UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Optimal Control and Reinforcement Learning

Flexible Assignment - Group 23

Professor: **Giuseppe Notarstefano**
Dr: **Marco Falotico**

Students:

Alessandro Binci
Alessandro Tampieri
Lorenzo Tucci

Academic year 2025/2026

Abstract

This report presents the design and implementation of optimal control strategies for a flexible robotic link. The system is modeled as an underactuated double pendulum characterized by a non-linear rotational stiffness between the links. The project addresses the complete control pipeline, starting from the discretization of the non-linear dynamics and their linearization.

The primary objective is to generate an optimal trajectory that allows the system to transition between two equilibrium configurations. This is achieved using an optimal control method based on Newton's algorithm coupled with an Armijo line search to ensure convergence. Particular attention is given to the initialization strategy, employing a trapezoidal trajectory reference to handle the physical constraints of the flexible dynamics.

Subsequently, the report explores trajectory tracking strategies to stabilize the system along the computed optimal path. A Time-Varying Linear Quadratic Regulator is designed and tested to reject disturbances and correct initial state perturbations. Finally, a Model Predictive Control (MPC) scheme is implemented to handle constraints explicitly. Numerical simulations performed in Python demonstrate the effectiveness of the proposed methods and analyze the trade-off between control effort and tracking precision.

Contents

Introduction

The control of flexible robotic manipulators represents a significant challenge in the field of automation and robotics.

This project focuses on the optimal control of a *flexible robotic link*, modeled as a planar two-link manipulator with a passive elastic joint. The system can be essentially described as an under-actuated double pendulum where the first joint is actuated by a motor, while the second joint is passive and interacts with the first links.

Problem Statement

The state of the system is defined by the angular positions and velocities of the two links, $x = [\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]^T$, while the control input is the torque applied to the first joint, $u = \tau$.

The main objectives of this project are:

- **Modeling and Discretization:** Deriving the state-space equations of motion and discretizing them using the **Forward Euler** method to obtain a discrete-time model suitable for numerical optimization.
- **Trajectory Generation:** Computing an optimal feed-forward trajectory (state and input) to transition the system from an initial equilibrium configuration (at $\theta_1 = 0°$) to a final target configuration (at $\theta_1 = 90°$) tracking a previous defined reference curve. This is achieved using Newton's Method for optimal control in closed loop form. Note that the equilibrium configuration of the second link is imposed from the position of the first one, thus its angle will always be the opposite of the first one (i.e. $\theta_2 = -\theta_1$).
- **Stabilization and Tracking:** Designing feedback control laws, specifically a Time-Varying Linear Quadratic Regulator and Model Predictive Control (MPC), to stabilize the system along the generated trajectory in the presence of disturbances.

Report Structure

The report is organized to mirror the project tasks assignment:

- **Chapter 1 (Task 0)** discretize the dynamics of the flexible link and write the discrete-time state-space equations using the Forward Euler method.
- **Chapter 2 (Task 1)** details the first trajectory generation phase, including the numerical computation of equilibrium points and the application of Newton's algorithm in closed-loop form.
- **Chapter 3 (Task 2)** presents the refined trajectory generation strategy, introducing a trapezoidal reference curve to improve physical feasibility, algorithm convergence and to make the previous reference curve smoother.
- **Chapter 4 (Task 3)** focuses on the design of the tracking controller using a Time-Varying Linear Quadratic Regulator and analyzes the closed-loop simulation results.
- **Chapter 5 (Task 4)** discusses the implementation of the Model Predictive Control (MPC) strategy for constraints handling and analyze the online simulation results.
- **Chapter 6 (Task 5)** describes the visualization tools developed to animate the flexible link manipulator based on the results of Task 3, providing a graphical verification of the resulting.
- **Conclusions** summarize the findings and the performance of the proposed control strategies.

Contributions

DA CHIEDERE AL TUTOR

All authors contributed equally to the mathematical derivation, code implementation, and the writing of this report.

Chapter 1

Mathematical Modeling (Problem setup -Task 0)

The first phase of the project (Task 0) involves the discritezation of the dynamics given by the equations provided by the assignment.

1.1 Physical System Description

The system under analysis is a planar two-link robotic arm. It is modeled as a double pendulum where:

- The first joint (base) is actuated by a motor providing a torque τ.
- The second joint is **passive** (under-actuated).

Let $q = [\theta_1, \theta_2]^T$ be the vector of generalized coordinates, where:

- θ_1 is the absolute angle of the first link with respect to the vertical axis where the origin coincides with the first joint.
- θ_2 is the relative angle between the first and the second link.

1.2 Equations of Motion

The dynamics is derived using the Euler-Lagrange formalism. The general form of the equations of motion for a robotic manipulator is:

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + F\dot{q} + G(q) = Bu \tag{1.1}$$

where:

- $M(q) \in \mathbb{R}^{2\times 2}$ is the symmetric, positive-definite inertia matrix.
- $C(q, \dot{q})\dot{q} \in \mathbb{R}^2$ represents Coriolis and centrifugal forces.

- $F \in \mathbb{R}^{2\times 2}$ is the diagonal viscous friction matrix.
- $G(q) \in \mathbb{R}^2$ is the vector of potential forces (gravity and elasticity).
- $u = [\tau]$ is the scalar control input and $B = [1, 0]$ maps the input to the actuated joint.

Based on the symbolic derivation implemented in Python (using the `SymPy` library), the specific components of the dynamic model are detailed below.

1.2.1 Inertia Matrix

The mass matrix $M(q)$ describes the inertial properties of the links:

$$M(q) = \begin{bmatrix} I_1 + I_2 + l_{c1}^2 m_1 + m_2(l_1^2 + 2l_1 l_{c2}\cos(\theta_2) + l_{c2}^2) & I_2 + l_{c2} m_2(l_1\cos(\theta_2) + l_{c2}) \\ I_2 + l_{c2} m_2(l_1\cos(\theta_2) + l_{c2}) & I_2 + m_2 l_{c2}^2 \end{bmatrix} \tag{1.2}$$

1.2.2 Coriolis and Centrifugal Matrix

The matrix $C(q, \dot{q})$ accounts for the coupling effects between velocities:

$$C(q, \dot{q}) = \begin{bmatrix} -l_1 l_{c2} m_2 \dot{\theta}_2 \sin(\theta_2) & -l_1 l_{c2} m_2(\dot{\theta}_1 + \dot{\theta}_2)\sin(\theta_2) \\ l_1 l_{c2} m_2 \dot{\theta}_1 \sin(\theta_2) & 0 \end{bmatrix} \tag{1.3}$$

1.2.3 Gravity and Stiffness Vector

The vector $G(q)$ contains both the gravitational terms and the specific **nonlinear stiffness** model of the flexible joint. According to the assignment specifications, the elastic torque is modeled as $\tau_{el} = 3k\sin(\theta_2)\cos^2(\theta_2)$.

$$G(q) = \begin{bmatrix} g l_{c1} m_1 \sin(\theta_1) + g m_2(l_1 \sin(\theta_1) + l_{c2}\sin(\theta_1 + \theta_2)) \\ g m_2 l_{c2}\sin(\theta_1 + \theta_2) + \underbrace{3k\sin(\theta_2)\cos^2(\theta_2)}_{\text{Elastic Torque}} \end{bmatrix} \tag{1.4}$$

1.2.4 Viscous Friction

Energy dissipation is modeled via a diagonal viscous friction matrix:

$$F = \begin{bmatrix} f_1 & 0 \\ 0 & f_2 \end{bmatrix} \tag{1.5}$$

1.2.5 System Parameters

The numerical validation of the model and the control strategies is performed using the parameters listed in Table 1.1, corresponding to Set 1 of the assignment.

Parameter	Symbol	Value
Link Masses	m_1, m_2	1.0 [kg]
Link Lengths	l_1, l_2	1.0 [m]
Center of Mass Distances	l_{c1}, l_{c2}	0.5 [m]
Moments of Inertia	I_1, I_2	0.33 [kg·m^2]
Viscous Friction	f_1, f_2	1.0 [Ns/rad]
Stiffness Coefficient	k	0.5 [Nm/rad]
Gravity	g	9.81 [m/s^2]

Table 1.1: Physical parameters of the flexible link manipulator.

1.3 State-Space Representation

To facilitate numerical simulation and control design, the system is converted into state-space form $\dot{x} = f(x, u)$. The state vector is defined as $x = [\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]^T$. By inverting the inertia matrix, the acceleration $\ddot{q}$ is computed as:

$$\ddot{q} = M(q)^{-1} \left(Bu - C(q, \dot{q})\dot{q} - F\dot{q} - G(q) \right) \tag{1.6}$$

Thus, the continuous-time system dynamics is:

$$\dot{x} = \frac{d}{dt} \begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ M(q)^{-1}(Bu - C\dot{q} - F\dot{q} - G(q)) \end{bmatrix} \tag{1.7}$$

1.4 Discretization Strategy (Forward Euler)

The continuous-time dynamics derived in the previous section, $\dot{x} = f_{CT}(x, u)$, must be discretized to be implemented in a simulation environment and to solve the optimal control problem numerically. For this project, we adopted the **Forward Euler** explicit integration scheme (which is implemented for Task 0 and used later on in other tasks).

1.4.1 Discrete-Time Dynamics

The Forward Euler method approximates the time derivative of the state vector as a finite difference. Using the notation t for the discrete time step:

$$x_{t+1} = x_t + \delta \cdot f_{CT}(x_t, u_t, t) \tag{1.8}$$

where x_t denotes the state at time step t. In our Python implementation (`dynamics.py`), this step is constructed symbolically using `simpy` library, while it is calculated numerically using `numpy` library. The choosen discretization time step for this project is equal to 0.005 and it is possible to change it in `dynamics.py` at line 11 (`dt = 5 * 1e-3`).

1.4.2 Linearization and Jacobians

A crucial part of the Newton-based optimal control algorithm (Task 1) is the computation of the linearized dynamics around a nominal trajectory. The discrete-time linearized system is defined as:

$$\Delta x_{t+1} = A_t \Delta x_t + B_t \Delta u_t \tag{1.9}$$

where A_t and B_t are the following Jacobian matrices:

$$A_t = \nabla_x f_d(x,u)\Big|_{\bar{x_t},\bar{u_t}} = \frac{\partial f_d(x,u)}{\partial x}\Big|_{\bar{x_t},\bar{u_t}} \tag{1.10}$$

$$B_t = \nabla_u f_d(x,u)\Big|_{\bar{x_t},\bar{u_t}} = \frac{\partial f_d(x,u)}{\partial u}\Big|_{\bar{x_t},\bar{u_t}} \tag{1.11}$$

Chapter 2

Optimal Trajectory Generation (Task 1)

The objective of Task 1 is to compute two equilibrium configurations of the flexible arm and, subsequently, to generate a feasible and optimal trajectory that drives the flexible link manipulator from the first equilibrium point to the second one over a time horizon of $T = 30.5\,\text{s}$. This choice comes from an evaluation of the trajectories along different time values, resulting the best tradeoff between the optimal results and the computation efficiency.

This problem is solved in three steps:

1. Computation of the equilibrium configuration for states and inputs.
2. Definition of a reference trajectory (Step function).
3. Solution of the non-linear optimal control problem using Newton's-like algorithm in closed-loop version.

2.1 Equilibrium Computation

Before starting the optimization, it is necessary to define the boundary conditions.

First of all, two equilibrium configurations must be determined and two different approaches are employed to compute them. In the first approach, a desired torque equal to zero is imposed, and an initial configuration of $\theta_1 = 45°$ is selected. Newton's Method algorithm for root-finding is then applied. Through this iterative procedure, the first equilibrium configuration is obtained. A point is considered as an equilibrium one when the error between its own current value and its own value at the next iteration is less or equal than a certain tolerance (1×10^{-8}) defined in `equilibrium.py`.

The second equilibrium configuration is determined by first prescribing a desired equilibrium posture with $\theta_1 = 90°$. Given that actuation is available

only at the first joint, the second joint angle is assumed to be opposite to the first one, i.e., $\theta_2 = -\theta_1$.

The corresponding equilibrium torque τ is then obtained by solving the inverse dynamics of the system. In conclusion, considering the state vector defined as:

$$\mathbf{x} = [\theta_1,\ \theta_2,\ \dot{\theta}_1,\ \dot{\theta}_2]^\top,$$

the resulting equilibrium configurations, developed in `equilibrium.py`, are:

$$\mathbf{x}_{eq,1} = [0,\ 0,\ 0,\ 0]^\top \quad \text{with } \tau = 0 \text{ N m}$$

$$\mathbf{x}_{eq,2} = [90^\circ,\ -90^\circ,\ 0,\ 0]^\top \text{with } \tau = 14.715 \text{ N m}\ .$$

2.2 Reference Trajectory (Step)

The design of the step reference trajectory is defined in a piecewise-constant manner inside `reference_curve.py`. Specifically, the reference is set equal to the first equilibrium configuration for the initial 15.25 s (T/2) and equal to the second one for the remaining time over the total horizon, that it is equal to $T = 30.5$ s. In this way, a step reference trajectory is obtained.

