UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Optimal Control

Optimal Control of an Actuated Flexible Surface

Professor: **Giuseppe Notarstefano**

Students:
Daniyar Zhakyp
Michele Culin
Giulio Rosadi

Academic year 2024/2025

Abstract

This report describes the development of a control system for an under-actuated flexible surface. Two different approaches are tested: Linear Quadratic Regulator (LQR) and Model Predictive Control (MPC) algorithms. The objective is to generate a desired optimal trajectory and follow it properly using both methods in Python environment. In the project, the desired curve's transitions between two system equilibria are connected by a 5^{th}-order polynomial and optimized via Newton's method. The obtained optimal trajectory has been successfully tracked by both LQR and MPC methods, achieving close to zero tracking error within small time for perturbed initial conditions. The animation in Python has clearly demonstrated the smooth transitions of the flexible surface between its equilibrium points.

Contents

Introduction

This project focuses on the optimal control of an underactuated flexible surface system, where only specific points can be directly manipulated through actuators. The system considered consists of a section of a flexible surface with four distinct points, characterized by their vertical displacements, each of which has a mechanical coupling with the others.

In this project, we begin by developing a discrete-time state-space model of the system dynamics, which accounts for the vertical forces and stiffness interactions between surface points (assuming that only vertical forces are generated). Then, two different control approaches are applied: an optimal feedback controller that uses a Linear Quadratic Regulator (LQR) algorithm to track a reference trajectory, and a Model Predictive Control (MPC) to track the same trajectory.

This report is structured as follows:

- **Chapter 1 - Problem setup**
  In this chapter, the problem statement is presented in detail. The primary step of the project is also explained, which is the discretization of the system dynamics.

- **Chapter 2 - Trajectory generation (I)**
  This chapter describes the generation of a first simple trajectory between exactly two equilibrium points. Next, it sheds light on the closed-loop version of Newton's method, which is used for optimal trajectory generation based on the reference signal. Lastly, the chapter shares the results of optimal tracking by providing the plots of optimal and reference trajectories side by side.

- **Chapter 3 - Trajectory generation (II)**
  This chapter describes the generation of a second, more complex reference trajectory that has transitions between equilibrium points. The trajectory generation described in Chapter 1 is also applied here to generate the optimal desired curve for LQR and MPC tracking.

- **Chapter 4 - Trajectory tracking via LQR**
  In this chapter, the control problem is approached through a linear-

quadratic regulator. This optimal feedback controller is applied to track the desired trajectory generated in Task 2.

- **Chapter 5 - Trajectory tracking via MPC**
  In this chapter, the control problem is solved with the MPC approach by solving a constrained optimal control problem at every timestep, while considering only a short horizon.

- **Chapter 6 - Animation**
  This chapter presents the results of the animation simulation for the optimal tracking of the system trajectories.

- **Conclusions**
  This chapter provides an analysis of the results obtained, along with a presentation of any issues related to the solution found.

Contributions

The project was developed in collaboration with each other at every stage of the project. The problem has been approached collectively by each member of the team to define the necessary steps for a successful completion of the code and the report. After elaboration on what should be done in each task of the problem assignment, each student took the responsibility for writing specific parts of the code, which were discussed and edited collectively in a shared coding environment. In this way, each team member had the chance to contribute to all the tasks in both the conceptual and technical aspects. It is worth noticing that most of the time, the tasks could be developed in parallel: in this way, we were able to make progress simultaneously on different parts of the project.

Chapter 1

Problem setup

1.1 Model of the system

The problem consists of controlling a flexible surface that can be modeled in a way shown in Figure 1.1. Four different points characterize the surface, with every point being equally spaced and having the distance between direct neighbors as d. It is assumed that only p_2 and p_4 are actuated. There are no on-plain forces, and the stiffness interaction between the points of the surface produces only forces acting in a vertical direction.

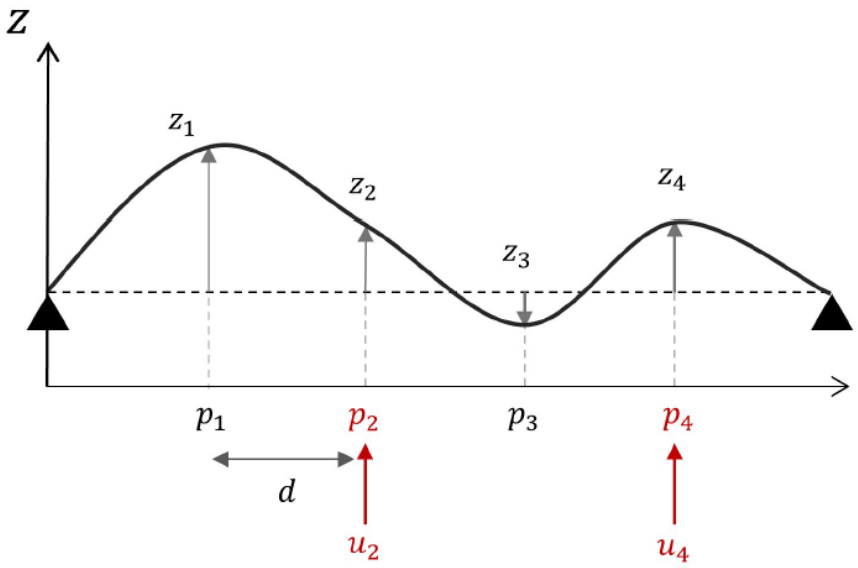

Figure 1.1: Model of a section of a flexible surface

Dynamics of the system are described by the following equation:

$$\ddot{z}_i = \frac{1}{m_i}\left[F_i - \alpha \sum_{j \in A} L_{ij} \frac{z_i - z_j}{L_{ij}^2 - (z_i - z_j)^2} - c\dot{z}_i\right], \tag{1.1}$$

where each z_{i} represents the vertical displacement of one of the points of the surface. Given the under-actuated nature of the system, $\mathrm{F_i} = \mathrm{u_i}$ only for the actuated points, and $\mathrm{F_i} = 0$ for all the others. A is the set of all the points indexes, $\mathrm{L_{ij}}$ is the distance between points i and j, and α is a mechanical coupling coefficient.

In addition, some useful numerical parameters are defined in Figure 1.2, such as masses, distances, etc.

Parameters: Set 3	
m	0.1
m_{act}	0.4
d	0.30
α	$128 \cdot 0.15$
c	0.1

Figure 1.2: Model parameters

1.2 Dynamics Realization in Python

The first step requires implementing the flexible surface dynamics presented in the form of the acceleration of a point i in a Python function named `fs_dynamics_staticps.py`. In our problem, the state-space vector is $\mathbf{x} = [z_1\, z_2\, z_3\, z_4\, \dot{z}_1\, \dot{z}_2\, \dot{z}_3\, \dot{z}_4]^T$, where the first four state z_1, z_2, z_3, z_4, representing the vertical displacements of the surface points, are labeled as `xx[0], xx[1], xx[2], xx[3]` "sympy" symbolic variables, respectively.

It is worth mentioning that besides the four moving points, the two static points located at both ends of the surface have also been incorporated into the dynamics equation. The one static point is at the distance d, and the other one is at the distance $4d$ from the 1^{st} point p_1. The effect of the static points has been incorporated into the acceleration equations of each point p_i.

1.3 Dynamics Discretization

The next step consists of transferring our flexible surface dynamics written in the continuous-time into the discrete-time domain. For that purpose,

we use the forward Euler discretization method, which is described by the following equation:

$$x_{t+1} = x_t + dt \cdot f_{CT}(x_t, u_t, t), \tag{1.2}$$

where $dt > 0$ sufficiently small discretization step and $f_{CT}(x_t, u_t, t)$ continuous-time dynamics function. For our project, we use the discretization step equal to 10^{-3}, which is small enough, and at the same time ensures a faster code compilation. Having our state space vector $\mathbf{x}$ and the acceleration equations, we can construct our state space model in the continuous time domain as follows:

$$\begin{cases} \dot{x_1} = x_5, \\ \dot{x_2} = x_6, \\ \dot{x_3} = x_7, \\ \dot{x_4} = x_8, \\ \dot{x_5} = \ddot{z}_1, \\ \dot{x_6} = \ddot{z}_2, \\ \dot{x_7} = \ddot{z}_3, \\ \dot{x_8} = \ddot{z}_4. \end{cases} \tag{1.3}$$

Subsequently, using the equation (1.2), we discretize our state space dynamics in Python, which has the set of equations shown in Equation 1.4.

$$\begin{cases} x_{1,t+1} = x_{1,t} + dt \cdot x_{5,t}, \\ x_{2,t+1} = x_{2,t} + dt \cdot x_{6,t}, \\ x_{3,t+1} = x_{3,t} + dt \cdot x_{7,t}, \\ x_{4,t+1} = x_{4,t} + dt \cdot x_{8,t}, \\ x_{5,t+1} = x_{5,t} + dt \cdot \ddot{z_{1,t}}, \\ x_{6,t+1} = x_{6,t} + dt \cdot \ddot{z_{2,t}}, \\ x_{7,t+1} = x_{7,t} + dt \cdot \ddot{z_{3,t}}, \\ x_{8,t+1} = x_{8,t} + dt \cdot \ddot{z_{4,t}}. \end{cases} \tag{1.4}$$

1.4 Gradient Computations

Towards the end of `fs_dynamics_staticps.py` function, we have added the gradients computations with respect to the states and inputs using the symbolic notation of the variables. The gradients will later be used to linearize our dynamics around a desired trajectory. Lastly, the gradients have been converted into the `NumPy` arrays for the convenience of subsequent operations with them.

Chapter 2

Trajectory generation (I)

In order to apply a trajectory tracking algorithm, a desired trajectory has to be defined. In this chapter, we compute exactly two equilibrium points for system and define a simple reference curve between them. It is important to mention that the desired trajectory does not have to satisfy the system's dynamics.

2.1 Implementation

2.1.1 Equilibrium Computation

To find the equilibria of the system, we need to solve for states' positions, where all velocities and accelerations are zero, having specific input signals applied. We used `Sympy's nsolve` function, which uses a Newton's method or its variant to find these equilibria numerically. The equilibrium conditions are therefore:

$$\dot{z}_1 = \dot{z}_2 = \dot{z}_3 = \dot{z}_4 = 0 \tag{2.1}$$

$$\ddot{z}_1 = \ddot{z}_2 = \ddot{z}_3 = \ddot{z}_4 = 0 \tag{2.2}$$

The equilibrium solver takes an initial guess for the state values and values for the control inputs, then iteratively finds a solution satisfying (2.1) and (2.2). In order to effectively compute two different equilibrium points, we pass the initial conditions resembling of what we expect system to be in the equilibria. We anticipate the surface to have a slight positive displacement at the points p_1, p_2, and p_4, and the negative displacement at the point p_3 for the 1^{st} equilibrium point, whereas the opposite behavior is assigned for the 2^{nd} equilibrium point. Along this, we pass the values of the control inputs that are small enough to ensure the convergence to the solution for state equilibria. Satisfying all of these conditions, the flexible surface in the

equilibria would resemble the transition between two sinusoids that are in anti-phase.

2.1.2 Trajectory Generation

Following the project guidelines, a reference trajectory must consist of these three phases:

1. Constant initial phase at the first equilibrium
2. Smooth transition period
3. Constant final phase at the second equilibrium

Additionally, the transition has to be symmetric to ensure a smooth movement of the surface. Based on these requirements, we have employed the 5^{th}-order polynomial for our reference trajectory pertained both for states and inputs. This choice was made due to its smooth C^2 continuity, which is suitable for controlling the flexible surface without unnecessary oscillations and transients. Thus, the polynomial trajectory for each state component has the equation below:

$$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \tag{2.3}$$

We need to find a set of polynomial coefficients pertained to each of the state reference trajectories. It is accomplished by imposing initial and final conditions on the position, velocity and acceleration of the system. The coefficients have been solved in the matrix form in the `compute_5th_order_coeffs.py` function, where the matrix A contains the symbolic coefficients of the 5^{th}-order polynomial equation and its 1^{st}- and 2^{nd}-order derivative equations, where the content of the matrix b is our initial conditions. We have imposed zero velocity and acceleration initial and final conditions, and our two equilibrium points as the initial and final position configurations of the system. The desired 5^{th}-order polynomial trajectories have been derived for each state by applying a specific set of coefficients obtained from the `compute_5th_order_coeffs` function. To be mentioned, the similar approach is adopted for the two input reference trajectories.

2.1.3 Optimal Control via Newton's Method

Quasi-Static Trajectory

As an initial guess for the optimization process we used a quasi-static trajectory, which is basically a collection of points equal to our first equilibrium point. Thus, for the initial trajectory that is to be optimally tracked we have a function that is constant along the whole duration. The quasi-static trajectory serves multiple purposes in our implementation:

1. **Initial Guess:** It provides a physically meaningful starting point for the optimization algorithm, helping avoid convergence to poor local minima.

2. **Kickstart of the Optimal Trajectory Generation:** The linearization around the final point of the reference trajectory enables the computation Q_T via the discrete algebraic Riccati equation (DARE). Using the quasi-static trajectory as the initial system trajectory and the matrix Q_T, we are able to solve the costate and difference Riccati equations using the backward propagation starting from time T to obtain all the necessary matrices for the initial iteration.

3. **Stability Guarantee:** By starting from a sequence of equilibrium states, we ensure that our initial guess satisfies the basic stability requirements of the system.

