UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Distributed Autonomus System

Course Project

Professor:

Giuseppe Notarstefano
Ivano Notarnicola

Students:

Andrea Alboni
Federico Calzoni
Emanuele Monsellato

Academic year 2024/2025

Abstract

A key challenge in the optimal control of robotic systems is the generation of optimal trajectories while considering the system's dynamics and constraints. This project focuses on the design and implementation of an optimal control law for a flexible robotic arm, modeled as a planar two-link robot with torque applied to the first joint.

The setup phase involves discretizing the robot's dynamics and formulating the discrete-time state-space equations.

The first task focuses on generating an optimal trajectory between two equilibrium points of the robotic arm. The equilibrium states are computed using a root-finding algorithm and a symmetric reference trajectory is defined between them.

The second task extends the first by introducing a smooth desired trajectory, for which a quasi-static trajectory is computed as an initial guess. This trajectory is refined through optimal control methods to minimize the cost function.

The third task involves trajectory tracking using Linear Quadratic Regulator (LQR) control. By linearizing the system dynamics around the optimal trajectory, a closed-loop controller is designed to track the reference trajectory while compensating for initial perturbations.

The fourth task leverages Model Predictive Control (MPC) to achieve trajectory tracking. The controller is developed to handle constraints dynamically, showcasing robustness to perturbed initial conditions.

Finally, the fifth task includes the visualization of the robotic arm's motion by animating the results of the LQR trajectory tracking task.

Contents

Chapter 1

Introduction

Distributed Autonomous Systems (DAS) are becoming increasingly relevant in a variety of application domains, ranging from environmental monitoring to autonomous transportation and surveillance. These systems rely on the coordination and cooperation of multiple agents such as mobile robots or sensors that operate autonomously, but are connected through a communication network. The decentralized nature of DAS enables scalability, fault tolerance, and adaptability in complex environments
This report presents the development and simulation of algorithms that enable such systems to perform complex tasks in a distributed manner. In particular, the work is structured around two main tasks:

- **Task 1:** *Multi-Robot Target Localization*
- **Task 2:** *Aggregative Optimization for Multi-Robot Systems*

In the first task addresses the problem where a fleet of robots aims to cooperatively estimate the positions of unknown targets based on noisy local measurements. This task emphasizes the use of distributed consensus optimization techniques, with a focus on the Gradient Tracking algorithm, which allows each robot to iteratively refine its estimate by combining local information with data received from neighboring agents.
The second task explores a distributed optimization framework where each robot aims to minimize a local objective function that depends not only on its own state but also on a global aggregative quantity, typically the team's barycenter. The primary goal is to design and implement a distributed control algorithm that enables the robots to stay close to private targets while maintaining the cohesion of the fleet, despite communication constraints. This task employs the Aggregative Tracking algorithm to achieve coordination and consensus through purely local interactions, relying solely on local communications. The Aggregative Tracking algorithm is employed to solve this problem in a distributed fashion, and the strategy is further validated through a ROS 2 implementation.

The purpose of this project is to apply theoretical concepts learned during the DAS course to practical scenarios through the implementation of distributed algorithms, simulation of multi-agent systems, with a focus on convergence properties and performance evaluation.

Chapter 2

Multi-Robot Target Localization

Problem Description

Multi-robot target localization involves a team of robots collaboratively estimating the positions of one or more targets based on noisy sensor measurements. In this scenario a team of robots, each equipped with sensors for detecting targets within a limited sensing range, and these measurements are corrupted by noise. The goal is to design and implement a distributed strategy that enables the robots to cooperatively locate the targets and improve the accuracy of the estimation compared to individual measurements.

For visualization of the context, in general, we have the following.

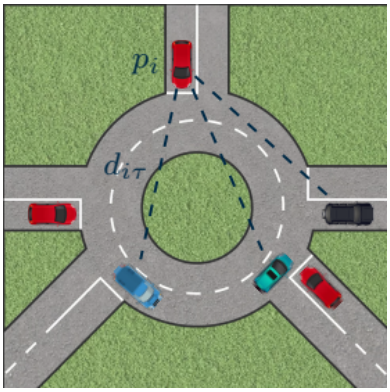


Figure 2.1: Example of multi-robot target localization problem. A team of robots observing multiple targets. Noisy measurements are represented with dashed lines.

where we have a $N \in N$ robots that estimate the positions of multiple unknown targets in a cooperative mode, using only local, noisy distance

measurements. Each robot is assumed to be aware only of its own position and of the distances (corrupted by noise) to nearby targets, and can exchange information solely with its neighbors in a communication graph.

To tackle this challenge, the problem is approached in two stages:

- **Task 1.1 - Distributed Consensus Optimization**: this focuses on the implementation and analysis of a distributed consensus optimization algorithm Gradient Tracking which allows robots to collaboratively minimize a global objective function.
- **Task 1.2 - Cooperative Multi-Robot Target Localization**: this builds an algorithm to solve the specific localization problem, defining suitable cost functions that encode the discrepancy between the measured distances and the estimated positions of the targets.

The proposed solution must be robust, scalable, and efficient, capable of handling different communication topologies and noisy sensor data, while guaranteeing convergence of the estimates across the network.

Code Structure Note

For the sake of modularity and clarity, the implementation has been organized into multiple files, each responsible for a specific aspect of the project. The code is organized as follows:

- `utils_graph.py`: File that contains utility functions for generating and handling different graph topologies (e.g., cycle, path, star,erdos_renyi) and computing Metropolis-Hastings weights.
- `utils_world_generation.py`: File for creating the simulated environment, including the generation of agents and targets within a bounded space.
- `gradient_tracking.py`: File to implementation of the Gradient Tracking algorithm used in both tasks, with same different.
- `cost_functions.py`: Definition of the local and global cost functions used in the optimization problem.
- `utils_visualization.py`: File dedicated to plotting the evolution of the cost function and gradients, and optionally animating robot movements.
- `main.py`: Entry point of the simulations, used to set parameters and run both tasks.

This modular structure enables the reuse of core components and makes the transition from consensus optimization (Task 1.1) to cooperative target localization (Task 1.2) seamless.

2.1 Distributed Consensus Optimization

2.1.1 Problem Statement

In this task, we address a distributed consensus optimization problem arising in multi-agent systems. The objective is to solve an unconstrained optimization problem of the form:

$$\min_{z \in \mathbb{R}^d} \sum_{i=1}^{N} \ell_i(z), \tag{2.1}$$

where:

- $z \in \mathbb{R}^d$ is the global decision variable to be optimized,
- $N \in \mathbb{N}$ is the number of agents (robots),
- $\ell_i : \mathbb{R}^d \to \mathbb{R}$ is a local objective function known only to agent i.

Each agent i has access only to its own cost function $\ell_i(z)$ and can communicate with a limited set of neighboring agents defined by a communication graph $\mathcal{G} = (\mathcal{I}, E)$, where $\mathcal{I} = \{1, \ldots, N\}$ is the set of nodes and $E \subseteq \mathcal{I} \times \mathcal{I}$ is the set of edges.
The goal is to design a distributed algorithm that allows all agents to cooperatively find a minimizer of the global cost function by exchanging information only with their immediate neighbors in the graph $\mathcal{G}$. To evaluate the performance of the distributed consensus optimization algorithm, are implemented a different simulation. In particular, consider different types of graph structures such as cycle, path, star, erdos_renyi graphs to model the communication network among the agents.
For each topology, the communication weights are defined using the *Metropolis Hastings rule*, ensuring the resulting weight matrix is symmetric and doubly stochastic, which is essential for convergence guarantees.
In this context, consensus optimization techniques are crucial as they enable a collective decision-making process, ensuring that all agents converge to a common optimal solution through local computations and communications.

2.1.2 Algorithm Implementation

ADD IMPLEMENTATION

2.2 Cooperative Multi-Robot Target Localization

In the second part of the task, leveraging the work done in Task 1.1, the goal is to implement the *Gradient Tracking* algorithm to enable a fleet of

robots to cooperatively localize multiple targets in the environment.
The main objective is to estimate the positions of multiple targets by using noisy distance measurements from the robots to the targets, while ensuring consensus among the robots on the final estimates.

2.2.1 Problem Statement

The scenario involves a set of N robots, each with an unknown position $p_i \in \mathbb{R}^d$, and N_T static targets whose positions are to be estimated. Each robot obtains noisy measurements of its distances $d_{i\tau} \in \mathbb{R}_{\geq 0}$ to each target $\tau = 1, \ldots, N_T$. The robots aim to collectively minimize a local cost function defined as:

$$f_i(z) = \sum_{\tau=1}^{N_T} \left(d_{i\tau}^2 - \|z_\tau - p_i\|^2\right)^2, \tag{2.2}$$

where $z = \mathrm{col}(z_1, \ldots, z_{N_T}) \in \mathbb{R}^{dN_T}$ is the concatenated vector of target position estimates $z_\tau \in \mathbb{R}^d$.

By implementing the Gradient Tracking algorithm, the robots iteratively update their local estimates while exchanging information with their neighbors, achieving consensus on the target locations.
So the algorithm minimize the local cost functions cooperatively, allowing the robots to share information and iteratively update their estimates.

2.2.2 Algorithm Implementation

ADD IMPLEMENTATION

2.3 Code Implementation

In this section presents the implementation details of the codebase developed to solve Task 1.1 and Task 1.2.

In particular, we distinguish the core modules, which are shared across the two tasks (e.g.utils_graph.py, utils_world_generation.py, ...), and specific function that define the optimization problem and implement the corresponding distributed algorithm (Distributed Gradient Descent for Task 1.1 and Gradient Tracking for Task 1.2).

What follows is an overview of the structure, key components, and how the codebase supports the objectives of each task.

2.3.1 Common Utility Function and Scripts

utils_graph.py

This module provides utility functions for generating and processing graphs

Functions:

- **`ensure_connected_graph(G)`**:
  Ensures that the input graph G is connected adding an Egees between if it disconnected;

- **`metropolis_hastings_weights(G)`**:
  Computes the Metropolis Hastings weight matrix A for a given graph G. If A is:
  - Not diagonal $A_{ij} = \frac{1}{(1+max(d_i,d_j)}$ if nodes i and j are connected;
  - It is diagonal $A_{ii} = 1 - \sum_{j \neq i} A_{ij}$

- **`generate_graph(num_agents, type, p_er=0.5)`**:
  `path`, `cycle`, `star`, or `erdos_renyi`. If the generated Erdos-Renyi graph is disconnected, minimal connections are added to make it connected. Returns the graph object G, its adjacency matrix Adj, and the Metropolis-Hastings weight matrix A.

utils_world_generation.py

This module provides helper functions to generate agents and targets in a bounded d-dimensional environment. It also simulates noisy distance measurements within a given field of view (FOV)
Functions:

- **`is_in_fov(agent_pos, target_pos, radius_fov)`**
  Determines whether the target is within the agent's field of view radius.

- **`spawn_agent_near_target(target, existing_agents, existing_targets, world_size, d, radius_fov)`**
  Randomly spawns an agent within the FOV of a given target, avoiding overlap with existing agents and targets.

- **`spawn_candidate(existing_agents, existing_targets, world_size, d)`**
  Generates a random position in the world that does not overlap with any existing agent or target.

- **`generate_agents_and_targets(num_targets, ratio_at, world_size, d, radius_fov)`**
  Creates a set of targets and agents such that each target is seen by at least 3 agents within the FOV. Additional agents are added randomly to achieve a target-agent ratio of `ratio_at`. The positions are normalized with respect to the world size.

- `get_distances(agents, targets, noise_level, bias_param, radius_fov, world_size)`
  Computes both exact and noisy distances between each agent and target, limited to those within the FOV. The noise includes a uniform bias and Gaussian noise, scaled by the world size.

gradient_tracking_method.py

This module implements the **Gradient Tracking** algorithm for decentralized optimization, applicable to both localization and target estimation tasks. Each agent maintains and updates local estimates and gradients through message passing over a communication graph.

- `gradient_tracking_method(agents, targets, noisy_distances, adj, A, local_cost_function, alpha, max_iters)`:
  Executes the gradient tracking algorithm **Algorithm steps:**

  1. **Initialization:**
     The variable z_0 (local estimate) is initialized uniformly at random in $[0, 1]$. Gradients s_0 are initialized using the local cost function.
  2. **Iterative updates:** For each iteration k:
     - *Estimate update:* z_i^{k+1}
     - *Gradient tracking update:* s_i^{k+1}
  3. **Monitoring quantities:**
     - *Cost:* Local cost contributions summed across all agents.
     - *Gradient norm:* Norm of the global gradient estimate at each iteration.
     - *Error:* Euclidean distance between the agent's estimate z_{ij}^k and the true target j.

  Return values:

  - z: Local estimates of agents over time.
  - *cost*: Global cost function at each iteration.
  - *norm_grad_cost*: Norm of the aggregated gradients.
  - *norm_error*: Estimation error norms for each agent-target pair over time.

This method is suitable for both localization tasks (estimating an agent's position using anchors) and distributed target tracking (estimating moving or static targets via multiple agents).

utils_visualization.py

This file contains functions to visualize the graph topology, the agents' and targets' positions in the world, and the evolution of decentralized estimation algorithms over time.

- **`visualize_graph(G)`**
  Draws the structure of the input graph G using `networkx`.

- **`plot_gradient_tracking_results(z, cost, norm_grad_cost, agents, targets, norm_error)`**
  Produces a set of plots related to the gradient tracking algorithm:

  – Evolution of the cost function.

  – Norm of the gradient over iterations.

  – Error norms for each target, shown across all agents.

  All plots are rendered using logarithmic scale for better visualization of convergence trends.

- **`visualize_world(agents, targets, world_size, d)`**
  Visualizes the spatial configuration of agents and targets in 1D, 2D, or 3D space. The entities are rescaled by the world size and displayed with:

  – Blue circles for agents.

  – Red crosses for targets.

  The function automatically adapts the axes and labels according to the dimensionality d.

- **`animate_world_evolution(agents, targets, z_history, type, world_size, d, speed=50)`**
  This function are usefull for the evolution of agent estimations z over time using `matplotlib`'s `FuncAnimation`. It supports 1D, 2D, and 3D visualizations, and:

  – Shows agents and targets as static reference points.

  – Dynamically updates estimation points in green.

  – Allows to visually inspect convergence of estimations to target locations.

  The graph topology type is included in the plot title for clarity.

2.3.2 Adaptation of the Method to Different Tasks

cost_functions.py

The script defines two separate functions, in particular define two separated local cost functions, for each tasks, used during optimizatio.

- `local_cost_function_task1(z, p_i, distances_i = None)`
  The agent seeks to estimate the target positions based only on relative displacement, without depending on any form of measurement. The cost is formulated as a weighted quadratic penalty on the distance between the agent's position p_i and each estimated target position z_j. A positive definite matrix Q scales the contribution of each coordinate, and an affine term b ensures numerical stability:

  $$\text{cost} = \sum_{j=1}^{m} (p_i - z_j)^\top Q (p_i - z_j) + b.$$

  The corresponding gradient is linear, and the same for all targets, as it does not depend on any observation or measurement.

- `local_cost_function_task2(z, p_i, distances_i)`
  The local cost depends on the difference between the estimated distances (based on current position estimates) and the actual noisy distance measurements received by each agent. The cost penalizes the squared error between measured and estimated squared distances:

  $$\text{cost} = \sum_{j=1}^{m} \left(\|z_j - p_i\|^2 - d_{ij}^2 \right)^2.$$

  The gradient, unlike Task 1, is computed separately for each target and depends nonlinearly on both the estimated position z_j and the measured distance d_{ij}. This makes it more sensitive to measurement noise, but also more informative when such data is available.

Practical Considerations. Both functions return the local cost and its gradient. In Task 2, special care is taken to handle missing or undefined distance values using `np.isnan`, setting the corresponding gradient to zero when necessary. This ensures stability and robustness to missing data.

main.py

The `main.py` script serves as the central entry point for the simulation, coordinating data generation, task execution, and result visualization. Its structure is modular and task-agnostic, making it adaptable for different estimation problems.

- **Parameter Setup**
  The simulation settings are defined in the `PARAMETERS` dictionary, including number of targets, agent-to-target ratio, dimension of the environment, type of communication graph, noise level, and field of view radius.

- **Task Selection and Configuration**
  The script supports different tasks through the `task_settings` dictionary, which maps each task identifier to its specific cost function and corresponding step size (α). The `task_to_run` list allows multiple tasks to be executed sequentially in a single run.

- **Environment and Graph Initialization**
  Agents and targets are randomly generated using `generate_agents_and_targets()`, and their connections are defined through `generate_graph()` according to the specified topology (e.g., cycle). Distance measurements are simulated by `get_distances()`, which produces both real and noisy measurements, subject to field-of-view constraints.

- **Task Execution Loop**
  For each selected task:

  1. The corresponding cost function and step size are selected.
  2. The `gradient_tracking_method()` is invoked to perform distributed optimization and return the full history of target estimates, costs, gradients, and errors.
  3. The resulting performance is visualized using:
     - `visualize_world()` for initial layout of agents and targets.
     - `plot_gradient_tracking_results()` for convergence diagnostics.
     - `animate_world_evolution()` to display the temporal evolution of estimates.

Chapter 3

Aggregative Optimization for Multi-Robot Systems

Chapter 4

Problem setup

4.1 Overview of the Flexible Robotic Arm Model

This work addresses the optimal control of a planar two-link robotic manipulator characterized by its underactuated nature. The system dynamics evolve in a two-dimensional plane and consist of two rigid links connected by revolute joints, with actuation provided only at the first joint through an applied torque u.

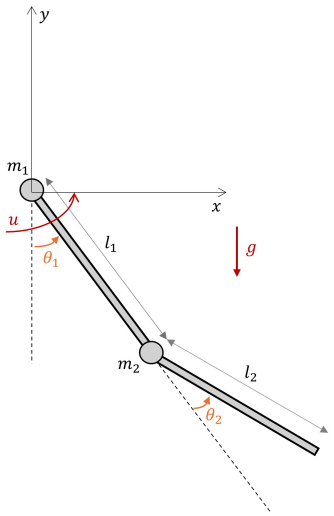


Figure 4.1: Model of the flexible arm.

The system state is defined as $\mathbf{x} = [\dot{\theta}_1, \dot{\theta}_2, \theta_1, \theta_2]^\top \in \mathbb{R}^4$, where θ_1 denotes the angle of the first link with respect to the vertical axis, θ_2 represents the relative angle between the two links, and $\dot{\theta}_1$, $\dot{\theta}_2$ are their respective angular velocities. The control input $u \in \mathbb{R}$ acts as a torque at the first joint only, making the system underactuated.

The equations of motion are described by:

$$\mathbf{M}(\theta_1, \theta_2) \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{bmatrix} + \mathbf{C}(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) + \mathbf{F} \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} + \mathbf{G}(\theta_1, \theta_2) = \begin{bmatrix} u \\ 0 \end{bmatrix} \tag{4.1}$$

where $\mathbf{M}(\theta_1, \theta_2) \in \mathbb{R}^{2\times 2}$ is the inertia matrix, $\mathbf{C}(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) \in \mathbb{R}^2$ accounts for Coriolis and centrifugal forces, $\mathbf{G}(\theta_1, \theta_2) \in \mathbb{R}^2$ describes gravitational effects, and $\mathbf{F} \in \mathbb{R}^{2\times 2}$ represents viscous friction through a diagonal matrix. The dynamics parameters used in this analysis are presented in Table 4.1.

Table 4.1: System Dynamics Parameters

Parameter	Value
M_1	2.0 [kg]
M_2	2.0 [kg]
L_1	1.5 [m]
L_2	1.5 [m]
R_1	0.75 [m]
R_2	0.75 [m]
I_1	1.5 [kg·m^2]
I_2	1.5 [kg·m^2]
g	9.81 [m/s^2]
F_1	0.1 [N·m·s/rad]
F_2	0.1 [N·m·s/rad]

4.2 Discretization of Dynamics

4.2.1 Discretization Using Euler Method

The continuous-time dynamics described in Equation 5.1 are discretized using the Euler method for implementation in a digital control system. The time derivative is approximated as:

$$\dot{x} \approx \frac{x_{k+1} - x_k}{\Delta t}$$

where Δt is the sampling time. Rearranging this yields the discrete-time update:

$$x_{k+1} = x_k + \Delta t \cdot dx$$

where dx is computed as:

$$dx = Ax_k + Bu_k + c$$

- A is the system matrix derived from the inertia, damping, and coupling effects, with terms dependent only on $\mathbf{x}$,
- B is the input matrix, with terms dependent only on $\mathbf{u}$,
- c encapsulates non-linear effects such as Coriolis, centrifugal, and gravitational forces.

The explicit equation of dx is the following:

$$\begin{bmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} = \begin{bmatrix} -\mathbf{M}^{-1}\mathbf{F} & 0_{2\times 2} \\ I_{2\times 2} & 0_{2\times 2} \end{bmatrix} \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \theta_1 \\ \theta_2 \end{bmatrix} + \begin{bmatrix} \mathbf{M}^{-1} & 0_{2\times 2} \\ 0_{2\times 2} & 0_{2\times 2} \end{bmatrix} \begin{bmatrix} \tau_1 \\ \tau_2 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -\mathbf{M}^{-1}(\mathbf{C}+\mathbf{G}) \\ 0_{2\times 1} \end{bmatrix} \tag{4.2}$$

Since the system is underactuated, τ_2 is always zero.

Chapter 5

Trajectory generation (I)

5.1 Equilibrium Points

To determine the equilibrium points of the flexible robotic arm, the system is analyzed under the condition where all velocities and accelerations are zero:

$$\dot{\theta}_1 = \dot{\theta}_2 = \ddot{\theta}_1 = \ddot{\theta}_2 = 0.$$

Substituting these conditions into the system dynamics simplifies the equations to:

$$\mathbf{G}(\theta_1, \theta_2) = \begin{bmatrix} \tau \\ 0 \end{bmatrix}$$

This equation implies that the equilibrium points are the configurations (θ_1, θ_2) where the torques due to gravity are balanced by the applied torque τ.

To solve the equilibrium points numerically, Newton's method is employed to find the roots of the non-linear equations:

$$\mathbf{G}(\theta_1, \theta_2) - \begin{bmatrix} \tau \\ 0 \end{bmatrix} = 0$$

This approach iteratively refines an initial estimate of the equilibrium state to converge on a solution. The process begins with an initial guess:

$$z_0 = \begin{bmatrix} \theta_{1,0} \\ \theta_{2,0} \\ \tau_0 \end{bmatrix},$$

and updates the estimate using the formula:

$$z_{k+1} = z_k - (\nabla r(z_k))^{-1} r(z_k),$$

The algorithm should steer the decision vector towards the desired equilibrium point z_{eq}, which satisfies:

$$z_{\text{eq}} = \begin{bmatrix} \theta_{1,\text{eq}} \\ \theta_{2,\text{eq}} \\ \tau_{\text{eq}} \end{bmatrix}.$$

5.2 Reference Curve

In this task the reference curve is a step function. During the initial phase of the simulation, the system remains at the first equilibrium, after which it instantaneously transitions to the second equilibrium.

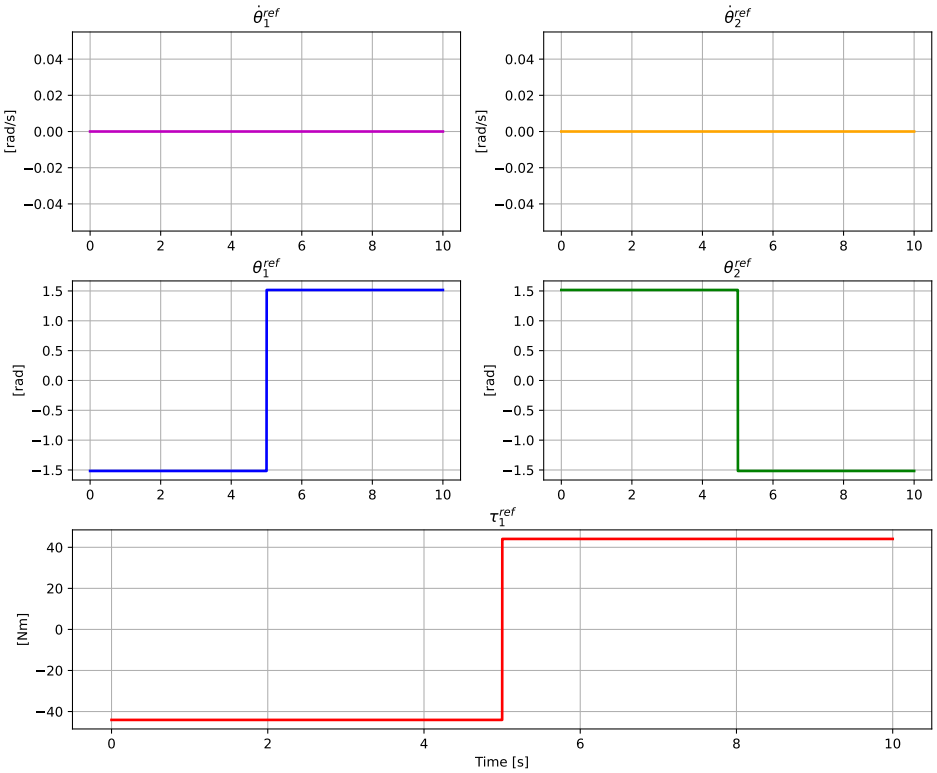


Figure 5.1: Step reference curve.

