton's method and then uses the MPC solution to stabilize the system.

$$\begin{cases} \min_{\Delta x \Delta u} \ \sum_{\tau=t}^{t+T-1} \Delta x_\tau^T Q^{reg} \Delta x_\tau + \Delta u_\tau^T R^{reg} \Delta u_\tau + \Delta x_{t+T}^T Q_T^{reg} \Delta x_{t+T} \\ subj.to \quad \Delta x_{\tau+1} = A_\tau \Delta x_\tau + B_\tau \Delta u_\tau \\ \qquad\qquad \Delta x_0 = x_t + n_m \\ \\ u_t = u_t^{mcp} + u_t^{star} + n_a \\ x_{t+1} = f_d(x_t, u_t) \end{cases} \tag{22}$$

The weights in the regulator cost have been selected as:

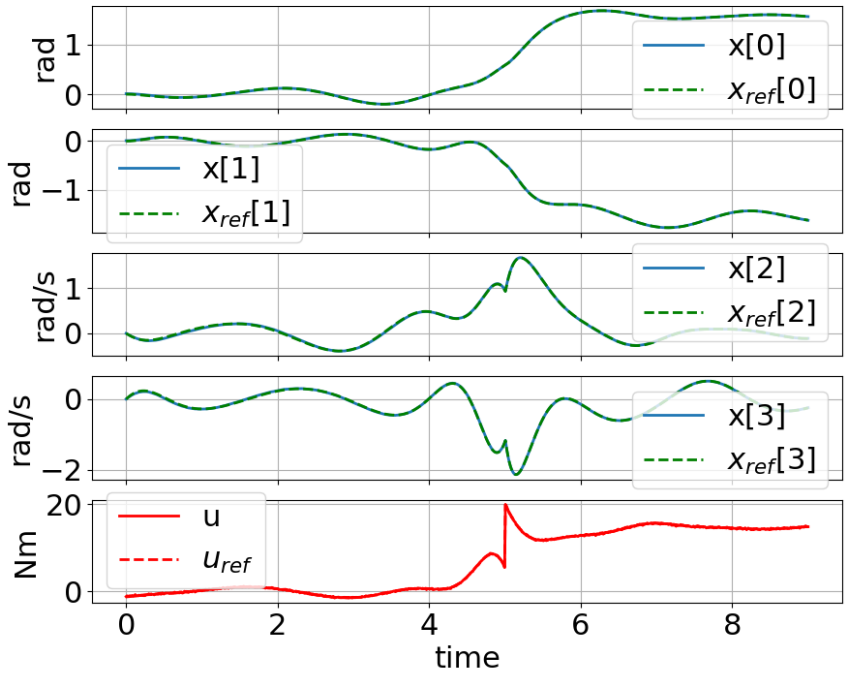


Figure 19: Reference Tracking with MPC

$$Q_t^{mpc} = \begin{bmatrix} 1000 & 0 & 0 & 0 \\ 0 & 1000 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix} = Q_T^{mpc}$$

$$R_t^{mpc} = 10$$

The time window we have chosen is 1 second (100 samples). We selected it by trying various values and chose a trade-off between performance and computational time.

As we can see from the plots, the system follows the reference, and the control loop is robust with respect to noise and a perturbed initial condition.

Python code

The MPC solver is defined in the Python file `solver.py`. The function calls the LQR function to obtain a gain K^*, and then uses it to compute the optimal input as $\Delta u_t = K_t \cdot \Delta x_t$.

- **solver_linear_mpc(dyn, A_f, B_f, QQ, RR, QQf, xx_t, xx_star, uu_star, dt, T_pred = 20)** : Compute the optimal gain matrix K_t^* for each time instant.

  Arguments:

  – dyn : dynamic package imported from Dinamics.py.