Therefore, the reference state x_{ref} and input u_{ref} are defined as:

$$x_{ref,t} = \begin{cases} x_{eq,1} & \text{if } t < 15.25 \text{ s} \\ x_{eq,2} & \text{if } t \geq 15.25 \text{ s} \end{cases}, \quad u_{ref,t} = \begin{cases} u_{eq,1} & \text{if } t < 15.25 \text{ s} \\ u_{eq,2} & \text{if } t \geq 15.25 \text{ s} \end{cases} \tag{2.1}$$

2.3 Newton's Method for Optimal Control in Closed-loop form

Newton's method is an iterative second-order optimization method used to solve optimal control problems for trajectory tracking (`main_task1.py`). The method relies on the solution of an affine LQR problem in order to compute an optimal and feasible trajectory that tracks the reference curve. It can be decomposed into the following three main steps:

- **Initialization:** an initial guess for the state and input trajectories, denoted as $(\mathbf{x}^0, \mathbf{u}^0)$, is selected.
- **Computation of the descent direction:** the descent direction is obtained by solving the associated affine LQR problem.
- **Trajectory update:** a new state–input trajectory is computed by applying the Armijo step-size selection rule.

2.3.1 Initialization

Due to the discontinuity of the reference trajectory, a direct use of the reference as initial guess would be infeasible. Therefore, Newton's method is initialized using a **static trajectory** corresponding to the initial equilibrium configuration:

$$\mathbf{x}_t^{(0)} = x_{eq,1}, \quad \mathbf{u}_t^{(0)} = u_{eq,1} \quad \forall t \in [0,\, T]. \tag{2.2}$$

This initialization guarantees feasibility of the initial iterate and enables the optimizer to converge towards the desired step transition.

2.3.2 Computation of the descent direction

Subsequently, the maximum number of Newton's method iterations is set to 20 and the cost time-invariant matrices $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{S}$ are imported from `cost.py`.

To find the optimal feasible trajectory, we formulated a finite-horizon optimal control problem minimizing the following cost function:

$$l(\mathbf{x}, \mathbf{u}) = \sum_{t=0}^{T-1} l_t(x_t, u_t) + l_T(x_T) \tag{2.3}$$

where the stage cost penalizes the deviation from the step reference:

$$l_t(x, u) = \frac{1}{2}(x - x_{ref,t})^T Q (x - x_{ref,t}) + \frac{1}{2}(u - u_{ref,t})^T R (u - u_{ref,t}) \tag{2.4}$$

while the terminal cost does the same of the stage cost at final time T:

$$l_T(x) = \frac{1}{2}(x - x_{ref,T})^T Q (x - x_{ref,T}) \tag{2.5}$$

In particular, in `cost.py` a quadratic tracking cost function and the corresponding gradients are defined.

The weighting matrices are then tuned through a trial-and-error procedure, leading to the following selection:

$$\mathbf{Q} = \mathrm{diag}(100,\, 100,\, 1,\, 1), \quad \mathbf{R} = 1, \tag{2.6}$$

which prioritizes accurate tracking of the joint angles over the angular velocities. The high penalties on the first two diagonal elements of Q ensure that the solver focuses on reaching the desired configuration, while the lower weights on velocities allow for faster transients. The input penalty R is tuned to avoid actuator saturation while permitting sufficient control authority. Then the problem is solved using **Newton's Method**.

At each iteration, the nonlinear optimal control problem is locally approximated by an affine LQR subproblem obtained through linearization of

the dynamics and local expansion of the tracking cost around the current trajectory.

So at each Newton iteration, the system dynamics are linearized along the current trajectory by repeatedly evaluating the Jacobian matrices returned by the functions defined in `dynamics.py`.

The trajectory update is then performed by applying the feedback gain $\mathbf{K}$ and the feedforward term $\boldsymbol{\sigma}$, which are obtained by solving the time-varying Riccati equations associated with the affine LQR subproblem in `ltv_solver_LQR.py`.

2.3.3 Trajectory Update

Subsequently, a new candidate trajectory is generated by performing a forward rollout of the original nonlinear dynamics in closed-loop form. The control input is updated according to the law

$$\mathbf{u}_t^{k+1} = \mathbf{u}_t^k + \gamma^k {\boldsymbol{\sigma}^k}_t + K_t^k \left(\mathbf{x}_t^{k+1} - \mathbf{x}_t^k \right), \tag{2.7}$$

where $\gamma \in (0, 1]$ denotes the step size.

The step size γ is selected using an Armijo backtracking line search strategy. The initial step size is set to $\gamma_{\text{init}} = 1$, and it is iteratively reduced by a scaling factor $\beta = 0.7$ until the Armijo sufficient decrease condition, with acceptance parameter $c = 0.5$, is satisfied.

Once the condition holds, the updated trajectory is accepted and used as the nominal trajectory for the subsequent Newton iteration.

2.4 Simulation Results

The algorithm successfully converged, transforming the discontinuous step reference into a smooth, optimal trajectory. The results are analyzed below, detailing the final optimal solution and the evolution of the algorithm.

2.4.1 Optimal Trajectories

The following figures illustrate the final computed solutions. The optimizer finds a trajectory that anticipates the transition and manages the link flexibility without violent oscillations. While the reference (dashed black line) commands an instantaneous jump, the optimal control (solid colored lines) creates a smooth transition for both link positions (θ_1, θ_2) and the torque input.

This result has been improved by choosing the trajectory duration time equal to $30.5s$; indeed, a higher horizon is helpful for tracking purposes since the algorithm has more time to achieve better results. It is possible to notice such behavior by comparing two trajectories with different time horizons.

Moreover, in the figures corresponding to the $T = 10\,\mathrm{s}$ horizon, different values of the cost matrices were adopted with respect to those selected for the final solution. In particular, the previous tuning employed $\mathbf{Q} = \mathrm{diag}(100, 100, 0.1, 0.1)$ and $\mathbf{R} = 0.1$.

To enforce such result, for Task 1, the comparison between plots of the $10s$ trajectory and the $30s$ one will be shown.

In the following figures (2.1 and 2.2) the results about positions and input can be seen:

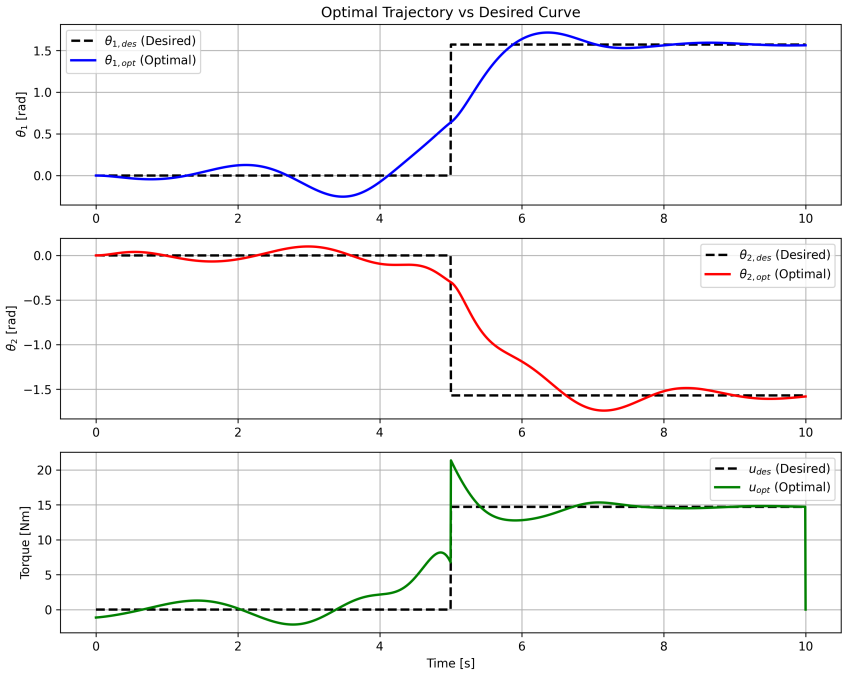

Figure 2.1: $10s$ - Optimal State Trajectories vs Desired Step.

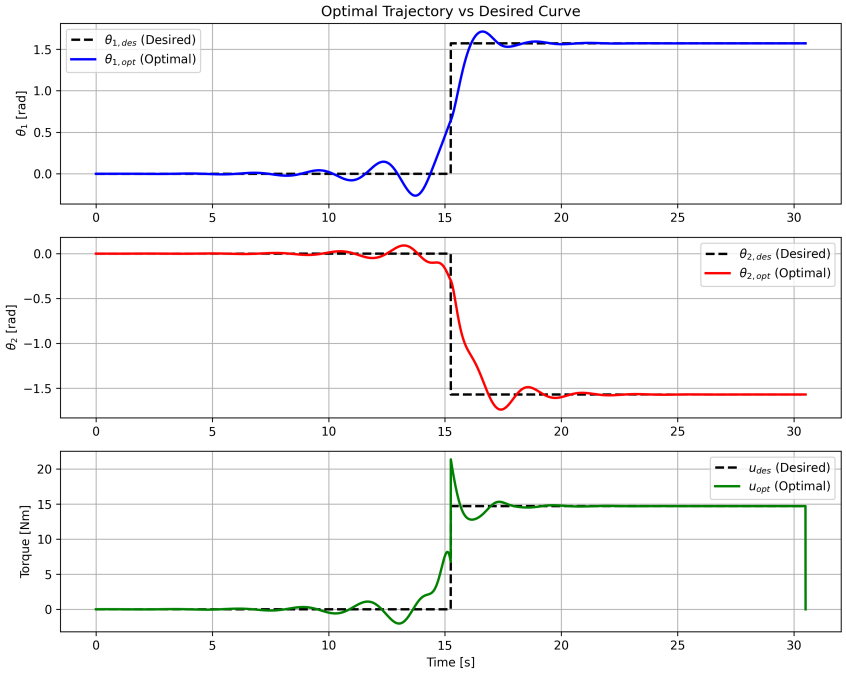

Figure 2.2: $T = 30.5s$ - Optimal State Trajectories vs Desired Step.

In addition to positions, it is crucial to verify the behavior of the velocities. The following figures (2.3 and 2.4) show the angular rates of the two links.

The velocities, in the second case, remain bounded and smooth throughout the motion, confirming the physical feasibility of the computed control law. Indeed, the maximum peak of $\dot{\theta}_1$ decreases from approximately 2 to about 1.30 due to the effect of the changed values in the cost matrices.

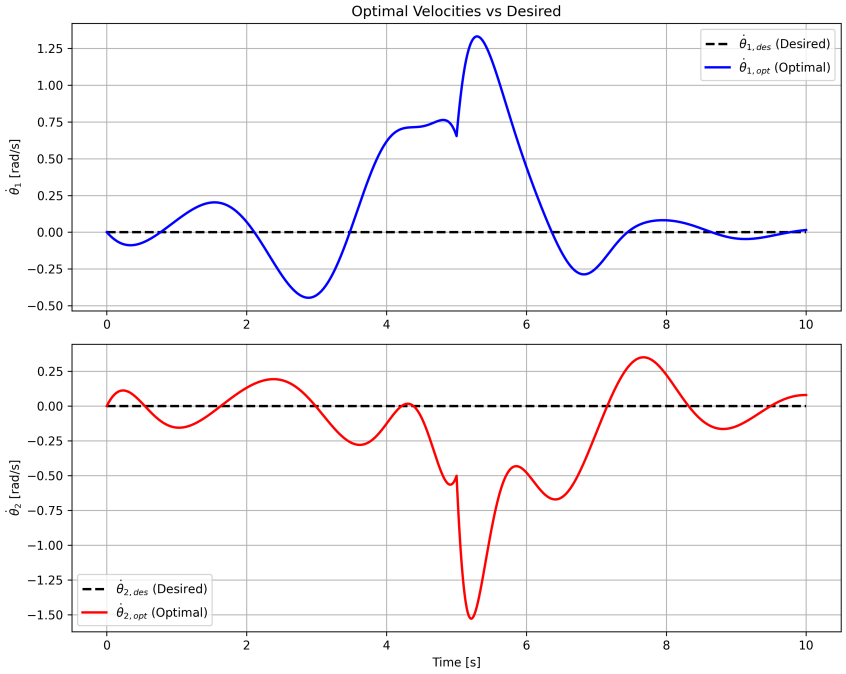

Figure 2.3: $T = 10s$ - Optimal Velocity Profiles vs Desired Velocities.