5.3 Cost Function

The cost function is quadratic and can be written as follows:

$$J(x,u) = \frac{1}{2}\sum_{t=0}^{T-1}\left[(x_t - x_{\text{ref},t})^T Q_t (x_t - x_{\text{ref},t}) + (u_t - u_{\text{ref},t})^T R_t (u_t - u_{\text{ref},t})\right] + \frac{1}{2}(x_T - x_{\text{ref},T})^T Q_T (x_T - x_{\text{ref},T}) \tag{5.1}$$

The cost matrices are time-varying to better capture the desired behavior of the system. This approach ensures precision near the equilibrium points and allows flexibility during transitions. In particular, higher weights are assigned to emphasize accuracy at equilibrium, while lower weights are used during transitions to allow smoother adjustments. This variation is implemented in a smooth manner to ensure proper and stable control. For the purposes of Task 1, the cost matrices behave as follows:

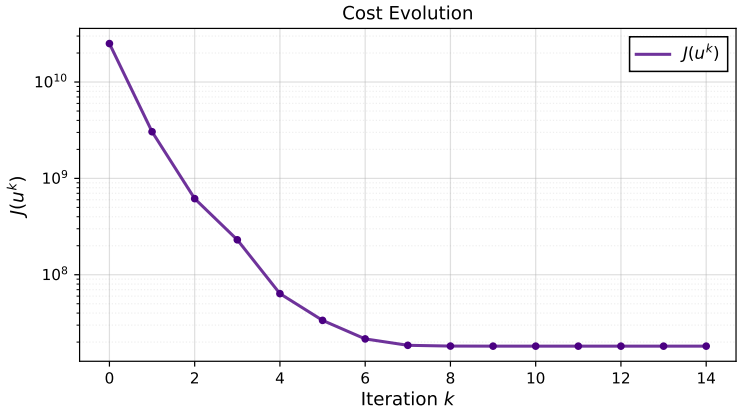


Figure 5.2: Evolution of cost matrices.

5.4 Optimal Transition: Newton's-like Algorithm

To achieve an optimal transition between equilibrium points, the Newton's-like algorithm for optimal control is applied in a closed-loop configuration. The goal is to minimize a cost function that penalizes deviations from the desired trajectory and excessive control effort. The optimization problem states as follows:

$$min_{\mathbf{x},\mathbf{u}} \quad J(\mathbf{x},\mathbf{u})$$
$$s.t. \quad x_{t+1} = f(x_t, u_t), \quad t = 0, 1, ..., T-1$$

The algorithm follows a two-step iterative procedure that involves solving the co-state equations, computing the necessary feedback gains, and updating the state-input trajectory until convergence is achieved.

In the first step, the algorithm computes the descent direction, which is essential to reduce the cost function. This involves evaluating the following terms:

$$\nabla_1 f_t(x_t^t, u_t^t) \quad \nabla_2 f_t(x_t^t, u_t^t) \quad \nabla_1 \ell_t(x_t^t, u_t^t) \quad \nabla_2 \ell_t(x_t^t, u_t^t) \quad \nabla \ell_T(x_T^T)$$

Once these gradients are evaluated, the co-state equations are solved backward in time, starting from $\nabla \ell_T(x_T^T)$. The goal is to calculate the matrices Q_t^k, R_t^k, S_t^k, and Q_T^k, which are essential to obtain the feedback control law. The feedback gains K_t^k and σ_t^k are then computed for all time steps from $t = 0$ to $t = T - 1$ by solving the optimal control problem in the backward direction.

The optimal solution of the problem reads

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

where

$$\begin{aligned} K_t^* &= -(R_t + B_t^\top P_{t+1} B_t)^{-1}(S_t + B_t^\top P_{t+1} A_t), \\ \sigma_t^* &= -(R_t + B_t^\top P_{t+1} B_t)^{-1}(r_t + B_t^\top p_{t+1} + B_t^\top P_{t+1} c_t), \\ p_t &= q_t + A_t^\top p_{t+1} + A_t^\top P_{t+1} c_t - K_t^{*\top} \left(R_t + B_t^\top P_{t+1} B_t\right) \sigma_t^*, \\ P_t &= Q_t + A_t^\top P_{t+1} A_t - K_t^{*\top} \left(R_t + B_t^\top P_{t+1} B_t\right) K_t^*, \end{aligned}$$

with $p_T = q_T$ and $P_T = Q_T$.

The second step of the algorithm involves using the computed feedback gains to update the state and control trajectories. This is done by applying the feedback control law:

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

Additionally, forward integration of the system dynamics is performed for all $t = 0, \dots, T-1$ to obtain the new trajectory. This is done by solving the system of equations:

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

To ensure convergence and prevent overly large updates that may destabilize the trajectory, a step-size selection rule is used, such as the Armijo rule. This rule adjusts the step size based on the reduction of the cost function at each iteration, ensuring that each new trajectory results in a decrease in the overall cost.

5.5 Plots of Generated Optimal Trajectory (I)

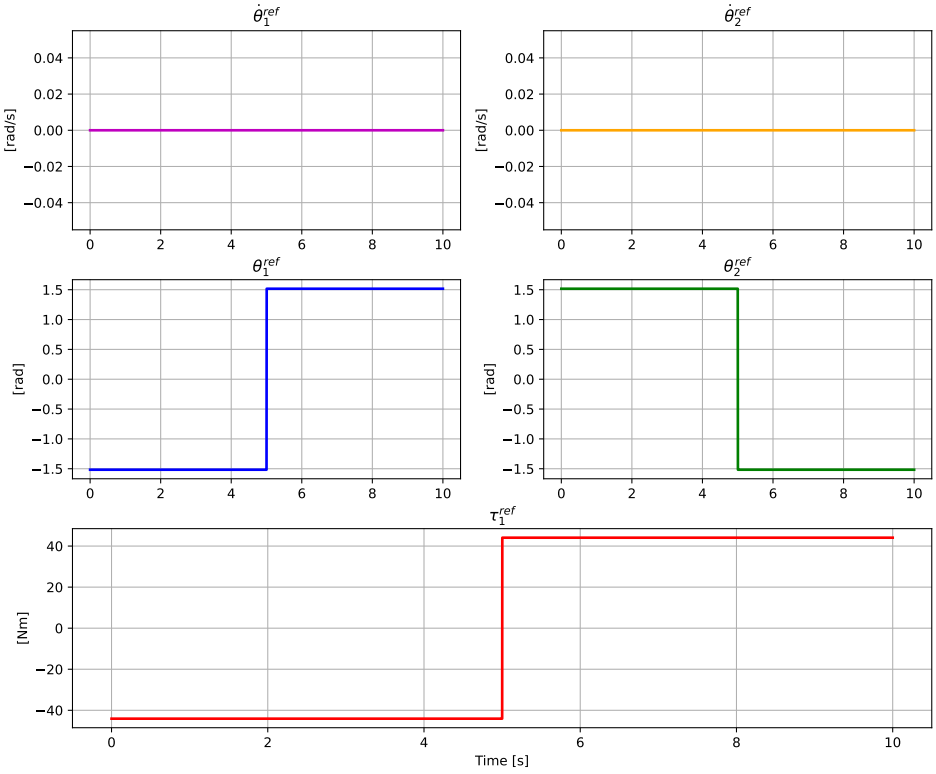


Figure 5.3: Generated optimal trajectory given a step reference.

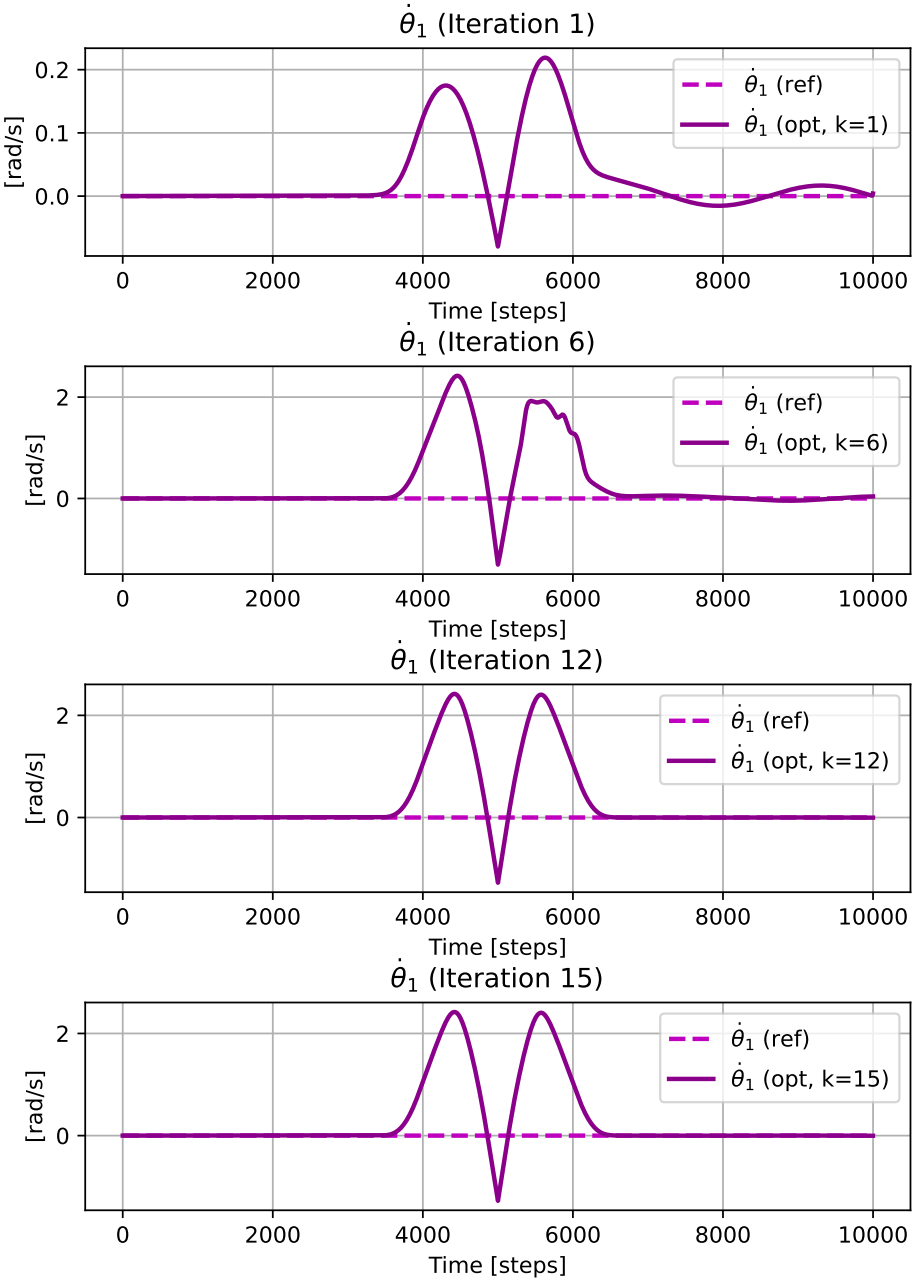


Figure 5.4: Evolution of $d\theta_1$.

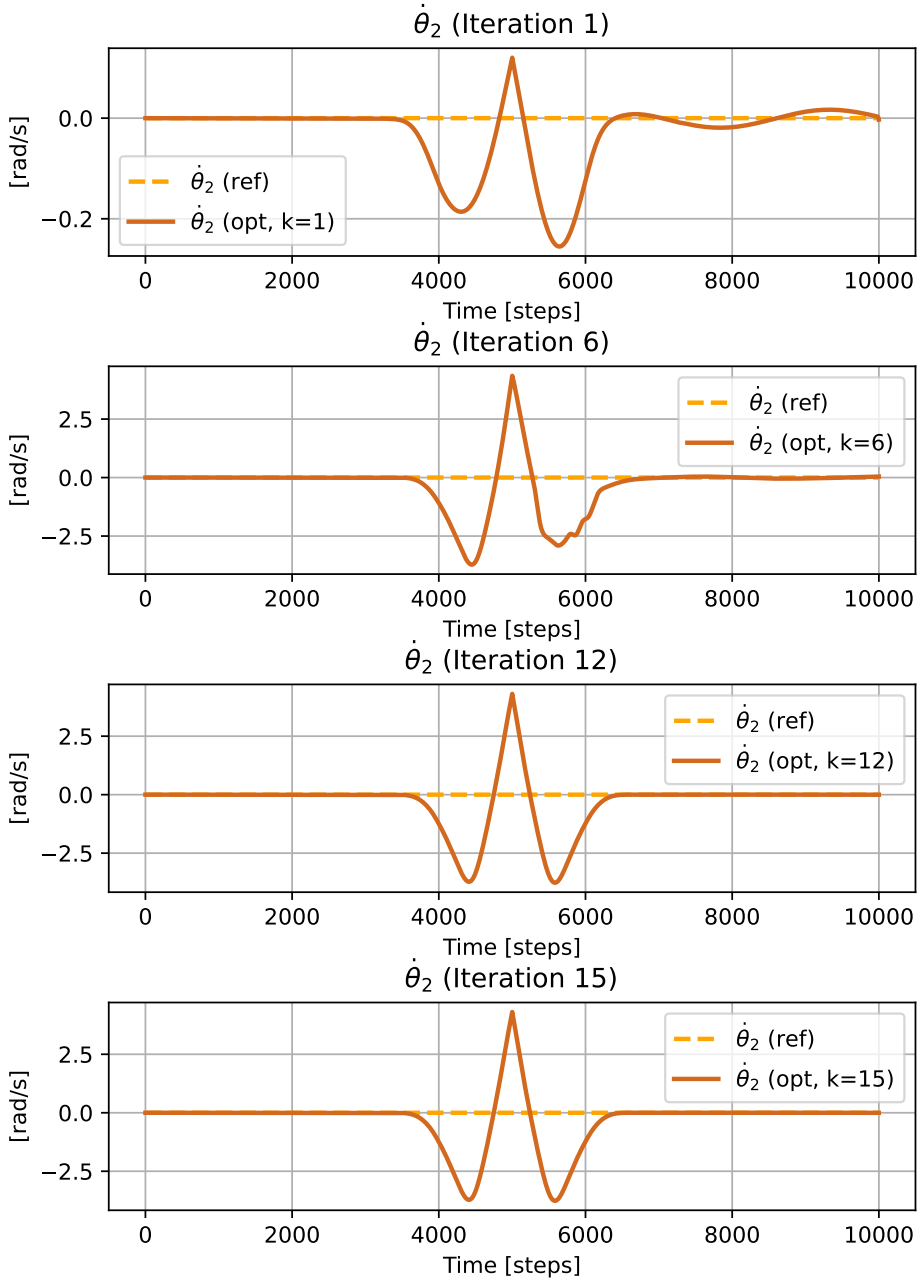


Figure 5.5: Evolution of $d\theta_2$.

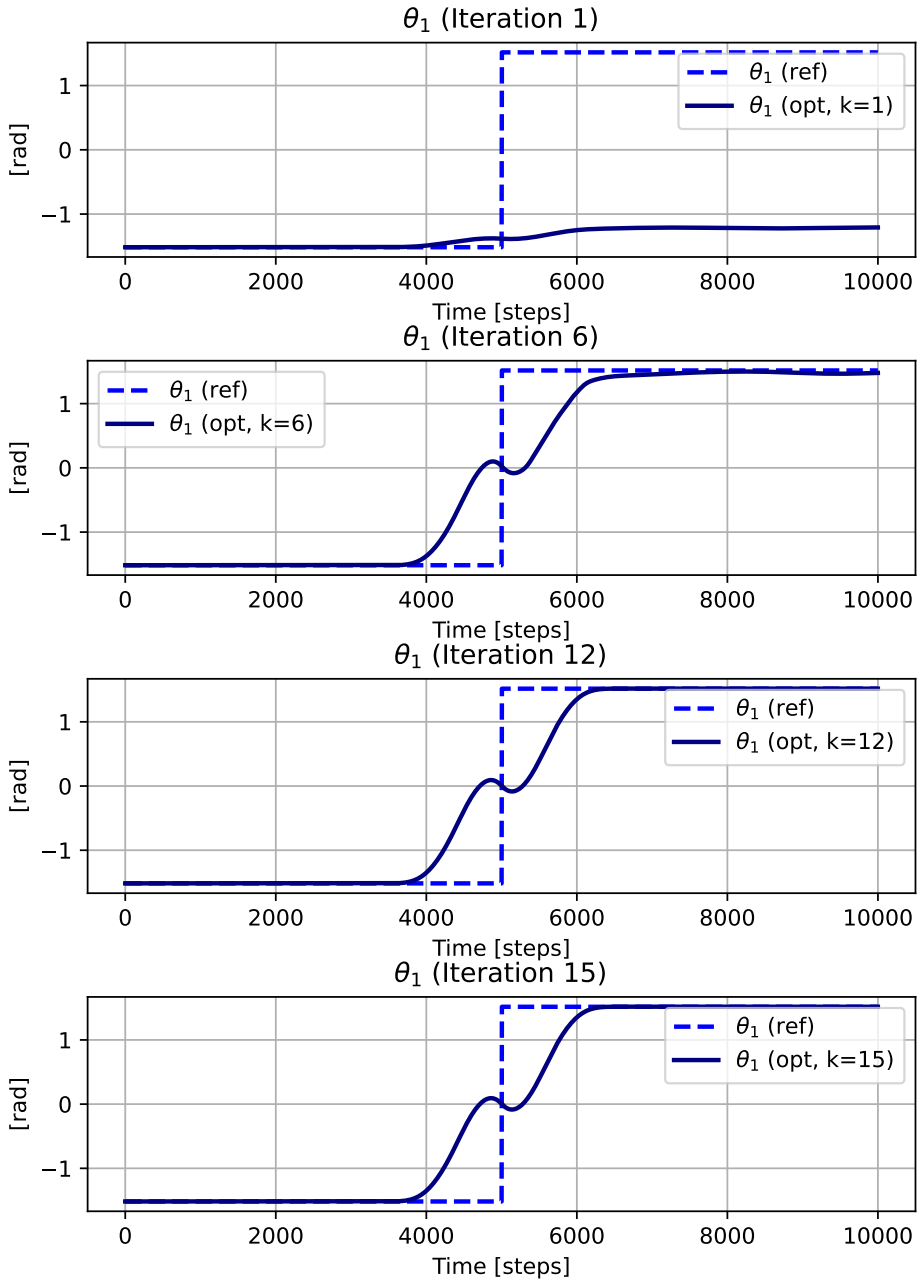


Figure 5.6: Evolution of θ_1.

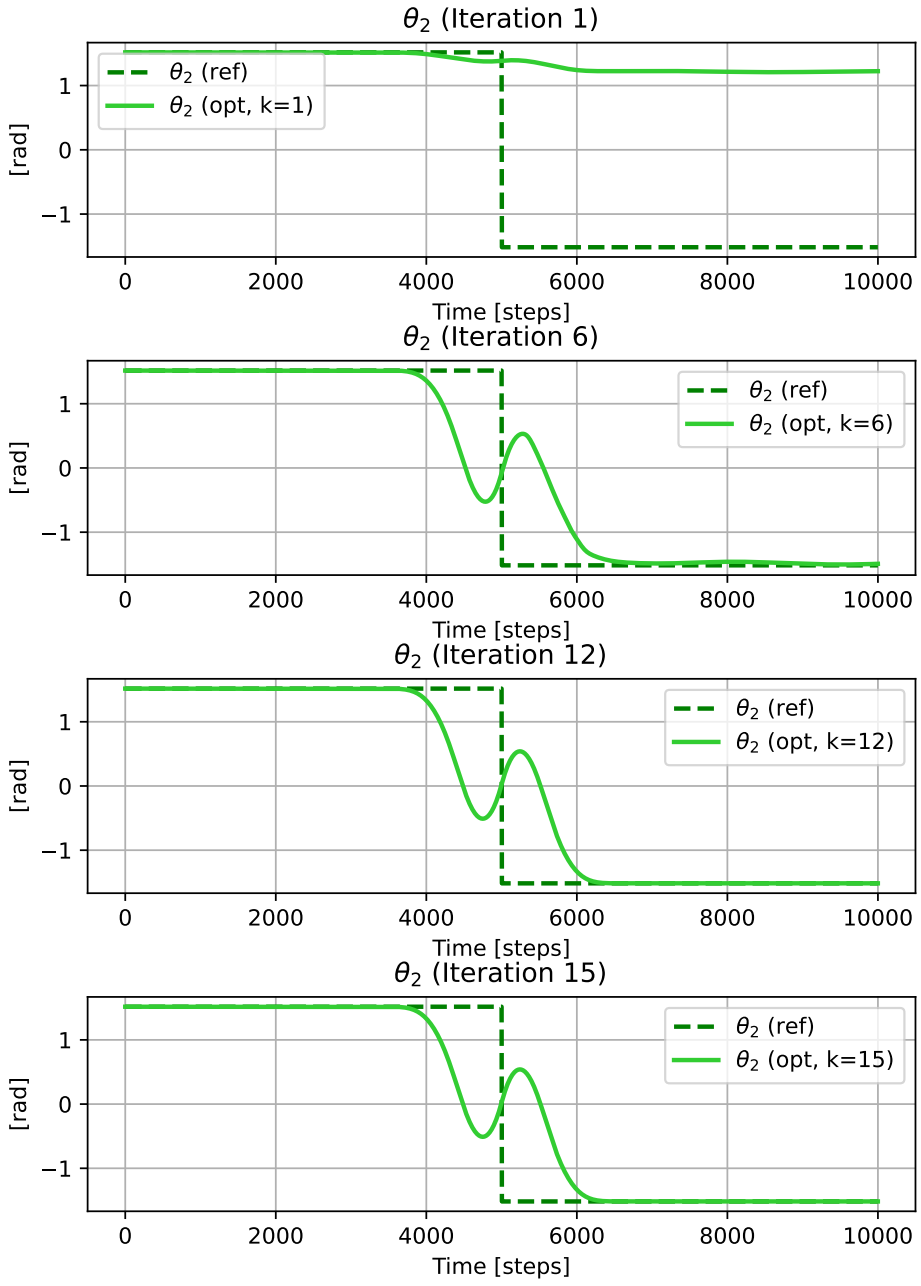


Figure 5.7: Evolution of θ_2.

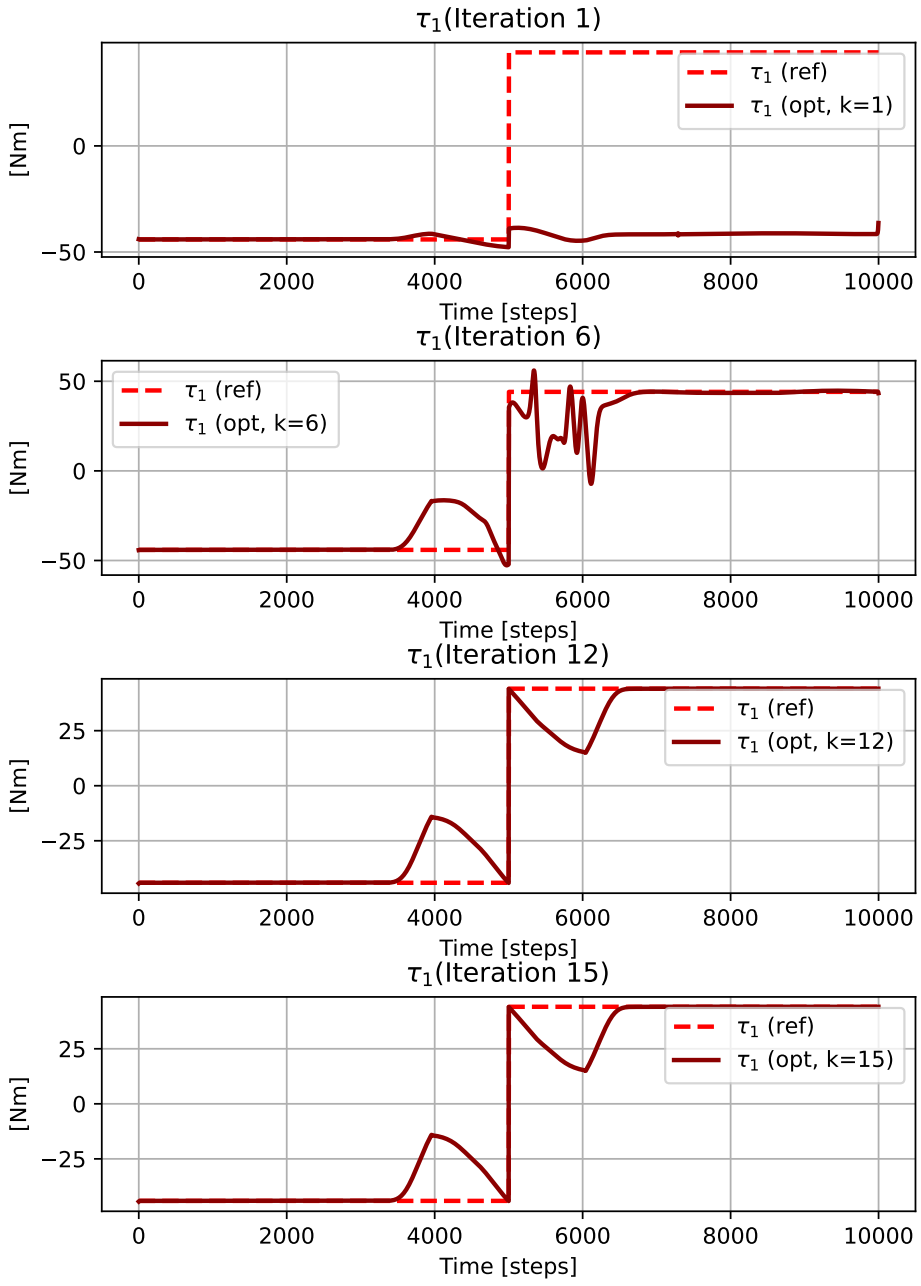


Figure 5.8: Evolution of τ_1.