Optimization Framework

Our optimization framework starts from introducing the cost function or our optimal control problem shown in Equation 2.4 below [2].

$$\begin{aligned}
&\min_{x_{1:T},u_{0:T-1}} \quad \sum_{t=0}^{T-1} \left(||x_t - x_t^{\text{ref}}||_Q^2 + ||u_t - u_t^{\text{ref}}||_R^2 \right) + ||x_T - x_T^{\text{ref}}||_{Q_T}^2 \\
&= \min_{x_{1:T},u_{0:T-1}} \quad \sum_{t=0}^{T-1} \ell_t(x_t, u_t) + \ell(x_T) \\
&\text{s.t.} \quad x_{t+1} = f(x_t, u_t), \quad t = 0, \ldots, T-1 \\
&x_0 = x_{\text{init}}
\end{aligned} \tag{2.4}$$

Via the shooting technique, we could achieve our optimal control problem to be dependent only on the control input u, while satisfying our optimization constraints. The reduced optimal control problem is described by Equations 2.5 [2].

$$\begin{aligned}
&\min_{u_{0:T-1}} \quad \sum_{t=0}^{T-1} \ell_t(\phi_t(\mathbf{u}), u_t) + \ell(\phi_T(\mathbf{u})) \\
&= \min_{u_{0:T-1}} \quad J(\mathbf{u})
\end{aligned} \tag{2.5}$$

We would then compute $\nabla J(\mathbf{u})$ solving the costate equation, which employs the gradients of the cost and dynamics functions with respect to the states and inputs. The gradient of J with respect to $\mathbf{u}$ ($\nabla J(\mathbf{u})$) will later be used to find the descent direction for the Armijo step selection rule [2].

Affine LQR

In the closed-loop version of the Newton's method for optimal control, the optimization task is to solve the affine LQR problem and obtain the gain matrices K_t^k and feed-forward actions σ_t^k for an iteration k by solving the difference Riccati equations. By employing the Riccati equations' formula presented in the class, we implement the LQR solver in `solver_LQR.py` file [2].

The optimal solution of the affine LQR problem is shown in Equation 2.6 [2].

$$\begin{aligned} \Delta u_t^* &= K_t^* \Delta x_t^* + \sigma_t^*, \quad t = 0, \dots, T-1 \\ \Delta x_{t+1}^* &= A_t \Delta x_t^* + B_t u_t^*, \quad t = 0, \dots, T-1 \end{aligned} \tag{2.6}$$

where K_t^* and σ_t^* are the optimal solutions of the Riccati equations for a particular iteration k. The Δu_t is of a particular interest because it is used to calculate the optimal descent direction of our cost function used in the Armijo rule and is computed as $\nabla J(\mathbf{u})^T \Delta u_t$.

Armijo Step Size Selection Rule

The Armijo step size selection rule helps to adaptively search for the best step size value, while minimizing our objective function from iteration to iteration. The algorithm reduces the value of the step size γ until the cost function value at iteration $k+1$ becomes less than a slightly modified 1^{st} order linearization of the cost at iteration k. The whole algorithm is described as follows [2]:

1. Set $\bar{\gamma}^0 > 0, \quad \beta \in (0,1), \quad c \in (0,1)$
2. While $J(\mathbf{u^k} + \mathbf{\Delta u}) = J(\mathbf{u}) + c\bar{\gamma}^i \nabla J(\mathbf{u^k})^T \Delta \mathbf{u}$:
   $\bar{\gamma}^{i+1} = \beta \bar{\gamma}^i$
3. Set $\gamma^k = \bar{\gamma}^i$

The Armijo algorithm has been implemented in the separate file called `armijo_rule.py`.

Lastly, we need to compute the new input and state trajectories for the iteration $k+1$ described by Equation 2.7 [2].

$$\begin{aligned} u_t^{k+1} &= u_t^k + K_t^k (x_t^{k+1} - x_t^k) + \gamma^k \sigma_t^k, \quad t = 0, \dots, T-1 \\ x_{t+1}^{k+1} &= f_t(x_t^k, u_t^k), \quad t = 0, \dots, T-1 \\ x_0^{k+1} &= x_0 \end{aligned} \tag{2.7}$$

where γ^k is the step-size selected by the Armijo rule for the iteration k. The forward integrate has been implemented as the last step in `main.py` for the optimal tracking of our desired input and state trajectories.

The entire framework is repeated for a particular number of iterations decided by the termination condition we set. The tracking is stopped when the value of the norm of the descent direction computed as $\Delta(u^k)^T \Delta u^k$ is less or equal than 10^{-6} - our termination condition.

2.2 Results

2.2.1 Equilibrium Computation

By giving the initial conditions for the states as $x_{0,E_1} = [0.001, 0.0005, -0.001, 0.0005, 0, 0, 0, 0]^T$ and $x_{0,E_2} = [-0.001, -0.0005, 0.001, -0.0005, 0, 0, 0, 0]^T$ and assigning the chosen inputs, our numerical solver successfully converged to two stable equilibrium configurations:

First Equilibrium (E_1):

$$\begin{aligned} x_{E_1} &= [1.86 \times 10^{-4}, 4.06 \times 10^{-4}, 7.30 \times 10^{-5}, -2.63 \times 10^{-4}, 0, 0, 0, 0]^T \\ u_{E_1} &= [0.5, -0.5]^T \end{aligned} \tag{2.8}$$

Second Equilibrium (E_2):

$$\begin{aligned} x_{E_2} &= [-1.86 \times 10^{-4}, -4.06 \times 10^{-4}, -7.30 \times 10^{-5}, 2.63 \times 10^{-4}, 0, 0, 0, 0]^T \\ u_{E_2} &= [-0.5, 0.5]^T \end{aligned} \tag{2.9}$$

2.2.2 Trajectory Generation

Having found two equilibria points for each state, we connect them with the 5^{th}-order polynomial, thus generating a smooth trajectory for tracking, as shown in Figures 2.1 and 2.2.

2.2.3 Optimal Control via Newton's Method

Optimal Control with the Fixed Step Size

Initially, we want to check how the Newton's method manages to track the desired trajectory using only a fixed step size predefined at the start. The step size is equal to 0.5. The optimal trajectory generation has been terminated after the 15 iterations, when the norm of the descent direction reached the termination condition value. The optimal and desired state trajectories for the 2^{nd}, 5^{th}, 8^{th}, and the last iterations have been plotted and shown in Figures 2.3, 2.5, 2.7, 2.9, respectively. Likewise, the optimal and desired input trajectories for the same iterations are shown in Figures 2.4, 2.6, 2.8, 2.10, respectively.

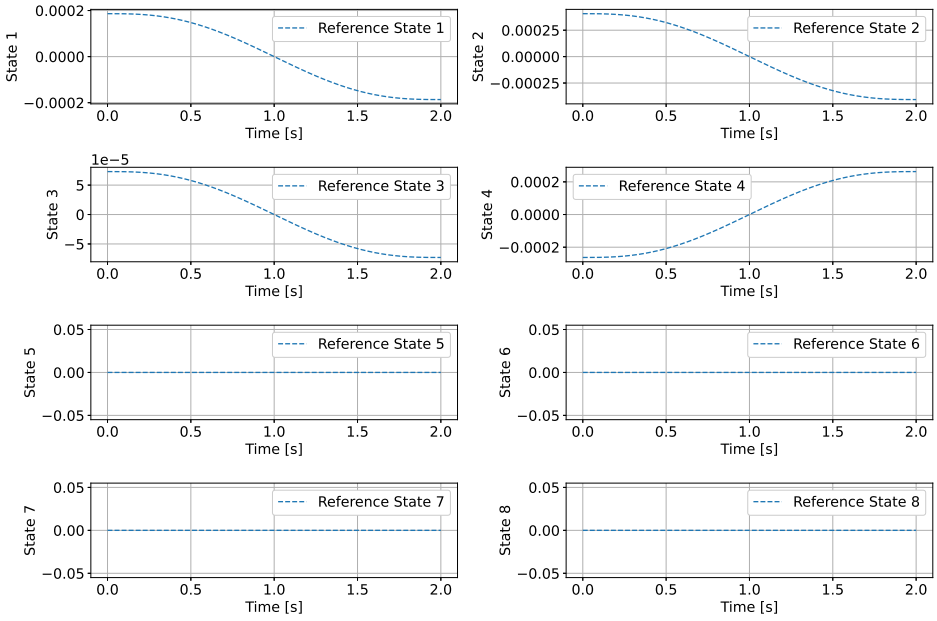

Figure 2.1: The reference state trajectories as the 5^{th}-order polynomials

Optimal Control with the Armijo Step Size

The next step is to apply the Armijo step selection rule to change the step size adaptively in the iterative scheme. The parameters have been chosen as follows: $\gamma^0 = 1.0$, $\beta = 0.7$ and $c = 0.5$. The algorithm has converged and terminated after 5 iterations. Figures 2.11, 2.14, and 2.17 show the Armijo plot for the 2^{nd}, 3^{rd}, and the last iterations, respectively. As you see from the plots, in the 2^{nd} and 3^{rd} iterations Armijo chooses the step size of 0.7, and in the last iteration it takes the value of 1.0, having the target cost function value lower than the value of the previous iteration already. Figures 2.12, 2.15, and 2.18 depict the optimal and desired state trajectories using Armijo for the aforementioned iterations, respectively. Figures 2.13, 2.16, and 2.19 depict the optimal and desired input trajectories using Armijo for the aforementioned iterations, respectively.

Lastly, the graphs of the norm of the descent direction and the cost function on the semi-logarithmic scale have been presented in Figures 2.20 and 2.21. From them we can observe, that both values constantly decrease during 5 iterations.

To summarize, when comparing the results of the Newton's method for the fixed step size and the Armijo step size, we observe that the final optimal trajectories are nearly the same. However, the convergence speed is substantially different for both approaches: 16 iterations for the fixed step size versus 5 iteration for the Armijo rule.

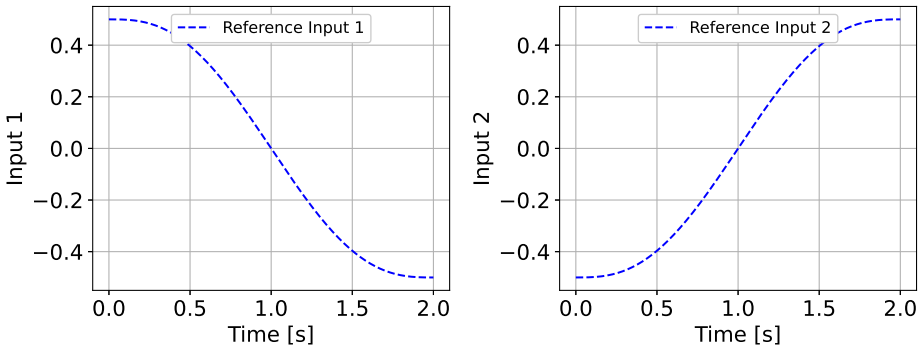

Figure 2.2: The reference input trajectories as the 5^{th}-order polynomials

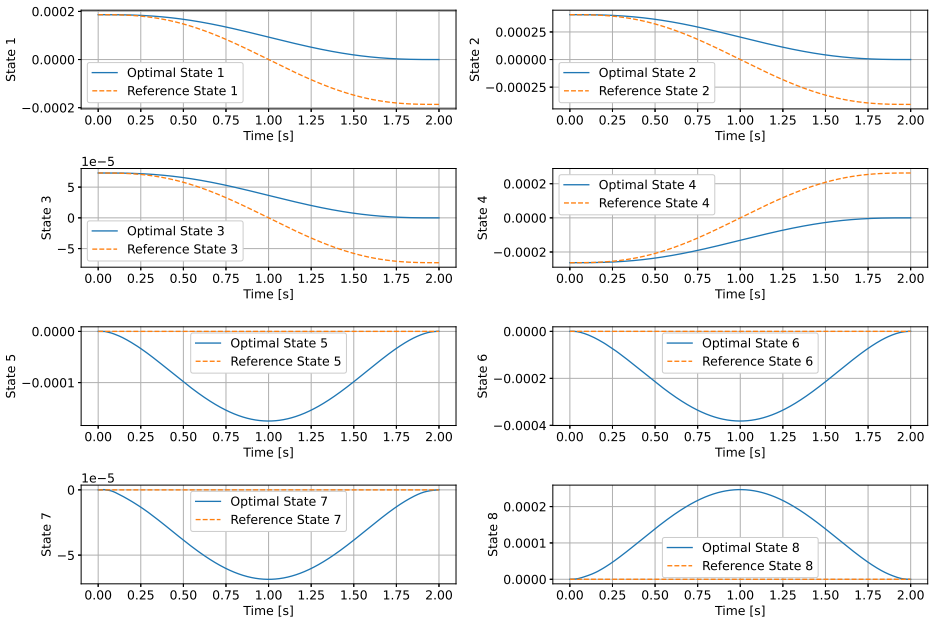

Figure 2.3: Optimal and desired state trajectories for the 2^{nd} iteration

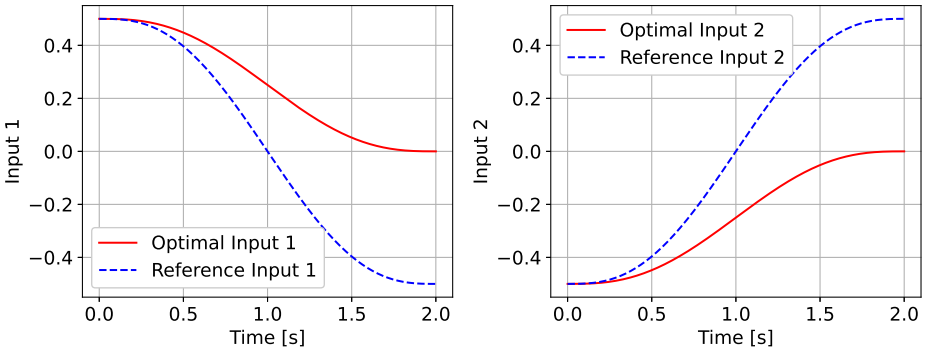

Figure 2.4: Optimal and desired input trajectories for the 2$^{\text{nd}}$ iteration