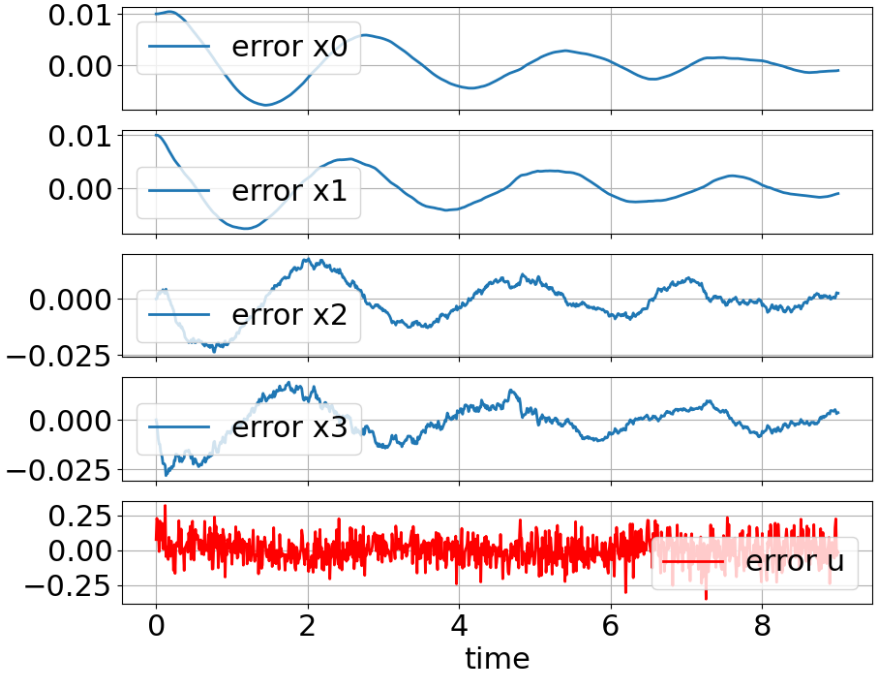


Figure 20: Tracking error

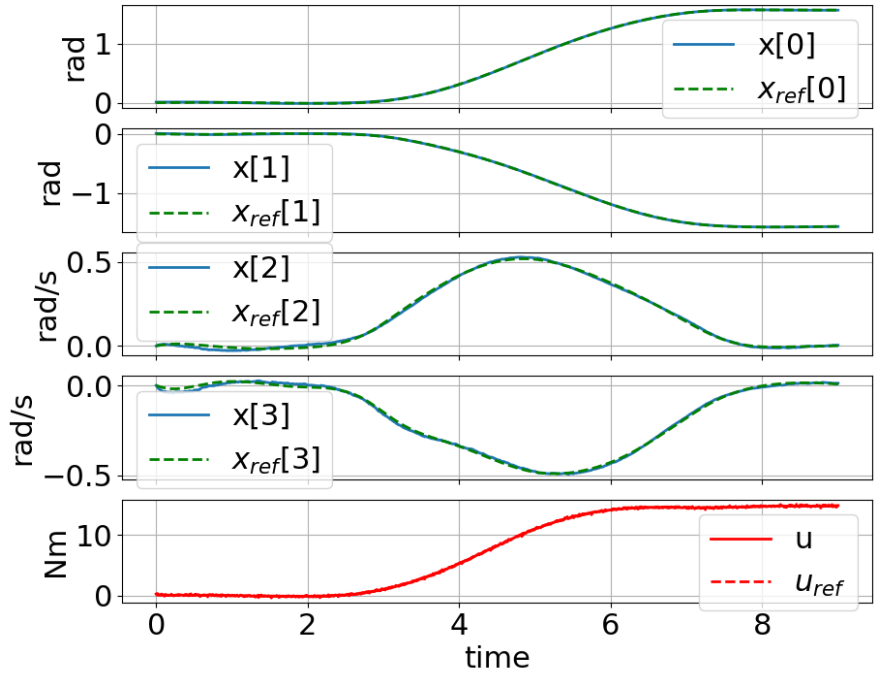


Figure 21: Smooth reference Tracking with MPC

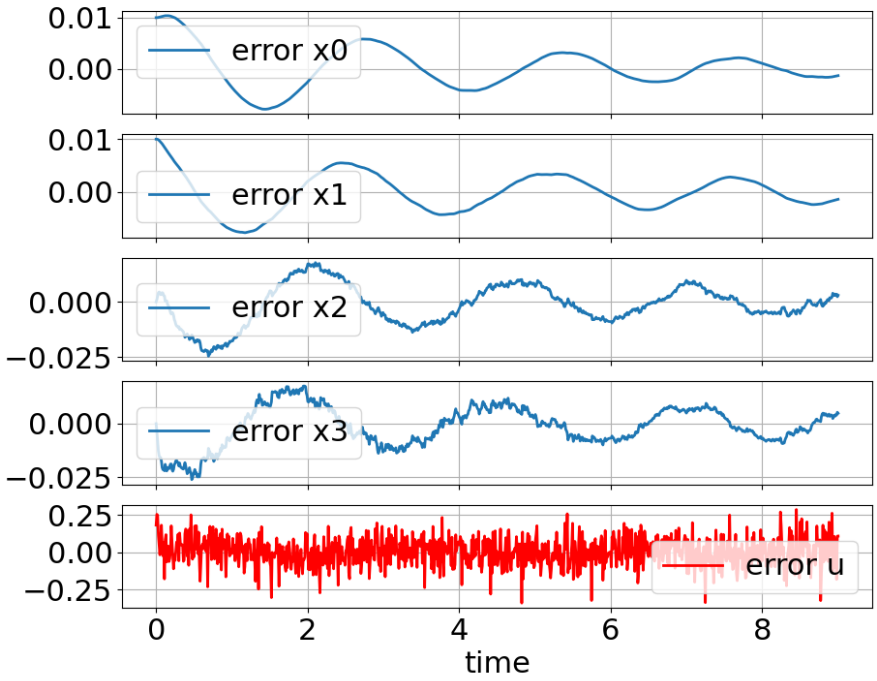


Figure 22: Smooth reference tracking error

- A_f : Symbolic rappresentation of the discretized state jacobian.
- B_f : Symbolic rappresentation of the discretized input jacobian.
- QQ : Weight matrix for the states.
- RR: Weight matrix for the inputs.
- QQf : Weight matrix for the states at the final time T.
- xx_t : measured input at time t.
- xx_star : Optimal state reference trajectory in the time window.
- uu_star : Optimal input reference trajectory in the time window.
- dt : Time of dicretization in seconds.
- T_pred : Time prediction window size in seconds.