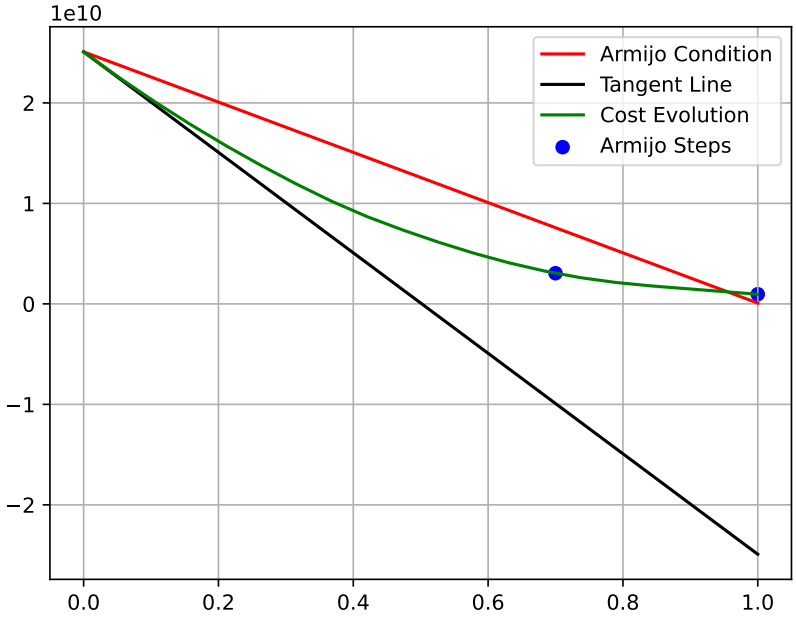


Figure 5.9: Armijo step-size selection: iteration 1.

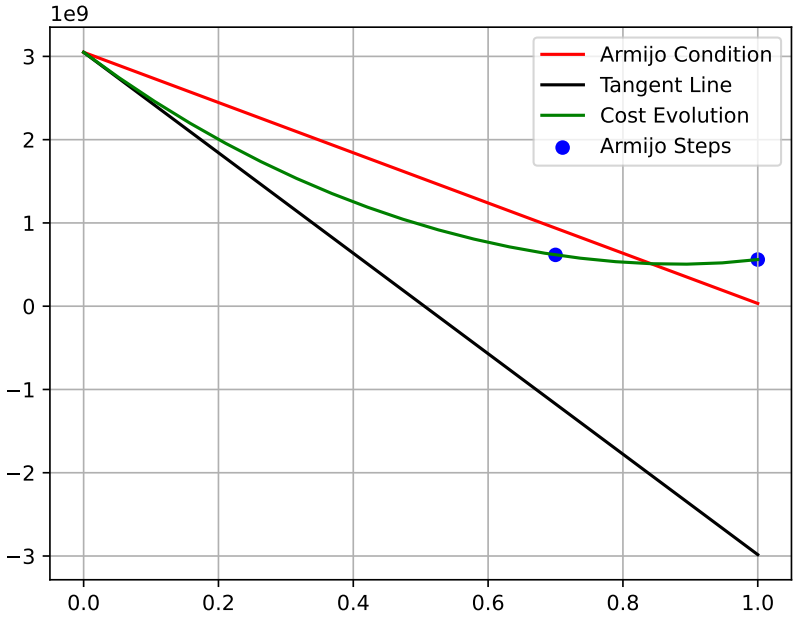


Figure 5.10: Armijo step-size selection: iteration 2.

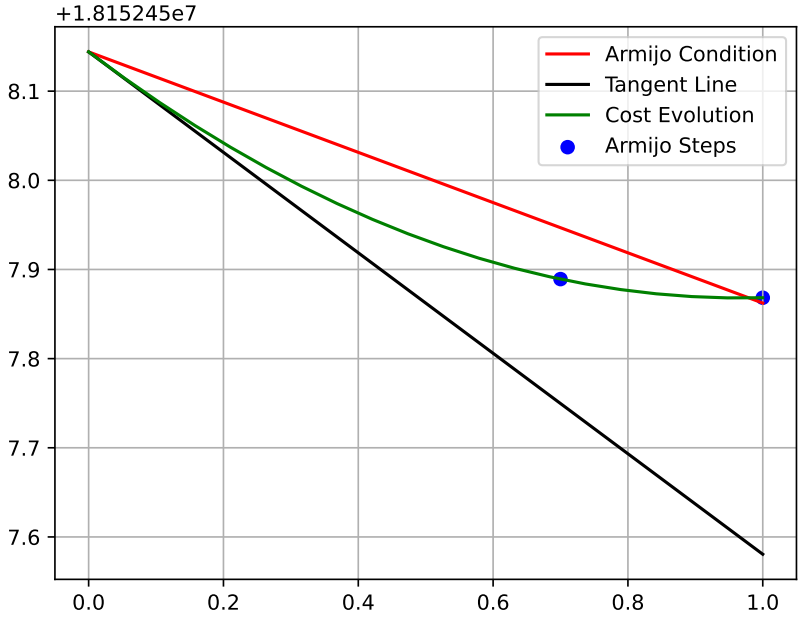


Figure 5.11: Armijo step-size selection: iteration 11.

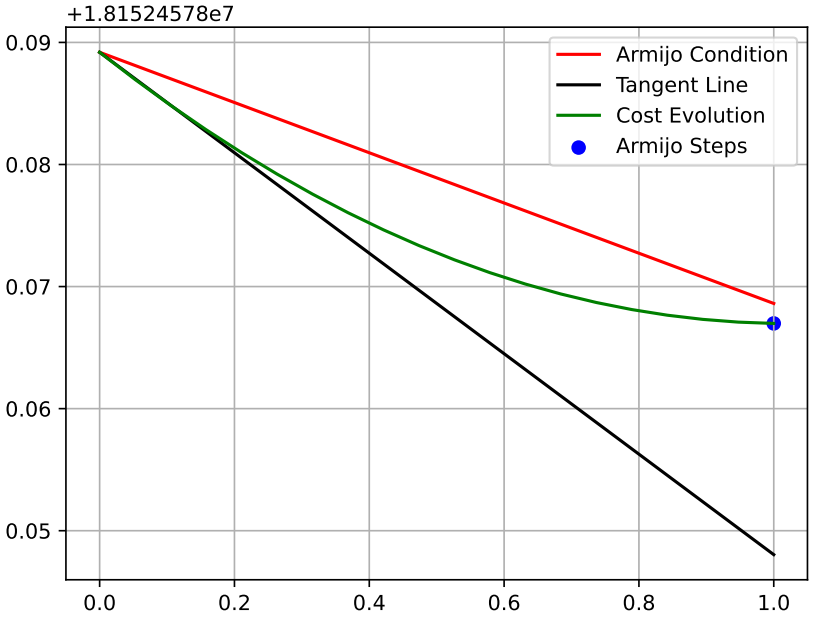


Figure 5.12: Armijo step-size selection: iteration 12.

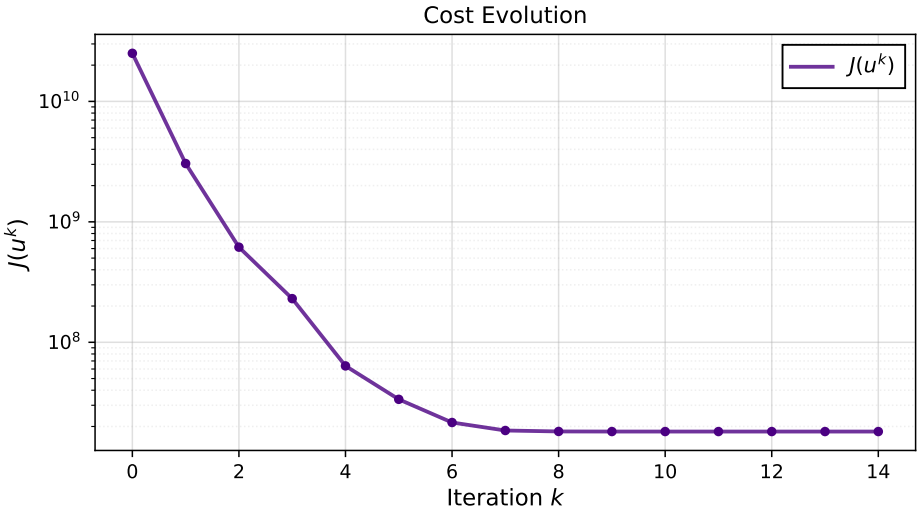


Figure 5.13: Cost evolution.

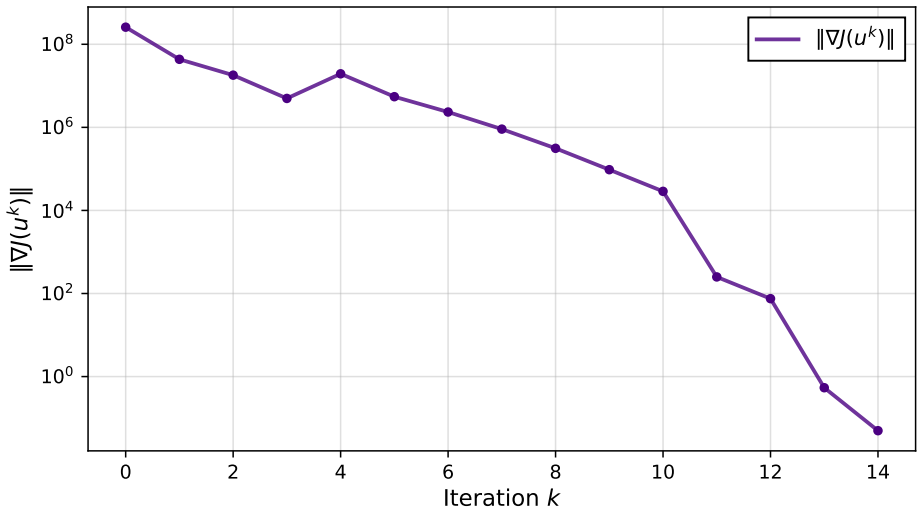


Figure 5.14: Cost gradient norm evolution.

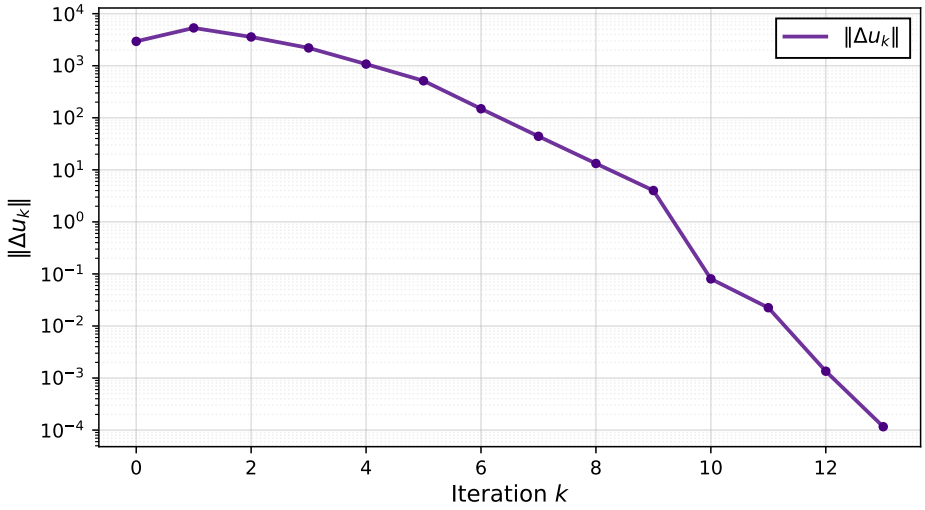


Figure 5.15: Evolution of $||\Delta u_k||$.

5.6 Constant Cost Matrices Scenario

The following cost matrices have been adopted:

$$Q_t = Q_T = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix} \tag{5.2}$$

$$R_t = \begin{bmatrix} 0.3 & 0 & 0 & 0 \\ 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0.3 & 0 \\ 0 & 0 & 0 & 0.3 \end{bmatrix} \tag{5.3}$$

Here, a summary of the results is presented.
The performances are obviously worse than the ones presented before.

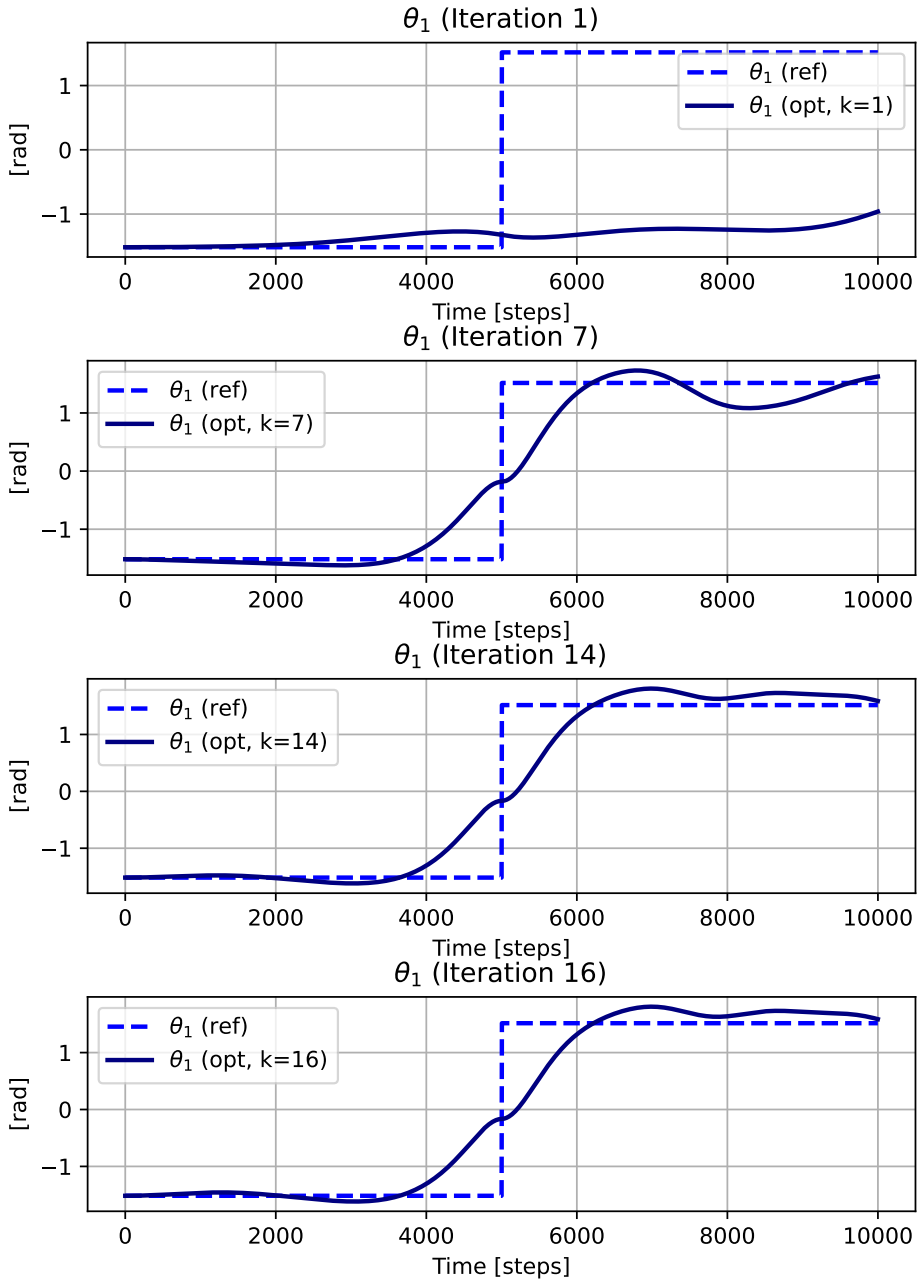


Figure 5.16: Evolution of θ_1 with constant Cost Matrices.

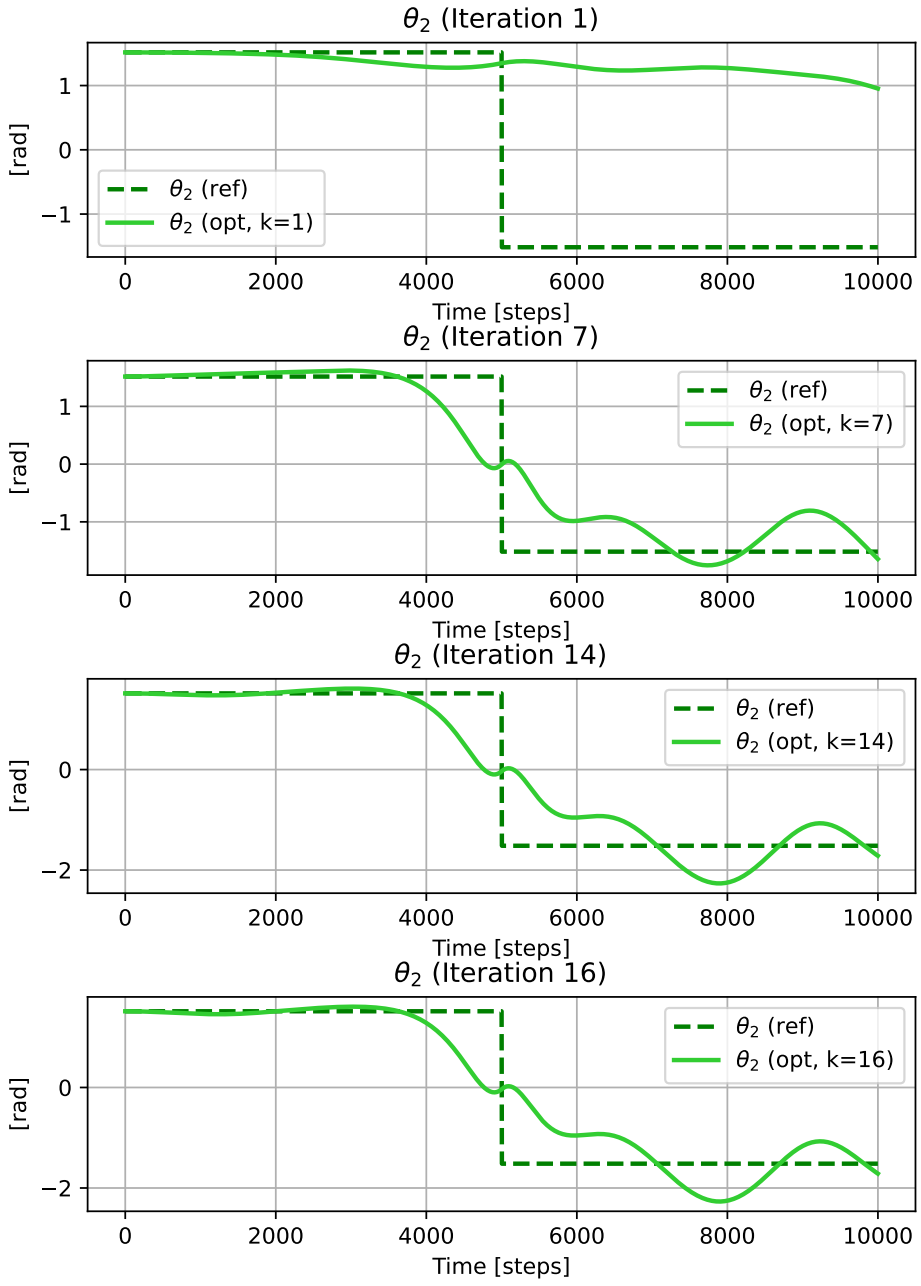


Figure 5.17: Evolution of θ_2 with constant Cost Matrices.

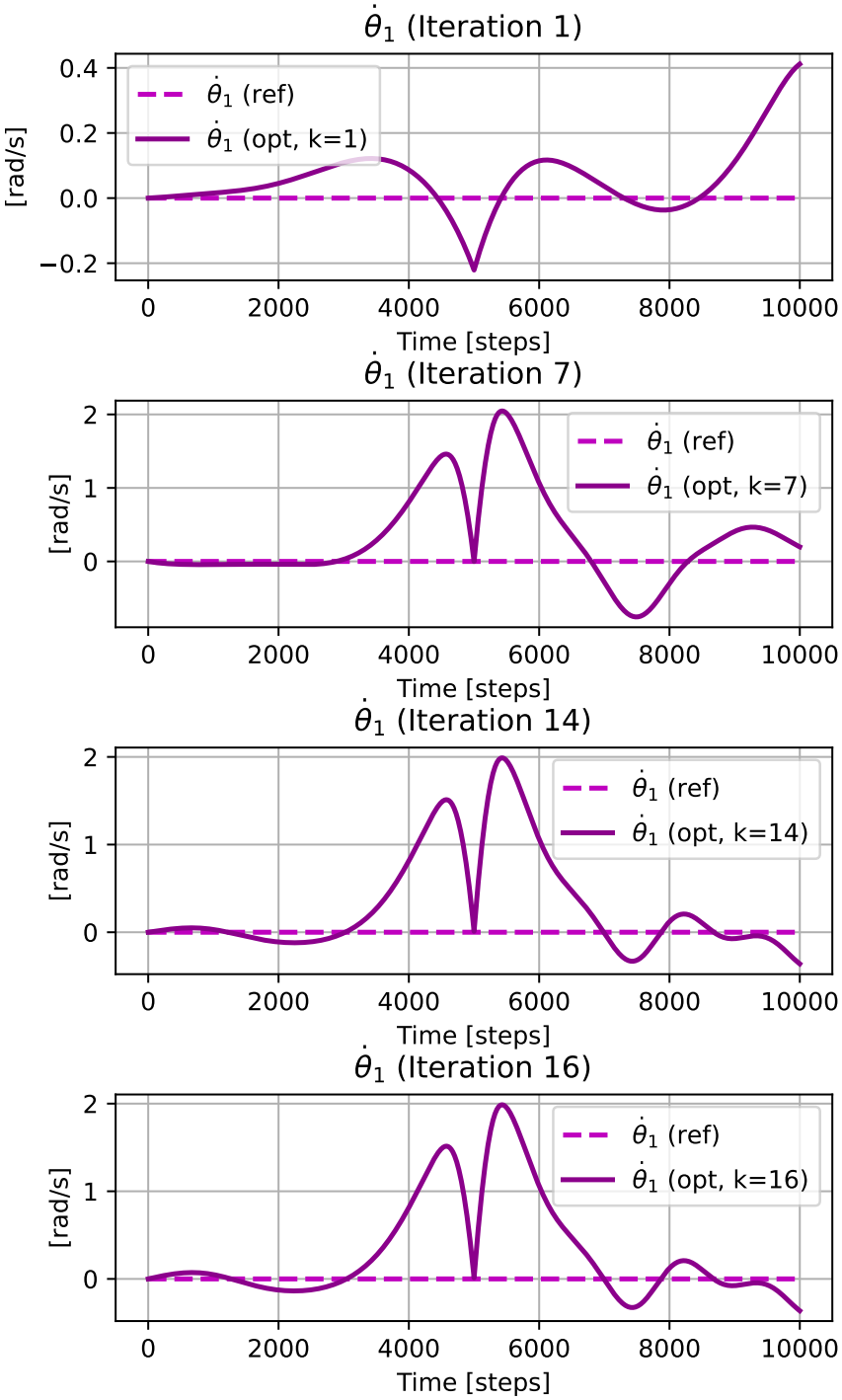


Figure 5.18: Evolution of $\dot{\theta}_1$ with constant Cost Matrices.

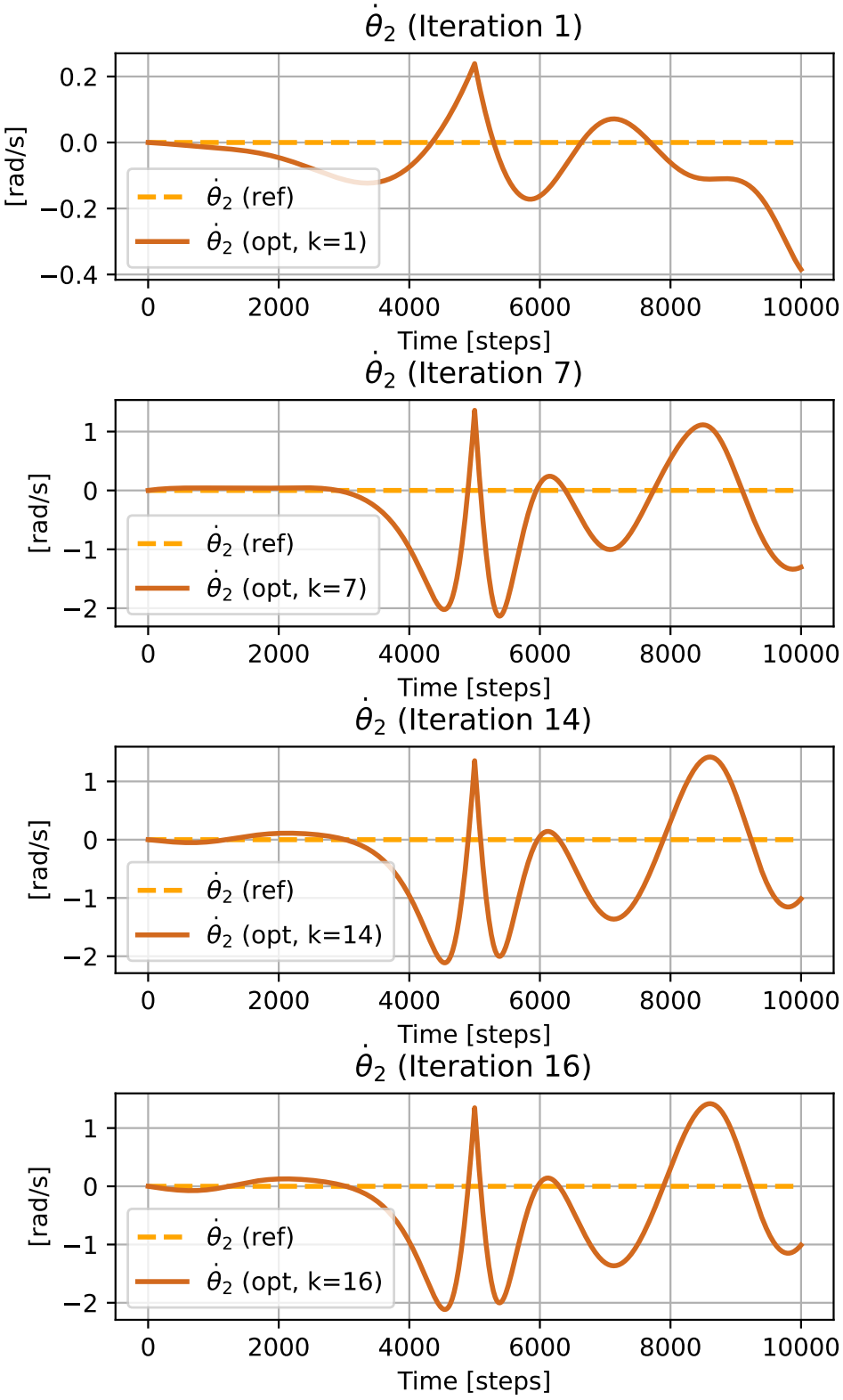


Figure 5.19: Evolution of $\dot{\theta}_2$ with constant Cost Matrices.