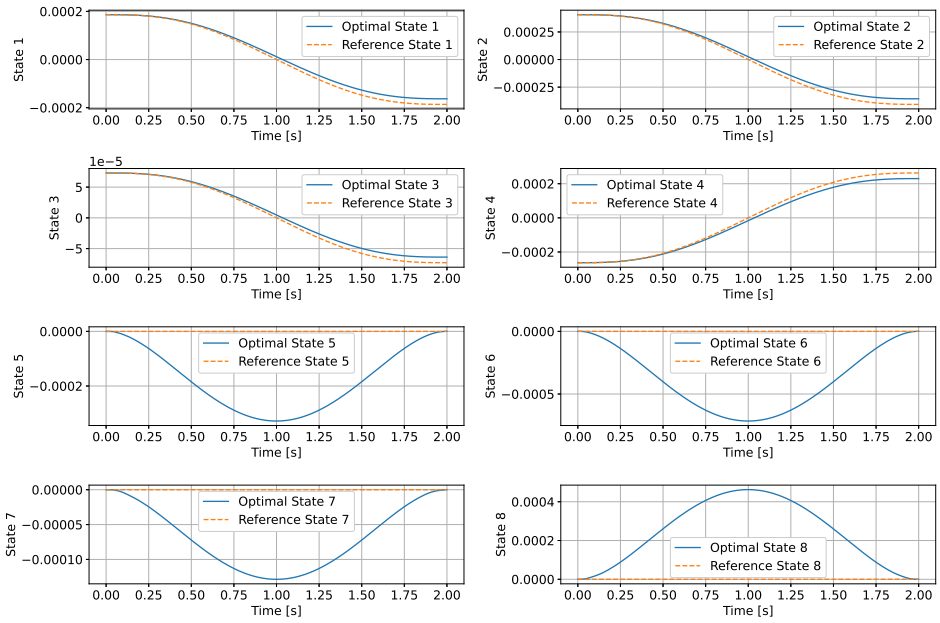

Figure 2.5: Optimal and desired state trajectories for the 5$^{\text{th}}$ iteration

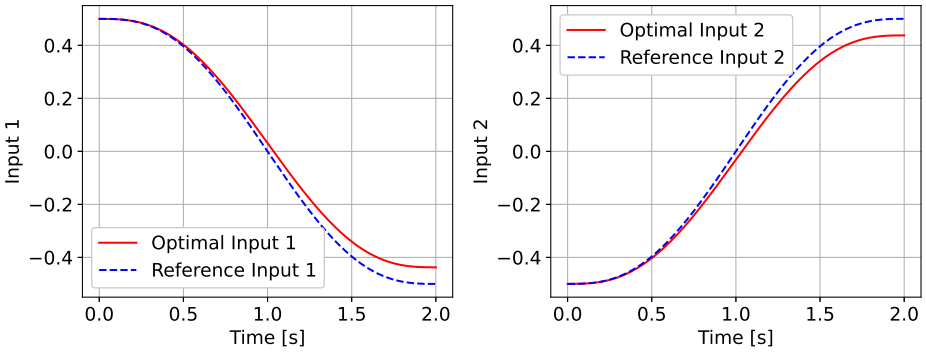

Figure 2.6: Optimal and desired input trajectories for the 5^{th} iteration

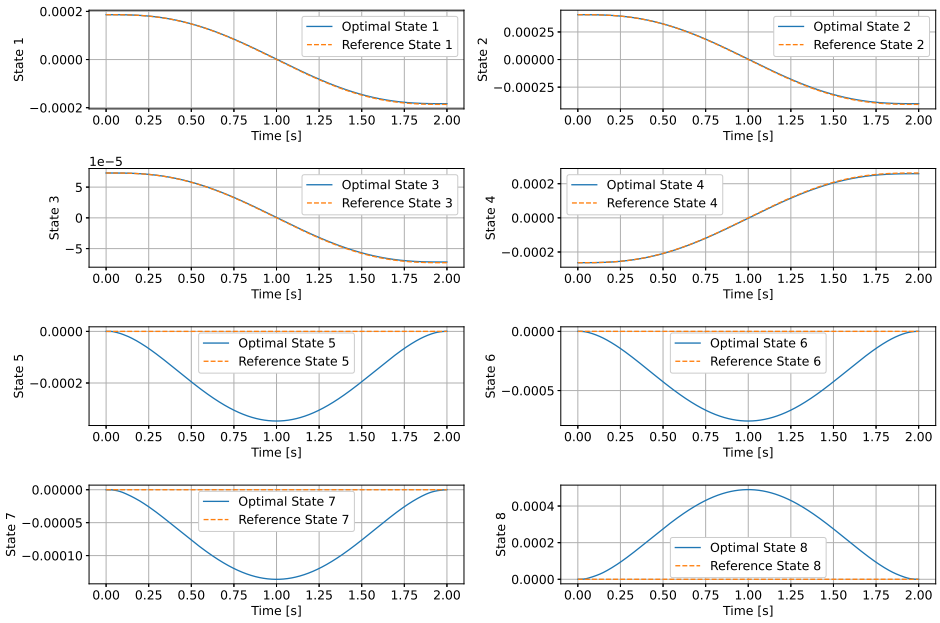

Figure 2.7: Optimal and desired state trajectories for the 8^{th} iteration

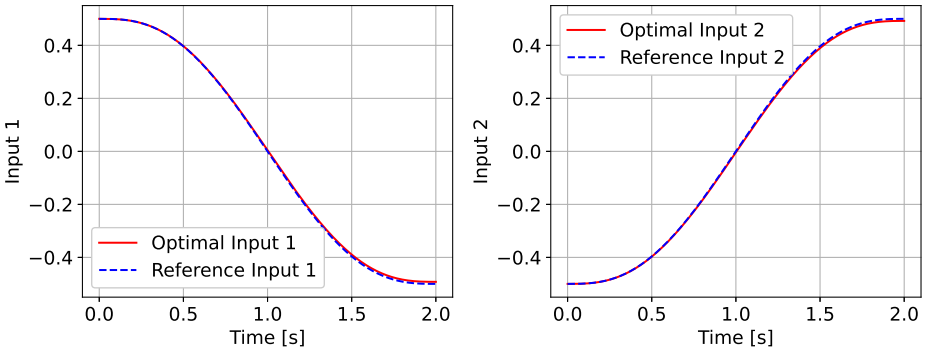

Figure 2.8: Optimal and desired input trajectories for the 8^{th} iteration

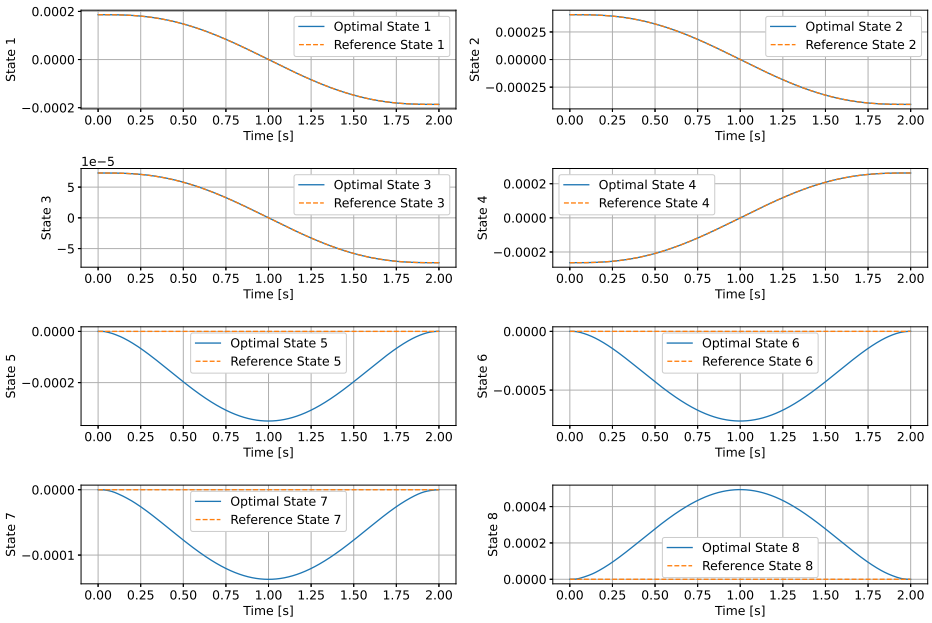

Figure 2.9: Optimal and desired state trajectories for the last iteration

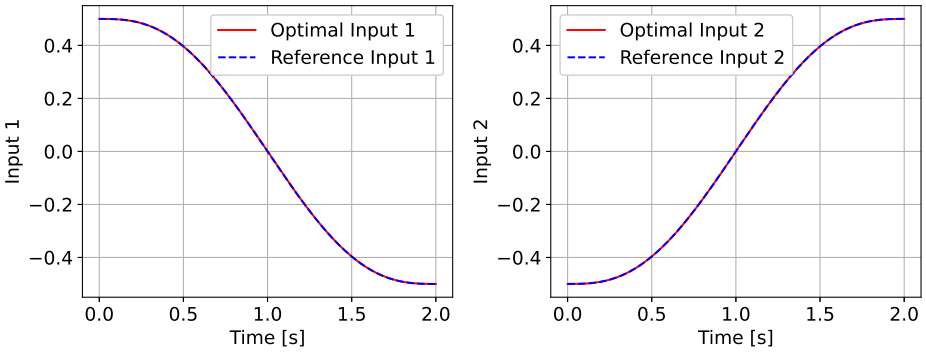

Figure 2.10: Optimal and desired input trajectories for the last iteration

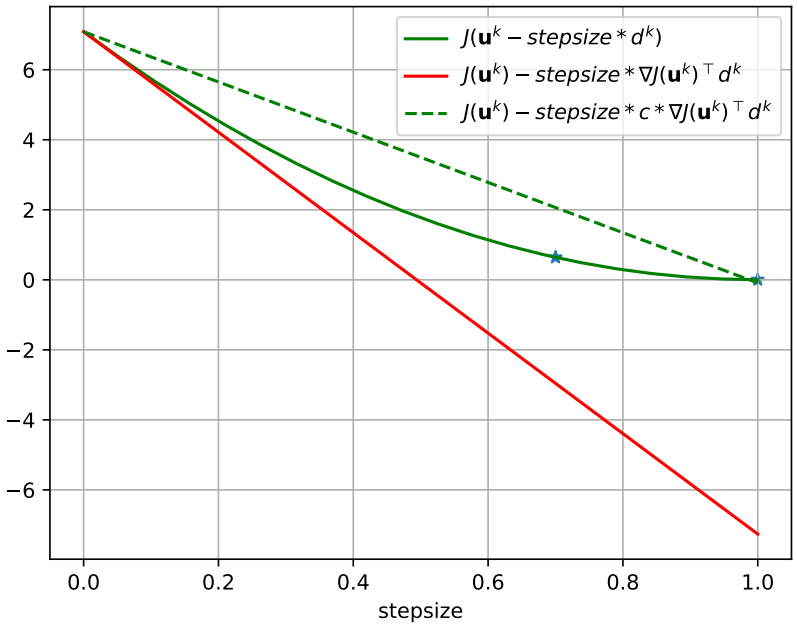

Figure 2.11: Armijo plot for the 2$^{\text{nd}}$ iteration

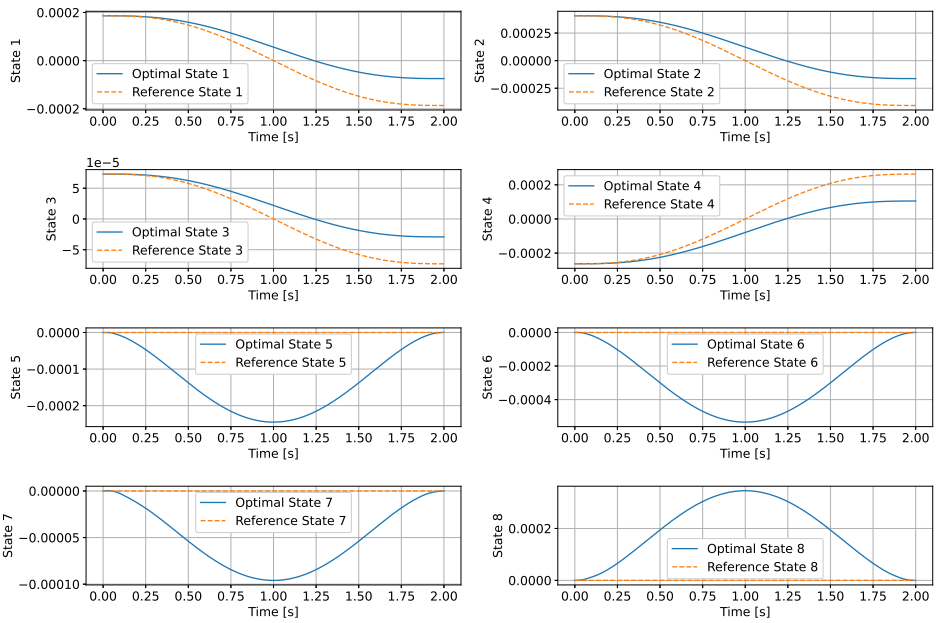

Figure 2.12: Optimal and desired state trajectories for the 2nd iteration using the Armijo rule

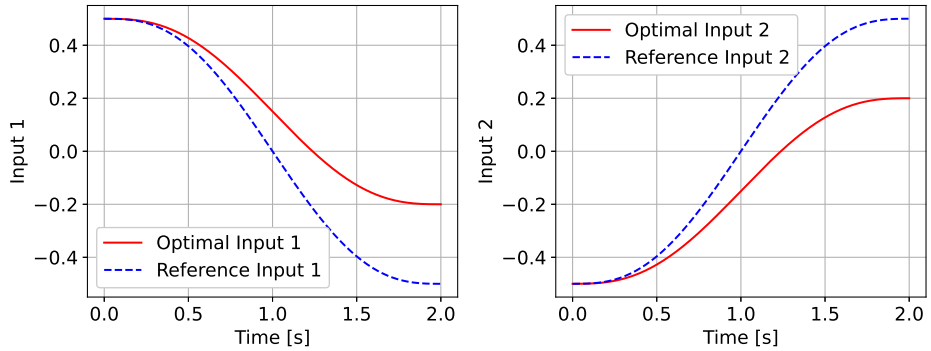

Figure 2.13: Optimal and desired input trajectories for the 2nd iteration using the Armijo rule