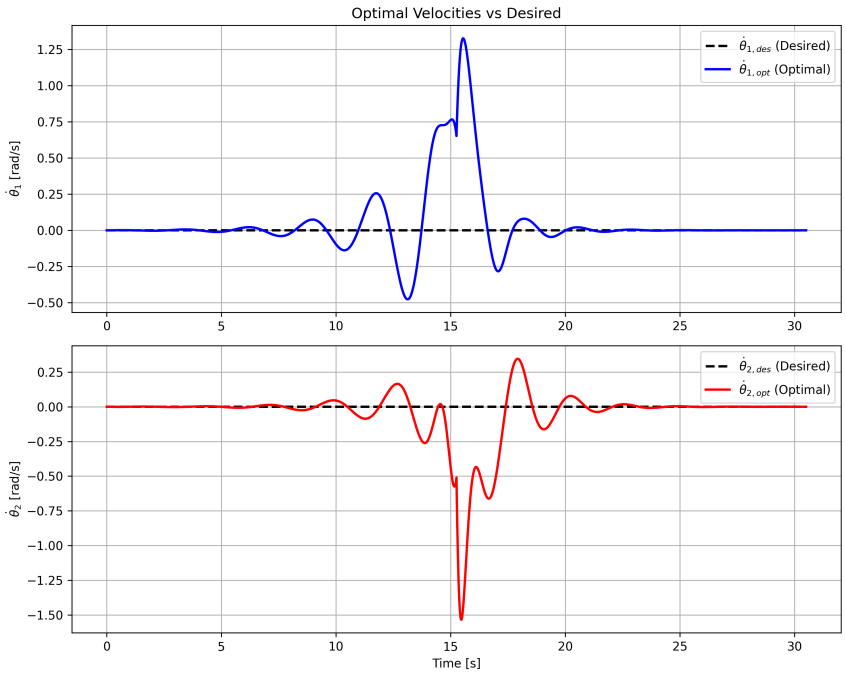

Figure 2.4: $T = 30.5s$ - Optimal Velocity Profiles vs Desired Velocities.

2.4.2 Algorithm Evolution

To understand how Newton's method reached the solution, it is shown the evolution of the trajectories over the iterations. Figure 2.5 shows how the solver modifies the initial guess (Iter 0, orange). In the very first iteration (Iter 1, blue), the algorithm already performs a significant correction, bringing the system close to the final optimal shape (red).

By comparing the two scenarios, it can be observed that, with respect to Fig. 2.5, the evolution shown in Fig. 2.6 exhibits a less pronounced improvement of the optimal trajectory during the initial iterations. Furthemore, its final solution achieves a more stable tracking behavior.

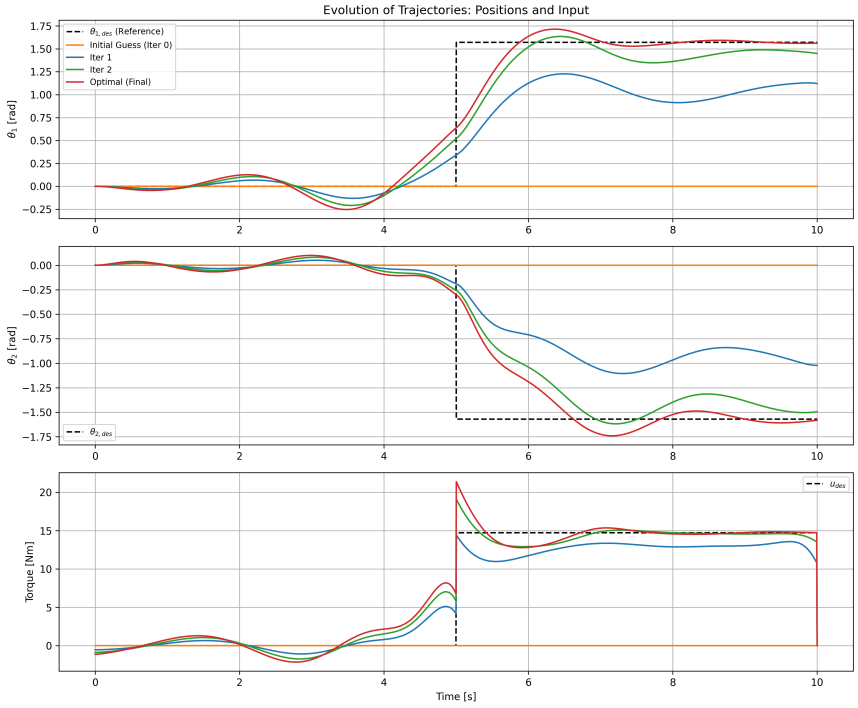

Figure 2.5: $T = 10s$ - Evolution of Positions and Input over iterations.

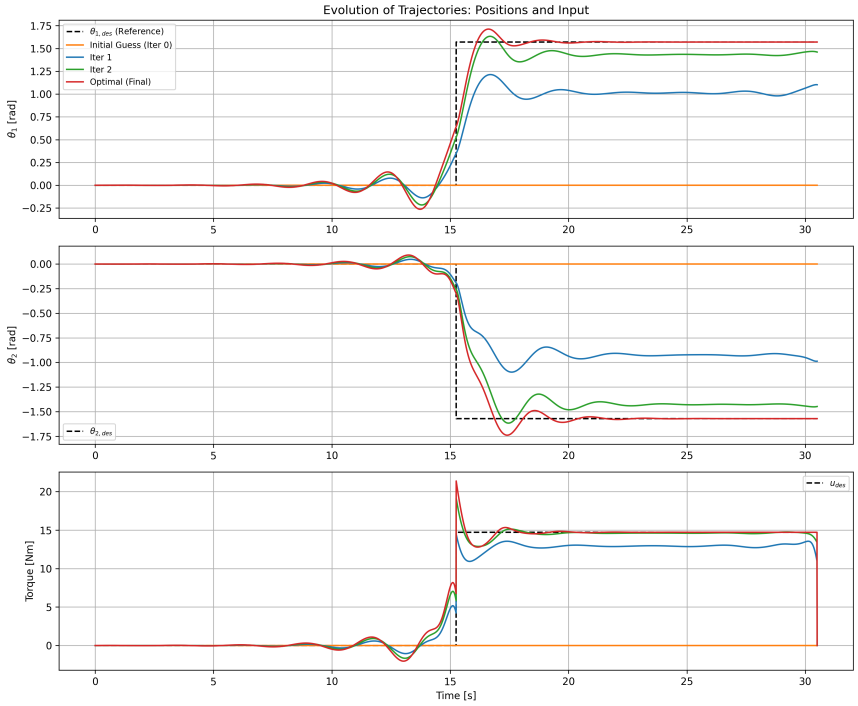

Figure 2.6: $T = 30.5s$ - Evolution of Positions and Input over iterations.

Similarly, Figure 2.7 displays the evolution of the velocity profiles. The initial guess had zero velocity everywhere; the solver introduces the necessary acceleration and deceleration phases to move the link. Note that, also in this case, adopting different values in the cost matrices, smaller peak values are obtained. In particular, in Figure 2.5 is possible to see an higher peak because the value of the $\mathbf{R}$ matrix is 0.1 instead of 1.

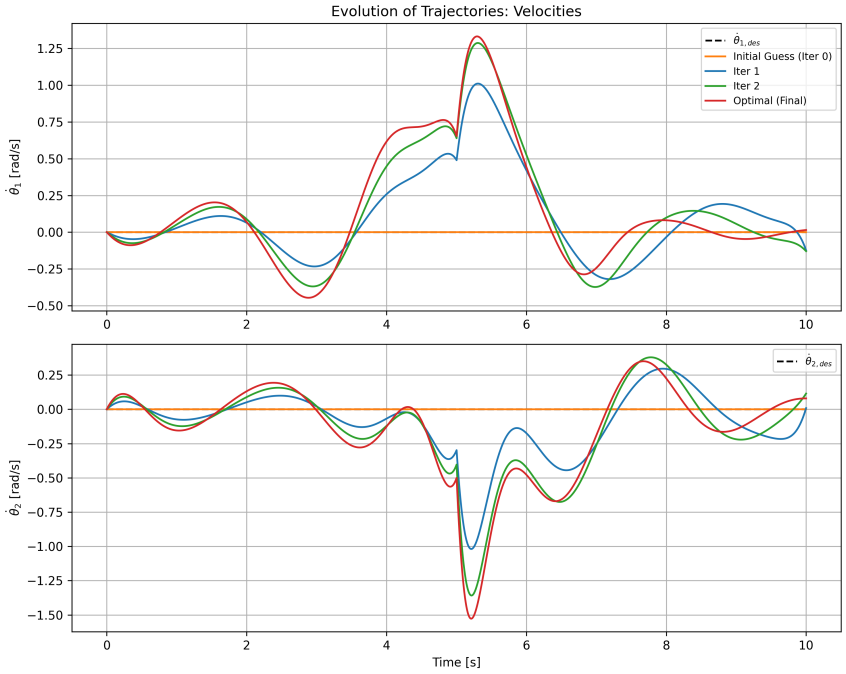

Figure 2.7: Evolution of Velocities over iterations.

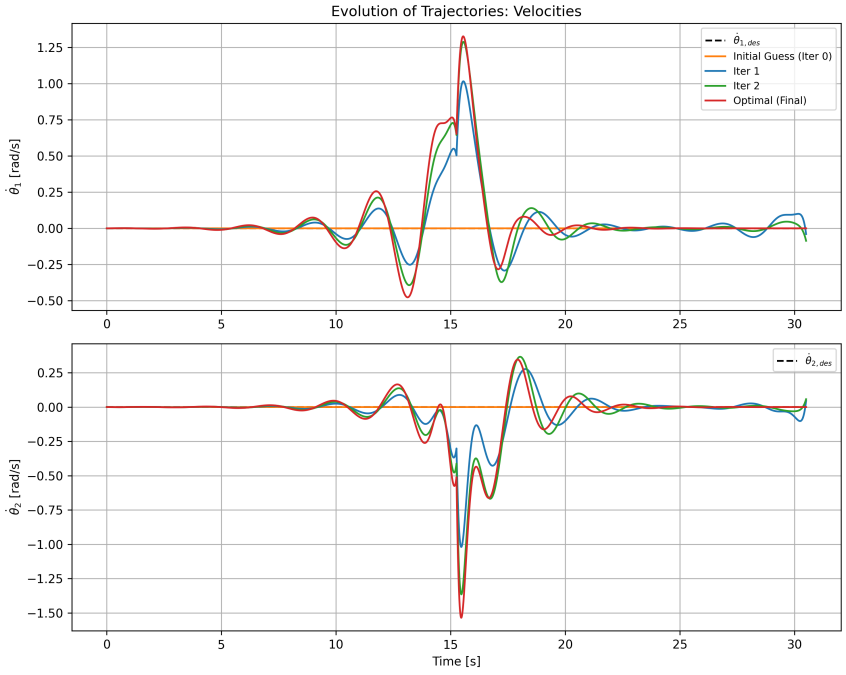

Figure 2.8: Evolution of Velocities over iterations.

2.4.3 Convergence Analysis

The numerical performance of the algorithm is evaluated through the cost function and the norm of Descent direction along iterations. Figure 2.9 shows the cost reduction in a semi-logarithmic scale. The high initial cost is drastically reduced within the first few iterations, demonstrating robust convergence.

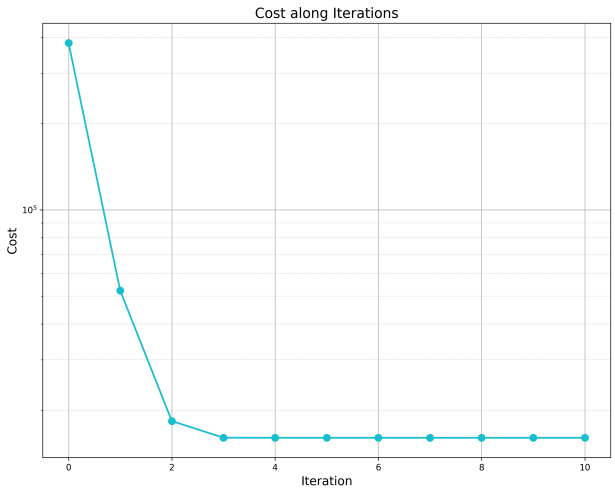

Figure 2.9: $T = 10s$ - Cost Function vs. Iterations (Log Scale).

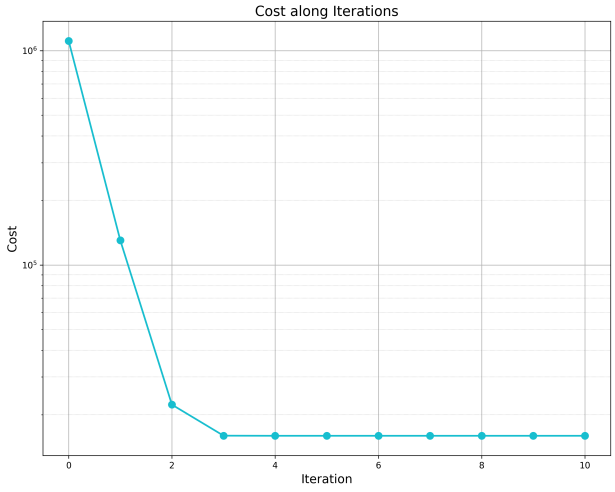

Figure 2.10: $T = 30.5s$ - Cost Function vs. Iterations (Log Scale).