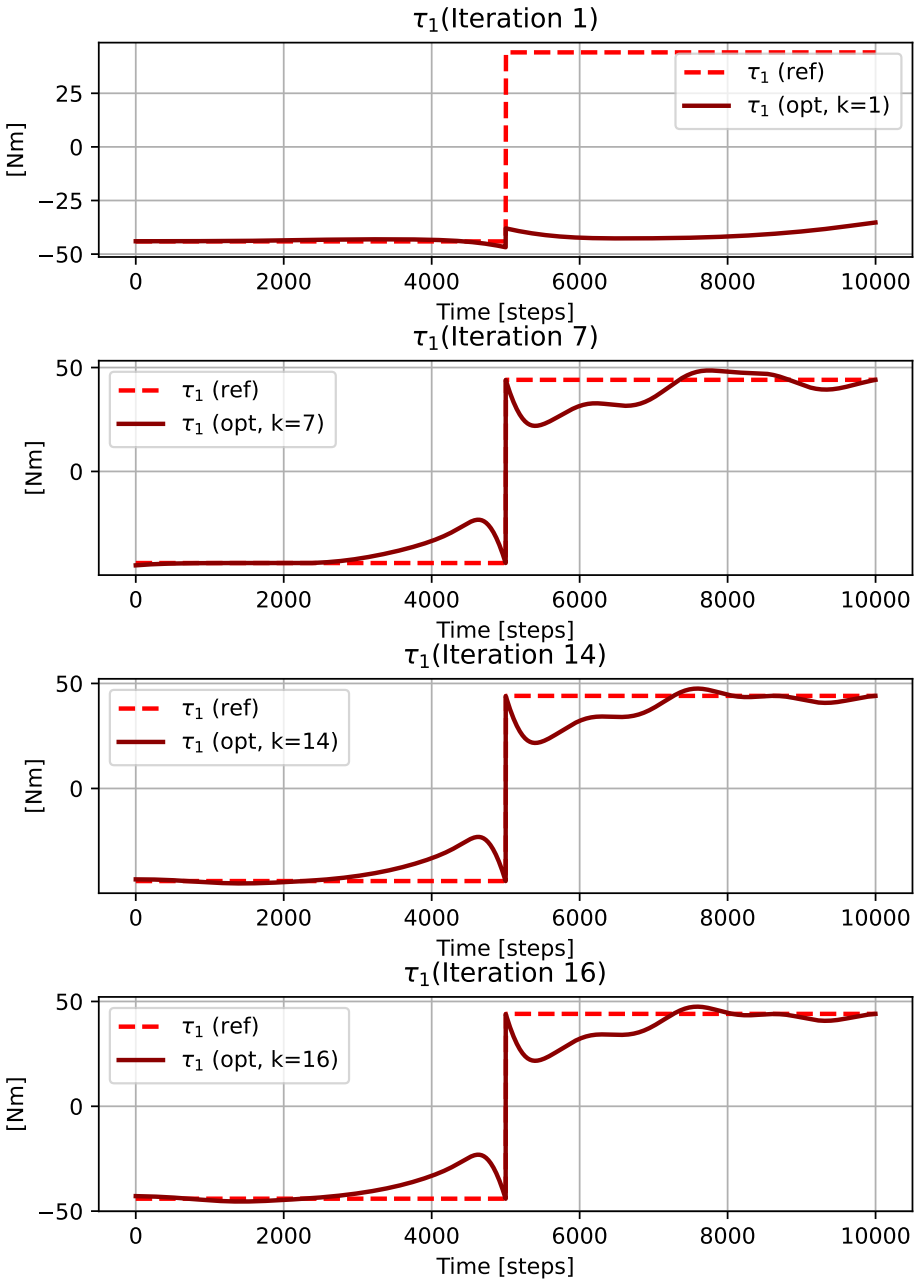


Figure 5.20: Evolution of τ with constant Cost Matrices.

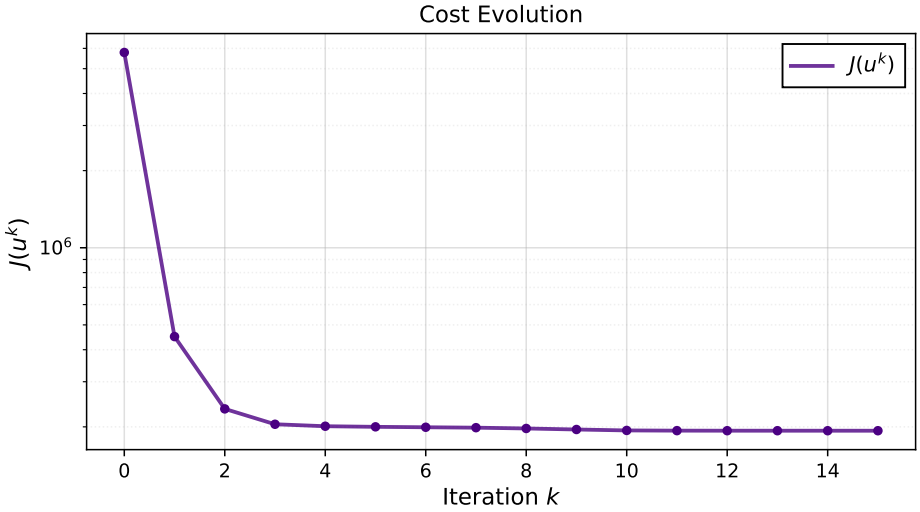


Figure 5.21: Evolution of cost function with constant Cost Matrices.

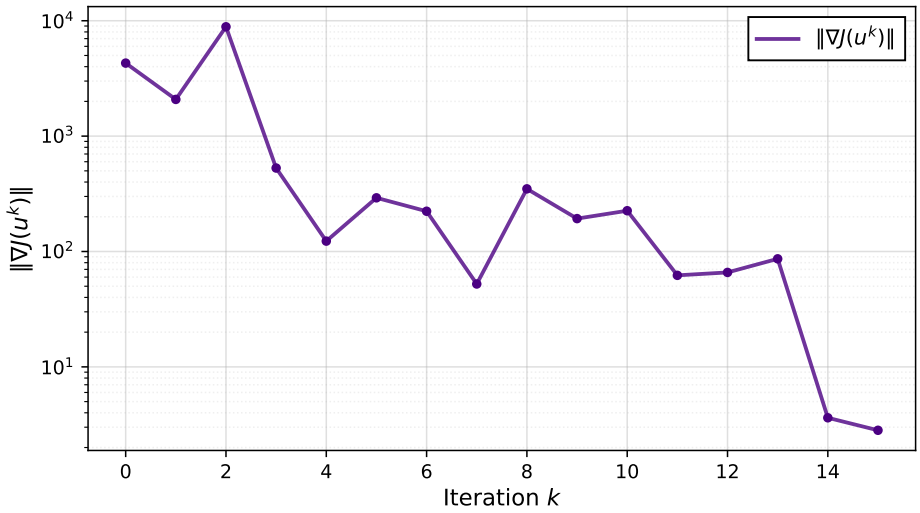


Figure 5.22: Evolution of $||\nabla J(u)||$ with constant Cost Matrices.

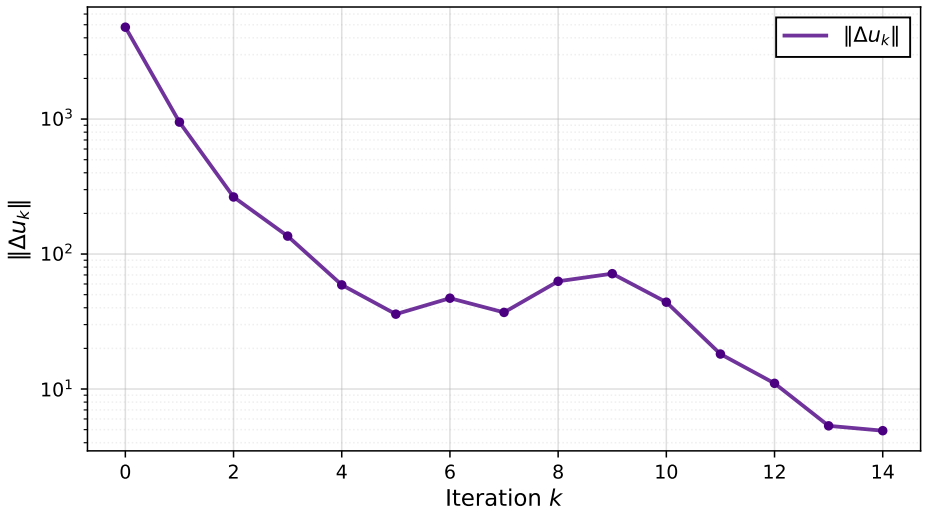


Figure 5.23: Evolution of $||\Delta u_k||$ with constant Cost Matrices.

Chapter 6

Trajectory generation (II)

6.1 Smooth Desired Trajectory

In this task, the reference curve is defined as a cubic spline that provides a smooth and continuous transition between a sequence of equilibrium points. The simulation starts with the system at the first equilibrium point. The reference trajectory guides the system through four smooth transitions between equilibrium points. At each intermediate equilibrium, the system briefly stabilizes before moving to the next. Once the system reaches the final equilibrium, it stabilizes completely and remains there for the rest of the simulation.

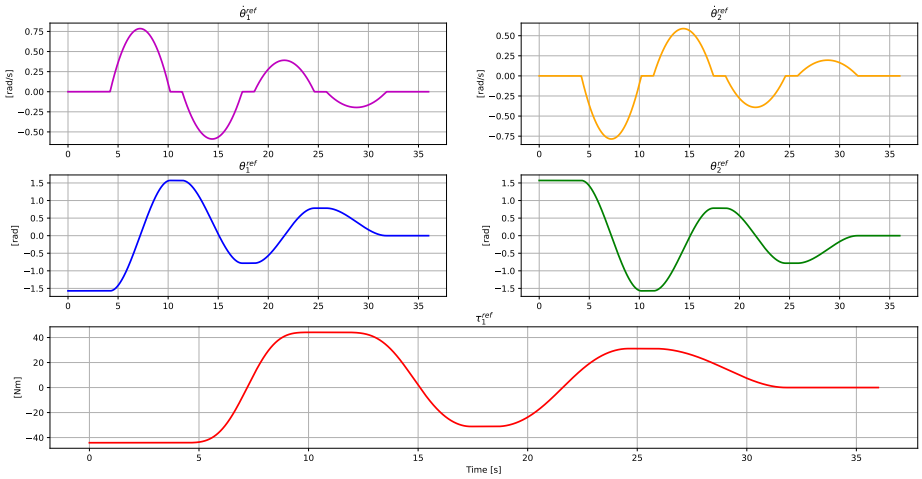


Figure 6.1: Smooth reference curve.

More precisely, this is a quasi-static trajectory; it represents a gradual and steady evolution of the system's states, during which the system remains in or near equilibrium at all times. This approach ensures that dynamic effects, such as transients or inertia, are negligible, as the system has sufficient

time to adjust to changes and maintain balance.

In the context of optimal control, quasi-static trajectories play an important role in the initialization process. They provide smooth and feasible transitions between equilibrium points, preventing abrupt changes that could lead to instability or infeasibility. By simplifying the dynamics and avoiding abrupt changes, they ensure stability, facilitate the computation of initial conditions, and improve the convergence of optimization algorithms.

6.2 Improved Optimal Transition

In Task 2, the same optimal control problem is addressed as in Task 1, with the distinction that the reference curve from which the system starts is a smooth transition between equilibrium points. The cost function is defined in Equation (2.1) and the cost matrices are time-varying, with the following behavior.

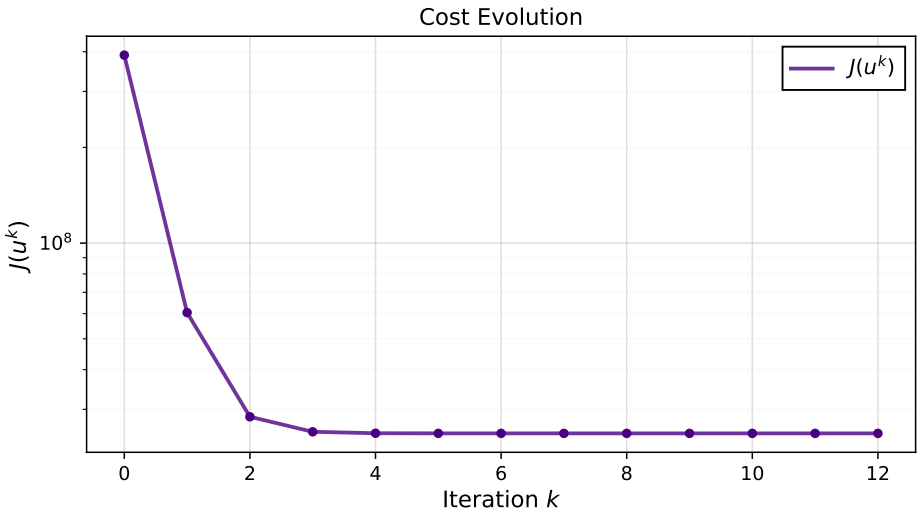


Figure 6.2: Evolution of cost matrices.

6.3 Plots for Trajectory Generation (II)

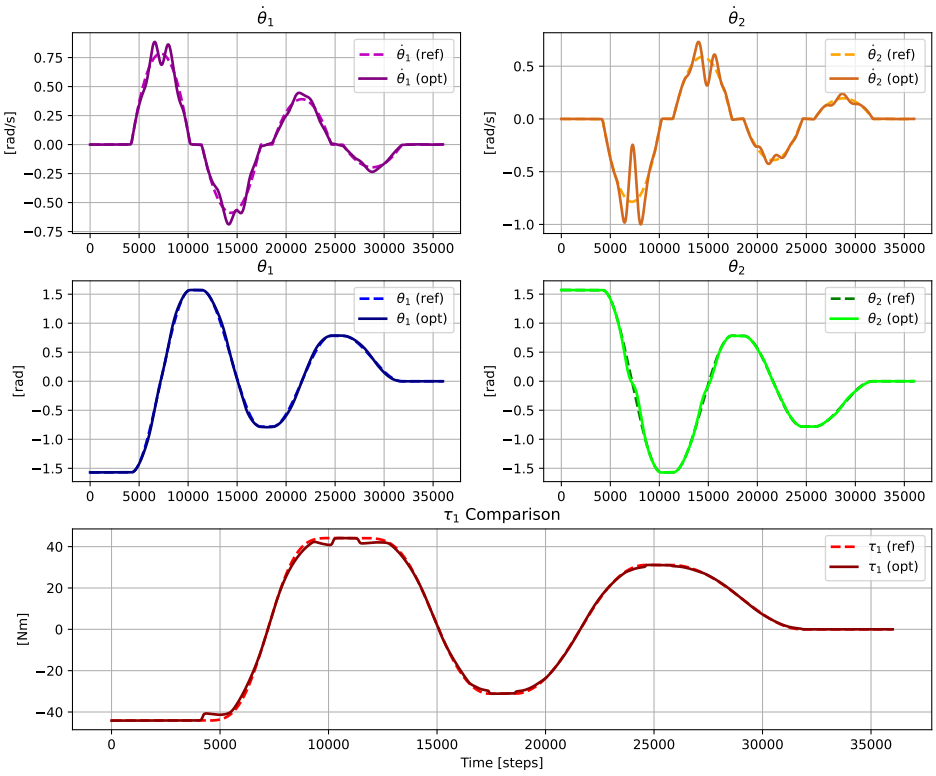


Figure 6.3: Generated optimal trajectory given a smooth reference.

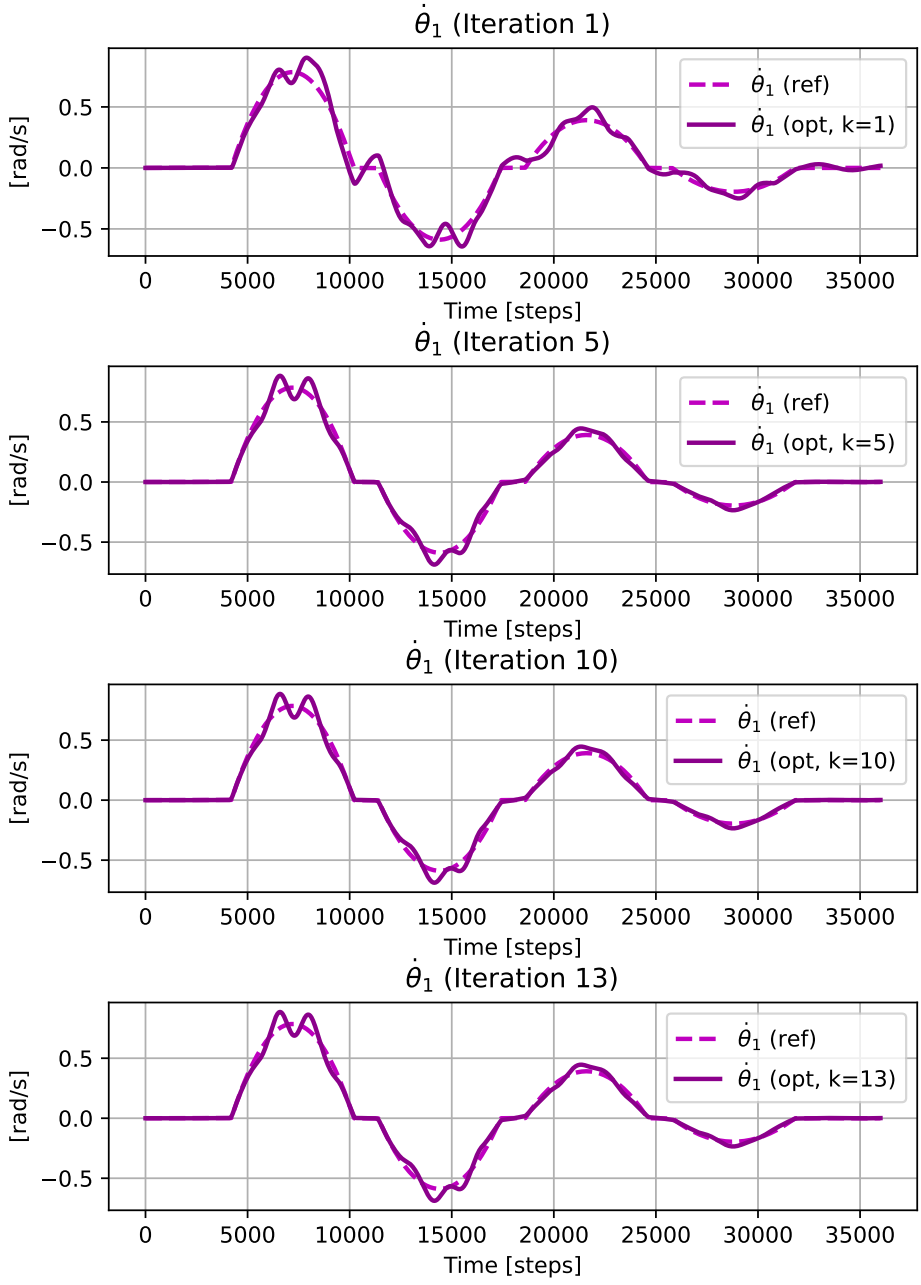


Figure 6.4: Evolution of $d\theta_1$.

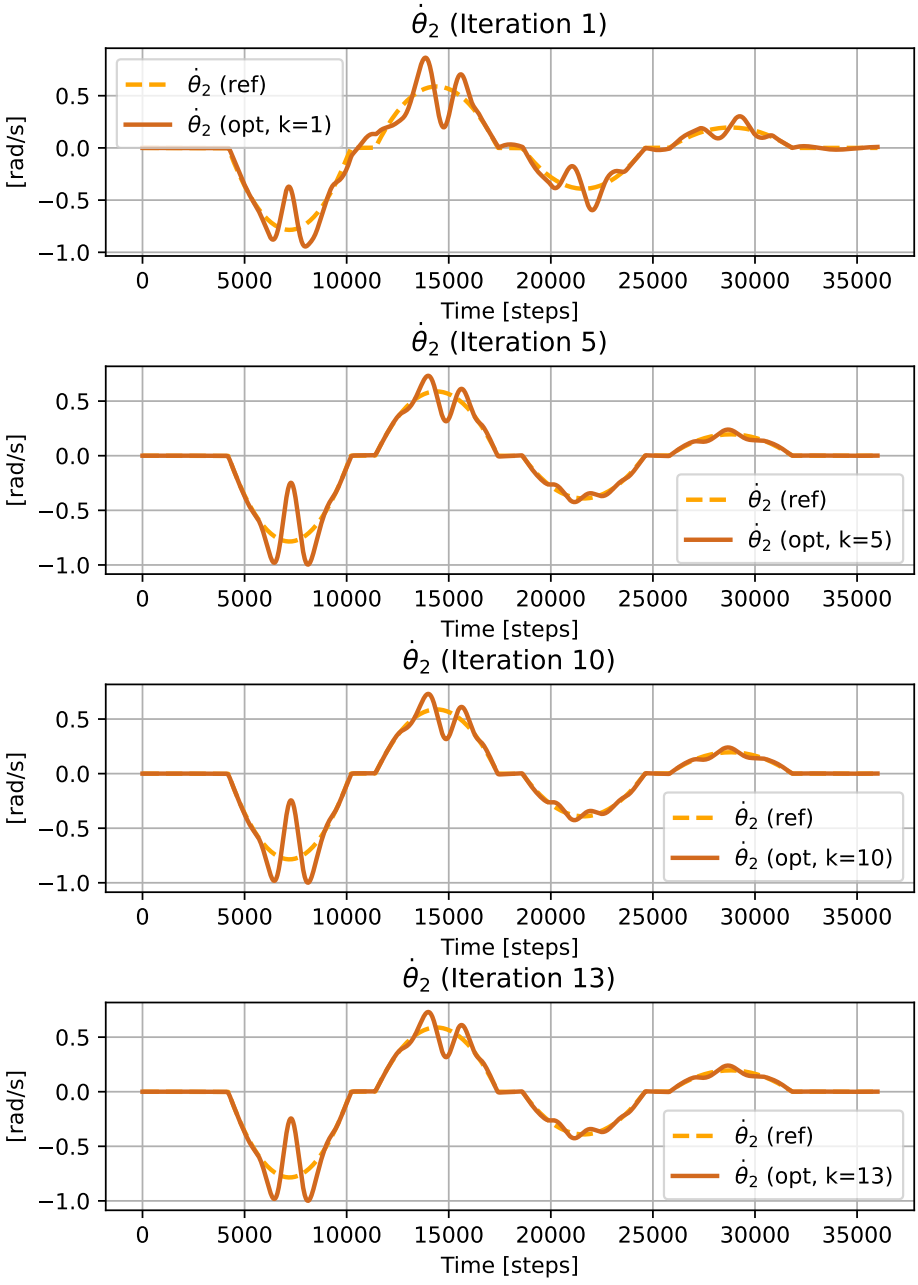


Figure 6.5: Evolution of $d\theta_2$.