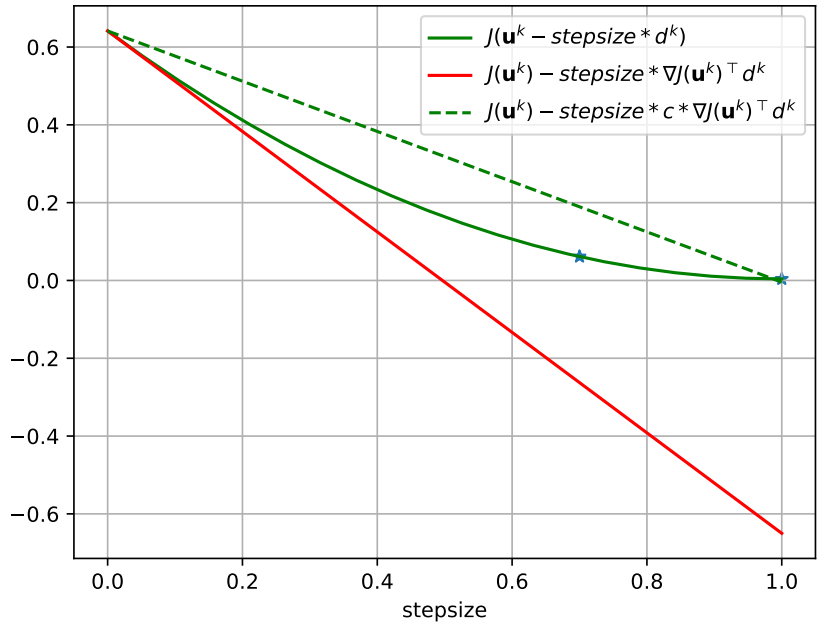

Figure 2.14: Armijo plot for the 3rd iteration

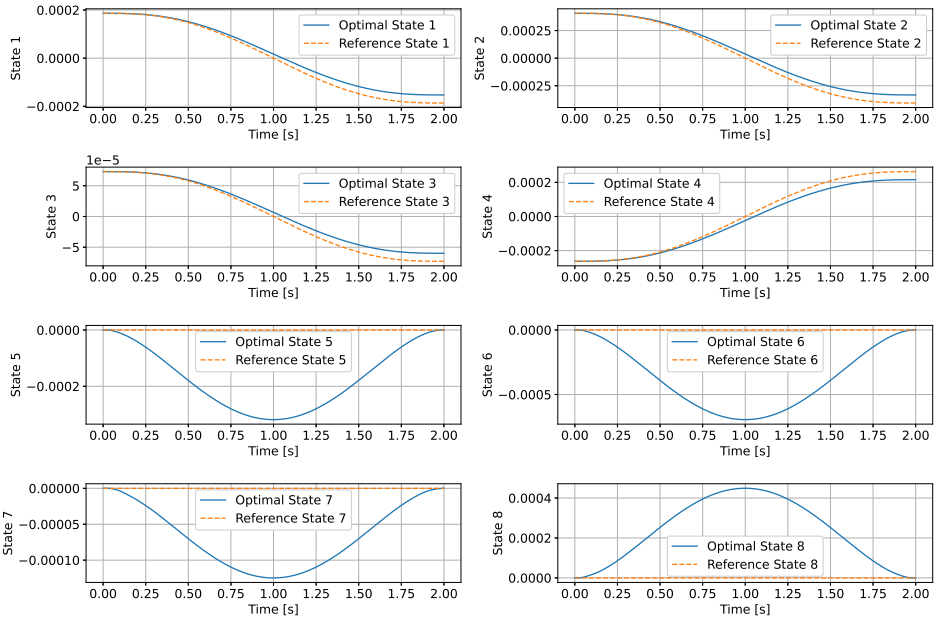

Figure 2.15: Optimal and desired state trajectories for the 3rd iteration using the Armijo rule

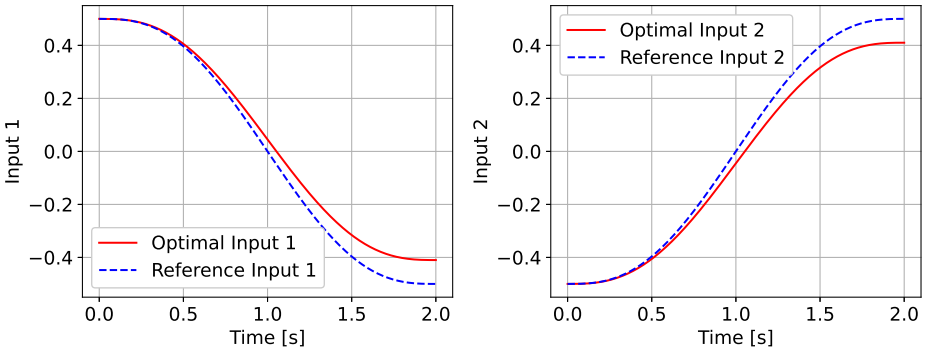

Figure 2.16: Optimal and desired input trajectories for the 3$^{\text{rd}}$ iteration using the Armijo rule

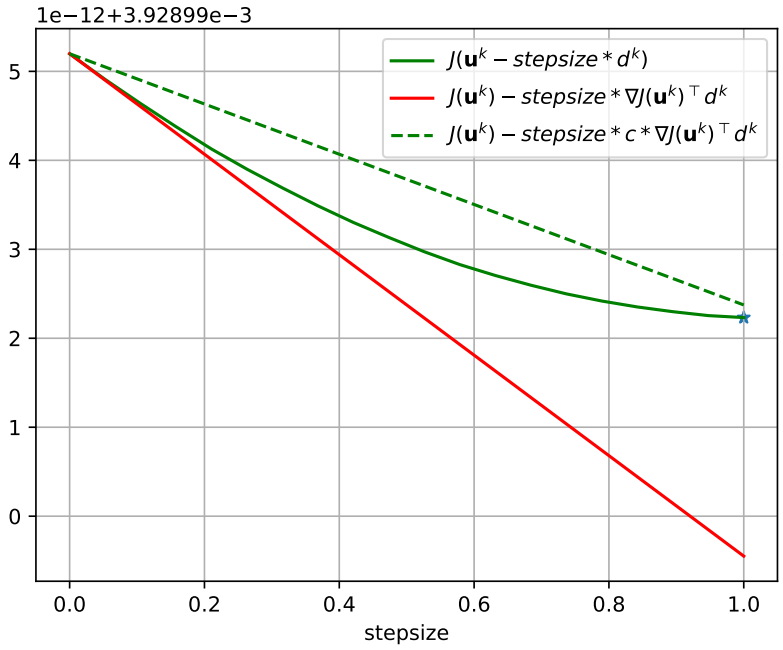

Figure 2.17: Armijo plot for the last iteration

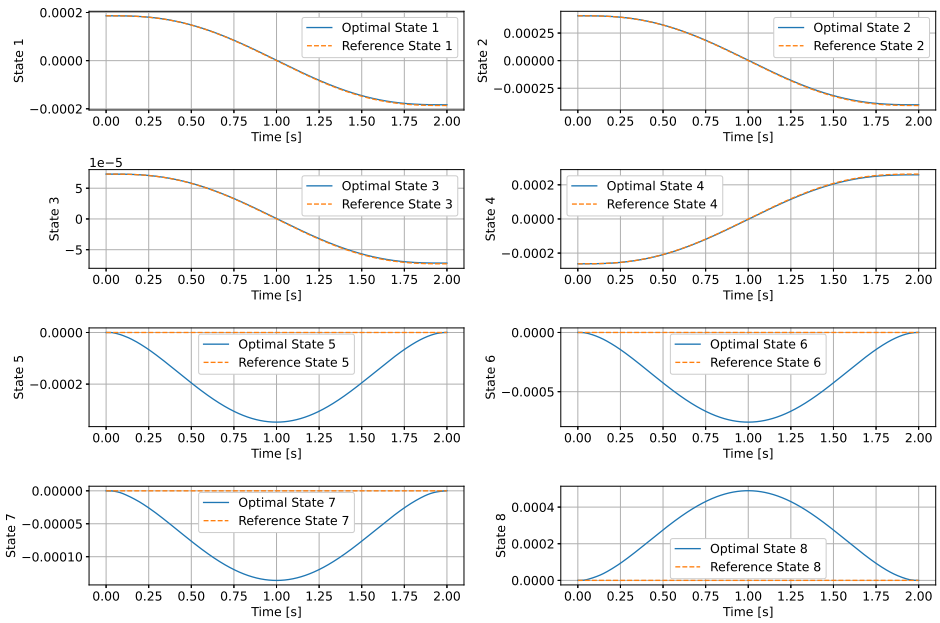

Figure 2.18: Optimal and desired state trajectories for the last iteration using the Armijo rule

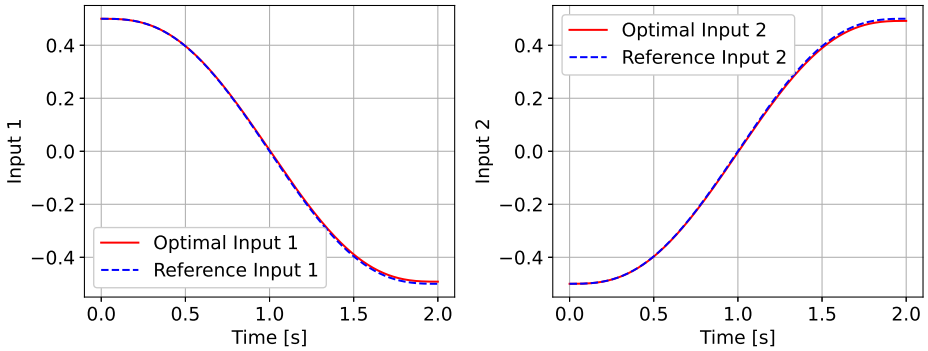

Figure 2.19: Optimal and desired input trajectories for the last iteration using the Armijo rule

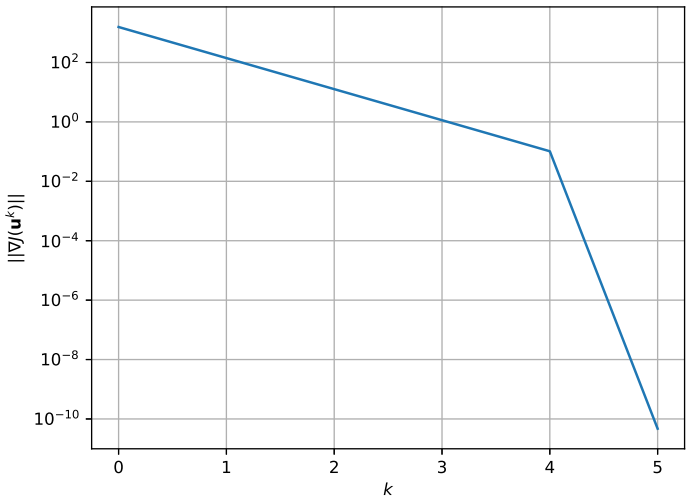

Figure 2.20: Norm of the descent direction graph

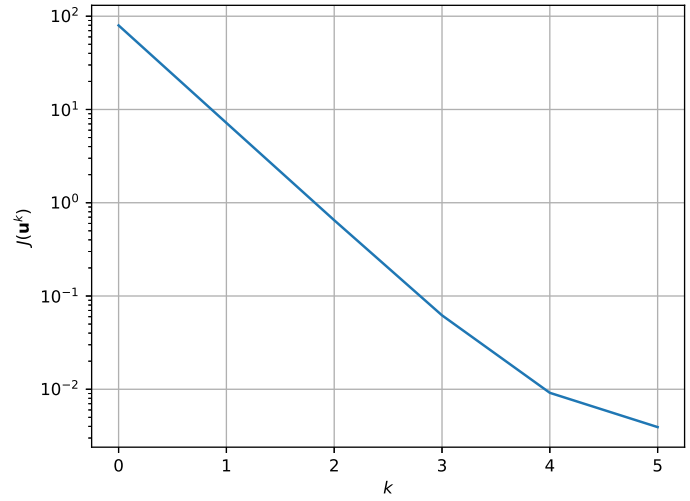

Figure 2.21: Cost function graph

Chapter 3

Trajectory generation (II)

3.1 Implementation

3.1.1 New Trajectory Generation

In Task 2 of the project, we have been asked to generate a new reference trajectory that is more complex than the previous one from Task 1 in a sense that there are multiple transitions between our equilibria points. Those transitions between the equilibria points have been defined by the same 5^{th}-order polynomial functions described in Chapter 2.

3.1.2 Optimal Control via Newton's Method

The optimal control via Newton's Method has the same steps for the new trajectory as for the previous one described in Chapter 2: starting from the quasi-static trajectory initialization and ending with the forward integrate. For the new trajectory, we have performed the optimal tracking using the Armijo step size selection with the same parameters: $\gamma^0 = 1.0, \beta = 0.7, c = 0.5$. The whole essence of Task 2 is to use this optimal trajectory derived by Newton's method to track it with the Linear Quadratic Regulator (LQR) and the Model Predictive Control (MPC) in the subsequent tasks using the perturbed initial conditions.

3.2 Results

3.2.1 New Trajectory Generation

We have slightly modified the code for the reference trajectory generation to make it transit from the 1^{st} equilibrium to the 2^{nd} one, stay there for 0.5 seconds, return back to the 1^{st} equilibrium and remain there till the end. The control input signals have been adjusted accordingly with the same

nominal values. The new reference trajectories for the states and the inputs are depicted in Figures 3.1 and 3.2.