From the plot in Figures 2.9 and 2.10, it is not possible to be fully confident that the optimal solution has been reached, since the stabilization of the cost function does not necessarily imply optimality. To further verify

convergence, the norm of the descent direction is also analyzed. As shown in Figure 2.11, this norm tends to zero (reaching numerical tolerance), which indicates that a local minimum has been found satisfying the necessary conditions for optimality.

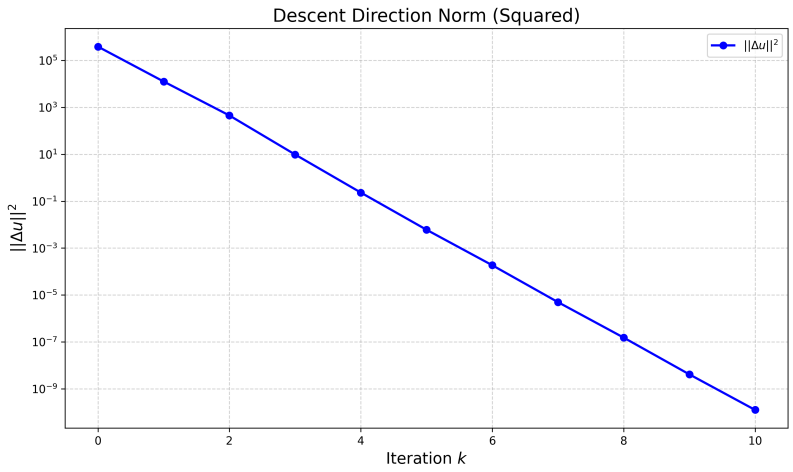

Figure 2.11: $T = 10s$ - Norm of the Descent Direction along iterations.

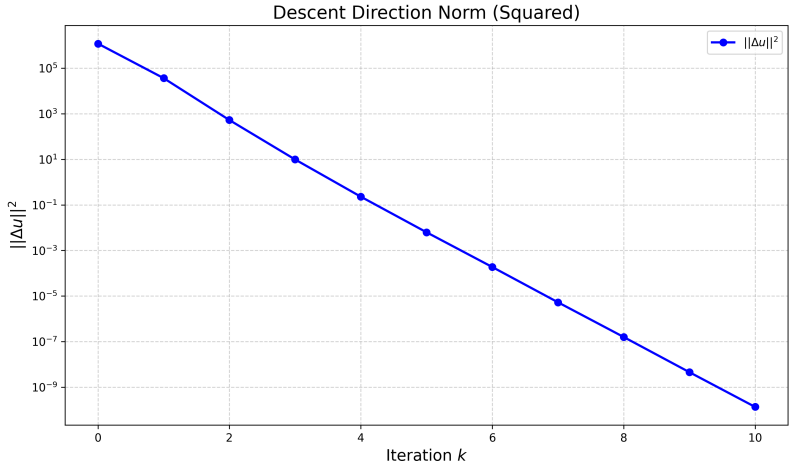

Figure 2.12: $T = 30.5s$ - Norm of the Descent Direction along iterations.

2.4.4 Armijo Line Search Analysis

A critical component of the Differential Dynamic Programming approach is the Line Search mechanism. Since we are optimizing a non-linear system, the full Newton step ($\gamma = 1$) might not always lead to a cost reduction or stability.

The following figures illustrate the line search process at selected iterations. The green curve represents the real cost $J(u+\gamma\Delta u)$, while the dashed line is the Armijo acceptance threshold. In the early iterations, the algorithm often explores smaller step sizes (orange dots) before accepting a value (red dot) that guarantees sufficient decrease. As the solution approaches the optimum, the quadratic approximation becomes accurate, and the full step $\gamma = 1$ is typically accepted. It can be observed that, by modifying the cost matrices, the cost function plotted during the Armijo line search exhibits a steeper slope.

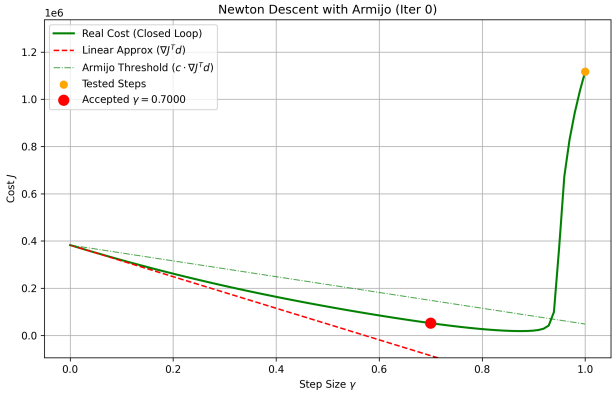

Figure 2.13: $T = 10s$ - Armijo Line Search at Iteration 0.

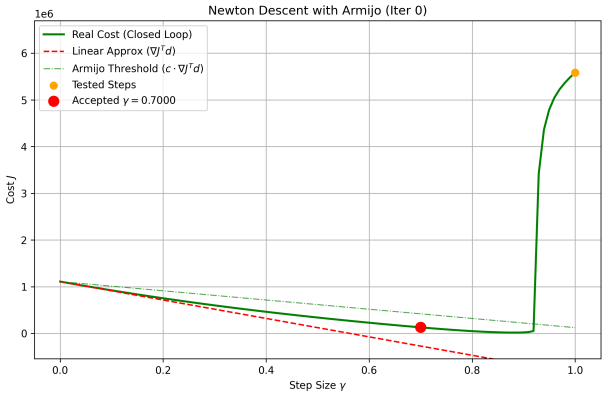

Figure 2.14: $T = 30.5s$ - Armijo Line Search at Iteration 0.

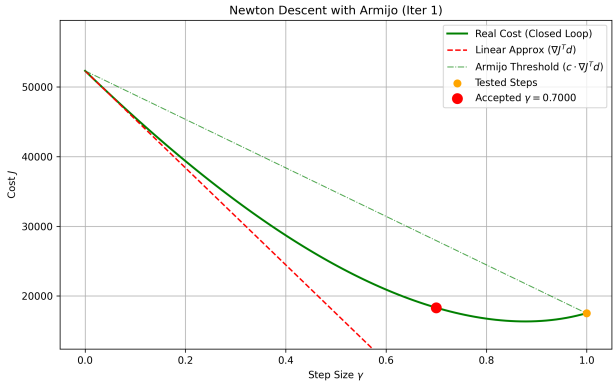

Figure 2.15: $T = 10s$ - Armijo Line Search at Iteration 1.

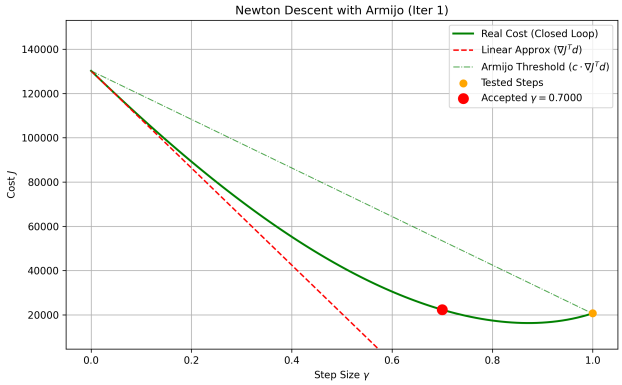

Figure 2.16: $T = 30.5s$ - Armijo Line Search at Iteration 1.

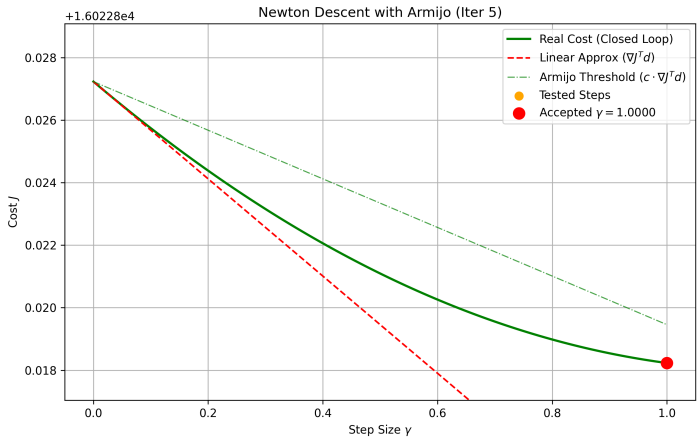

Figure 2.17: $T = 10s$ - Armijo Line Search at Iteration 5.

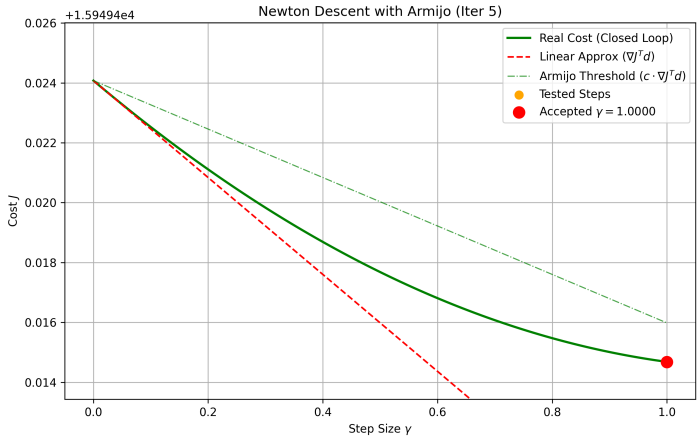

Figure 2.18: $T = 30.5s$ - Armijo Line Search at Iteration 5.

Chapter 3

Refined Trajectory Generation (Task 2)

The approach presented in Task 1 is based on a discontinuous step reference. From a practical control perspective, tracking a discontinuous signal would require infinite bandwidth and can negatively affect numerical stability and convergence properties of the optimization algorithm.

The goal of Task 2 is therefore to enhance the trajectory generation framework by adopting a **smooth reference trajectory**. In this formulation, the system is steered from the initial equilibrium configuration to the final one by means of a continuous transition.

3.1 Choice of Reference: Trapezoidal Velocity Profile

In optimal control applications, smooth reference trajectories are often constructed using sigmoid functions or fifth-order polynomials, such as minimum-jerk profiles. In contrast, this project adopts a **trapezoidal velocity profile**.

The motivation for this choice lies in its extensive adoption within **industrial robotics** because it is the standard approach for joint-space point-to-point motion planning in industrial manipulators.

The trapezoidal velocity profile consists in three parts: an initial constant acceleration phase, a constant velocity part and a final constant deceleration segment.

The objective is to assess whether such a kinematic strategy, traditionally associated with rigid robotic systems, can effectively support the optimal control of a flexible and under-actuated link.

Simulation results presented in Section 3.4 confirms the effectiveness of this approach, as it leads to a feasible optimal solution with favorable

convergence behavior. Therefore it's a robust alternative to higher-order polynomial reference trajectories.

3.2 Reference Trajectory Design

The smooth reference trajectory is generated through a dedicated function, denoted as `gen_smooth()`, implemented in `reference_curve.py`. The design idea is to construct a piecewise reference trajectory composed of equilibrium segments and a smooth transition phase.

Specifically, the reference trajectory is divided into three consecutive time intervals.

In the first interval $(0s - 13s)$, the reference state is kept equal to the first equilibrium configuration

$$\mathbf{x}_{eq,1} = [0,\ 0,\ 0,\ 0]^\top.$$

In the central interval $(13s - 17.5s)$, the reference evolves according to a trapezoidal motion profile defined in a dedicated function named `trapezoidal()`.

In the last interval $(17.5s - 30.5s)$, the reference is set equal to the second equilibrium configuration

$$\mathbf{x}_{eq,2} = [90^\circ,\ -90^\circ,\ 0,\ 0]^\top.$$

The resulting reference trajectory provides a smooth transition of the flexible robotic link from the initial equilibrium configuration to the final one, guaranteeing continuity of both position and velocity profiles.

Specifically, the trapezoidal position profile $x(t)$ is defined as:

$$x(t) = \begin{cases} x_{init} + \frac{1}{2}\ddot{x}(t)(t - t_{init}) & t_{init} \le t < t_{init} + t_{acc} \\ x_{init} + \ddot{x}(t)t_{acc}(t - t_{init} - t_{acc}) & t_{init} + t_{acc} < t \le t_{fin} - t_{acc} \\ x_{fin} - \ddot{x}_{max}(t)(t_{fin} - t) & t_{fin} - t_{acc} < t \le t_{fin} \end{cases} \tag{3.1}$$

3.3 Optimization with Smooth Reference

The Newton-based optimal control framework adopted for Task 2, implemented in `main_task2.py`, preserves the same structure as the algorithm used for Task 1 in `main_task1.py`. The only modification concerns the reference trajectory generation, where the function `gen_smooth()` replaces the step-reference function `gen()` previously employed.