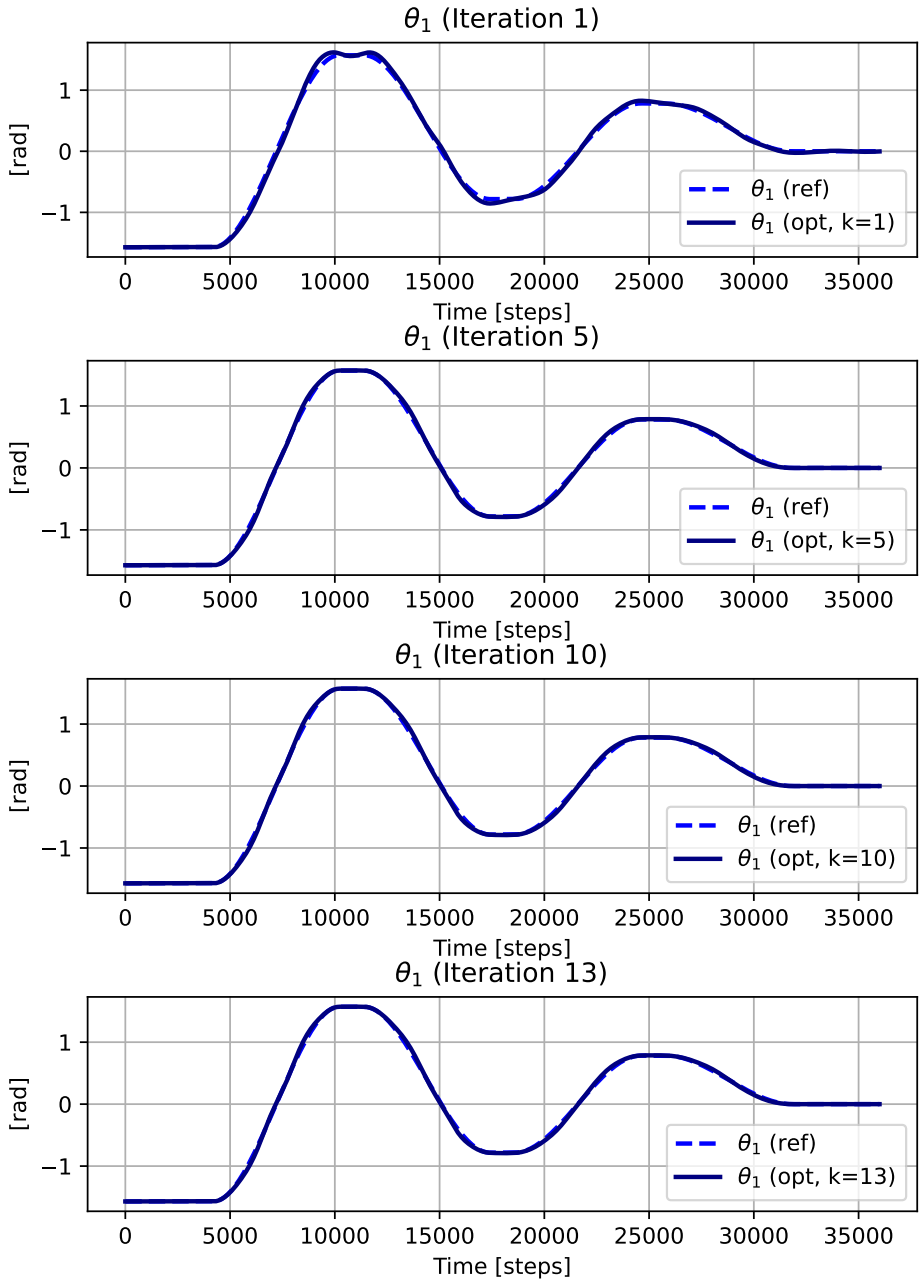


Figure 6.6: Evolution of θ_1.

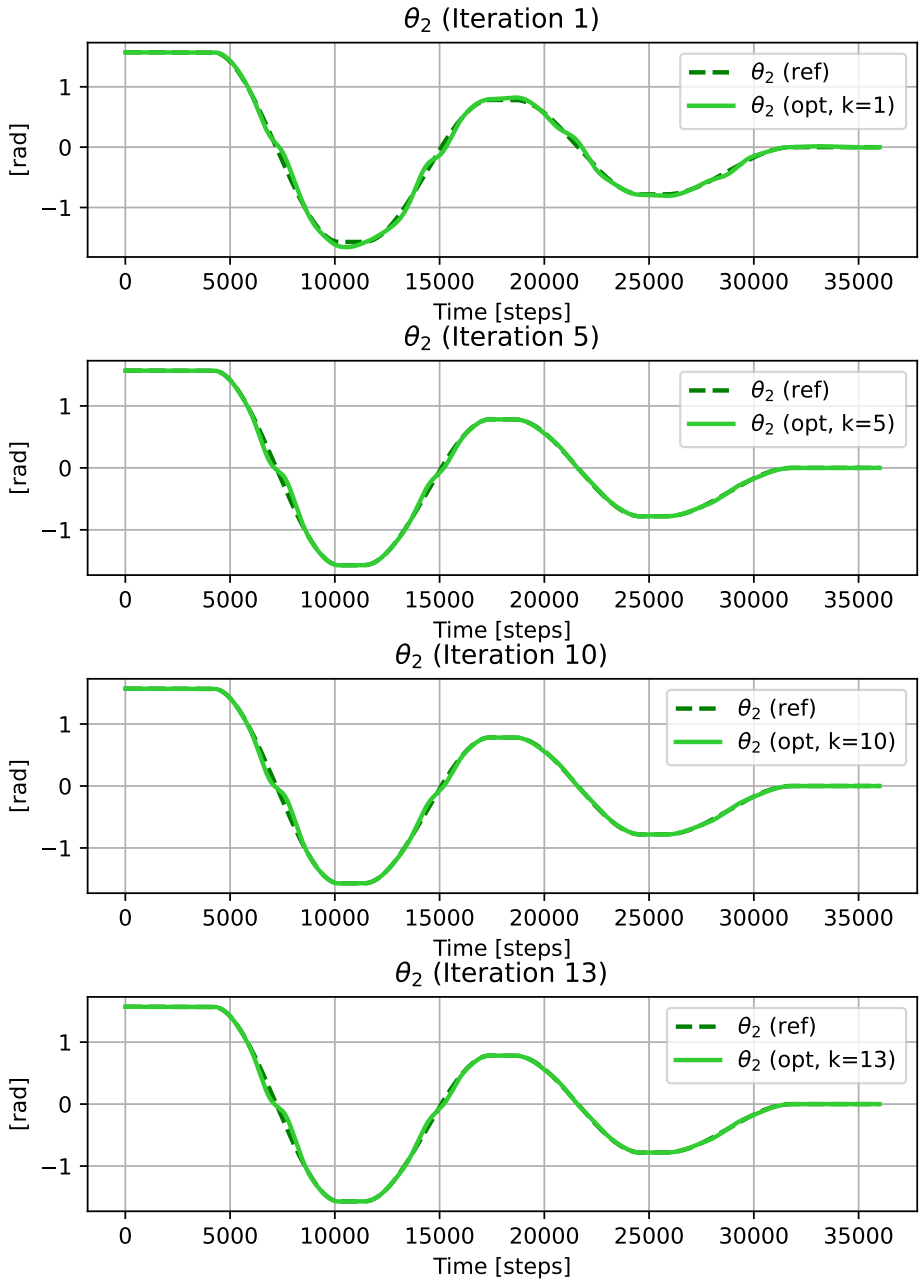


Figure 6.7: Evolution of θ_2.

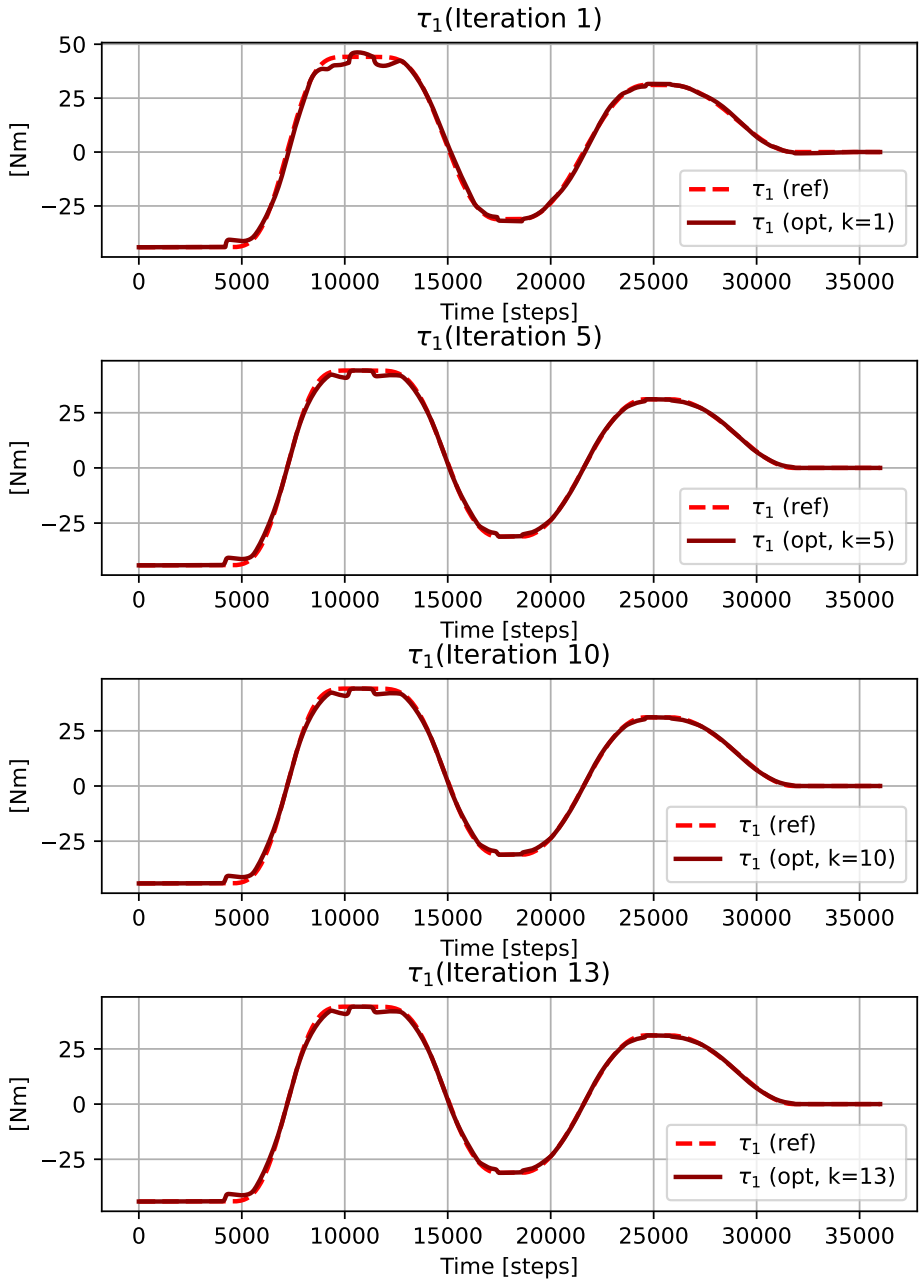


Figure 6.8: Evolution of τ_1.

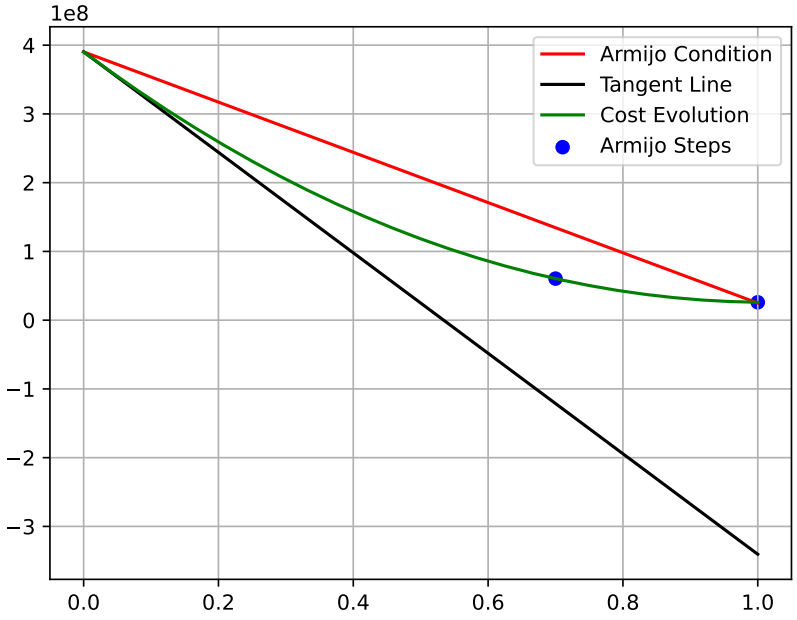


Figure 6.9: Armijo step-size selection: iteration 1

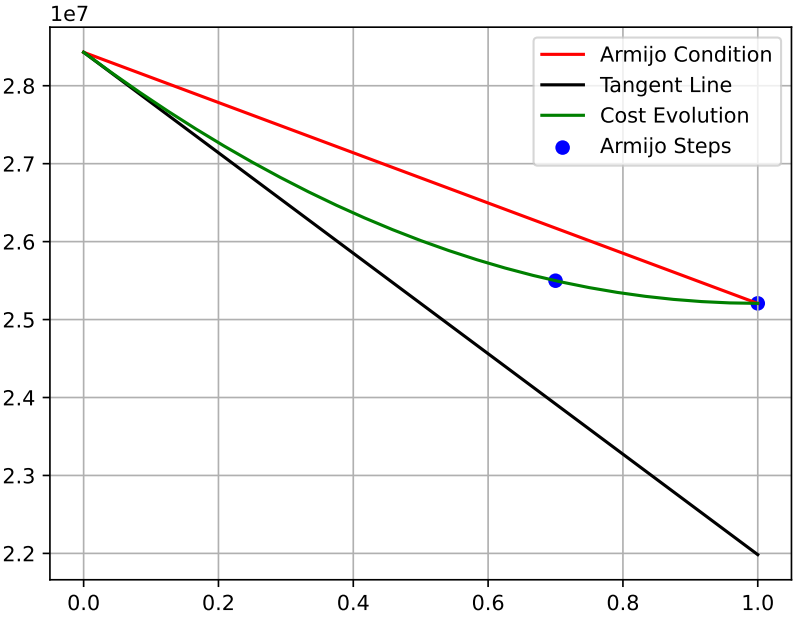


Figure 6.10: Armijo step-size selection: iteration 3

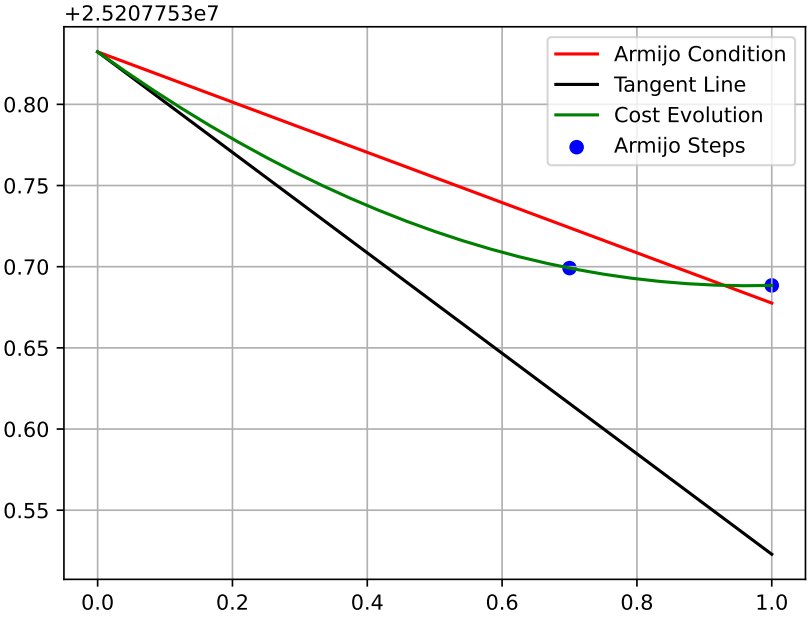


Figure 6.11: Armijo step-size selection: iteration 10

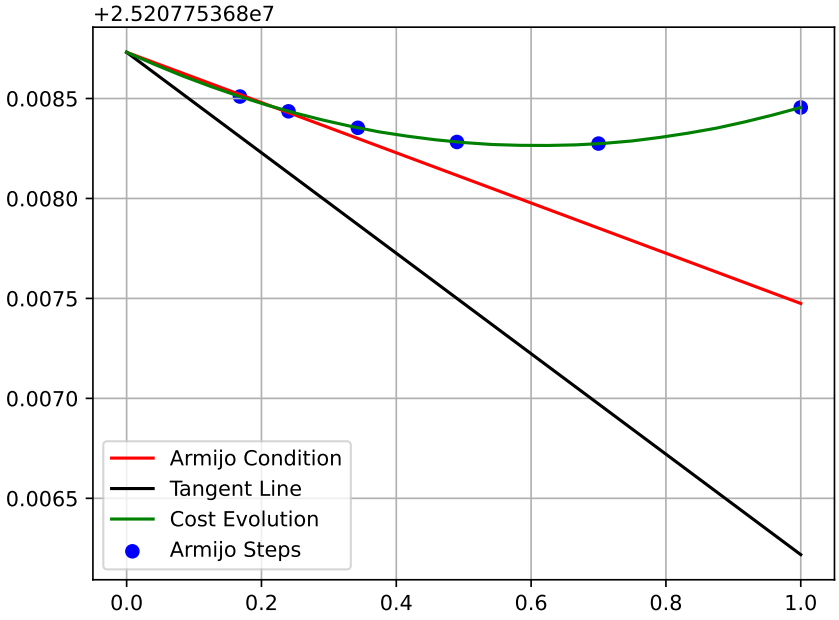


Figure 6.12: Armijo step-size selection: iteration: 12

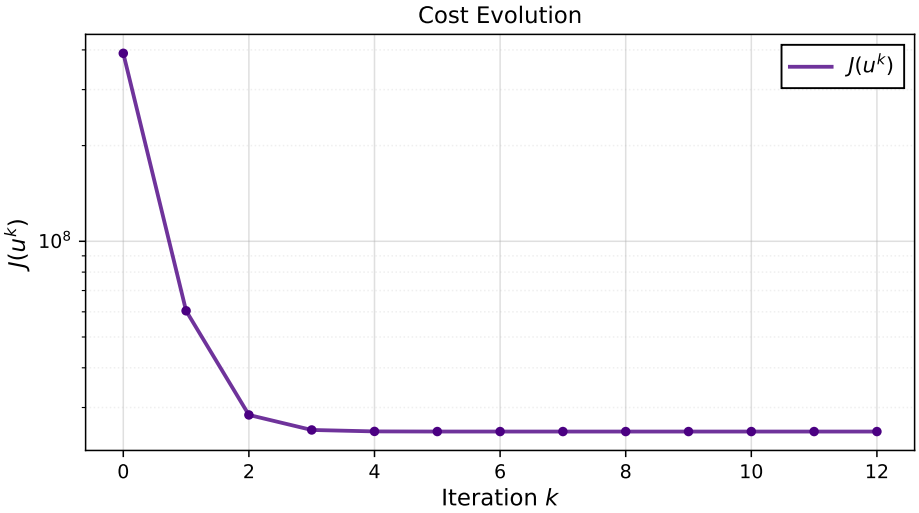


Figure 6.13: Cost evolution

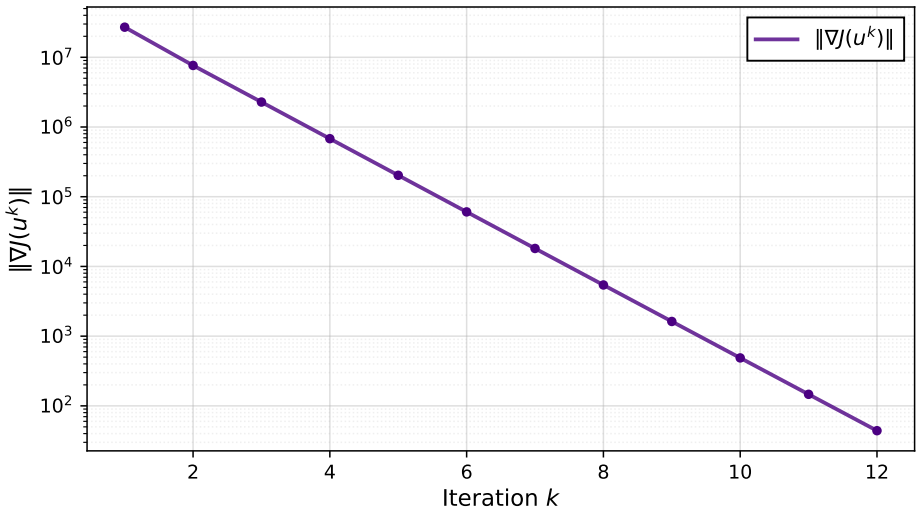


Figure 6.14: Cost gradient norm evolution

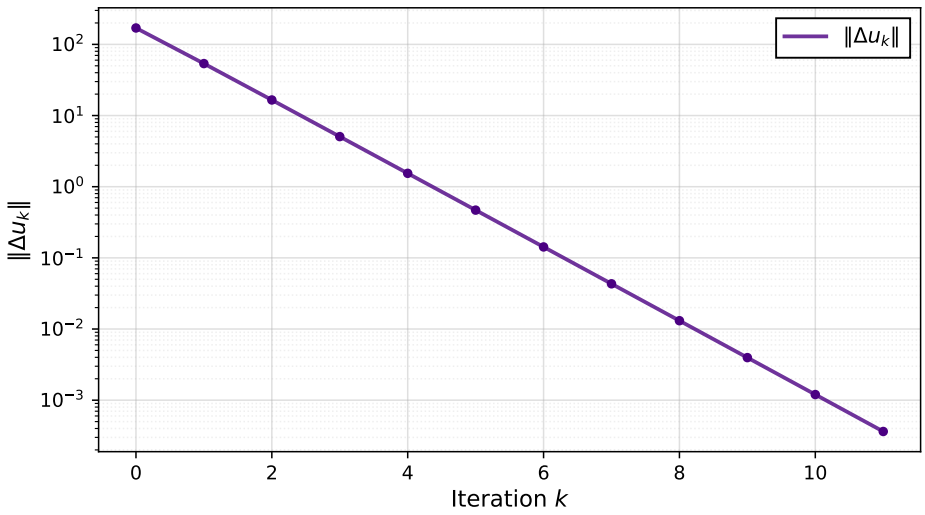


Figure 6.15: Evolution of $||\Delta u_k||$

6.4 Constant Cost Matrices Scenario

The following cost matrices have been adopted:

$$Q_t = Q_T = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix} \tag{6.1}$$

$$R_t = \begin{bmatrix} 0.3 & 0 & 0 & 0 \\ 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0.3 & 0 \\ 0 & 0 & 0 & 0.3 \end{bmatrix} \tag{6.2}$$

Here, a summary of the results is presented.
The performances are obviously worse than the ones presented before.

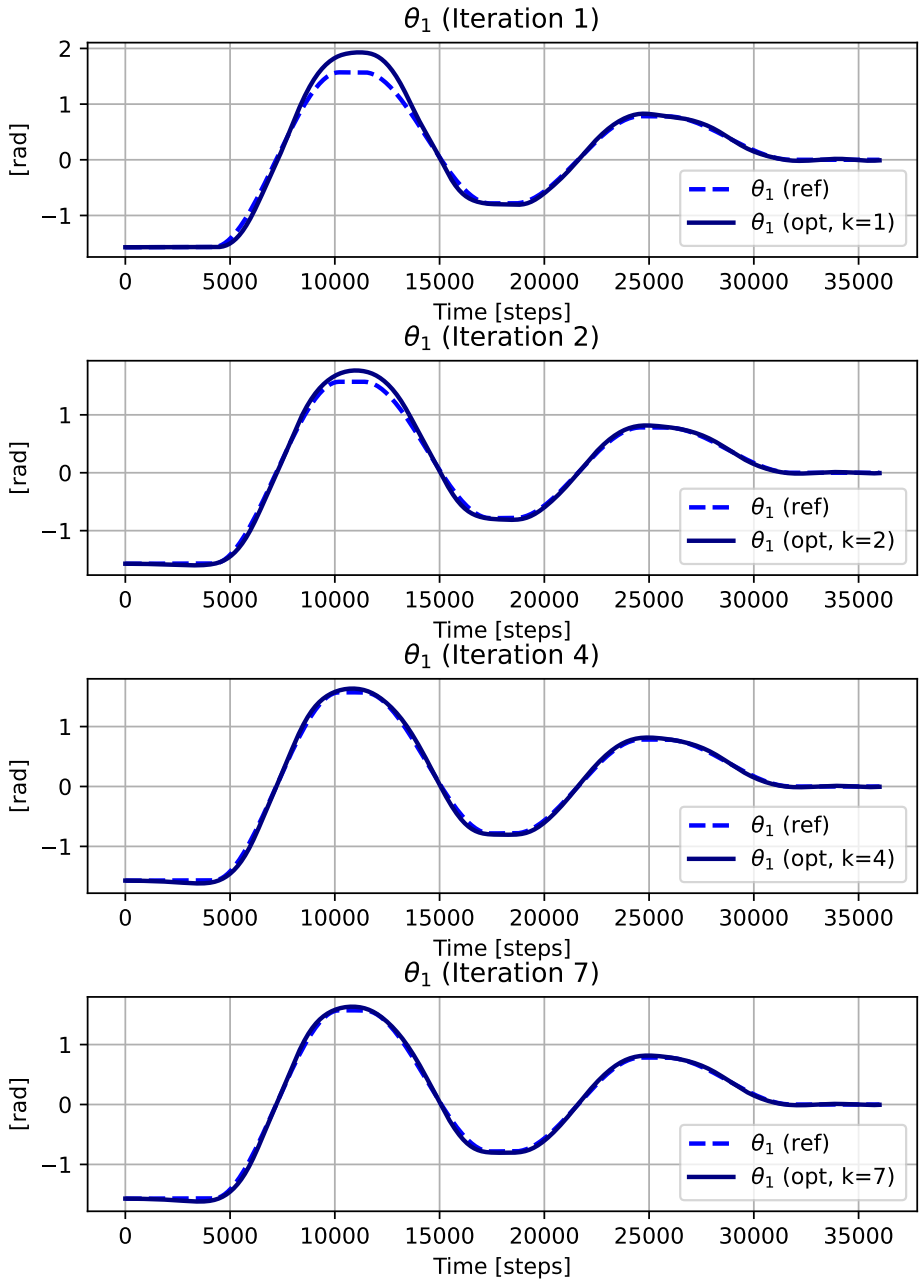


Figure 6.16: Evolution of θ_1 with constant Cost Matrices.