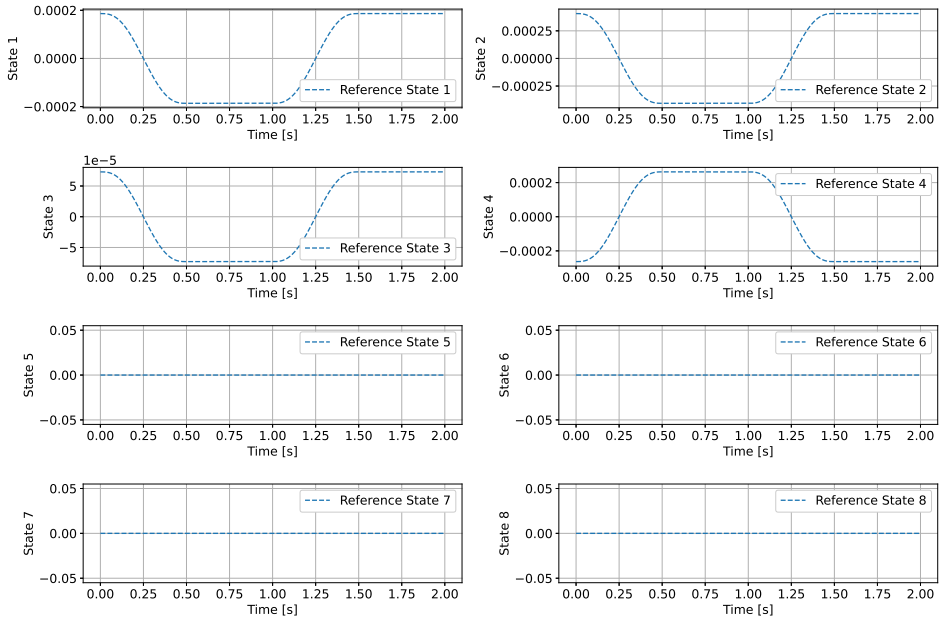

Figure 3.1: Desired state trajectories with multiple transitions between equilibria

3.2.2 Optimal Control via Newton's Method

The algorithm applied on the new trajectories has converged in 4 iterations. The Armijo plot for the 2^{nd}, 3^{rd}, and last iterations are shown in Figures 3.3, 3.6, and 3.9, respectively. The results for the optimal control via Newton's method for the state trajectories in the 2^{nd}, 3^{rd}, and last iterations are shown in Figures 3.4, 3.7, and 3.10, respectively. The results for the optimal control via Newton's method for the input trajectories in the 2^{nd}, 3^{rd}, and last iterations are shown in Figures 3.5, 3.8, and 3.11, respectively. Lastly, the norm of the descent direction and cost function graphs for the multiple equilibria case have been presented in Figures 3.12 and 3.13, respectively.

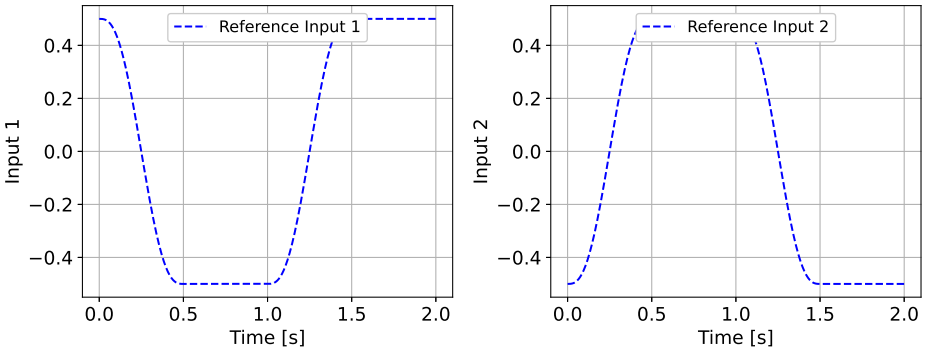

Figure 3.2: Desired input trajectories with multiple transitions between equilibria

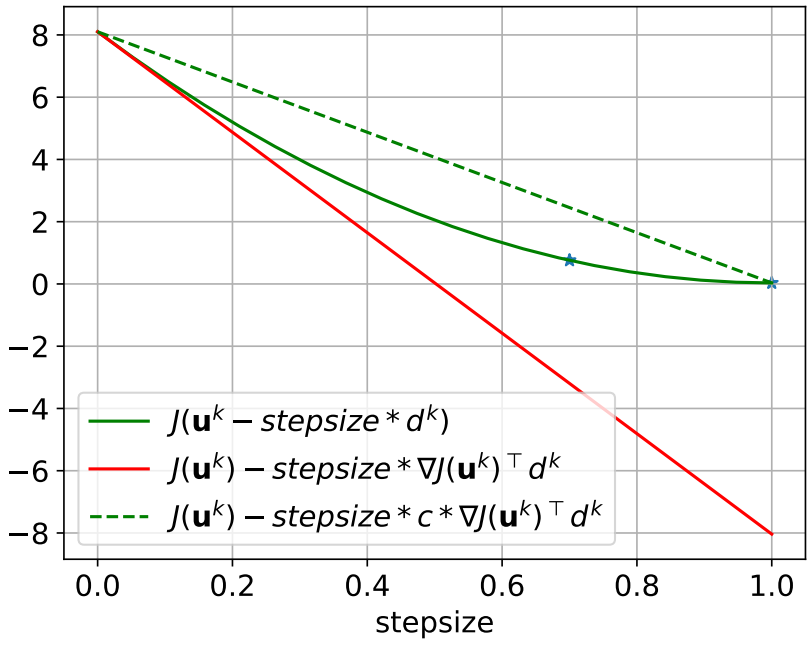

Figure 3.3: Armijo plot for the 2nd iteration

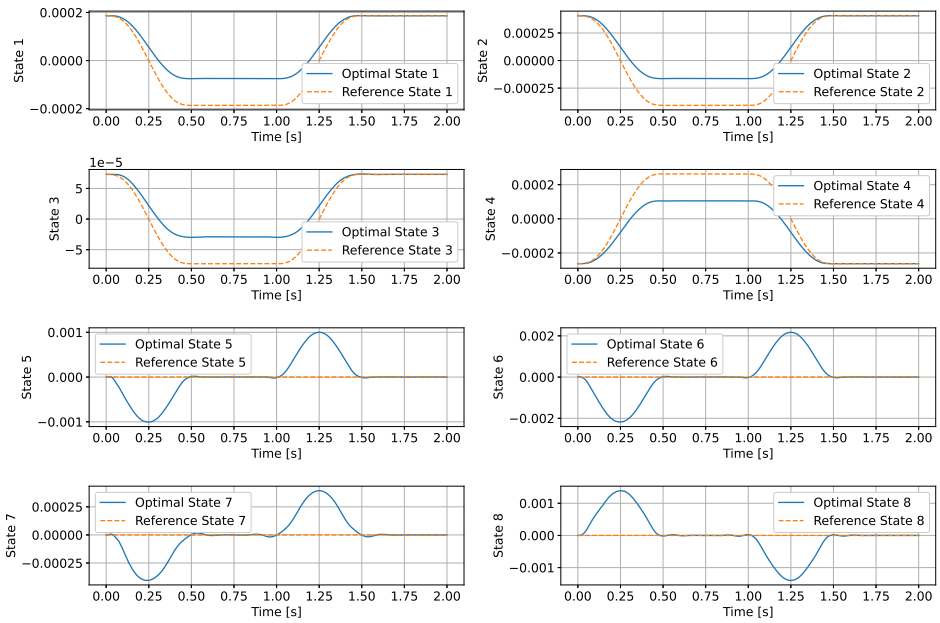

Figure 3.4: Optimal and desired state trajectories with multiple transitions between equilibria for the 2$^{\text{nd}}$ iteration

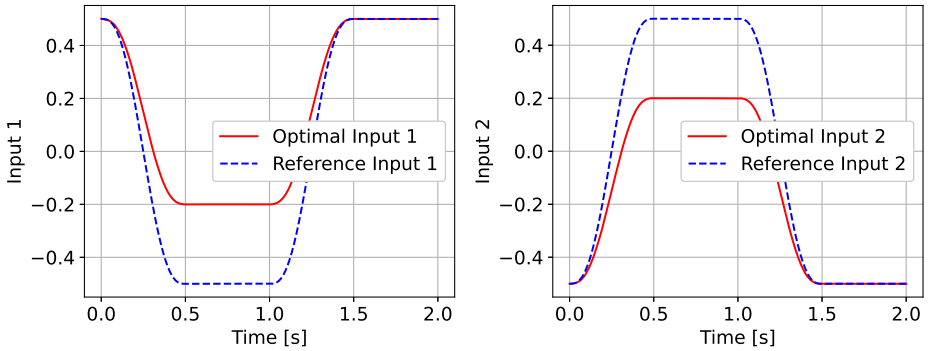

Figure 3.5: Optimal and desired input trajectories with multiple transitions between equilibria for the 2$^{\text{nd}}$ iteration

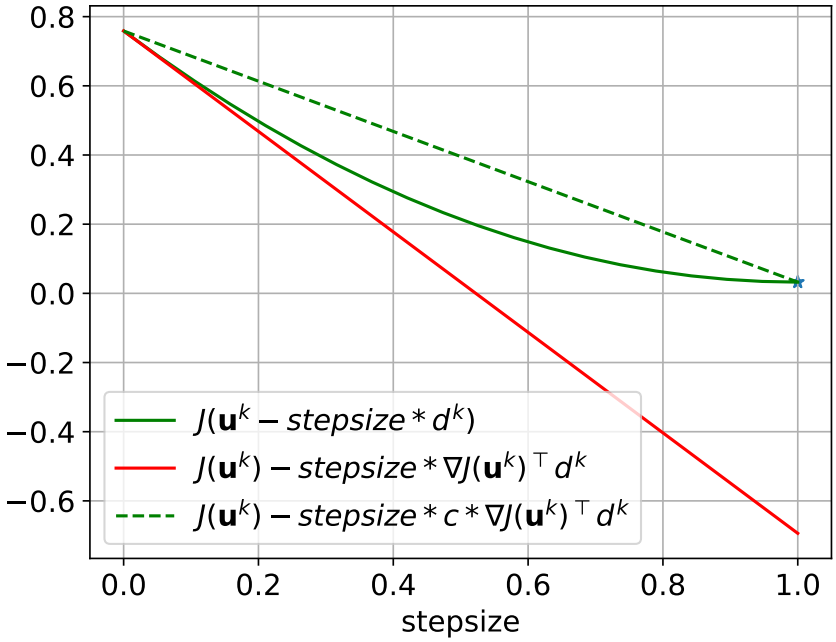

Figure 3.6: Armijo plot for the 3rd iteration

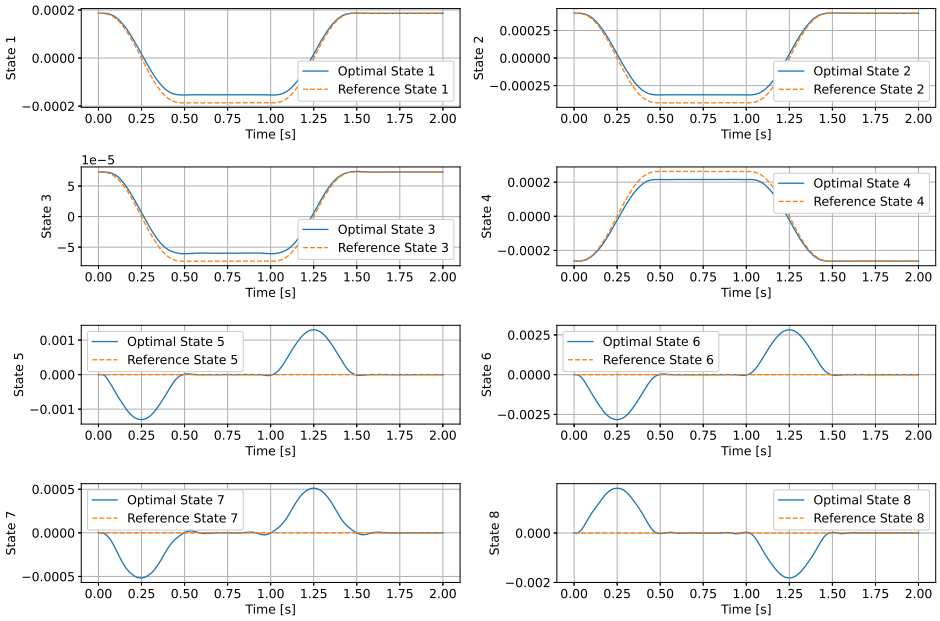

Figure 3.7: Optimal and desired state trajectories with multiple transitions between equilibria for the 3rd iteration

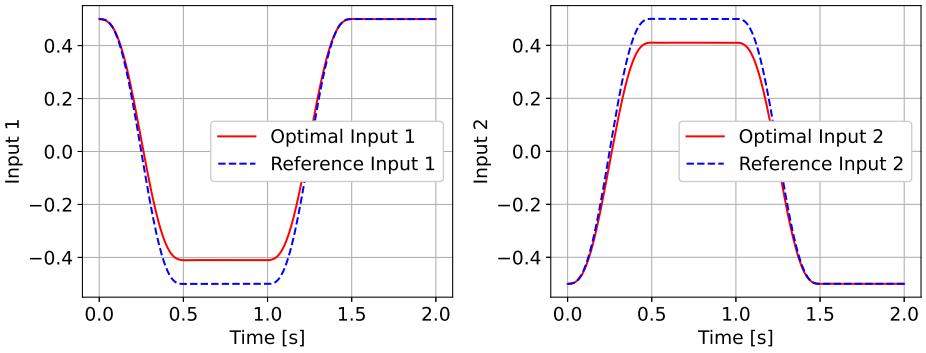

Figure 3.8: Optimal and desired input trajectories with multiple transitions between equilibria for the 3rd iteration