3.4 Simulation Results

The following figures summarize the main results obtained from the optimal trajectory generation using a smooth trapezoidal reference. The plots provide a comprehensive view of the evolution of the state and input trajectories throughout the Newton iterations, as well as a direct comparison between the final optimal solution and the desired reference curves.

3.4.1 Optimal Trajectories

Figure 3.1 shows that, by adopting a trapezoidal reference trajectory, a significantly smoother overall behavior is obtained. In particular, oscillations before and after the transition phase are substantially reduced. Moreover, the control input no longer exhibits sharp peaks and residual oscillations are almost completely eliminated.

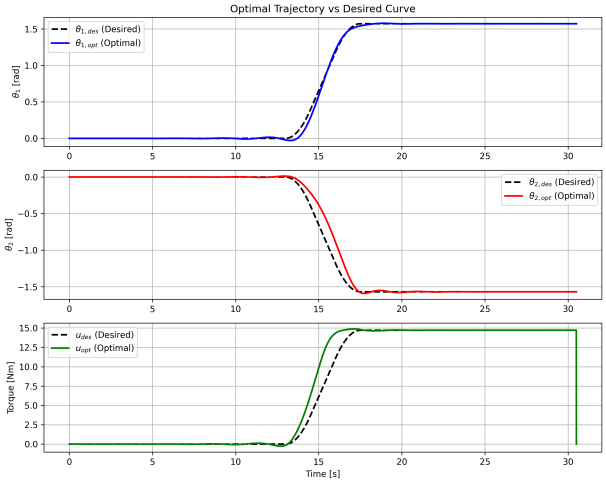

Figure 3.1: Optimal State Trajectories vs Reference Trajectories with Trapezoidal Transition.

The benefits of the smooth reference are evident in the velocity profiles shown in Figure 3.2. Similarly to the angular positions and the control input, the velocity profiles also exhibit a significantly smoother behavior. The optimal velocities follow a clear trapezoidal shape (acceleration, constant speed, deceleration) without the impulsive spikes observed in the step response. In particular, the peak value in $\dot{\theta}_1$ is noticeably reduced (for instance, from approximately $1.25 rad/s$ to $0.7 rad/s$), and oscillations are almost completely suppressed.

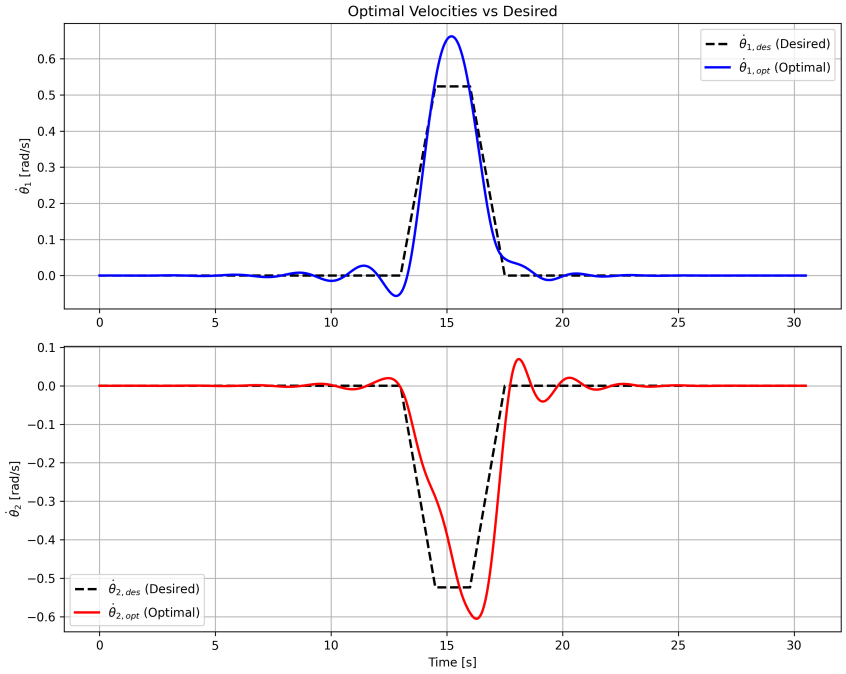

Figure 3.2: Optimal Velocity Profiles vs Velocity profiles with Trapezoidal Transition.

3.4.2 Algorithm Evolution (Warm Start)

In Figure 3.3 it's possible to observe the evolutions of the state and the input trajectories along the iterations. We can observe a significantly improvement in the first iterations as for the step function case, proving how Newton's algorithm reaches the optimal tracking trajectory.

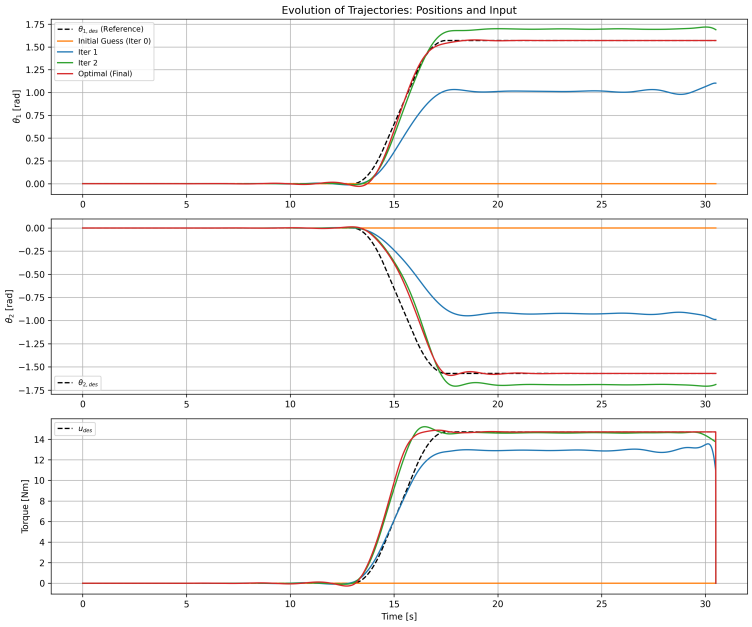

Figure 3.3: Evolution of State and Input Trajectories.

Figure 3.4 confirms this behavior for velocities as well. The solver, along iterations, simply refines the initial velocity guessed profile to satisfy the equations of motion. In this image is worth noticing that the velocity profile is better tracked on $\dot{\theta}_1$ due to the direct action of the input torque.

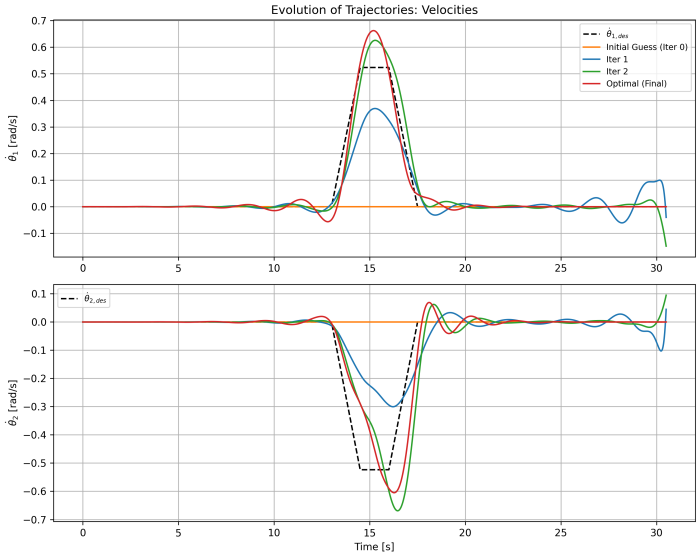

Figure 3.4: Evolution of Velocities over iterations.

3.4.3 Convergence Analysis

The same reasoning of Task 1 can be done regarding the convergence of the algorithm solution. Figure 3.5 shows the cost function evolution.

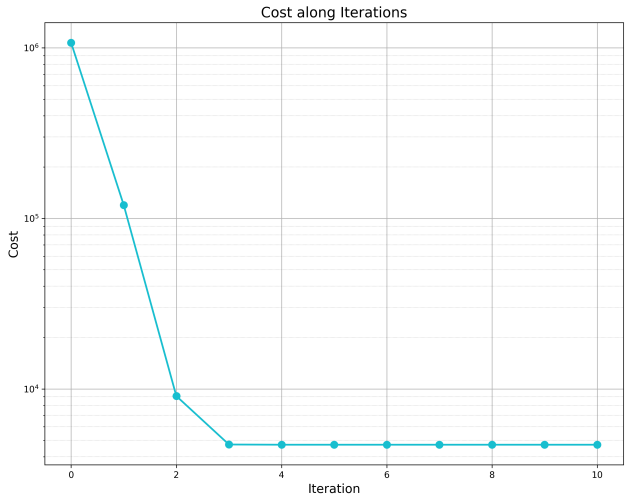

Figure 3.5: Cost Function vs. Iterations (Log Scale).

The decay of the descent direction norm to numerical tolerance levels in Figure 3.6 corroborates the findings of Figure 3.5, indicating that a valid local minimum has been attained.

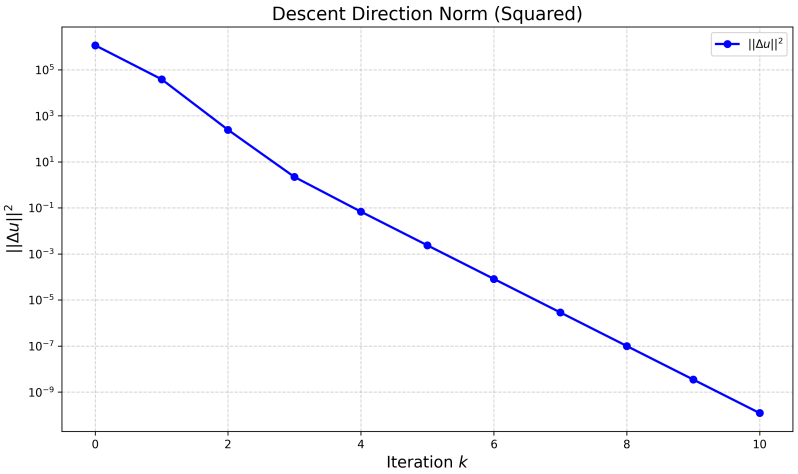

Figure 3.6: Norm of the Descent Direction along iterations.

3.4.4 Armijo Line Search Analysis

The analysis of the Line Search process provides further insight into the quality of the proposed initialization strategy.

Figures 3.7 through **??** display the cost reduction as a function of the step size γ. From Figure 3.8 it's notable that the algorithm does not need to heavily reduce the step size to ensure stability. As seen in Figure 3.7, even at Iteration 0, the Newton step is accepted with a large γ (close to or equal to 1), requiring minimal backtracking. The full step $\gamma = 1$ is already accepted from Iteration 1, indicating quadratic convergence.

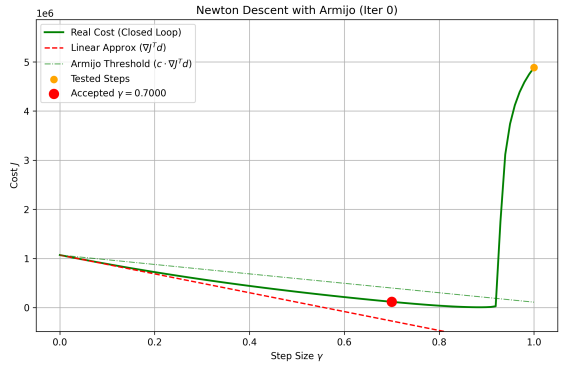

Figure 3.7: Armijo Line Search at Iteration 0 (Task 2).

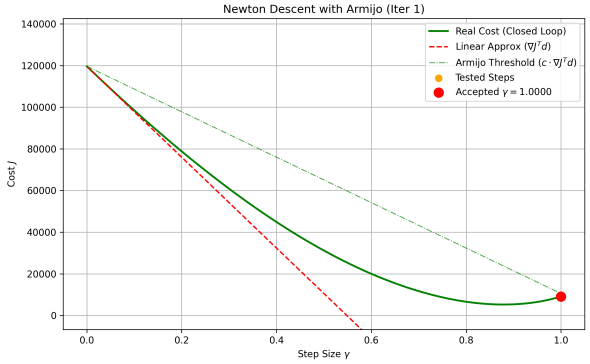

Figure 3.8: Armijo Line Search at Iteration 1.