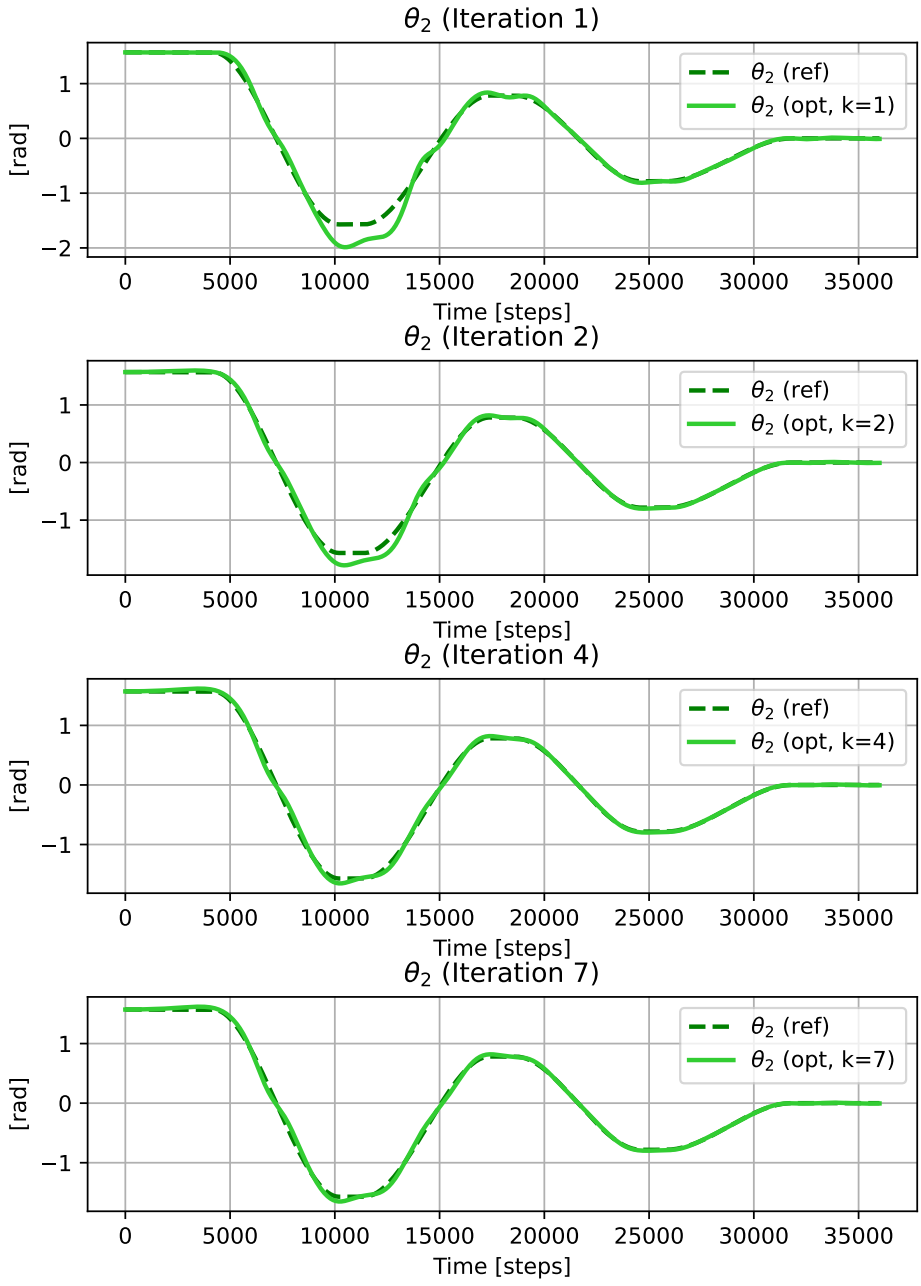


Figure 6.17: Evolution of θ_2 with constant Cost Matrices.

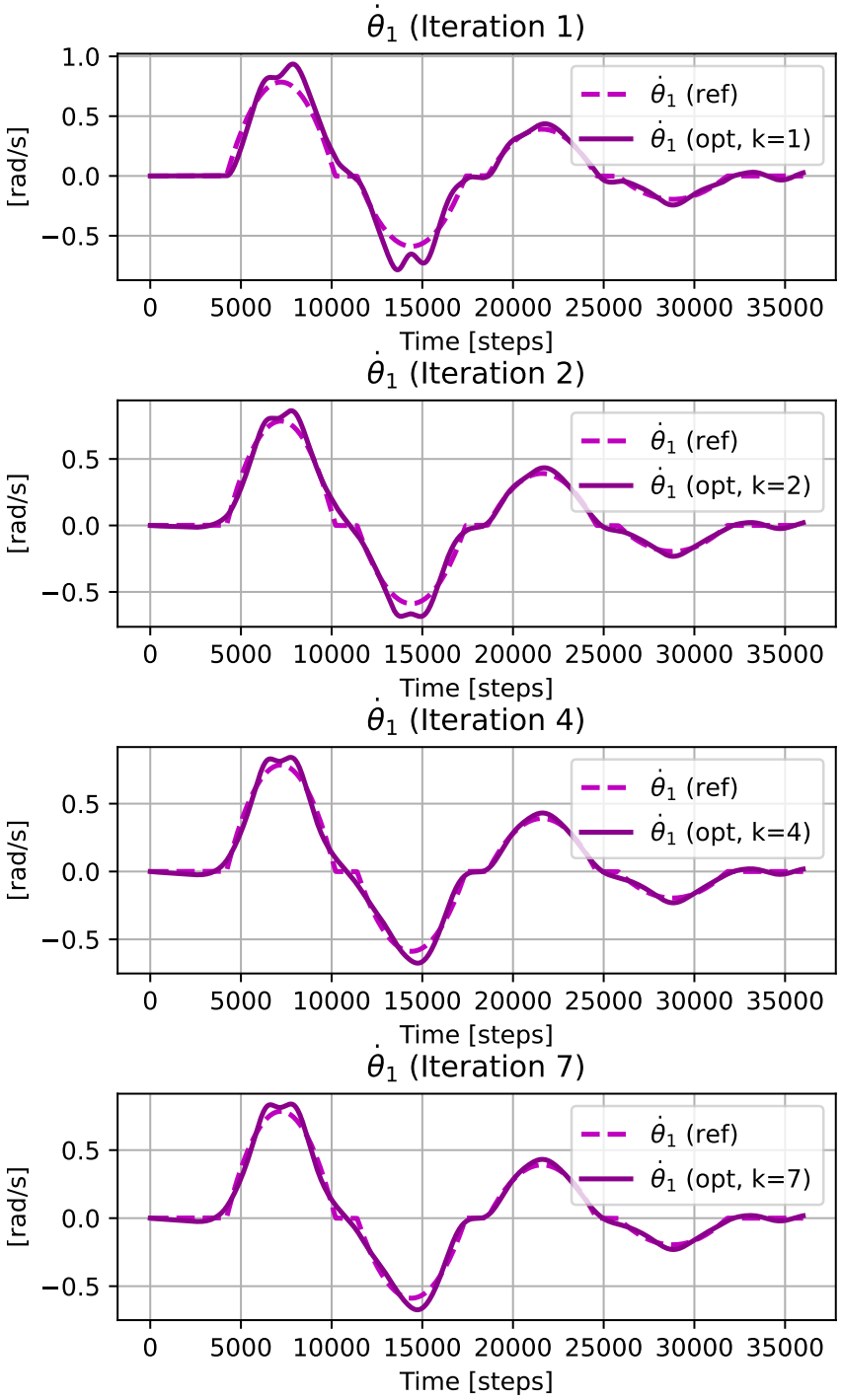


Figure 6.18: Evolution of $\dot{\theta}_1$ with constant Cost Matrices.

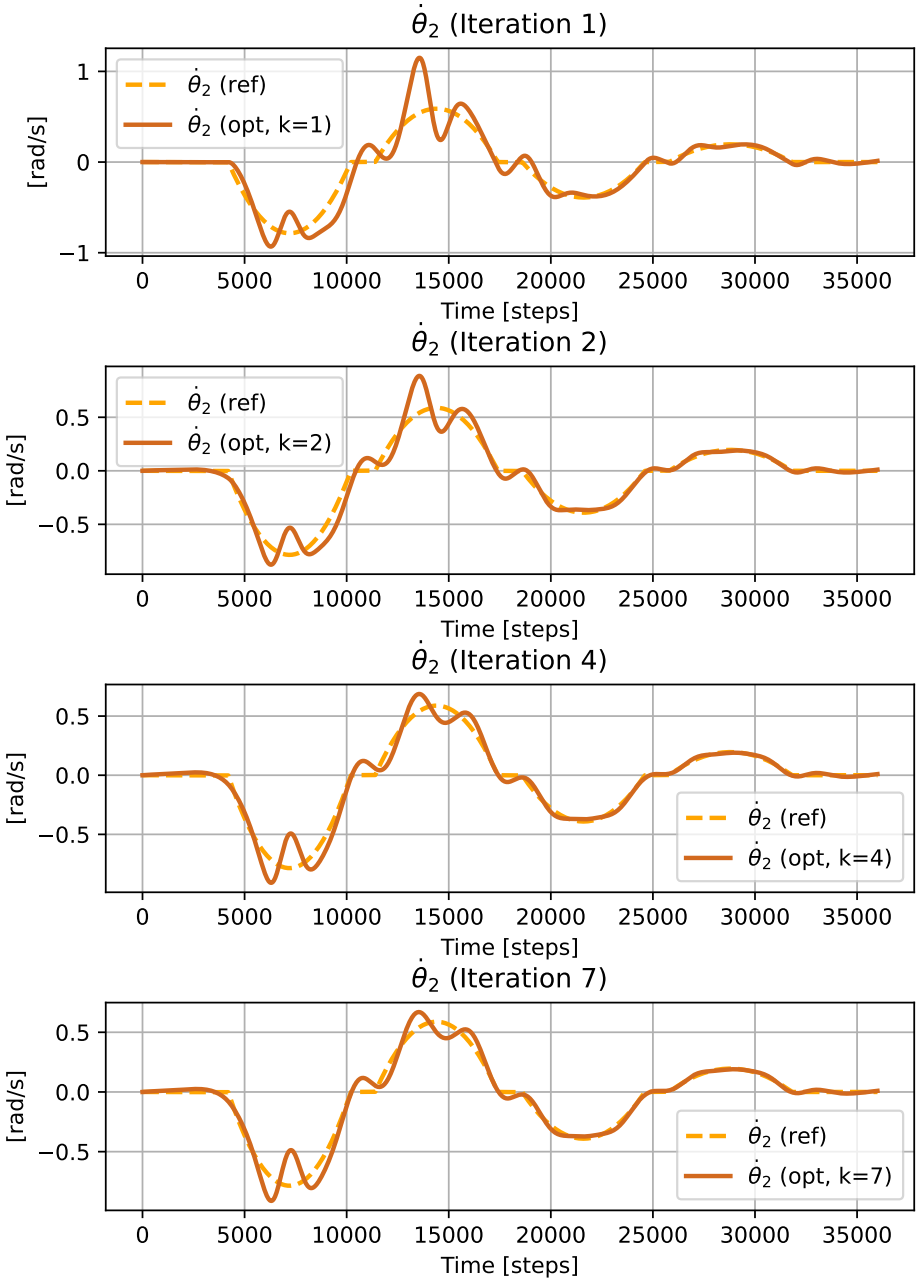


Figure 6.19: Evolution of $\dot{\theta}_2$ with constant Cost Matrices.

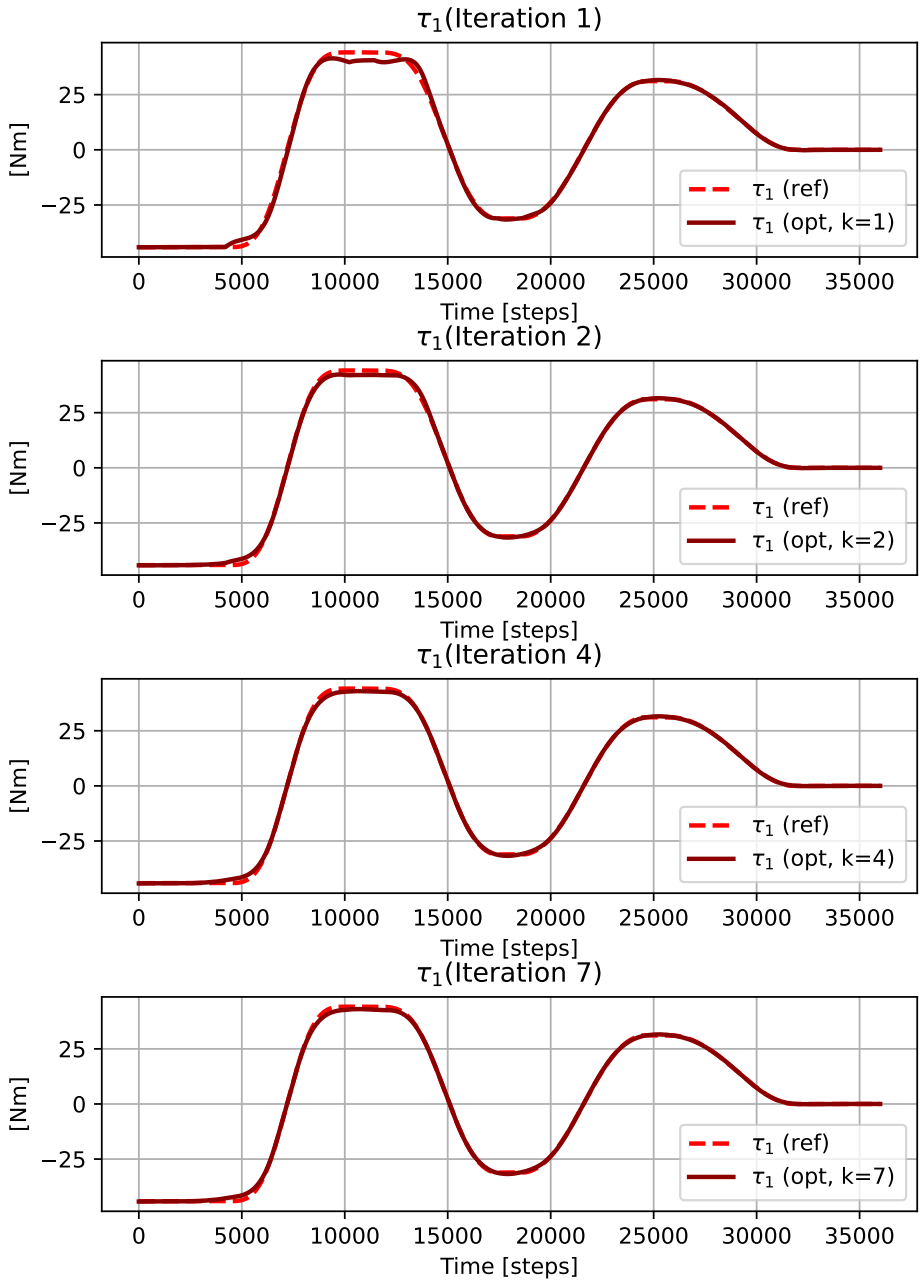


Figure 6.20: Evolution of τ with constant Cost Matrices.

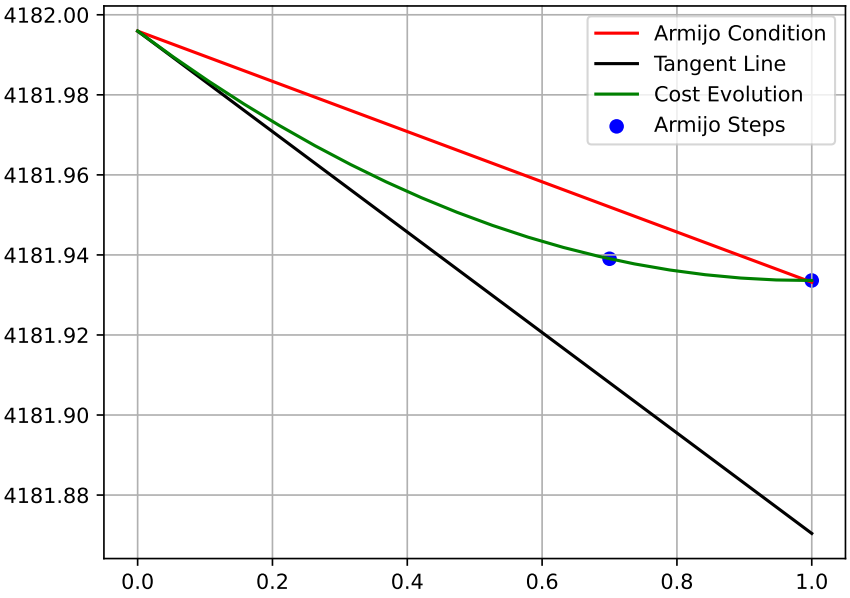


Figure 6.21: Armijo step-size selection: last iteration.

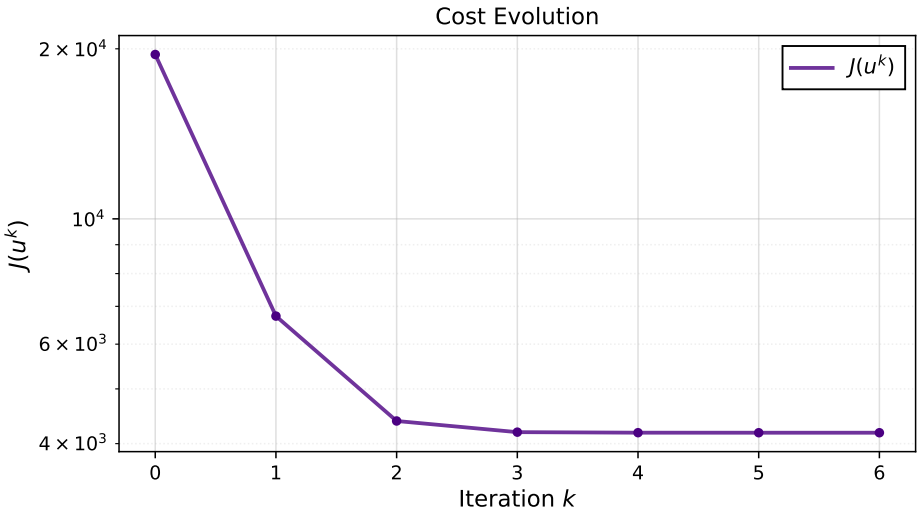


Figure 6.22: Evolution of cost function with constant Cost Matrices.

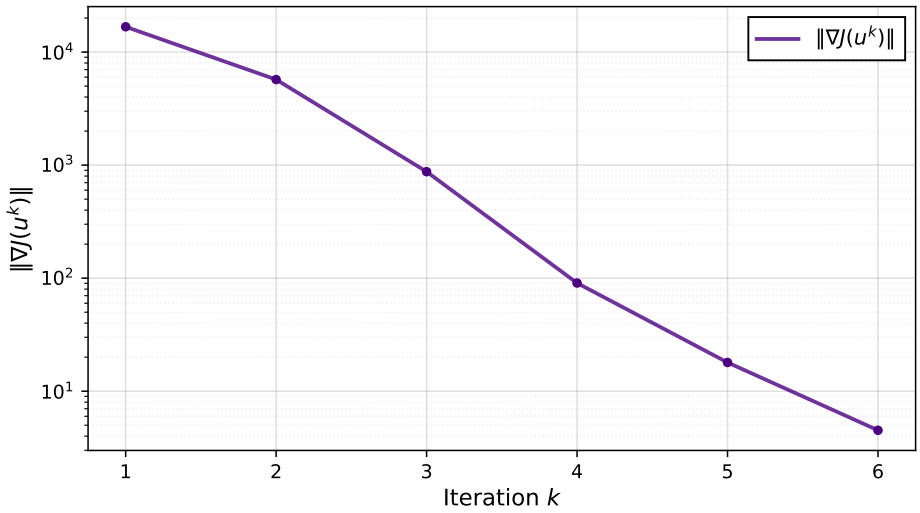


Figure 6.23: Evolution of $||\nabla J(u)||$ with constant Cost Matrices.

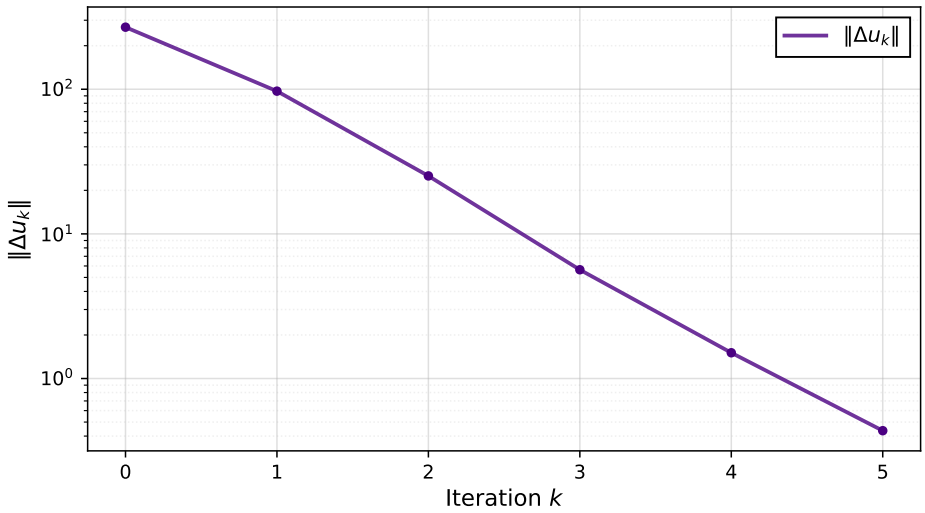


Figure 6.24: Evolution of $||\Delta u_k||$ with constant Cost Matrices.

Chapter 7

Trajectory tracking via LQR

7.1 Dynamics Linearization and LQR Design

In this task, the robot dynamics are linearized around the trajectory $(x^{\text{gen}}, u^{\text{gen}})$ computed in Task 2. Using the linearized model, the LQR algorithm is applied to design an optimal feedback controller for tracking the reference trajectory. The objective is to solve the following Linear-Quadratic (LQ) problem:

$$\min_{\substack{\Delta x_1,\ldots,\Delta x_T,\\ \Delta u_0,\ldots,\Delta u_{T-1}}} \quad \sum_{t=0}^{T-1} \Delta x_t^\top Q_t^{\text{reg}} \Delta x_t + \Delta x_t^\top R_t^{\text{reg}} \Delta u_t + \Delta x_T^\top Q_T^{\text{reg}} \Delta x_T$$

Subject to:

$$\Delta x_{t+1} = A_t^{\text{gen}} \Delta x_t + B_t^{\text{gen}} \Delta u_t, \quad t = 0, \ldots, T-1$$

$$x_0 = 0$$

Here, $\Delta x_t = x_t - x_t^{\text{gen}}$, where A_t^{gen} and B_t^{gen} are the matrices obtained by linearizing the nonlinear system dynamics around the reference trajectory $(x_{\text{gen}}, u_{\text{gen}})$. The cost matrices Q_{reg} and R_{reg} are tuning parameters that need to be defined during the regulator's setup. These matrices balance the trade-off between state deviations and control effort. In task 3 the behavior of the cost matrices is the following:

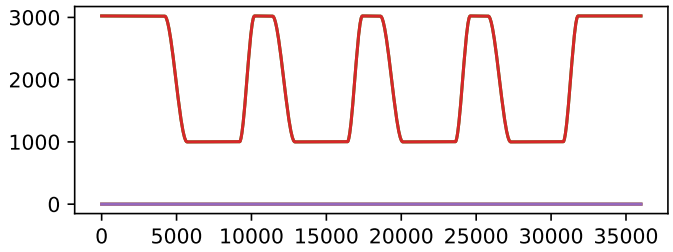


Figure 7.1: Evolution of cost matrices.

To compute the feedback gain, the discrete Riccati equation is solved backward in time. Starting with $P_T = Q_T^{\text{reg}}$, compute recursively for $t = T-1, \ldots, 0$:

$$P_t^{\text{reg}} = Q_t^{\text{reg}} + A_t^{\text{gen},\top} P_{t+1}^{\text{reg}} A_t^{\text{gen}}$$

$$- \left(A_t^{\text{gen},\top} P_{t+1}^{\text{reg}} B_t^{\text{gen}}\right) \left(R_t^{\text{reg}} + B_t^{\text{gen},\top} P_{t+1}^{\text{reg}} B_t^{\text{gen}}\right)^{-1} \left(B_t^{\text{gen},\top} P_{t+1}^{\text{reg}} A_t^{\text{gen}}\right)$$

Using the results from the Riccati equation, the feedback gain K_t^{reg} is calculated for all $t = 0, \ldots, T-1$:

$$K_t^{\text{reg}} = -(R_t^{\text{reg}} + B_t^{\text{gen},\top} P_{t+1} B_t^{\text{gen}})^{-1} (B_t^{\text{gen},\top} P_{t+1} A_t^{\text{gen}})$$

The LQR algorithm computes the control inputs in a feedback form, ensuring robustness. At each time step, the input is computed as a function of the deviation of the state from the reference trajectory:

$$u_t = u_t^{\text{gen}} + K_t^{\text{reg}} (x_t - x_t^{\text{gen}})$$

The system evolves according to:

$$x_{t+1} = f_t(x_t, u_t), \quad t = 0, 1, \ldots$$

To evaluate the tracking performance, the system should be tested with a perturbed initial condition, like an initial state that deviates from x_0^{gen}. The feedback controller should be able to compensate for this offset and guide the system back towards the reference trajectory, minimizing the tracking error over time.

7.2 Performance Analysis and Plots

In Task 3, to test the tracking performance, a perturbed initial condition is considered, with a state perturbation percentage.