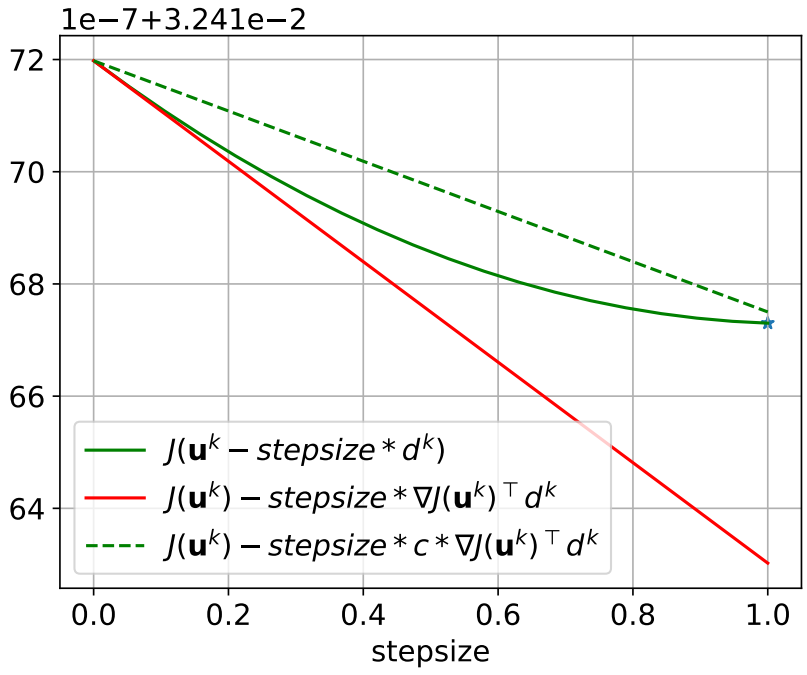

Figure 3.9: Armijo plot for the last iteration

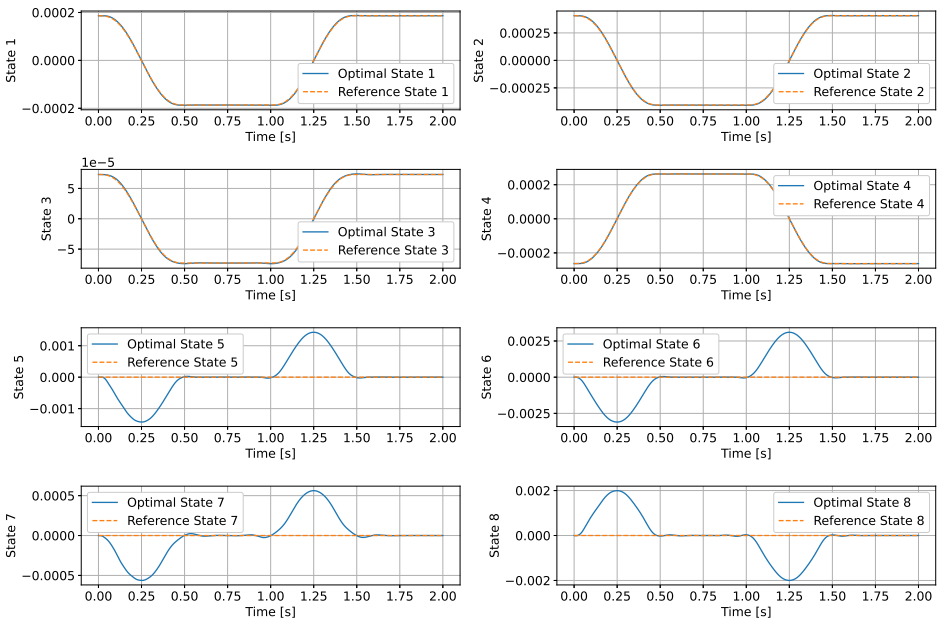

Figure 3.10: Optimal and desired state trajectories with multiple transitions between equilibria for the last iteration

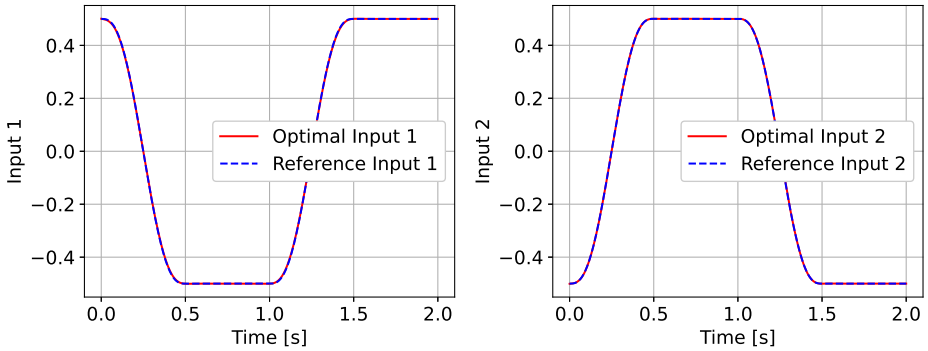

Figure 3.11: Optimal and desired input trajectories with multiple transitions between equilibria for the last iteration

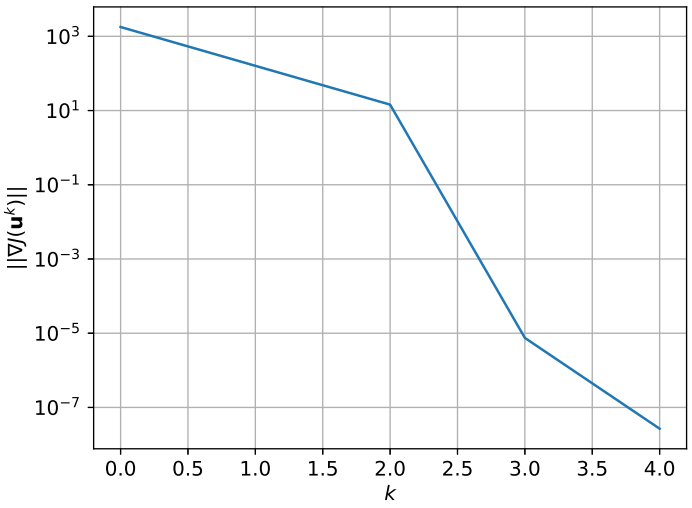

Figure 3.12: Norm of the descent direction graph for multiple equilibria case

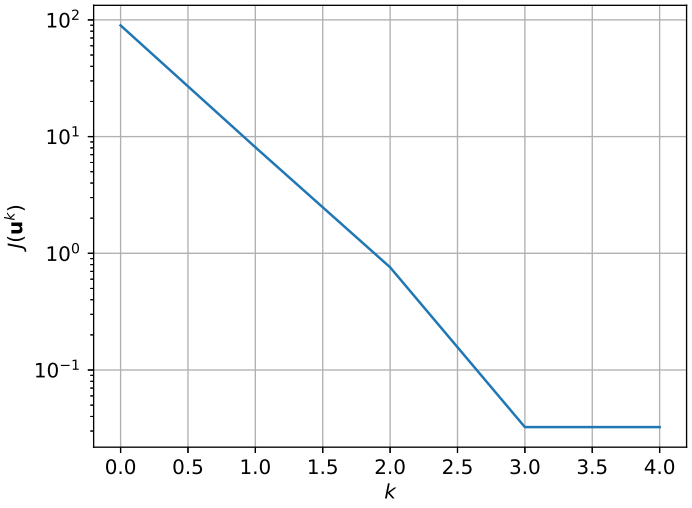

Figure 3.13: Cost function graph for multiple equilibria case

Chapter 4

Trajectory tracking via LQR

4.1 Control method description

In Task 3, a Linear Quadratic Regulator (LQR) is designed to track the reference trajectory computed in Task 2. The system dynamics are linearized about the reference trajectory $(x^{\text{ref}}, u^{\text{ref}})$, resulting in the dynamic model shown in Equation 4.1 [2].

$$\Delta x_{t+1} = A_t \Delta x_t + B_t \Delta u_t \tag{4.1}$$

The LQR cost function is defined in terms of deviations from the reference trajectory shown in Equation 4.2 [2].

$$J = \sum_{t=0}^{T-1} \Delta x_t^\top Q \Delta x_t + \Delta u_t^\top R \Delta u_t + \Delta x_T^\top Q_T \Delta x_T \tag{4.2}$$

The corresponding feedback control law becomes:

$$\Delta u_t = K_t \Delta x_t \tag{4.3}$$

4.2 Implementation

The gain sequence K_t was computed using a finite-horizon discrete-time LQR solver based on the linearized system dynamics. The simulation was performed in closed loop: at each time step, the control input was computed using the LQR feedback law, and the next state was propagated by applying the dynamics of the model.

To evaluate robustness, an additional simulation was performed with a small perturbation added to the initial state.

4.3 Results

4.3.1 LQR Tracking Plots

From Figures 4.1 and 4.2, we observe that both position states and control inputs closely follow their respective reference trajectories throughout the simulation. The system and reference trajectories are almost indistinguishable, indicating that the LQR feedback controller is highly effective in maintaining the system close to the optimal trajectory generated in Task 2.

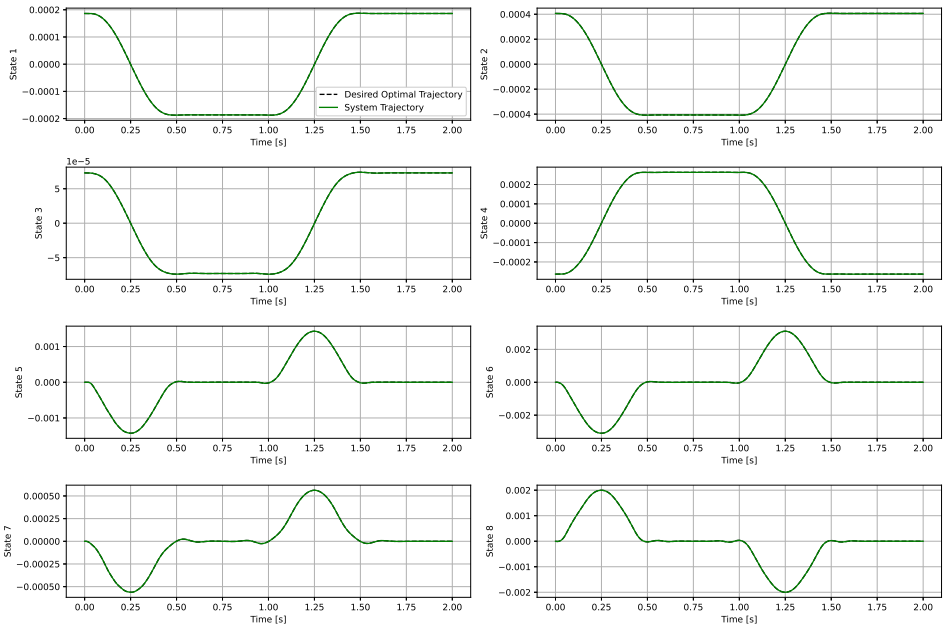

Figure 4.1: LQR tracking results for state variables

4.3.2 LQR Tracking Plots - Perturbed Case

To showcase tracking performances, the system was perturbed by applying different initial conditions: instead of applying x_0^{traj}, the following was added to the former initial conditions, i.e. the first equilibrium position coordinates:

$$perturbation = 0.2 \cdot x_{eq1}$$

As shown in Figure 4.3, the LQR controller successfully tracks states of the system even in the perturbed case because the feedback compensates well for initial deviation. Bigger oscillations are seen in the velocity plots, while position plots show only little deviation from the desired trajectory.

Figure 4.4 illustrates the tracking error for the first four states (corresponding to the position of the four points of the flexible surface). The error

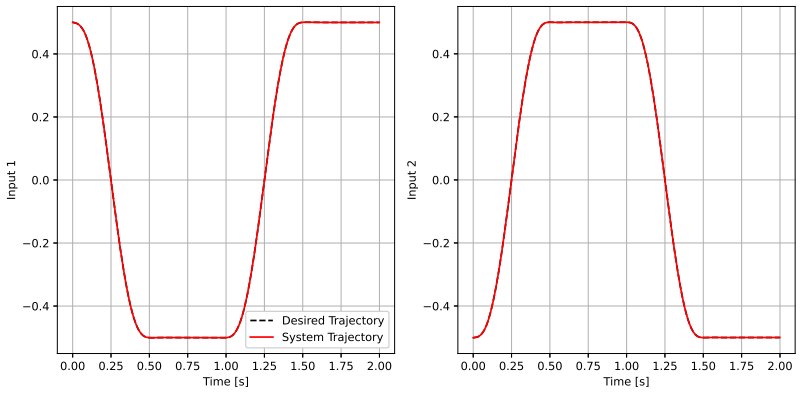

Figure 4.2: Control input trajectories

remains small and only a few oscillations are displayed at the beginning of the trajectories.

In Figure 4.5, the tracking performances of the control inputs are shown. Here as well, deviations are present only at the very beginning of motion.