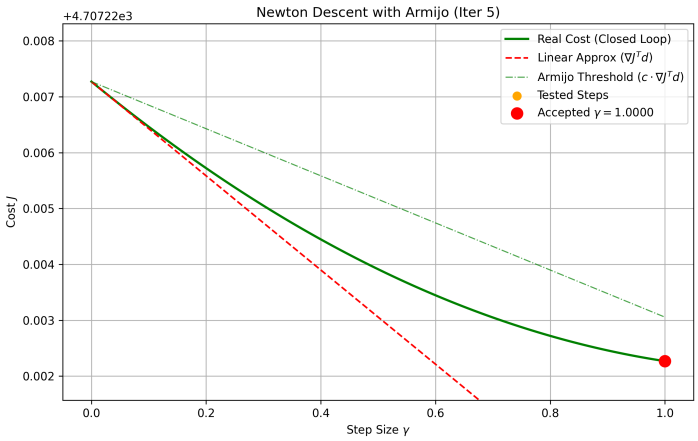

Figure 3.9: Armijo Line Search at Iteration 2.

Chapter 4

Trajectory Tracking via TV-LQR (Task 3)

The goal of Task 3, written in `main_task3.py`, is to design a feedback control law that ensures stabilization of the system around the optimal trajectory computed in the previous task.

This is achieved by adopting a Time-Varying Linear Quadratic Regulator (TV-LQR) which minimizes trajectory tracking errors based on a linearized representation of the system dynamics along the optimal solution.

This problem is solved in three steps:

1. Linearization of the system around the optimal trajectory of Task 2.
2. Calculation of the linear quadratic optimal controller on the finite horizon.
3. Tracking of the optimal trajectory.

4.1 Linearization of the system

The first step involves linearizing the system dynamics around the feasible trajectory $(x^{\text{traj}}, u^{\text{traj}})$ in order to obtain a linear time-varying (LTV) representation.

$$\Delta \mathbf{x}_{t+1} = \mathbf{A}_t^{\text{traj}} \, \Delta \mathbf{x}_t + \mathbf{B}_t^{\text{traj}} \, \Delta \mathbf{u}_t$$

with

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

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

The error variables are defined as the deviations of the system state and control input from the corresponding optimal trajectories:

$$\Delta x_t = x_t - x_t^{opt}, \quad \Delta u_t = u_t - u_t^{opt} \tag{4.3}$$

The goal is to drive these deviations to zero. Since the system operates in a neighborhood of the optimal trajectory, we can approximate the nonlinear dynamics $x_{t+1} = f(x_t, u_t)$ using a first-order Taylor expansion around (x_t^{opt}, u_t^{opt}).

4.2 Calculation of the linear quadratic optimal controller on the finite horizon

The trajectory tracking problem is posed as an infinite-horizon LQR problem, approximated over a finite time horizon T, for the corresponding error system. The goal is to compute the optimal control correction $\delta\mathbf{u}_t$ that minimizes a quadratic cost functional:

$$\sum_{t=0}^{T-1} \left(\Delta x_t^T Q_t^{reg} \Delta x_t + \Delta u_t^T R_t^{reg} \Delta u_t\right) + \Delta x_T^T Q_T^{reg} \Delta x_T \tag{4.4}$$

where $Q_t^{reg} \geq 0$ and $R_t^{reg} > 0$ and $Q_T^{reg} \geq 0$ are the weighting matrices for the regulation task. In particular, a trial-and-error tuning approach is employed and the same cost matrices used in the previous tasks are retained.

Furthermore, in this case, the LQR problem does not include affine terms, therefore, the linear components $\mathbf{q}_t$ and $\mathbf{r}_t$ are identically zero. Consequently, the cost function is formulated around the zero-error equilibrium.

To evaluate the system's tracking performance, a perturbed initial condition was introduced. The perturbation was generated using the `np.random.uniform()` function, applying a distinct value to each state component within the interval $[0, 0.5)$. The results presented in this chapter are based on the specific perturbation vector. The approximate values are reported below:

$$\delta_x = [0.441, 0.055, 0.413, 0.123]^T \tag{4.5}$$

4.2.1 Feedback Controller Design

The optimal feedback law is computed in `ltv_solver_LQR.py` by solving the Difference Riccati Equation backward in time ($t = T-1, ..., 0$):

$$P_t = Q_t^{reg} + A_t^{traj,T} P_{t+1} A_t^{traj} - (A_t^{traj,T} P_{t+1} B_t^{traj})(R_t^{reg} + B_t^{traj,T} P_{t+1} B_t^{traj})^{-1}(B_t^{traj,T} P_{t+1} A_t^{traj}) \tag{4.6}$$

with the terminal condition $P_T = Q_T^{reg}$.

The optimal gain is then:

$$K_t^{reg} = -(R_t^{reg} + B_t^{traj,T} P_{t+1} B_t^{traj})^{-1}(B_t^{traj,T} P_{t+1} A_t^{traj}) \tag{4.7}$$

4.2.2 Tracking of the Optimal Trajectory

In the final stage, the feedback control law derived from the linearized dynamics is applied to enforce tracking of the optimal trajectory.

The control input applied to the nonlinear system is expressed as

$$\mathbf{u}_t = \mathbf{u}_t^{\text{traj}} + \mathbf{K}_t^{\text{reg}} \left(\mathbf{x}_t - \mathbf{x}_t^{\text{traj}} \right). \tag{4.8}$$

Once the input is computed, we use the `dynamics_euler()` function to obtain the state at the next iteration:

$$x_{t+1} = f_t(x_t, u_t) \tag{4.9}$$

4.3 Simulation Results

The performance of the TV-LQR controller was validated by simulating the closed-loop non-linear system starting from the perturbed condition.

4.3.1 Tracking Performance

Figure 4.1 compares the optimal reference trajectory (black dashed) with the actual closed-loop trajectory (solid colored). Despite the significant initial error both on θ_1 and θ_2 due to the random initial perturbation, the controller rapidly drives the system back to the optimal path within the first $10s$. Even the input profile shows the initial corrective torque spike which smoothly converges to the desired torque u^{opt} as the error vanishes.

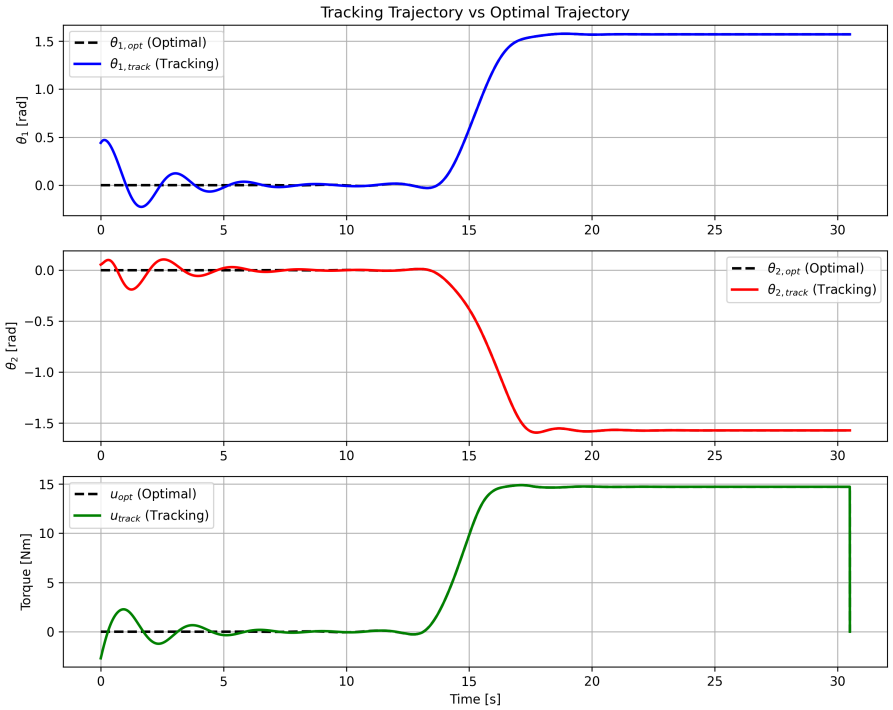

Figure 4.1: Tracking vs Optimal Trajectory: angular position and torque.

Figure 4.2 confirms that the velocities are also stabilized effectively. The transients are smooth and the system avoids dangerous oscillations during the recovery phase, proving the robustness of the linearized feedback gains K_t.

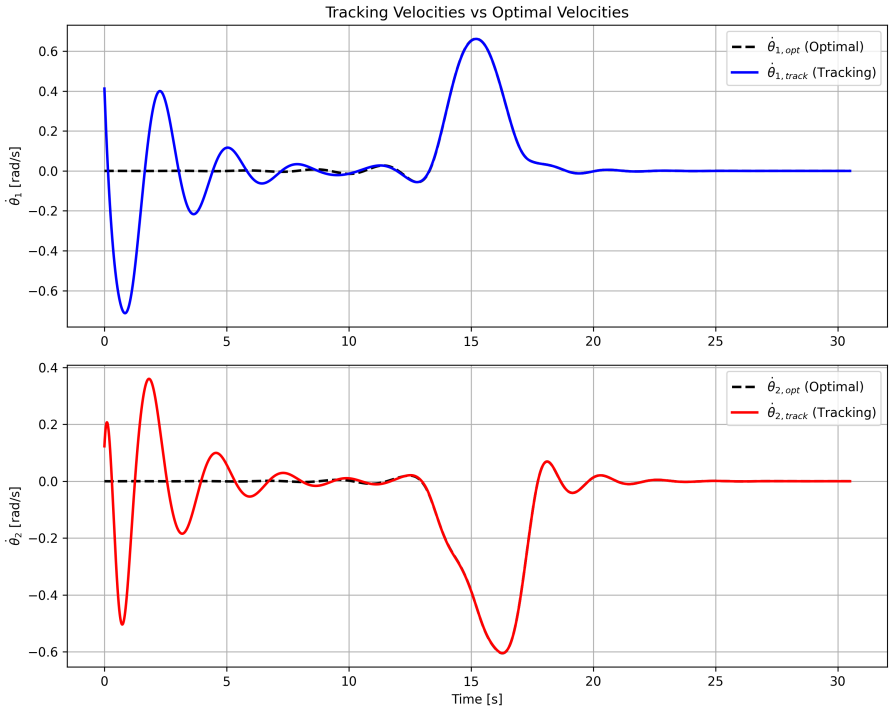

Figure 4.2: Tracking vs Optimal Velocities: angular velocity profiles of the two joints.

4.3.2 Error Analysis

To quantify the tracking accuracy, the time evolution of the tracking error $|\Delta x_t|$ is analyzed. Figure 4.3 displays the position errors. The error on θ_1 starts from the initial perturbation and decays exponentially. Interestingly, θ_2 also shows a transient error: although the second link was not perturbed initially, the dynamic coupling with the first link induces a deviation which is promptly rejected by the controller.

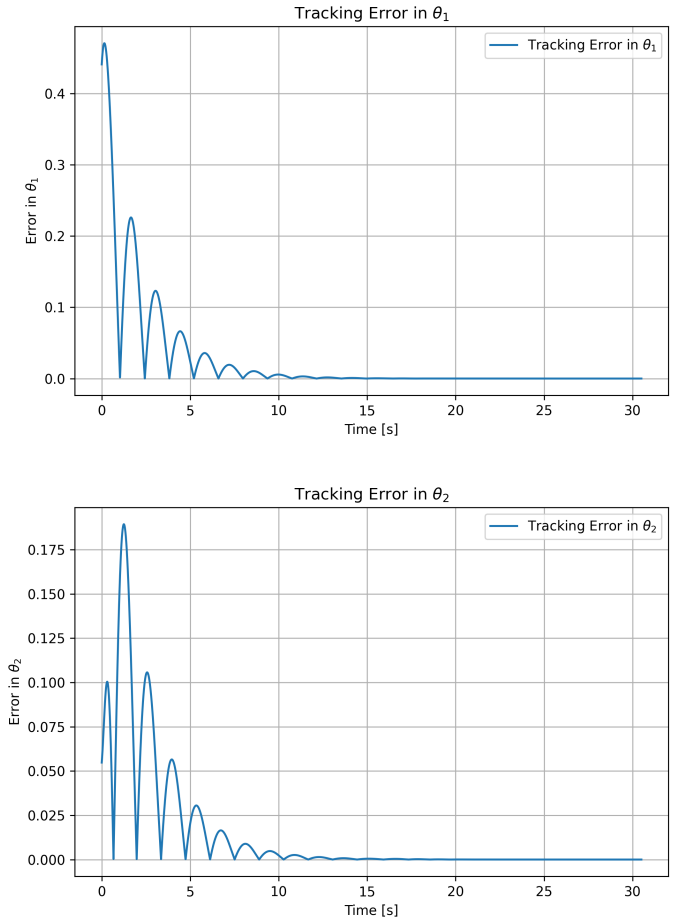

Figure 4.3: Position Tracking Errors.