Case	**State Perturbation Percentage**
Case 1	0.02
Case 2	0.05
Case 3	-0.1

Table 7.1: Perturbation values for different cases

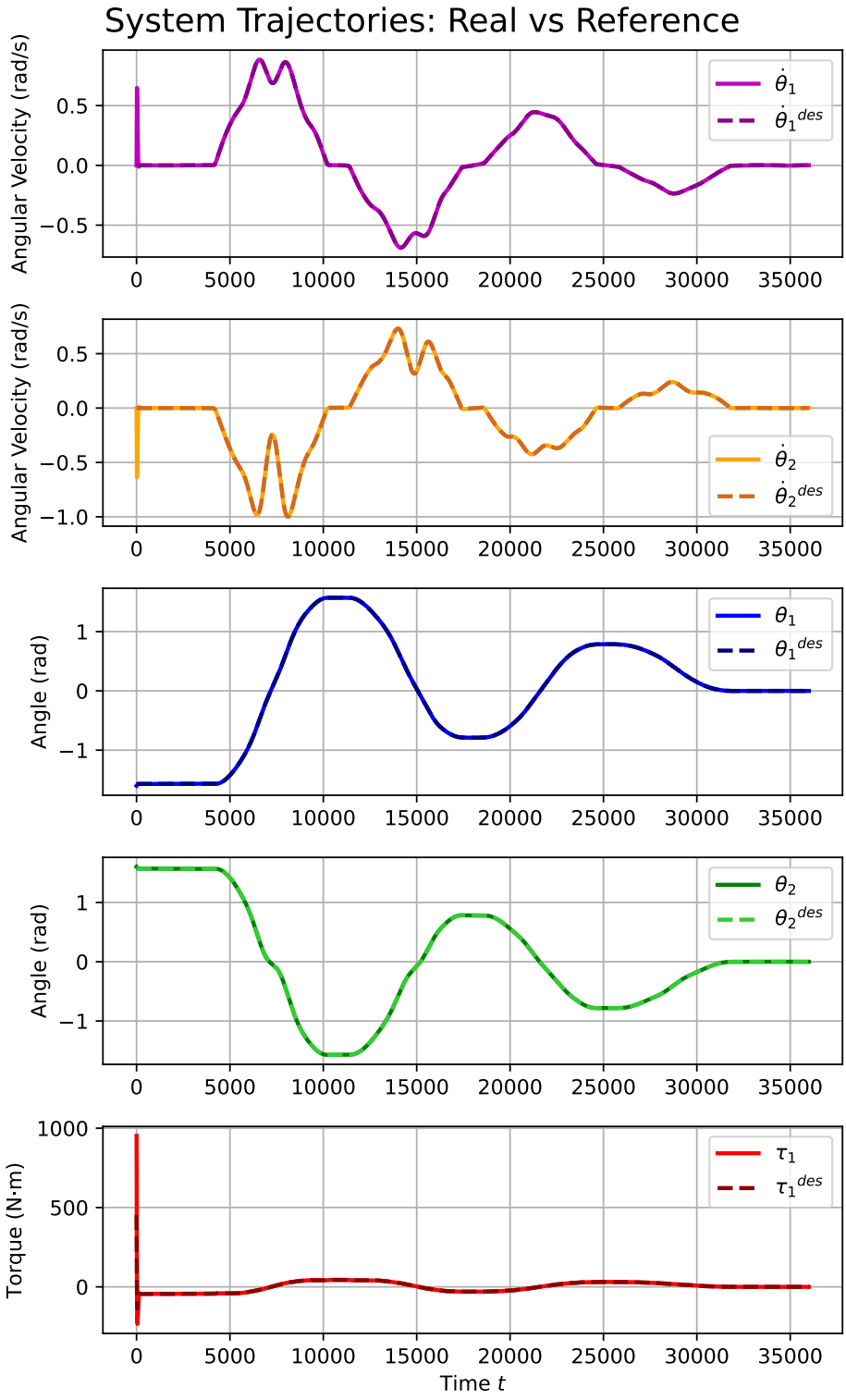


Figure 7.2: Trajectories case 1

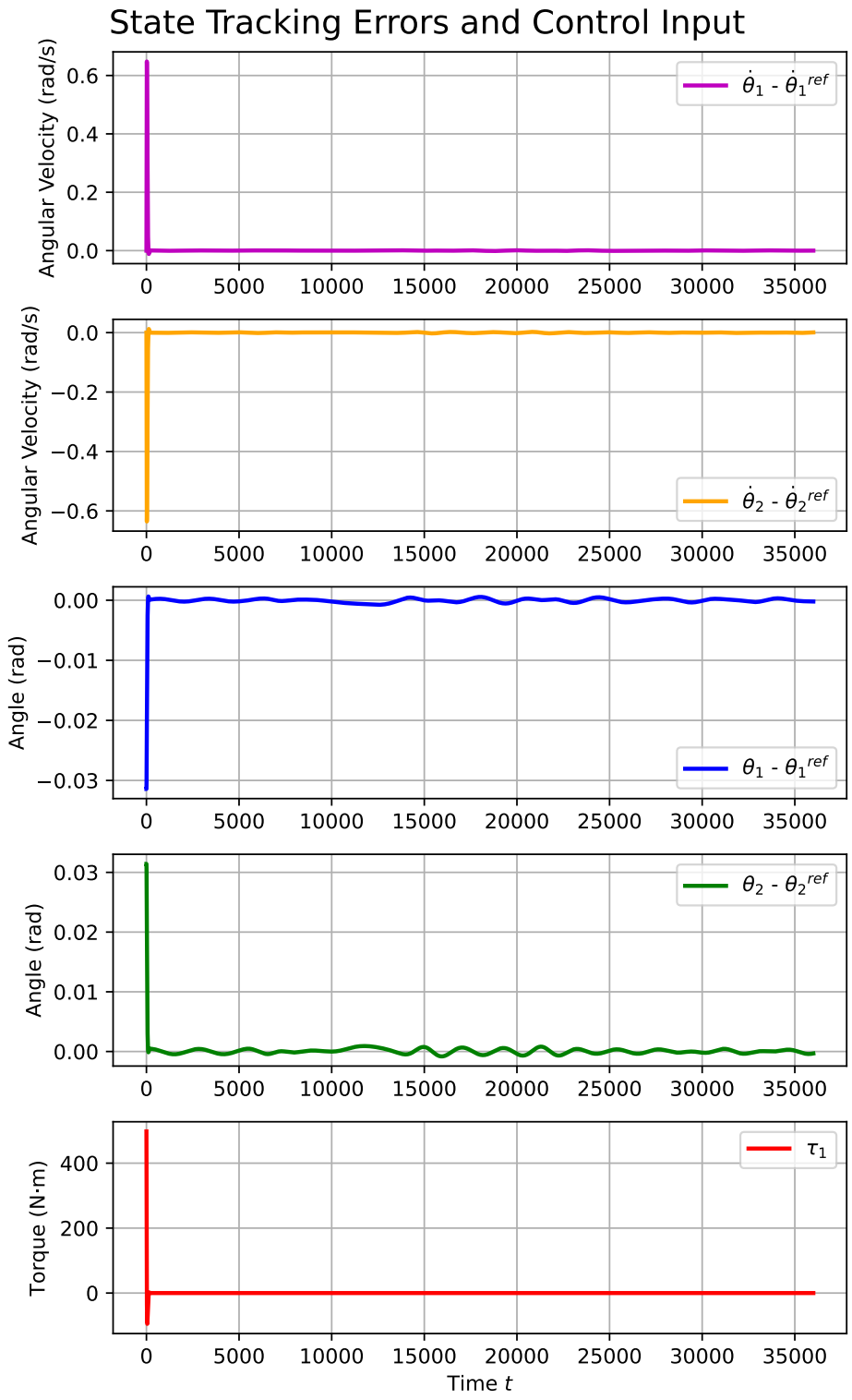


Figure 7.3: Tracking errors case 1

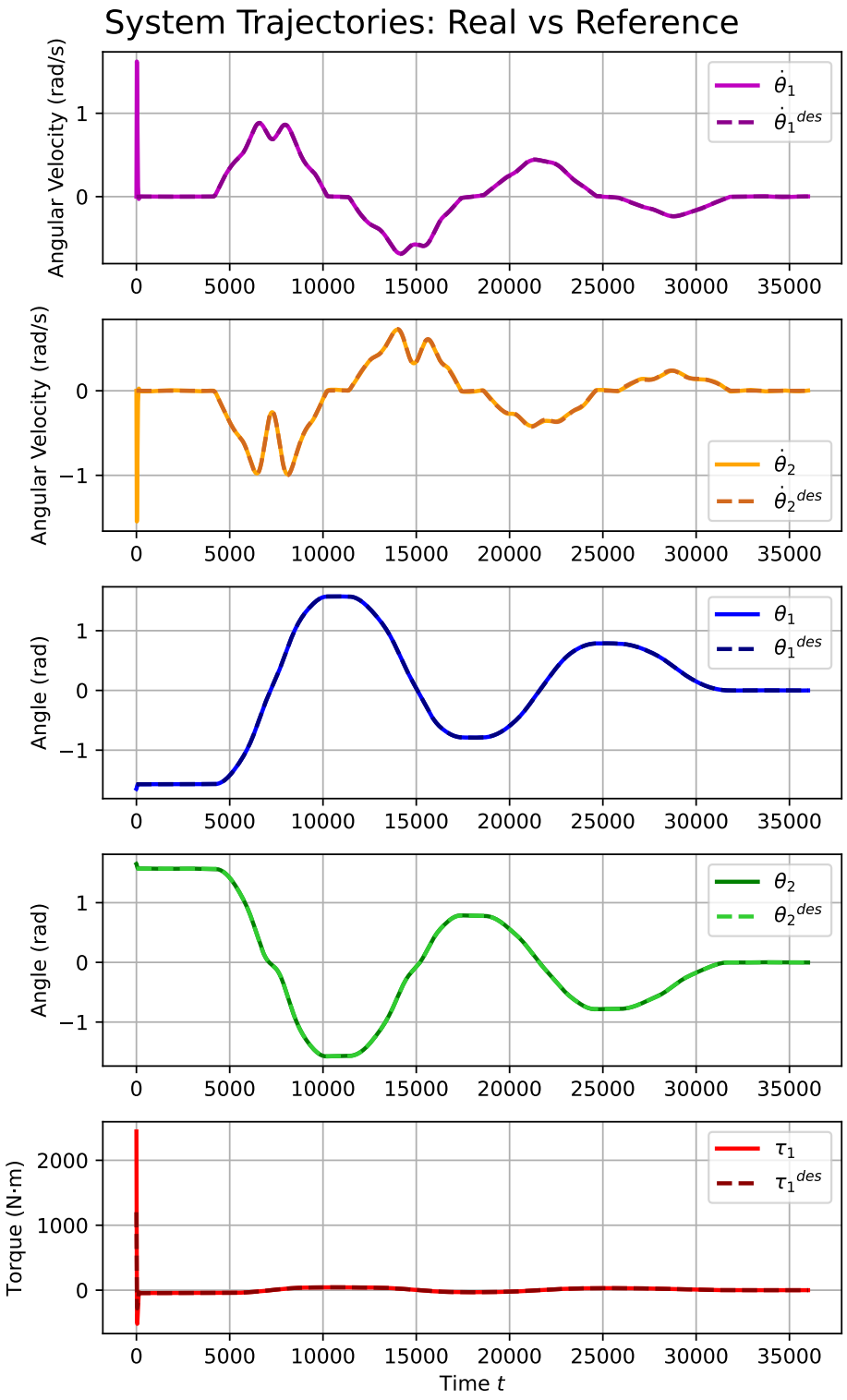


Figure 7.4: Trajectories case 2

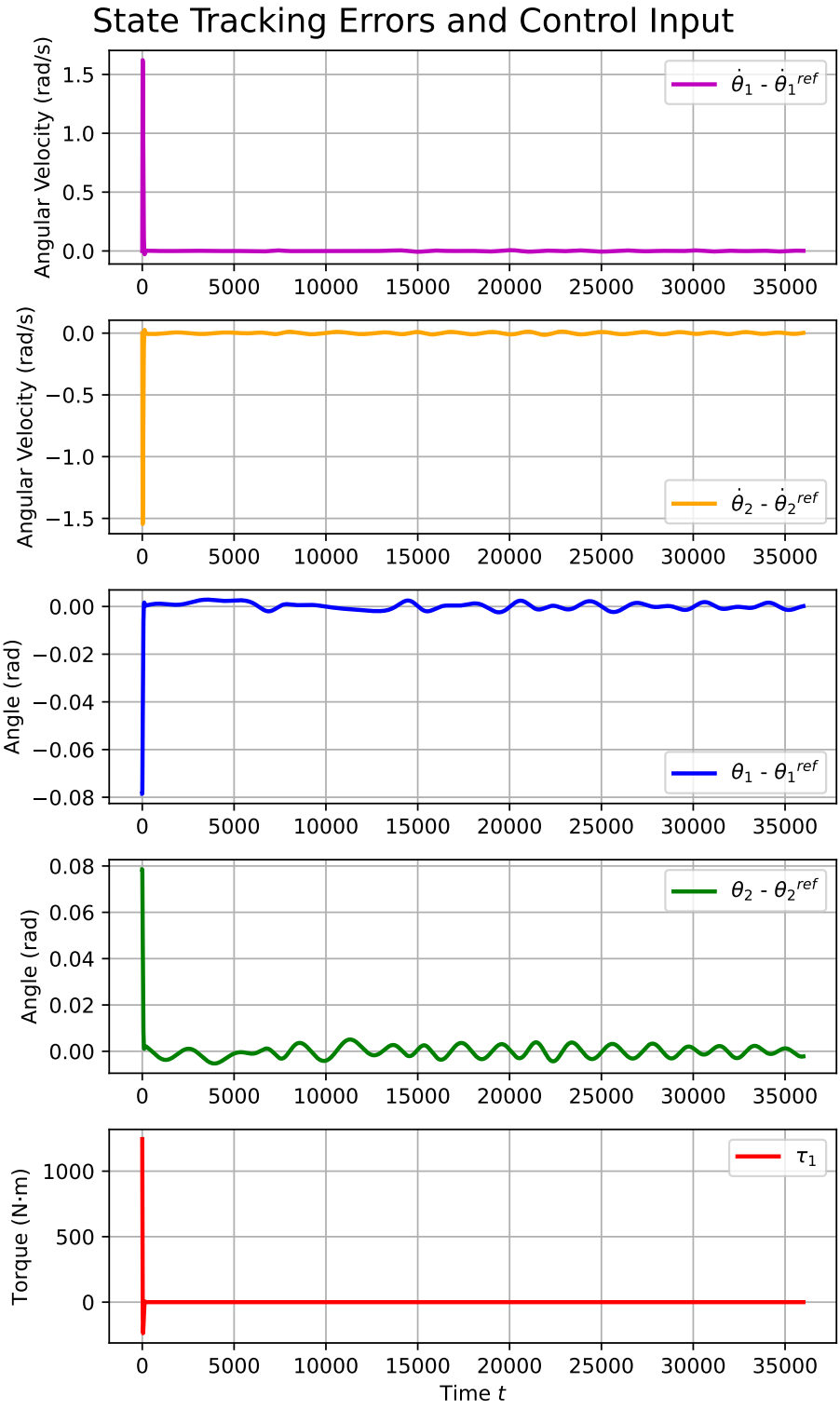


Figure 7.5: Tracking errors case 2

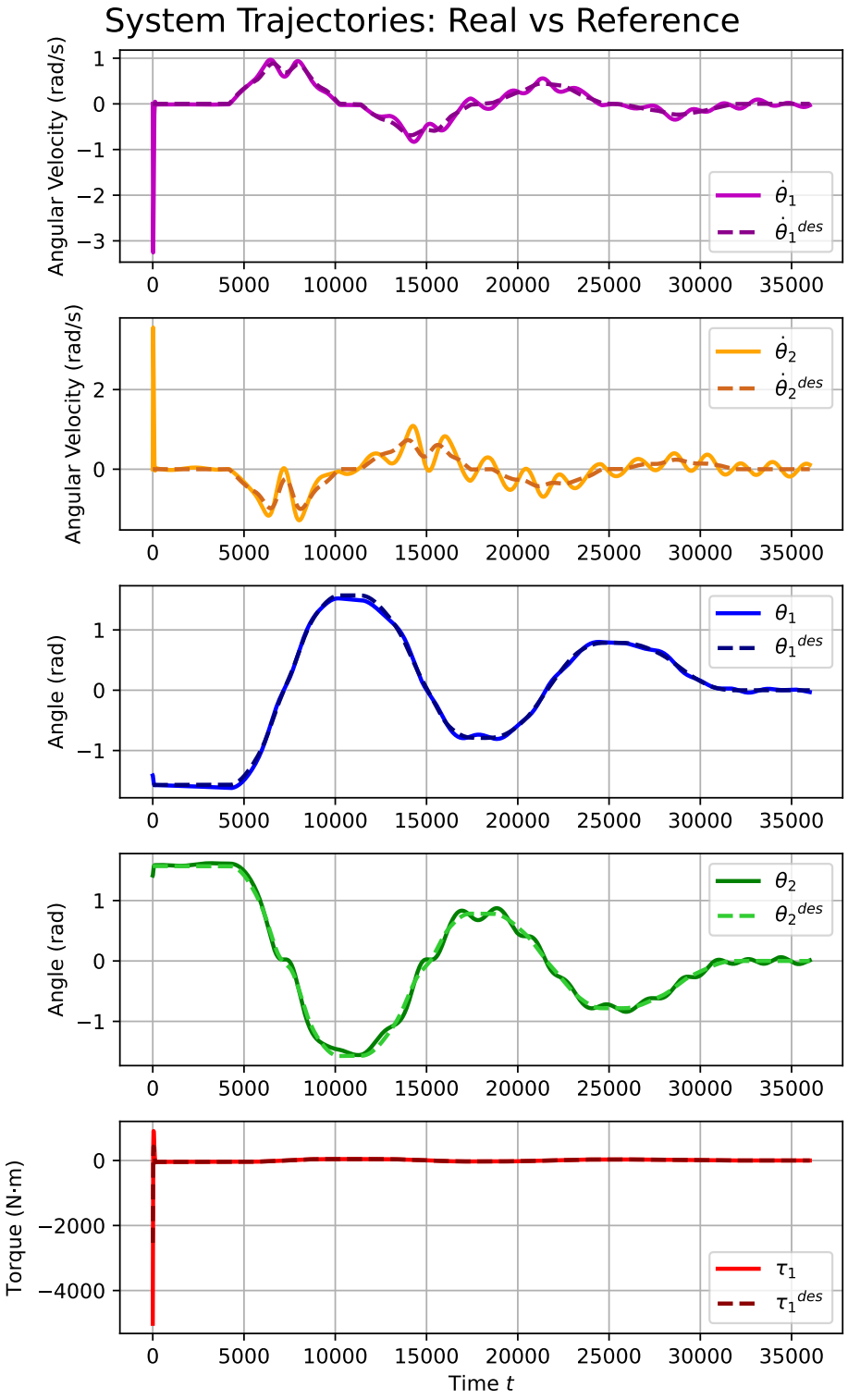


Figure 7.6: Trajectories case 3

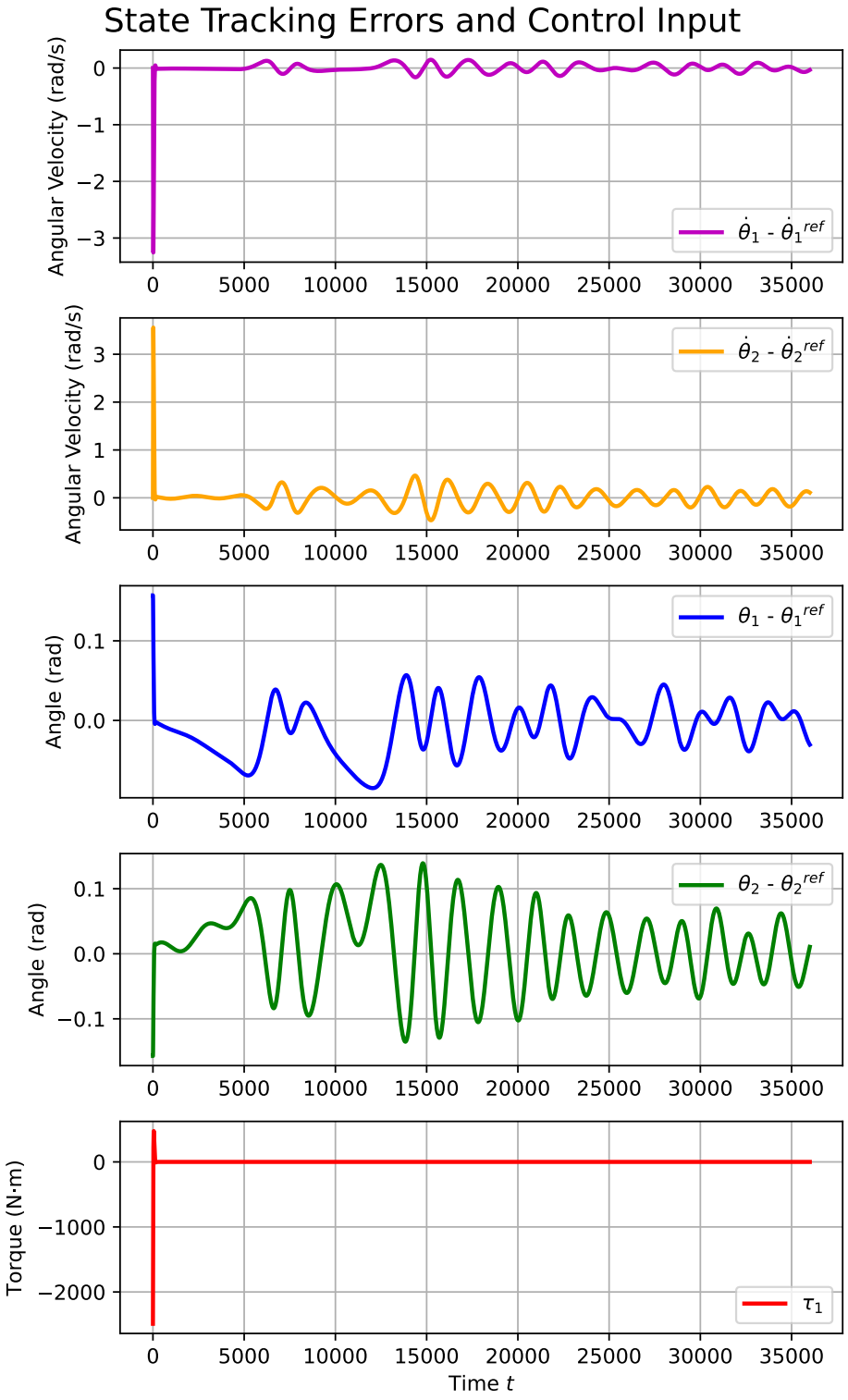


Figure 7.7: Tracking errors case 3

Chapter 8

Trajectory tracking via MPC

8.1 MPC Formulation and Implementation

The goal of this task is to apply a Model Predictive Control (MPC) framework to track the optimal trajectory $(x^{\text{opt}}, u^{\text{opt}})$, which was generated in Task 2. The MPC ensures robustness to disturbances in the perturbed initial condition x_0.

For MPC computations, the dynamics of the system is approximated around the optimal trajectory generated in the Task 2, resulting in time-dependent matrices A_t^{opt} and B_t^{opt}.

$$A_t^{\text{opt}} = \left.\frac{\partial f}{\partial x}\right|_{(x_t^{\text{opt}}, u_t^{\text{opt}})}, \quad B_t^{\text{opt}} = \left.\frac{\partial f}{\partial u}\right|_{(x_t^{\text{opt}}, u_t^{\text{opt}})} \tag{8.1}$$

The weighting matrices Q_t, R_t, and Q_T are chosen to penalize deviations in states and inputs. All these matrices are positive definite, ensuring stability and good tracking performance.

At each time step t, the initial condition for the predicted trajectory is taken as the current measured state x_t^{meas}.
The optimization problem minimizes a cost function defined over the prediction horizon T_{pred}:

$$\min_{\Delta x, \Delta u} \sum_{\tau=t}^{t+T-1} \left(\Delta x_\tau^\top Q_\tau \Delta x_\tau + \Delta u_\tau^\top R_\tau \Delta u_\tau \right) + \Delta x_{t+T}^\top Q_T \Delta x_{t+T}$$

subject to:

$$\begin{aligned} &x_{\tau+1} = A_t^{\text{opt}} x_\tau + B_t^{\text{opt}}\ u_\tau, && \tau = t, \ldots, t+T-1 \\ &x_\tau \in \mathcal{X}, \quad u_\tau \in \mathcal{U}, && \tau = t, \ldots, t+T \\ &x_t^{\text{mpc}} = x_t^{\text{meas}} \end{aligned}$$

Here, $\Delta x = x_t^{\text{mpc}} - x_t^{\text{opt}}$ and $\Delta u = u_t^{\text{mpc}} - u_t^{\text{opt}}$ represent deviations from the optimal reference trajectory.

The algorithm enforces specific constraints on the states and inputs to ensure safe and feasible operation:

$$u_{\min} \leq u_\tau \leq u_{\max}, \quad x_{\min} \leq x_\tau \leq x_{\max}, \quad \forall \tau \in [t, t+T] \tag{8.2}$$

These constraints include limits on angular velocities, angles, and control inputs.

At each time step t, the algorithm solves the optimization problem to compute the predicted control inputs and state trajectories. However, only the first control input u_t^{mpc} is applied to the system. At the next time step $t+1$, the state x_{t+1}^{meas} is re-measured, and the optimization problem is solved again. This process allows the MPC to dynamically adapt the control actions, ensuring robustness to disturbances and modeling inaccuracies.

8.2 Performance Analysis and Plots

In Task 4, to test the tracking performance, a perturbed initial condition is considered, with a state perturbation percentage. In addition, the model accounts for both measurement and actuation noise to provide a more realistic representation of real-time implementation challenges.