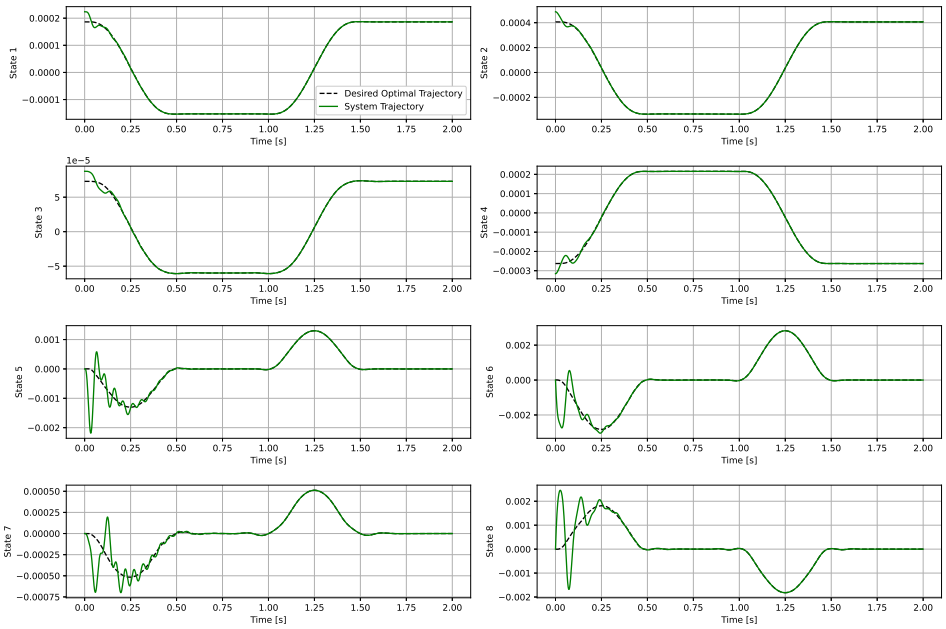

Figure 4.3: Perturbed LQR - Tracked and reference trajectories for system states

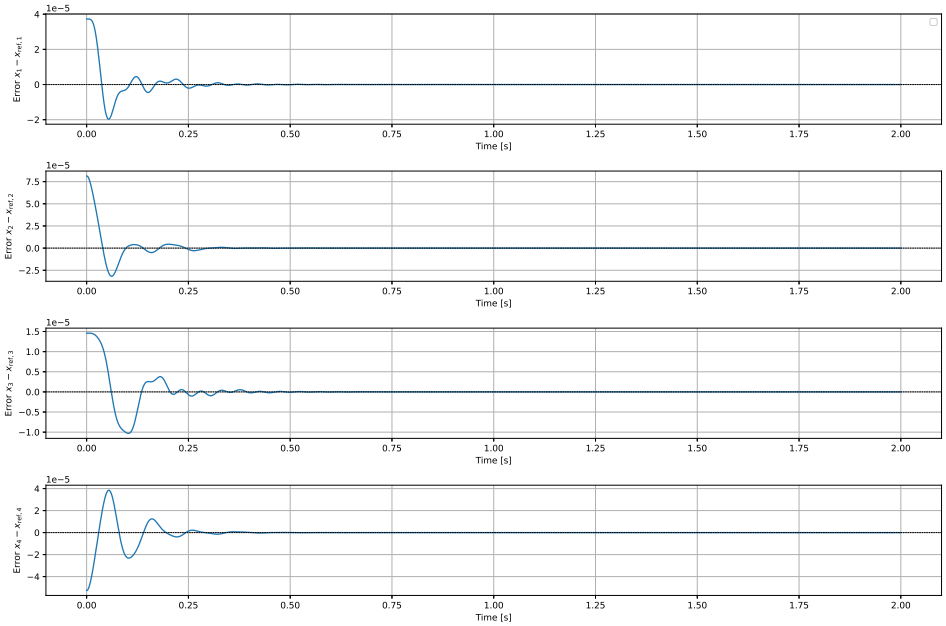

Figure 4.4: Perturbed LQR - Tracking error for position states

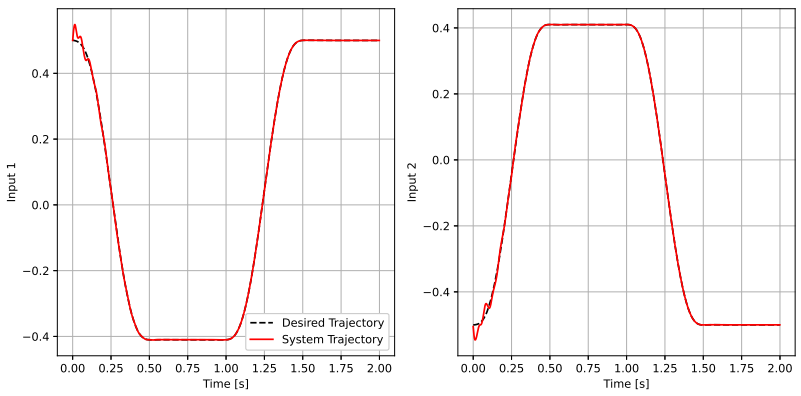

Figure 4.5: Perturbed LQR - Control input trajectories

Chapter 5

Trajectory tracking via MPC

5.1 Control method description

Model Predictive Control (MPC) is a model that enables a system to track a desired trajectory while respecting input and state constraints. This method predicts the system's future behavior over a limited time frame, known as the horizon [2]. The optimal inputs for the control are then computed thanks to these predictions. This process is repeated periodically, with the horizon shifting forward after each interval [2]. It is worth noticing that different types of constraints can be introduced, to limit the input values or to avoid undesired positions.

5.2 Implementation

In this section, we implement MPC to control our surface. We begin by defining the simulation parameters, including the time horizon, input constraints, and state constraints. The constraints reflect possible physical limitations of the actuators and desired limitations on the displacements. The MPC control was firstly developed in an unconstrained environment, only then we added these constraints to obtain a more realistically applicable solution.

The MPC loop runs for the entire simulation horizon: it solves the optimization problem at each step and applies the optimal control input to the system. At each iteration:

1. The MPC solver computes the optimal control input sequence based on the current state and the desired optimal reference trajectory.

2. Only the first control input from the computed sequence is applied to the system.

3. The system state is updated using the system dynamics.

Addressing the working mechanism of the MPC solver, a function basically formulates and solves the constrained optimization problem at each time step considering the chosen prediction horizon. The optimization problem is solved by minimizing the previously used cost function. The constraints must be implemented, so that the system dynamics are taken into account, input constraints and vertical displacements are not excessive, and the starting point is set to the current state. To compute a solution the OSQP solver was used, which is efficient in solving constrained quadratic programs [3]. Terminal cost is computed and the control input to be applied the current time step is returned, along with the complete sequence of states and optimal inputs over the whole prediction horizon.

An important design choice was done regarding the prediction time. Since the system is highly non-linear, using a very short prediction time - 5 time steps - resulted in a much better tracking of the trajectory. Longer horizons produced bad tracking performances and heavier computational load.

5.3 Results

5.3.1 MPC Tracking Plots

Similarly to LQR, the performance registered shows that in the unperturbed case, the tracking is almost perfect, displaying a trajectory overlapping with the desired optimal one (Figure 5.1 and 5.2).

5.3.2 MPC Tracking Plots - Perturbed Case

When applying different initial conditions to simulate a perturbation, the MPC model did not manage to catch up to the reference and showed high oscillations. To improve performances, positions are shown in Figure 5.3 and control inputs in Figure 5.5, the prediction horizon was lengthened to 7 time steps instead of 5. Following this change, the tracking improved greatly with oscillations comparable to LQR control. After a short transitory the tracking error is driven to zero and the oscillations are dampened within half a second (Figure 5.4).

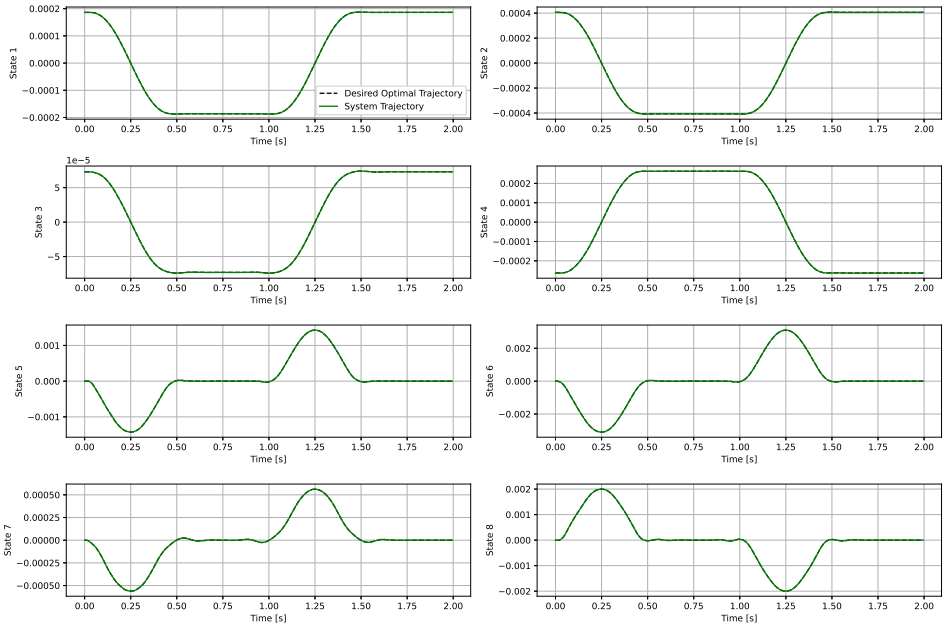

Figure 5.1: MPC - Tracking results for state variables

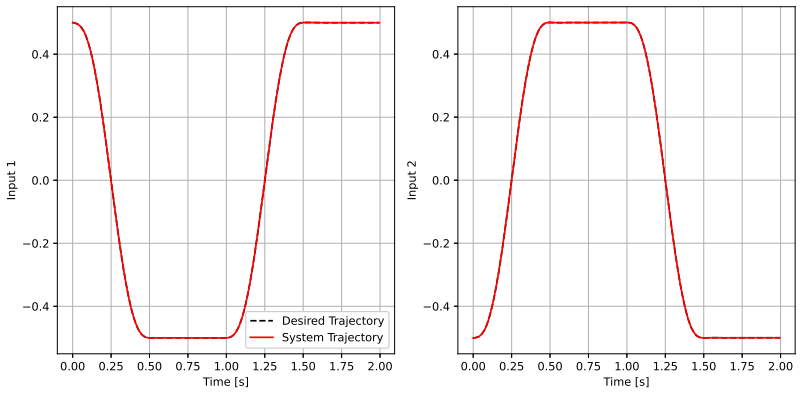

Figure 5.2: MPC - Control input trajectories

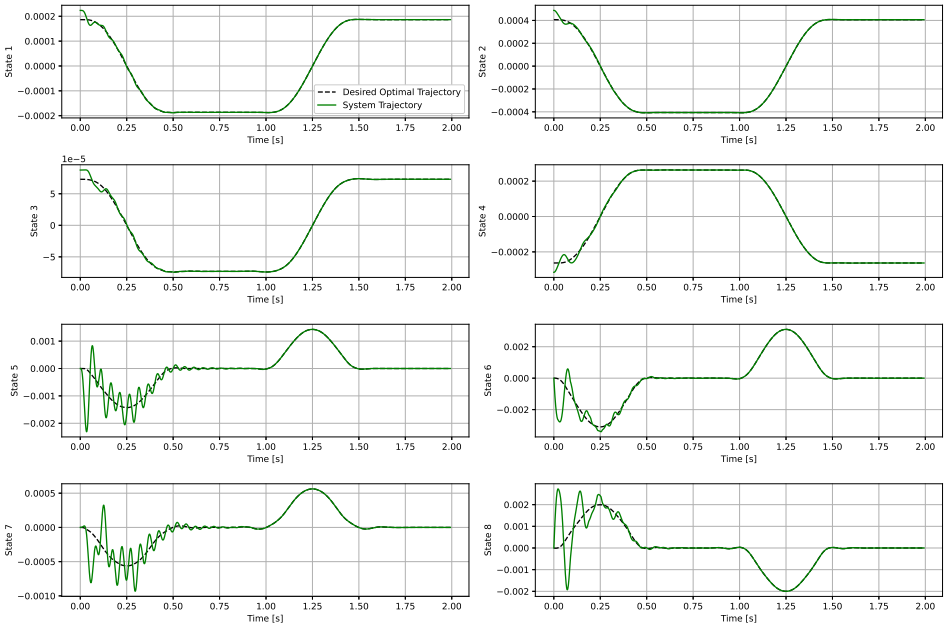

Figure 5.3: Perturbed MPC - Tracking results for state variables

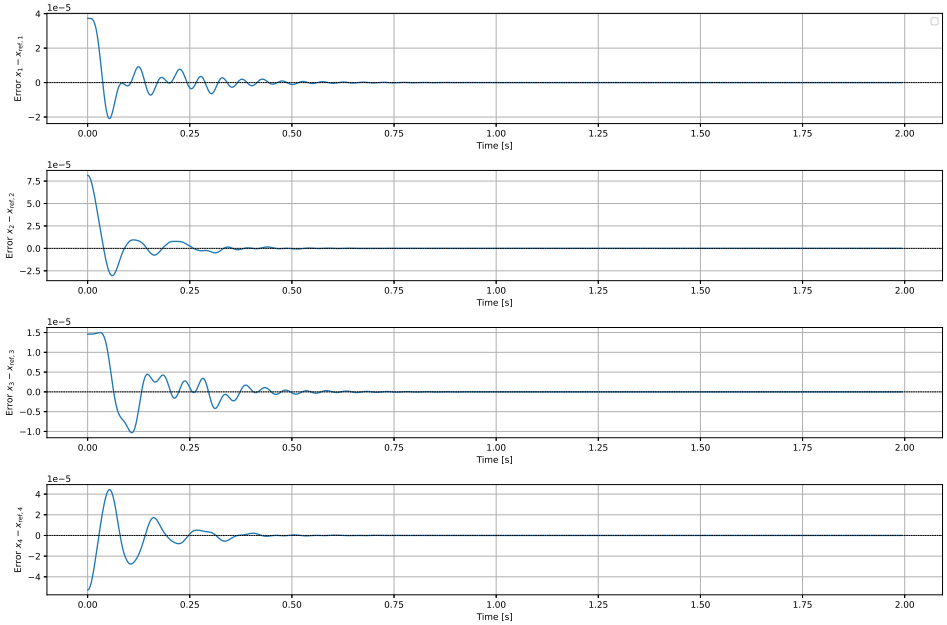

Figure 5.4: Perturbed MPC - Tracking error of the first four states

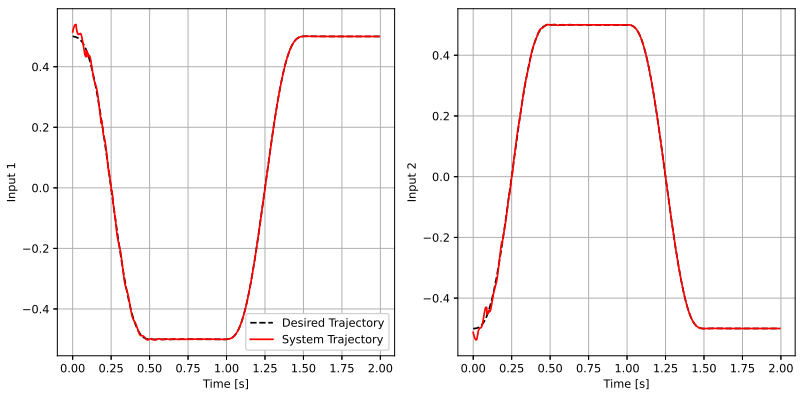

Figure 5.5: Perturbed MPC - Control input trajectories

Chapter 6

Animation

In order to visualize the behavior of the flexible surface system, we developed an animation exploiting the Matplotlib library [1]. Both the optimal and reference trajectories of the surface are represented, allowing for a direct comparison between the two.