Similarly, Figure 4.4 shows the velocity tracking errors. The convergence to zero confirms the asymptotic stability of the closed-loop time-varying system.

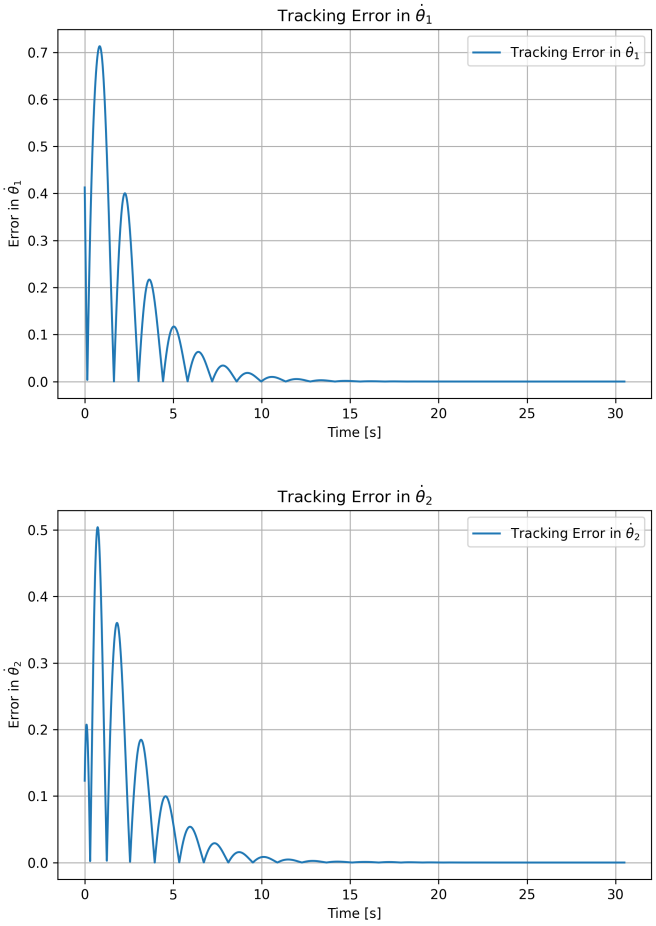

Figure 4.4: Velocity Tracking Errors.

Finally, Figure 4.5 shows the input tracking error $\Delta u_t = u_t - u_t^{opt}$. This represents the additional corrective torque required by the feedback controller to compensate for the perturbation. As expected, it goes to zero as the system realigns with the optimal trajectory.

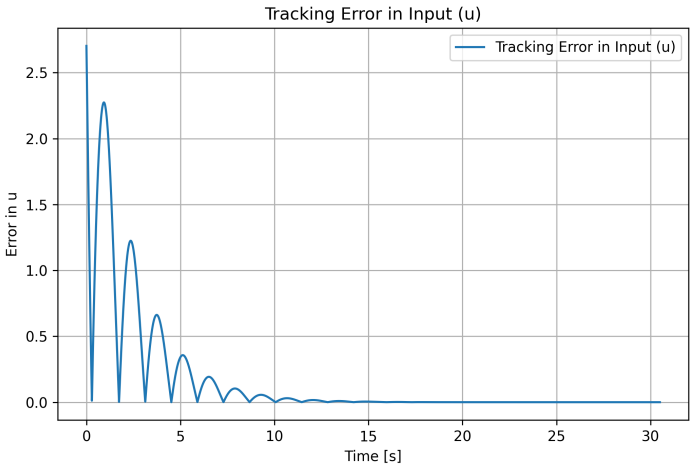

Figure 4.5: Input Tracking Error.

Chapter 5

Trajectory Tracking via MPC (Task 4)

While the Time-Varying LQR designed in the previous chapter provides an effective feedback law for stabilization, it has a fundamental limitation: it cannot explicitly handle physical constraints. In real-world robotic applications, actuators have saturation limits (torque bounds), and state variables may have safety ranges.

The goal of Task 4 is to upgrade the tracking strategy by implementing a **Linear Model Predictive Control (MPC)** scheme. Unlike LQR, which computes a static gain matrix offline, MPC solves a constrained optimization problem online at each time step, allowing it to predict future system behavior and respect actuation limits.

5.1 Problem Formulation

The control objective remains the tracking of the optimal trajectory (x^{opt}, u^{opt}) generated in Task 2. The problem is formulated as a finite-horizon optimization over the *error dynamics*.

Let N be the **prediction horizon**. At each sampling instant t, we measure the current state x_t and compute the current tracking error $\Delta x_t = x_t - x_t^{opt}$. Then we solve for the optimal sequence of control corrections $\Delta U = \{\Delta u_0, \Delta u_1, \ldots, \Delta u_{N-1}\}$ that minimizes the following cost function:

$$\sum_{\tau=t}^{t+N-1} \left(\Delta x_\tau^T Q_{mpc} \Delta x_\tau + \Delta u_\tau^T R_{mpc} \Delta u_\tau\right) + \Delta x_N^T Q_{mpc,N} \Delta x_N \tag{5.1}$$

subject to:

$$\Delta x_{\tau+1} = A_\tau \Delta x_\tau + B_\tau \Delta u_\tau, \quad \tau = t, \ldots, t+N-1 \tag{5.2}$$

$$u_{min} \leq u_\tau^{opt} + \Delta u_\tau \leq u_{max} \tag{5.3}$$

Consistent with Task 3, a perturbed initial condition is employed. The results presented here are based on the following perturbation vector (values are approximated):

$$\delta_x = [0.286, 0.452, 0.402, 0.247]^T \tag{5.4}$$

5.1.1 Linear Time-Varying Prediction Model

A key feature of this implementation is that the internal model used by the MPC is **Time-Varying**. Since the robot follows a complex trajectory, linearizing around a single equilibrium point is insufficient. At every time step t, the solver (`solver_MPC.py`) utilizes a specific "slice" of the Jacobian matrices A and B, which were pre-computed along the reference trajectory:

$$A_\tau = \nabla_x f(x,u)\big|_\tau, \quad B_\tau = \nabla_u f(x,u)\big|_\tau \tag{5.5}$$

This ensures that the prediction model evolves consistently with the reference motion.

5.1.2 Constraints Handling

The most critical addition in this task is equation (5.3). The solver ensures that the *total applied torque* (Feed-forward + Feedback correction) never exceeds the physical limits of the motor. Based on the optimization results from Task 2, we selected the following saturation limits to realistic values, ensuring the controller has enough authority to stabilize but is properly bounded:

$$u_{max} = 15\,\text{Nm}, \quad u_{min} = -1.5\,\text{Nm} \tag{5.6}$$

5.2 Implementation Details

The MPC solver is implemented in Python using the **CasADi** framework, which allows for efficient numerical optimization of the resulting Quadratic Programming (QP) problem. The implementation of the MPC problem is instead done in `main_task4.py`, where the following definitions are available:

- **Prediction Horizon:** We selected a window size of $N = 25$ steps (0.125 s). This horizon is sufficiently long to capture the short-term dynamics of the flexible link without excessive computational cost.
- **Weighting Matrices:** To achieve high-performance tracking, we increased the penalty on position errors compared to the LQR task:

$$Q_{mpc} = \text{diag}(200, 200, 1, 1), \quad R_{mpc} = 0.1 \tag{5.7}$$

- **Receding Horizon Loop:** At each simulation step t, the optimization problem is solved, and only the first element of the optimal sequence, Δu_0^*, is applied to the system ($u_t = u_t^{opt} + \Delta u_0^*$). The process is then repeated for $t+1$.

5.3 Simulation Results

The MPC controller was tested under the assumption of perturbed initial conditions as Task 3. The simulation proves the capability of the controller to reject disturbances while respecting the constraints at the same time.

5.3.1 Tracking Performance

Figure 5.1 shows the state trajectories. The controller successfully recovers from the initial perturbation and converges to the optimal path (black dashed line). The transient is fast and smooth, thanks to the predictive nature of the algorithm.

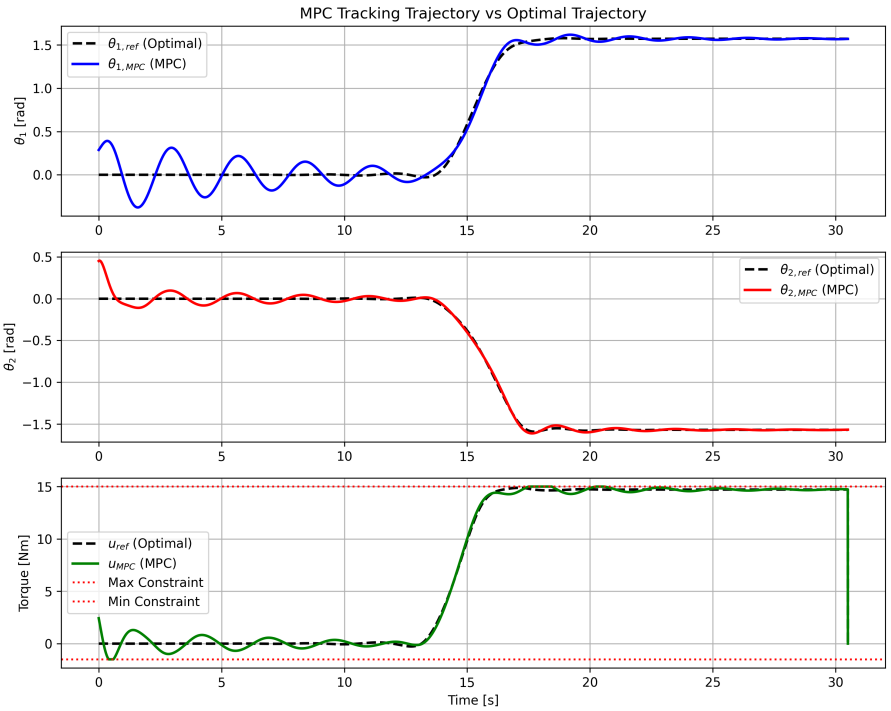

Figure 5.1: MPC Tracking vs Optimal Reference.

Crucially, the bottom subplot of Figure 5.1 confirms that the constraints are respected. Even if the initial error would require a large corrective

torque, the MPC saturates the input at the specified bounds (15 or -1.5 Nm) in an optimal manner, avoiding possible input peaks.

Figure 5.2 illustrates the velocity tracking. The velocities converge to the reference profile without inducing high-frequency chatter, indicating a stable closed-loop behavior.

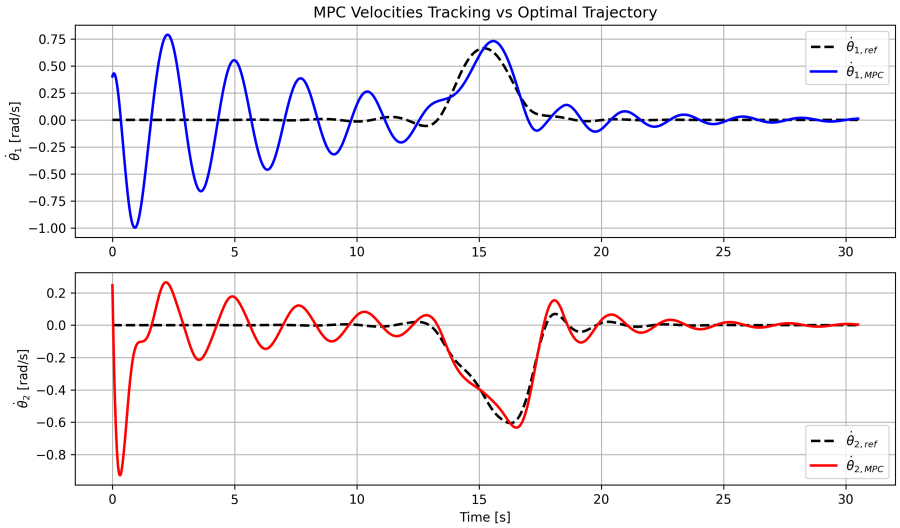

Figure 5.2: MPC Velocities Tracking vs Optimal Velocity Profiles.

5.3.2 Error Analysis

The tracking error decay is analyzed in Figure 5.3. The position errors θ_1 and θ_2 are driven closely to zero. It is observed that the error convergence might be slightly slower than the unconstrained LQR in the very first instants if the actuator hits the saturation limit, but this ensures the physical safety of the system.