Return:

- uu_mpc: Optimal input at time t.

Animation

For the animation, the subpackage `matplotlib.animation` has been used. The `FuncAnimation` function allows displaying consecutive plots as an animation. The robot is presented as a chain of two segments with dots at the extremities. The reference trajectory for the end-effector is plotted as a dashed line, allowing the motion to be appreciated from the beginning.

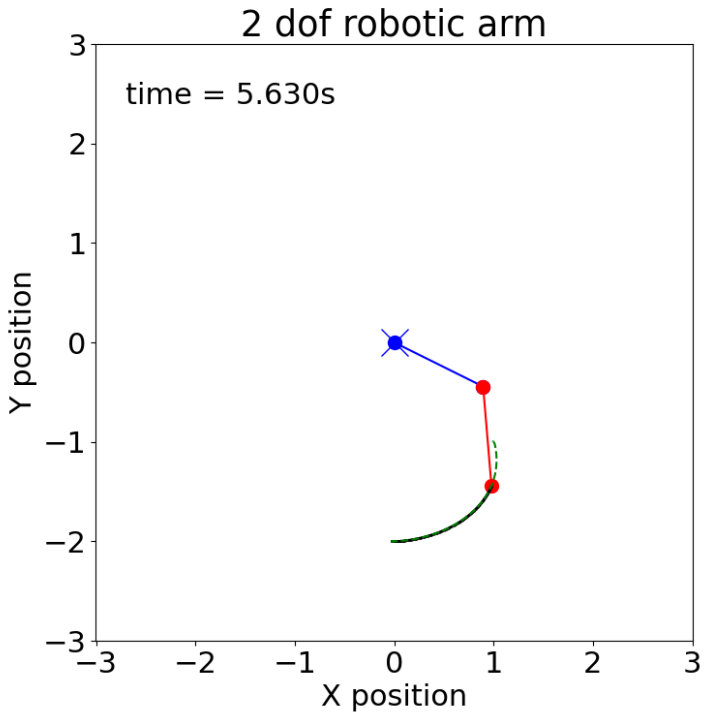


Figure 23: Animation for the smooth trajectory of Task 2

For the purpose of showing the various steps of the project, a simple trajectory has been presented in the previous figures. However, we have tested different trajectories that will be presented below with some plots and animations.