The code was originally intended to animate the LQR trajectory tracking, though the modularity of its implementation may allow for the MPC trajectory tracking to be visualized as well. We have indeed defined a function that generates the animation starting from the inputs as the optimal state trajectory xx^*, the reference trajectory $\mathrm{xx}_{\mathrm{ref}}$ and the sampling time.

The position of the surface is represented by discrete points, which are combined with the fixed endpoints (i.e., static endpoints). The line connecting these points needs to be characterized properly, therefore the flexible surface is treated as a continuous line interpolated between four points. Specifically, our implementation uses cubic spline interpolation to create a smooth representation of the surface, which describes the physical behavior of a flexible material in a more natural way, at least with respect to a simple linear interpolation.

In Figure 6.1, the animation is basically the appearance of the surface in its initial conditions, while Figure 6.2 shows how the control system performs and visualizes the actual motion of the surface. In the case of our implementation, the motion is smooth and the surface follows closely the desired trajectory. In Figure 6.3, one can see the surface going back to the starting position after the motion in under 2 seconds.

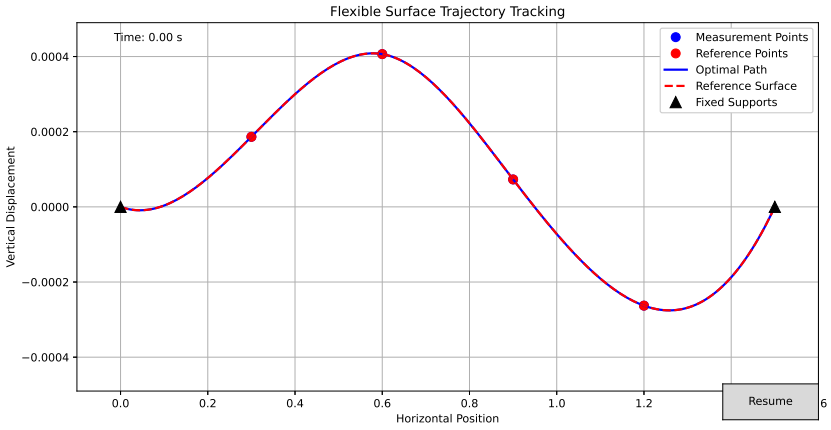

Figure 6.1: Animation of LQR tracking (t = 0s)

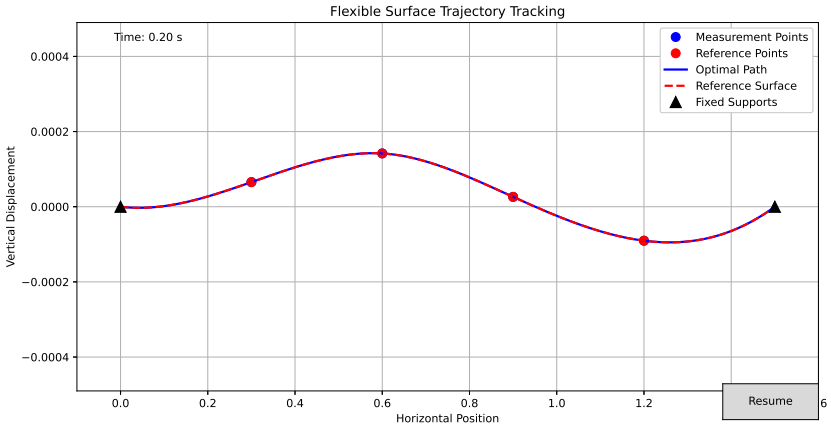

Figure 6.2: Animation of LQR tracking (t = 0.20s)

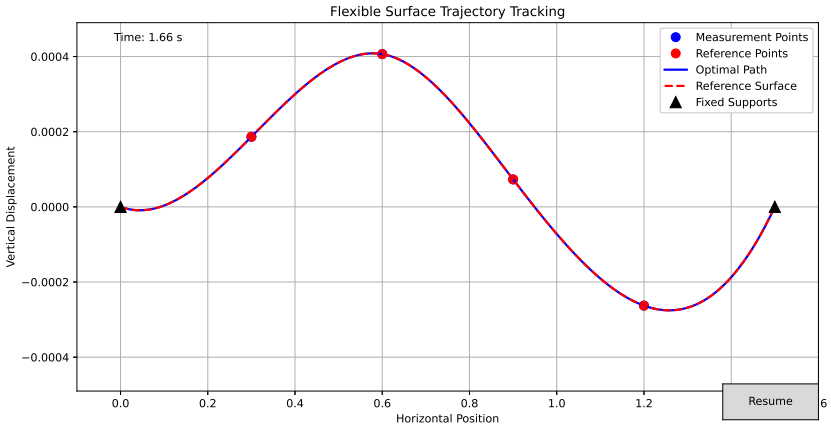

Figure 6.3: Animation of LQR tracking (t < 2s)

As described in Chapter 4, to further investigate the robustness of the LQR controller, we tested the control performance in case of perturbed initial conditions, which is a realistic scenario. Specifically, a small disturbance was introduced in the initial state vector, as shown in Figure 6.4.

The animation resulting from this simulation clearly illustrates the controller action: although the surface initially diverges from the reference due to the perturbation, it progressively converges to the reference trajectory until reaching again the configuration in Figure 6.7. By looking at Figure 6.5 and 6.6, one can observe that the surface is quite fast in compensating the initial error.

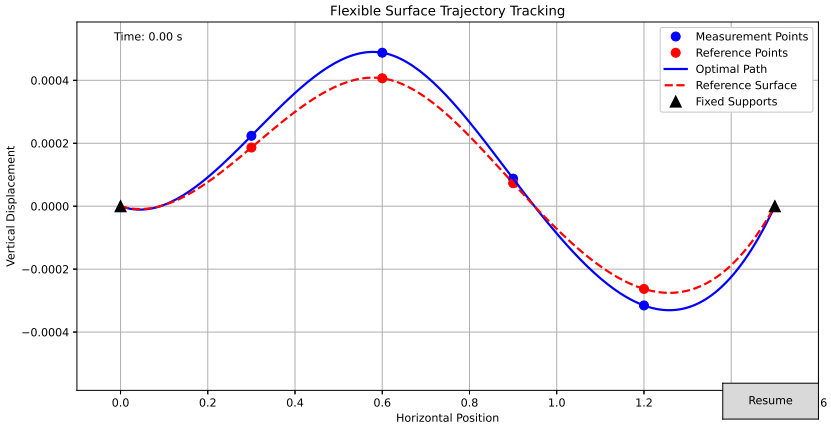

Figure 6.4: Animation of LQR tracking with perturbation (t = 0s)

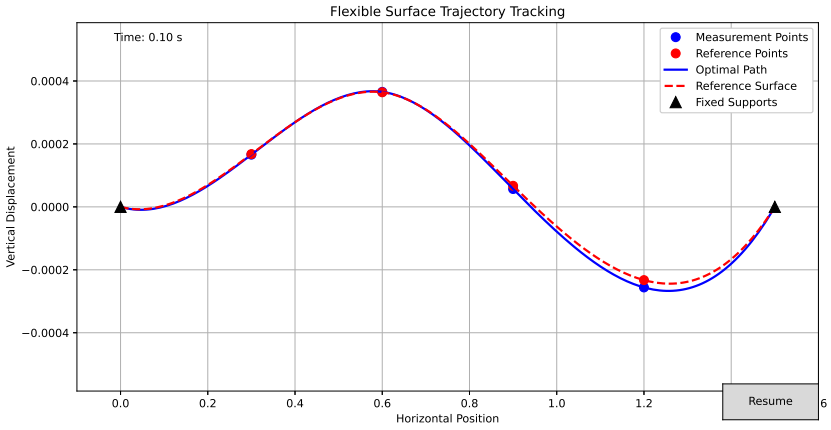

Figure 6.5: Animation of LQR tracking with perturbation (t = 0.10s)

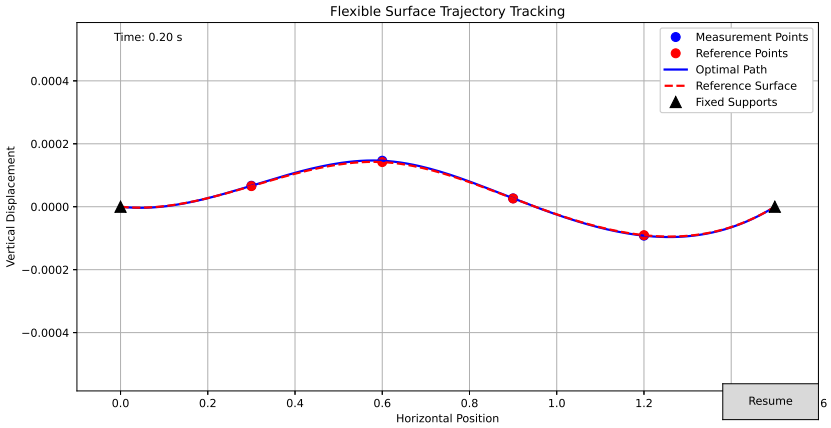

Figure 6.6: Animation of LQR tracking with perturbation (t = 0.20s)

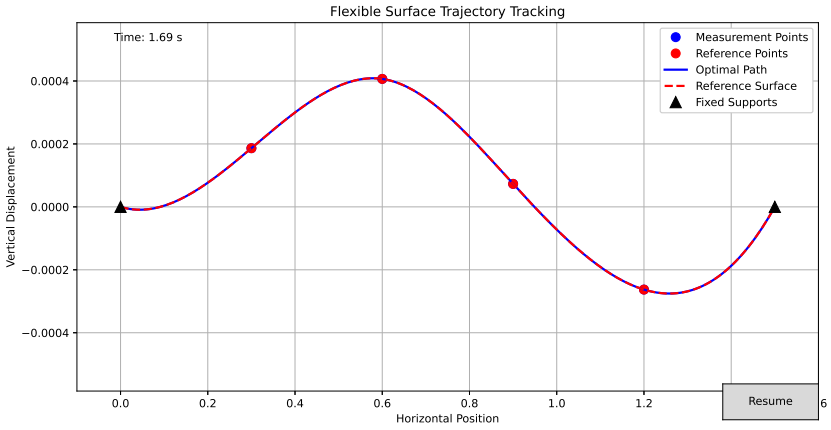

Figure 6.7: Animation of LQR tracking with perturbation (t $<$ 2s)

Conclusions

The main objective of this project was to implement the classical robust optimal control methods like Linear Quadratic Regulator (LQR) and Model Predictive Control (MPC) to achieve an optimal tracking of the desired optimal trajectory performed by the underactuated flexible surface in the Python environment. To sum up, the given problem has been tackled in multiple steps or tasks. In Task 0, the dynamics equations and the discretization of the system have been performed. In Task 1, a simple reference trajectory consisting of a 5^{th}-order polynomial connecting two equilibria points has been employed to later generate an optimal trajectory via the closed-loop version of the Newton's method. In Task 2, a more complex optimal trajectory using multiple transitions between equilibria has been generated. Using the Armijo step selection rule, the optimal task generation for the new reference trajectory has been terminated in only 4 iterations. In Task 3 and 4, the aforementioned control techniques like LQR and MPC have been utilized for optimal tracking of the desired trajectory. The results have shown a good tracking of the trajectories achieving close to zero tracking error for perturbed initial conditions using both LQR and MPC. Lastly, in Task 6, a visually comprehensive animation of the optimal control of the flexible surface has been prepared to showcase the performance of our code.

Despite the achievements of the major goals related to the project, there were plenty of obstacles that the group met along the implementation. The problems were faced in the initial stages, which resulted into incorrect equilibria derivations due to poor or unrealistic initial conditions choice. Some of the problems were encountered also in the final stages of the project, where the obtained graphs did not reflect the right behavior for the tasks. Nevertheless, the group constantly addressed these problems and managed to accomplish tangible results in the end with the important lessons learned on the way.

Bibliography

[1] J. D. Hunter. Animations using matplotlib. https://matplotlib.org/stable/users/explain/animations/animations.html, 2024. Accessed: May 2025.

[2] G. Notarstefano. Lecture notes and slides from optimal control course. https;//www.unibo.it/en/study/phd-professional-masters-specialisation-schools-and-other-programmes/course-unit-catalogue/course-unit/2024/468715, 2024. University of Bologna.

[3] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. OSQP: an operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 12(4):637–672, 2020.