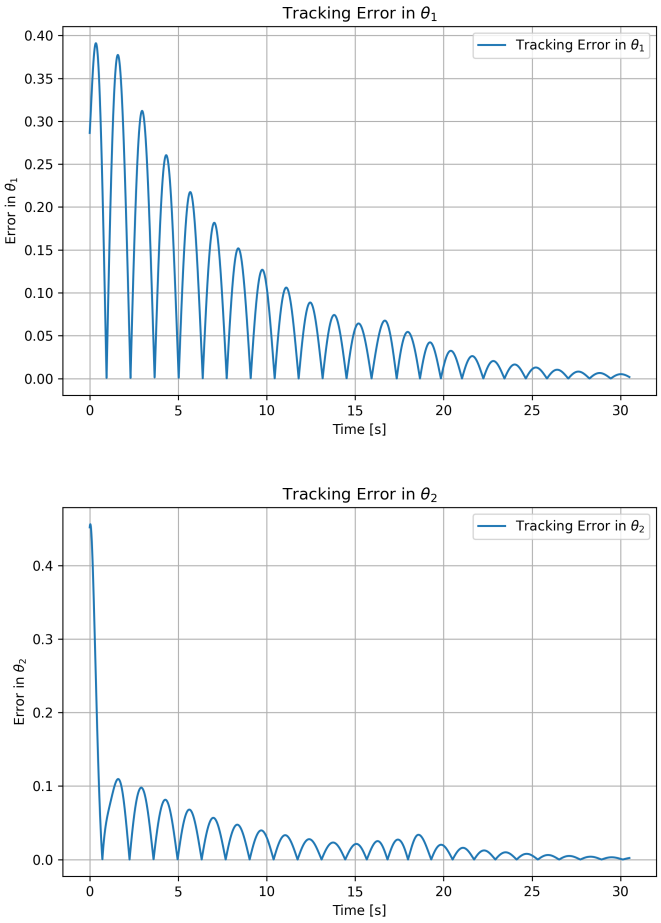

Figure 5.3: MPC Position Tracking Errors for θ_1 and θ_2.

Figure 5.4 shows the input correction δu. This value represents the effort the MPC adds on top of the feed-forward torque. As expected, it converges to zero as the system aligns with the planned trajectory.

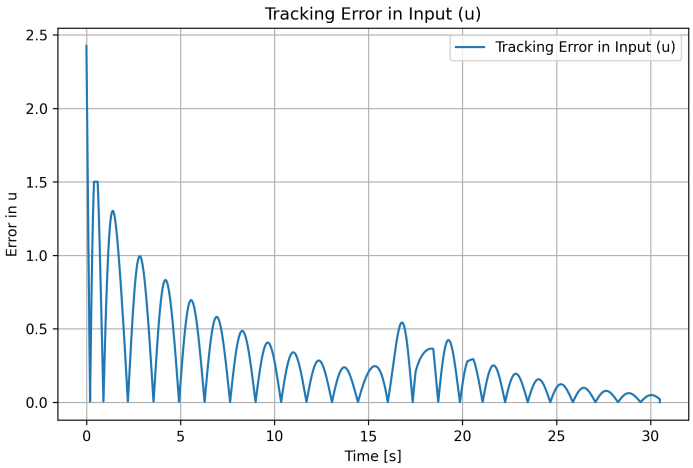

Figure 5.4: MPC Input Correction Error (δu).

Chapter 6

Animation (Task 5)

The objective of Task 5 is to provide a graphical validation of the closed-loop behavior obtained in Task 3 by developing a realistic animation of the flexible link manipulator. The animation is generated in `main_task5.py` by replaying a closed-loop simulation and rendering the robot motion through a CAD-like visualization based on `matplotlib` patches.

6.1 Implementation

The manipulator is rendered as a two-link planar mechanism with link lengths l_1 and l_2 taken from `dynamics.py`.

Given the state $\mathbf{x} = [\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]^\top$, the forward_kinematics function is used to compute the Cartesian coordinates of the joints:

$$p_0 = (0, 0), \qquad p_1 = \begin{bmatrix} l_1 \sin\theta_1 \\ -l_1 \cos\theta_1 \end{bmatrix}, \qquad p_2 = \begin{bmatrix} l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2) \\ -l_1 \cos\theta_1 - l_2 \cos(\theta_1 + \theta_2) \end{bmatrix}.$$

6.2 Visualization

The visualization is designed to resemble an industrial CAD rendering. In particular:

- each link is drawn as a thick rectangular bar rather than a thin line.
- joints are represented as circles.
- a trace of the end-effector is displayed to emphasize the executed path.
- a time label is shown during playback to indicate the current simulation time.

6.3 Details and animation Export

The animation is created using `FuncAnimation` from `matplotlib.animation` library. To speed up the rendering, frames are subsampled with a fixed skip factor (e.g., one frame every 5 simulation steps), while the frames-per-second is selected to preserve the real-time speed of the simulated motion.

Finally, the animation is exported as a `.gif` file using `PillowWriter`. The output directory `images/Task5` is created automatically if it does not exist, and the generated file is saved as:

`images/Task5/task5_animation.gif`.

This animation provides an immediate qualitative verification of the closed-loop tracking performance and of the physical plausibility of the resulting flexible-link motion.

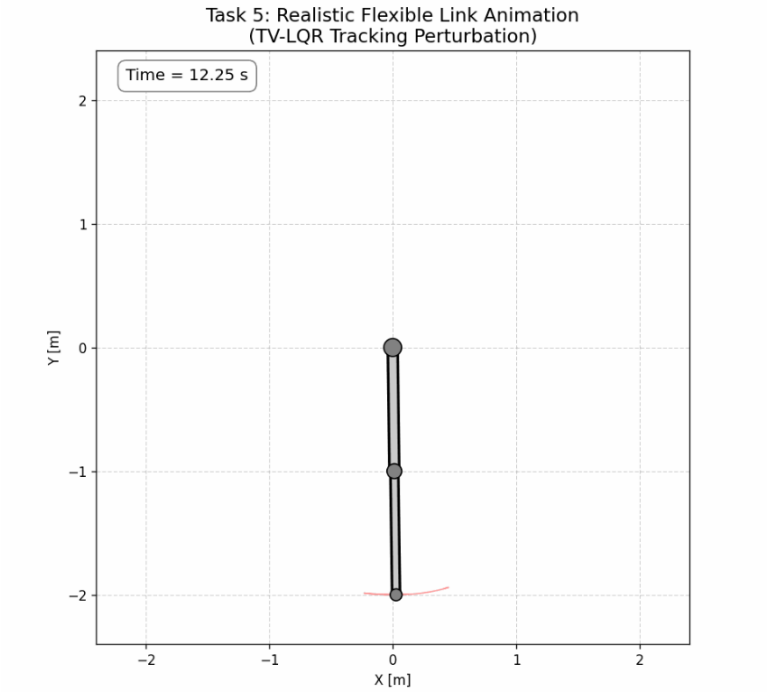

Figure 6.1: Image animation 1

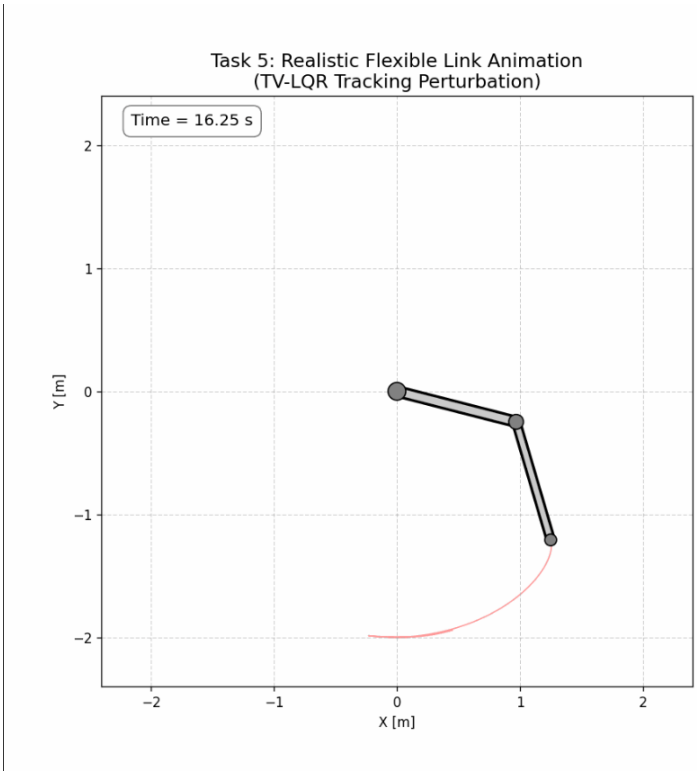

Figure 6.2: Image animation 2

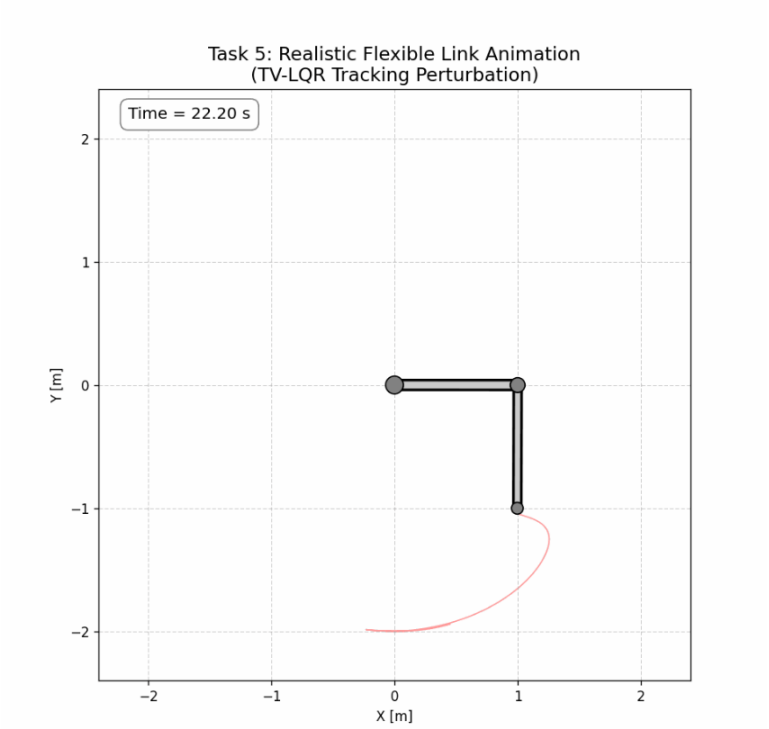

Figure 6.3: Image animation 3

Conclusions

In this project, the problem of optimal trajectory generation and tracking for a flexible two-link robotic manipulator was investigated. Starting from a nonlinear dynamic model, the system dynamics were discretized using the Forward Euler method and embedded within a Newton-based optimal control framework to compute dynamically feasible trajectories connecting two equilibrium configurations.

A first solution was obtained by tracking a discontinuous step reference; however, this approach highlighted limitations in terms of physical plausibility and numerical convergence. To overcome these issues, a refined trajectory generation strategy based on a trapezoidal velocity profile was introduced. This smooth reference ensured a physically feasible transition for the optimization algorithm, significantly enhancing convergence properties. Furthermore, it yielded smoother state and input trajectories, characterized by attenuated oscillations and reduced peak values.

Once the optimal feed-forward trajectory was computed, feedback control strategies were designed to ensure robust tracking. By linearizing the system dynamics along the optimal trajectory, a Time-Varying Linear Quadratic Regulator (TV-LQR) was synthesized to stabilize the system in the presence of initial state perturbations. Simulation results demonstrated convergence of the tracking errors to zero and confirmed the local validity of the linearized feedback law for the nonlinear system.

Finally, a Model Predictive Control (MPC) scheme was implemented to explicitly handle input constraints. The MPC controller successfully enforced actuator limits while maintaining accurate trajectory tracking, even under significant disturbances. Overall, the results validate the effectiveness of the proposed control pipeline, combining optimal trajectory generation, smooth reference design, feedback control and demonstrate its robustness and applicability to flexible and under-actuated robotic systems.

Bibliography

[1] G. Notarstefano. *Optimal Control and Reinforcement Learning - Course Lecture Notes*. University of Bologna, 2025. Lecture Slides.

[2] *Course Project: Optimal Control of a Flexible Robotic Link*. University of Bologna, 2025. Assignment Specifications.

[3] G. Notarstefano. *Lecture Slides Set 1: Nonlinear Optimization*. Optimal Control and Reinforcement Learning, University of Bologna, 2025.

[4] G. Notarstefano. *Lecture Slides Set 4: Linear Quadratic (LQ) Optimal Control*. Optimal Control and Reinforcement Learning, University of Bologna, 2025.

[5] G. Notarstefano. *Lecture Slides Set 8: Second-order and Closed-loop Methods for Optimal Control*. Optimal Control and Reinforcement Learning, University of Bologna, 2025.

[6] G. Notarstefano. *Lecture Slides Set 10: Optimal Control based trajectory generation and tracking*. Optimal Control and Reinforcement Learning, University of Bologna, 2025.

[7] G. Notarstefano. *Lecture Slides Set 11: Model Predictive Control*. Optimal Control and Reinforcement Learning, University of Bologna, 2025.