Swing up

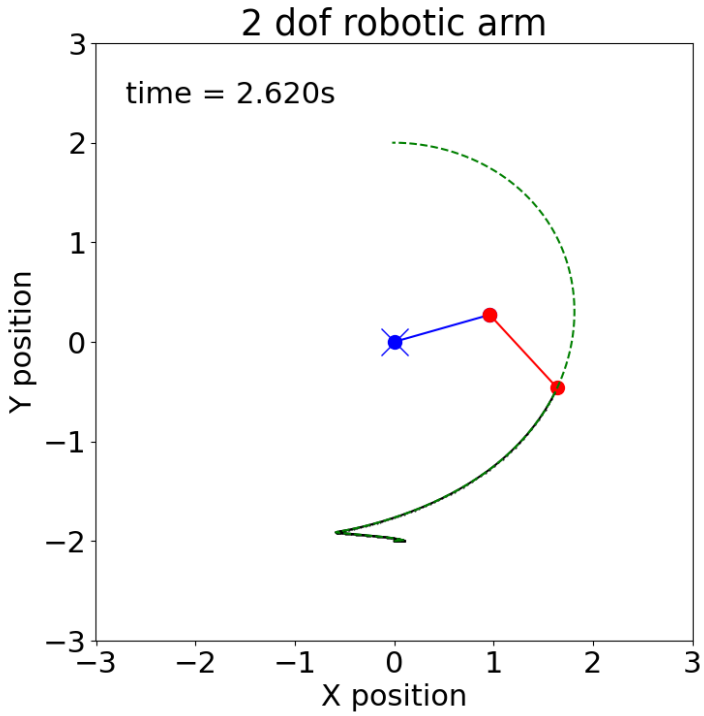


Figure 24: Animation for the swing up trajectory

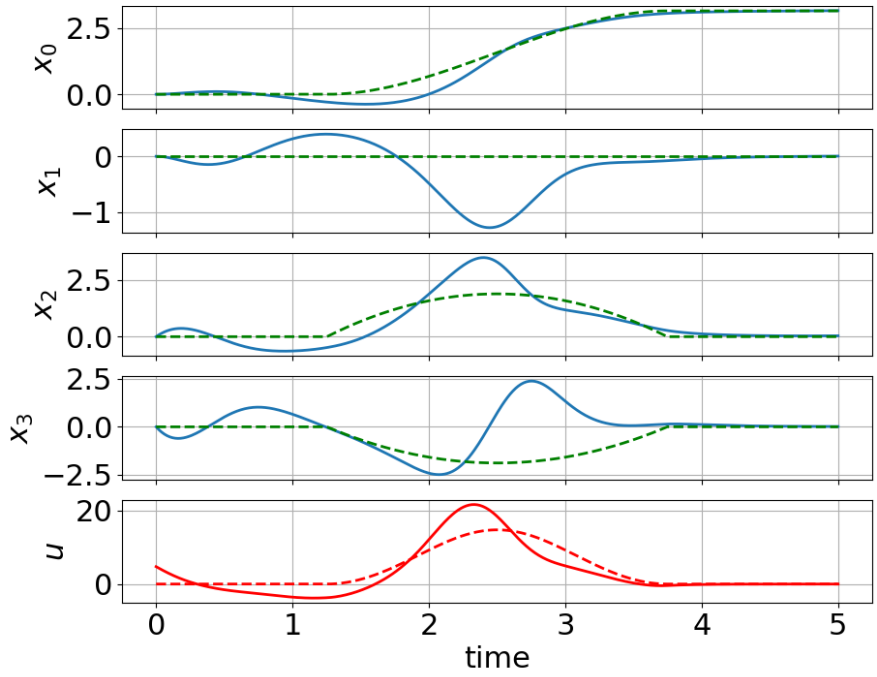


Figure 25: Swing up final result

Circular motion

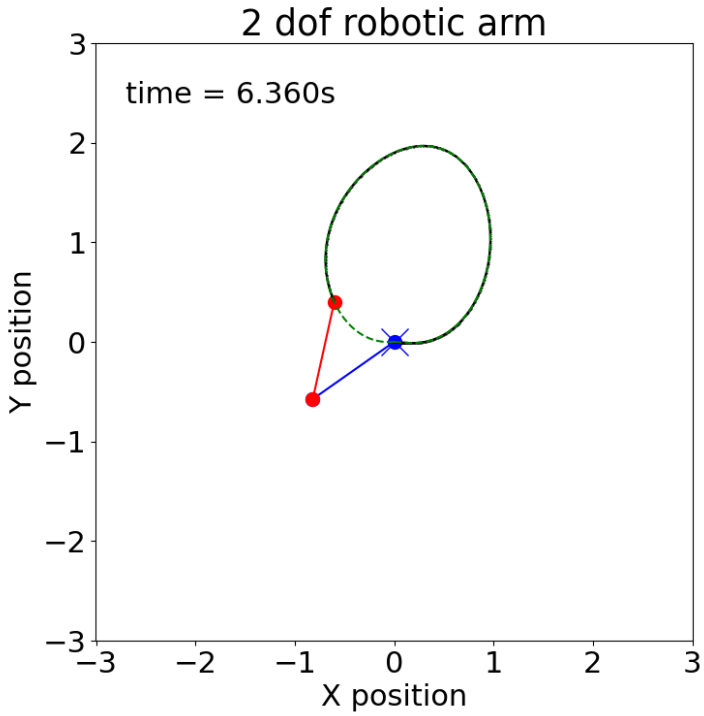


Figure 26: Animation for the circular trajectory

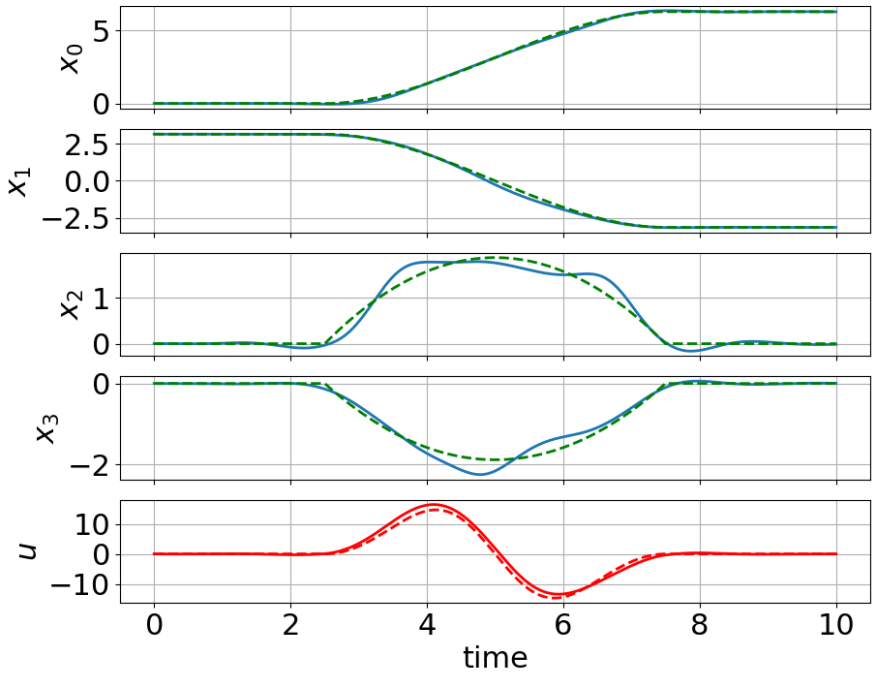


Figure 27: Circular motion final result

Conclusions

The project showed how the Optimal Control framework can effectively manage complex systems. It successfully controlled the underactuated manipulator and dealt with time-varying linear systems, such as the linearized model of the manipulator. Additionally, the framework demonstrated its capability to generate optimal trajectories for the system, ensuring smooth and efficient movement while satisfying the system's constraints.

The results proved that the control methods are robust, even when there is noise, making them reliable for practical use. Additionally, the exploration of different approaches highlighted the flexibility of the Optimal Control framework in solving similar problems.

Bibliography

[1] L. Sforni G. Noterstefano, S. Baroncini. *Optimal Control course slides*. University of Bologna, A.Y 2024 - 2025.