Case	**State Perturbation**	**Measurement Noise**	**Actuation Noise**
1	0.05	0	0
2	-0.1	0.005	0.005
3	-0.2	0.05	0.05

Table 8.1: State perturbation values, measurement noise, and attenuation noise for different cases

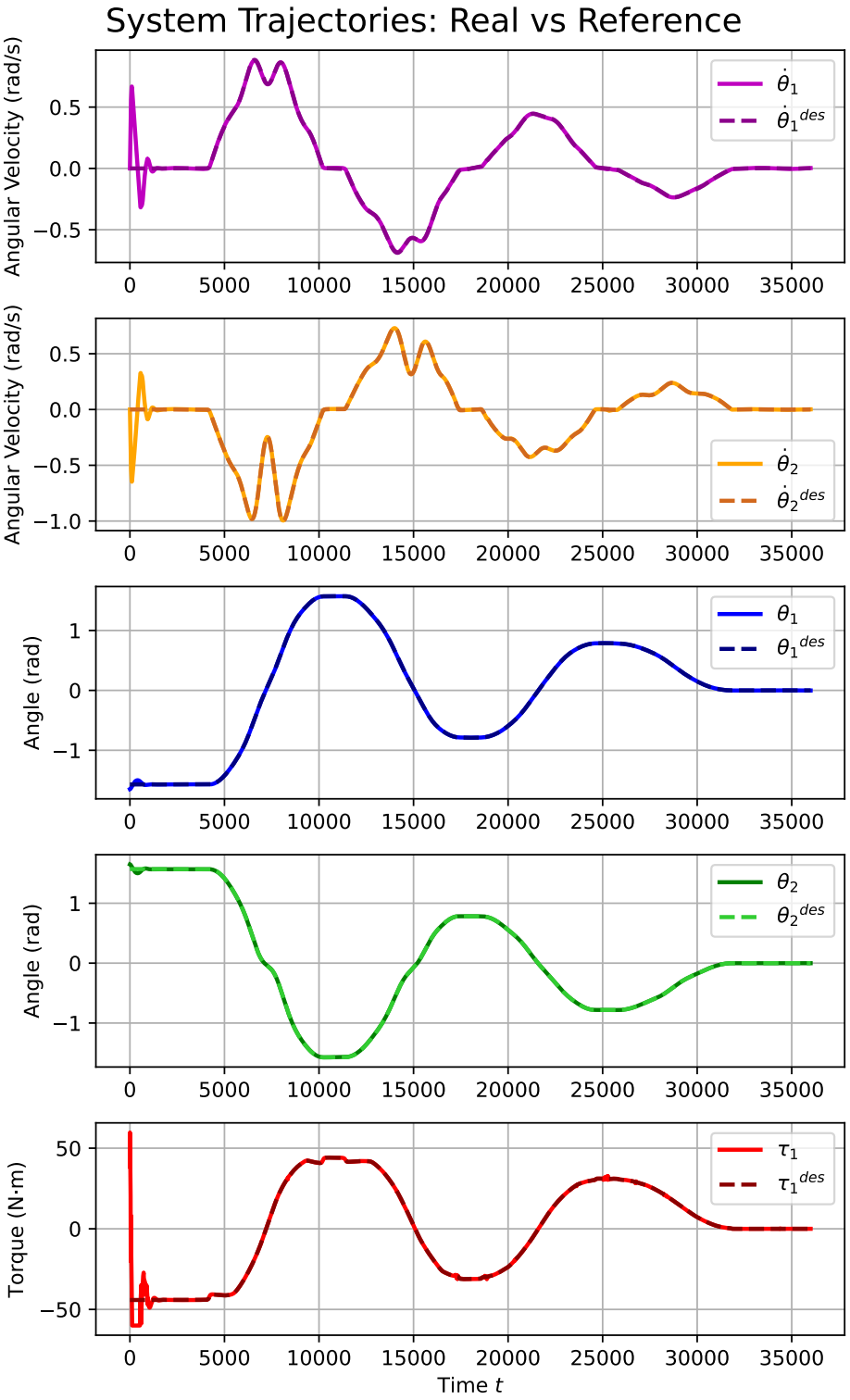


Figure 8.1: Trajectories case 1

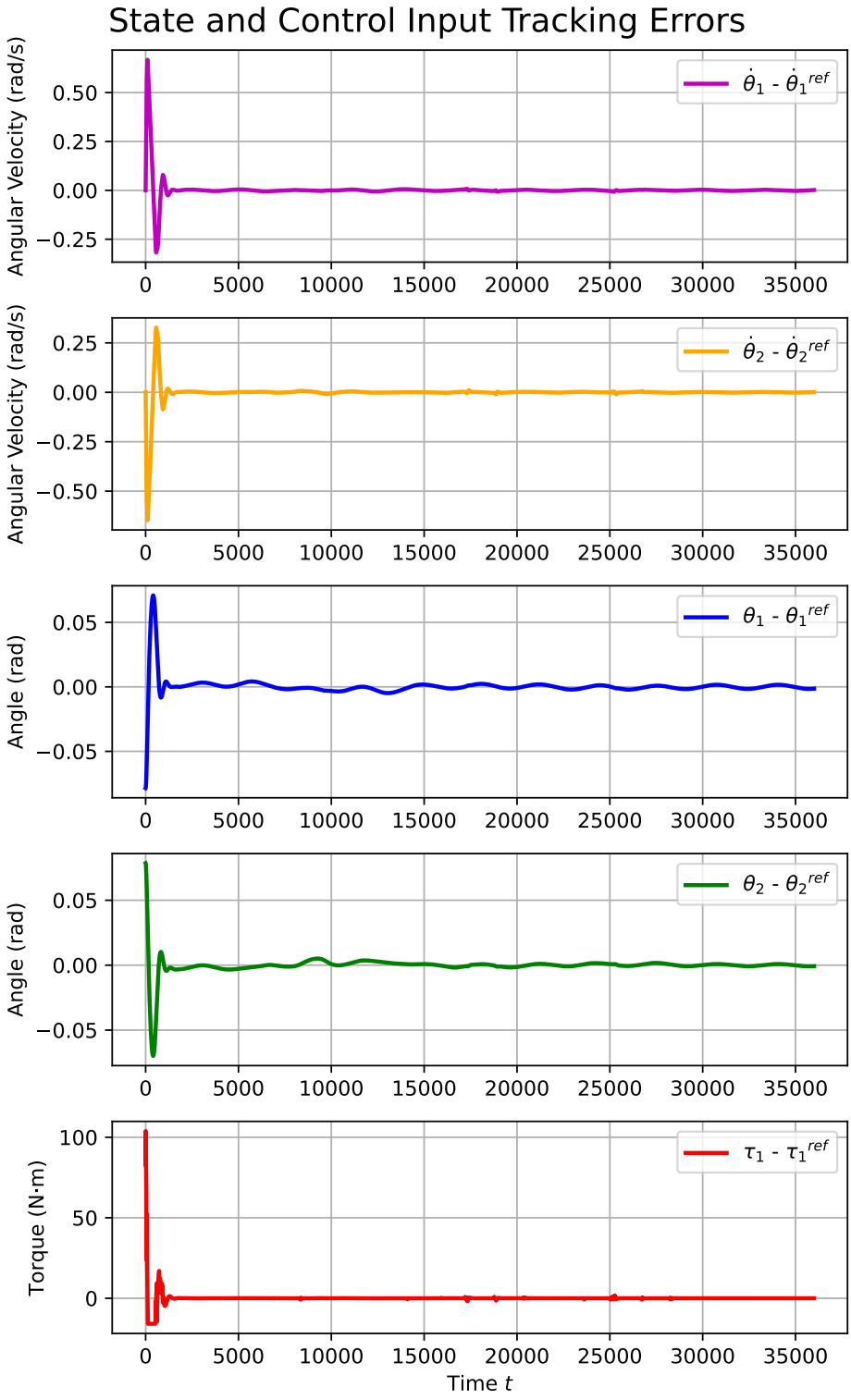


Figure 8.2: Tracking errors case 1

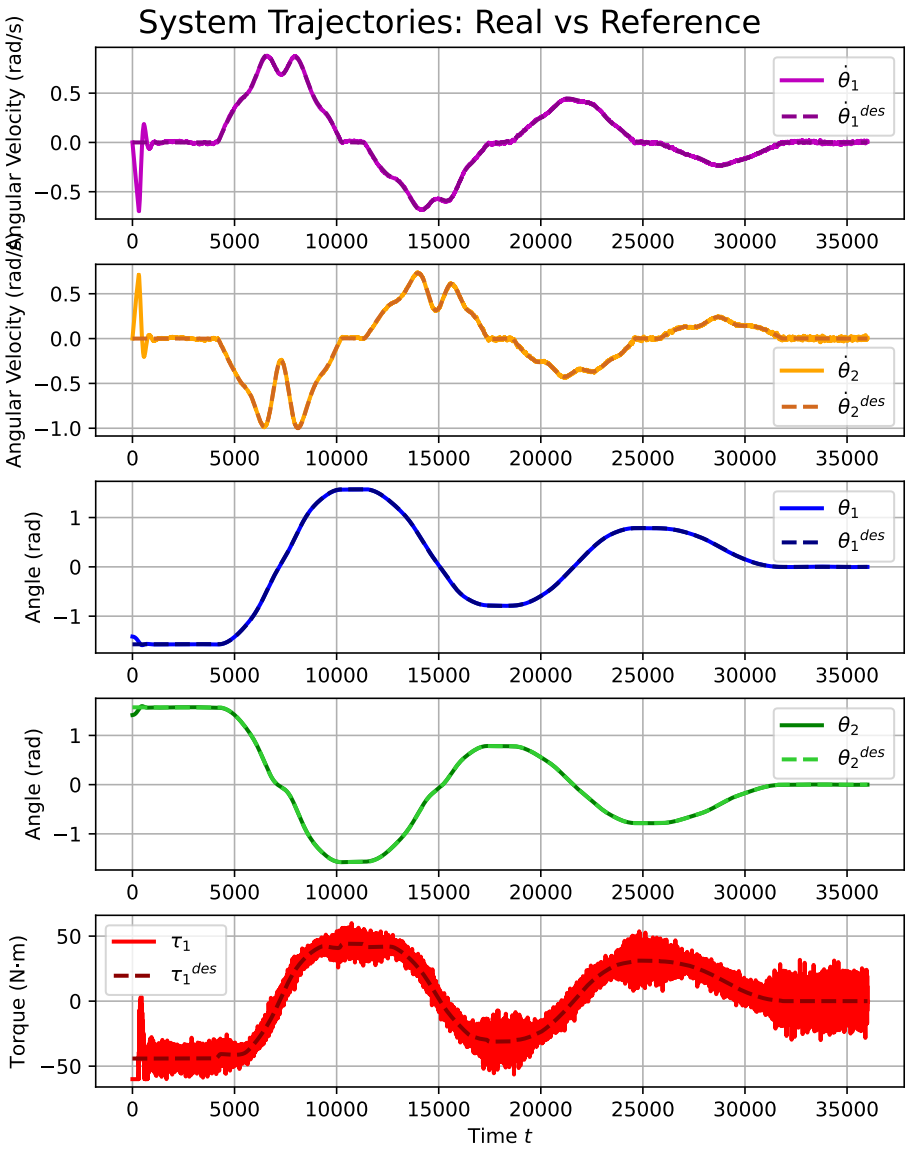


Figure 8.3: Trajectories case 2

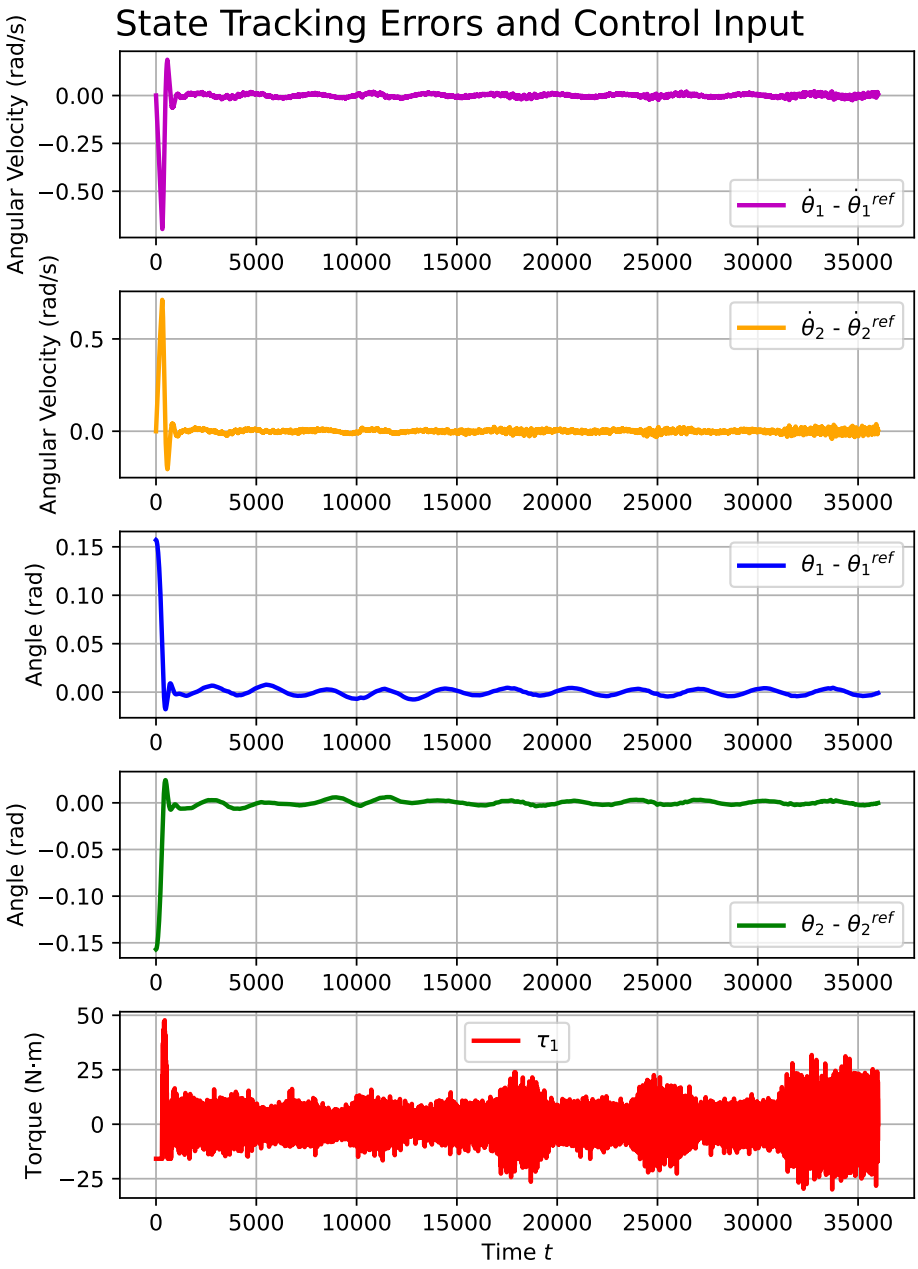


Figure 8.4: Tracking errors case 2

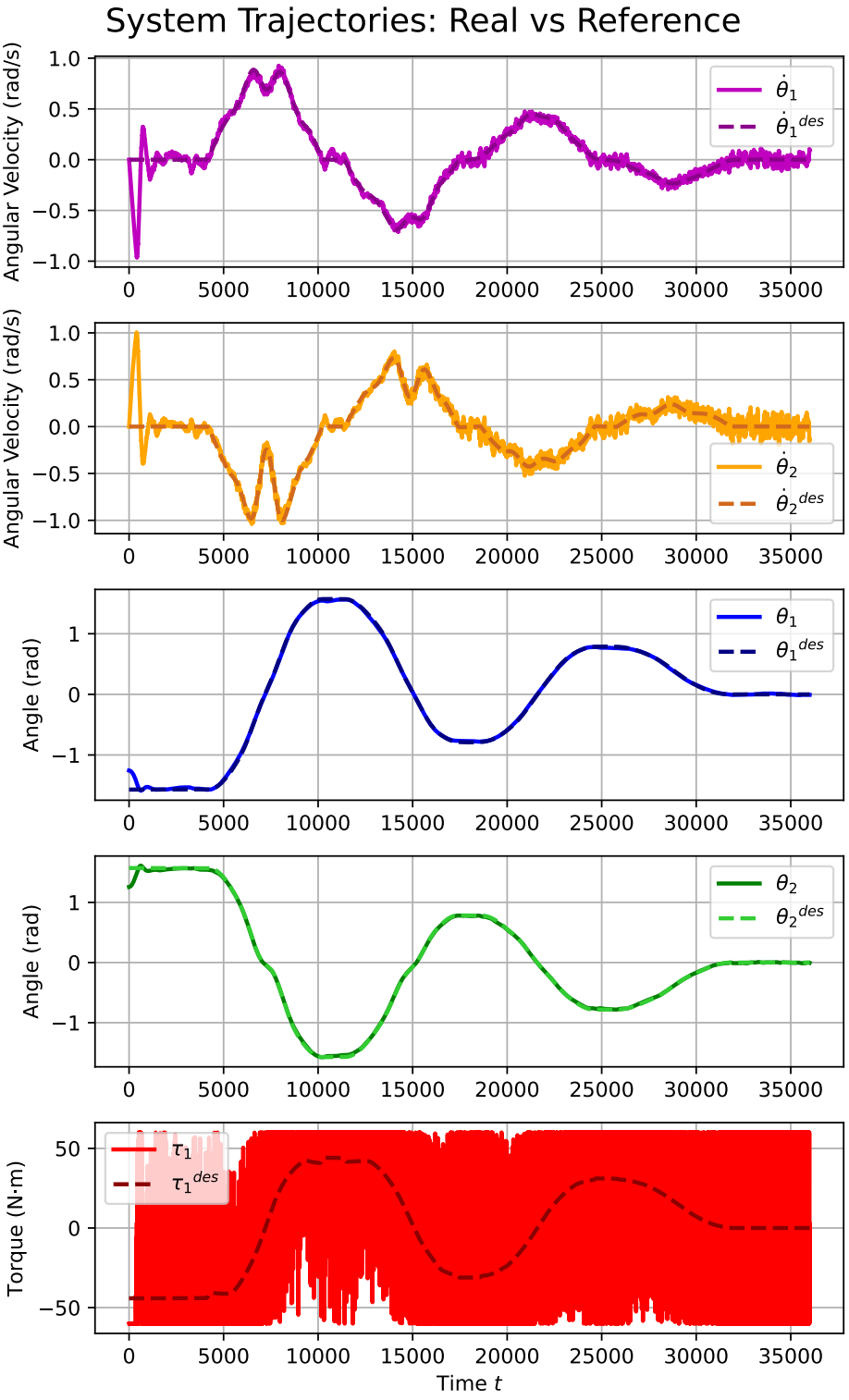


Figure 8.5: Trajectories case 3

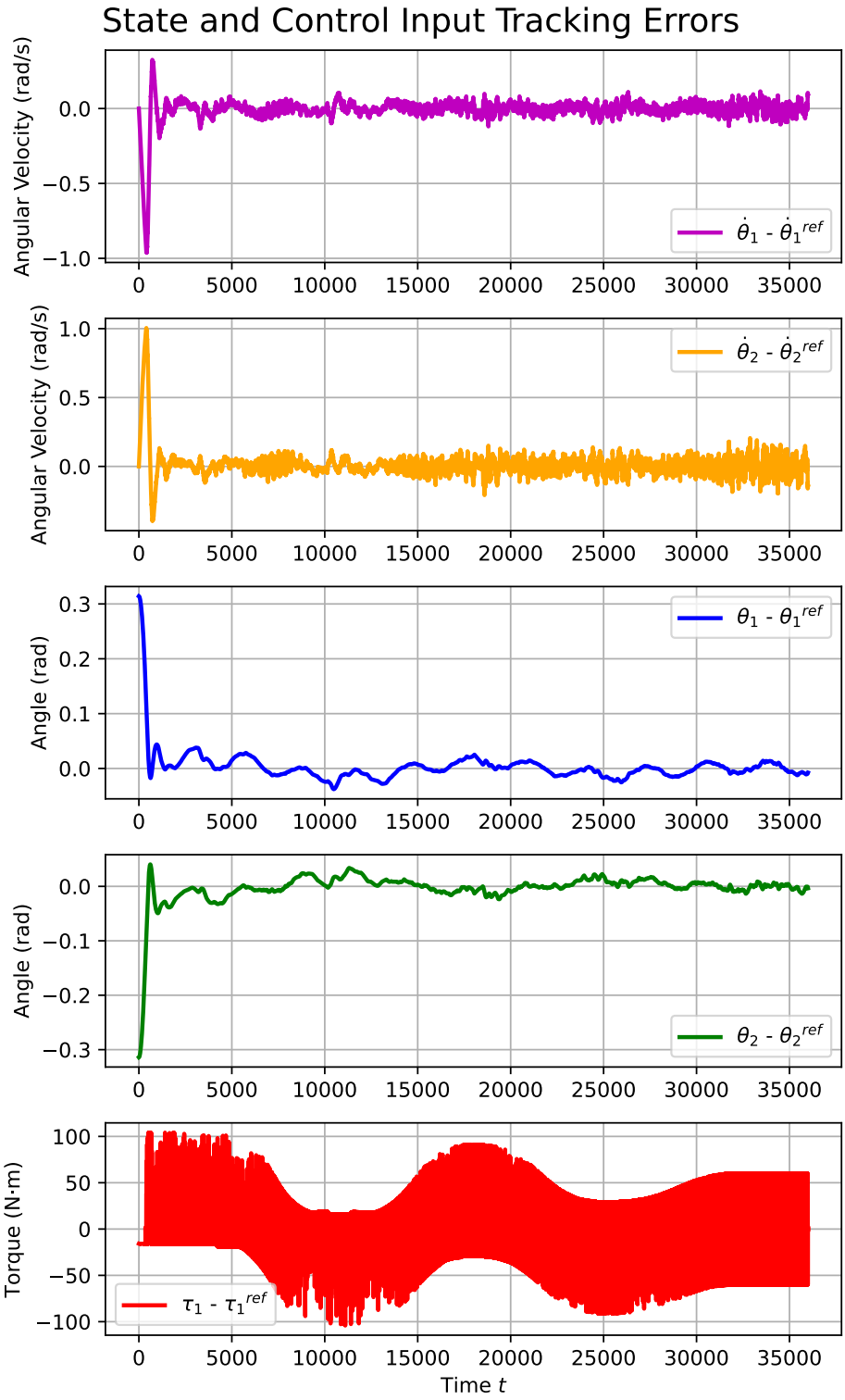
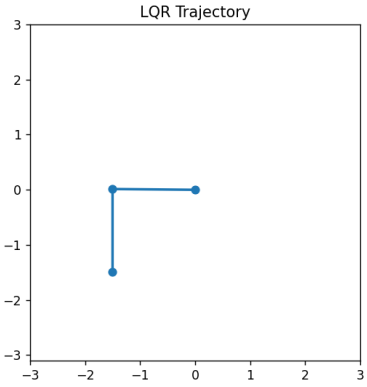


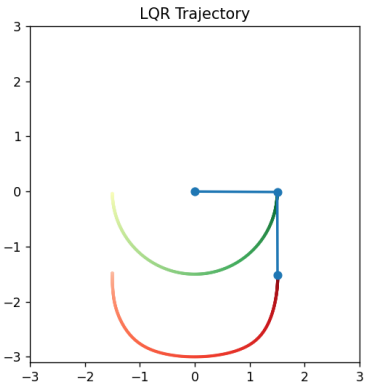
Figure 8.6: Tracking errors case 3

Chapter 9

Animation

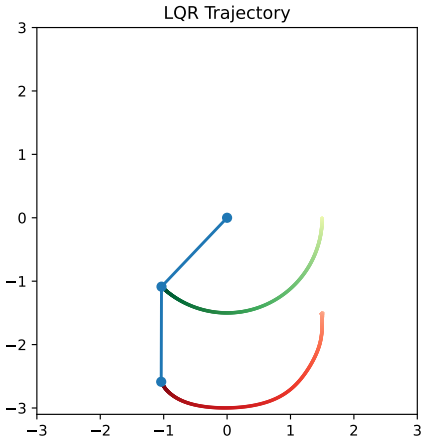


(a) Equilibrium point 1

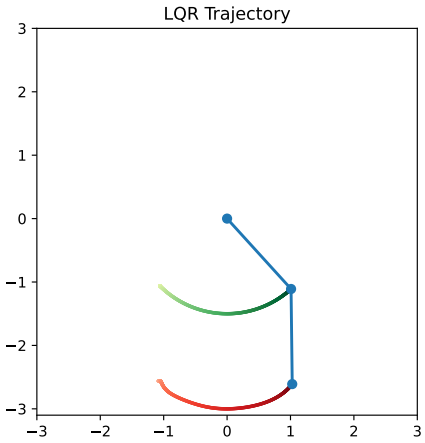


(b) Equilibrium point 2

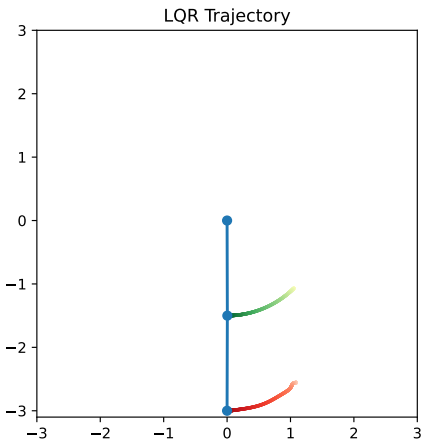
Figure 9.1



(a) Equilibrium point 3



(b) Equilibrium point 4



(c) Equilibrium point 5

Figure 9.2

Conclusions

This work addresses the optimal control of a planar underactuated two-link robotic manipulator. The system dynamics were discretized, and the system's evolution was observed to be consistent and promising. Newton's method for root finding, used to compute the equilibria, proved to be highly effective. Also the reference generation for a quasi-static trajectory also demonstrated strong performance. Additionally, the formulation of a quadratic cost function effectively penalized errors in both states and inputs.

In Task 1, although significant results were achieved, the Armijo plots were sometimes not representative in the last iterations of the algorithm. Several potential causes could explain this issue. One possibility is numerical instability, which can occur when handling extremely large or small values and the precision is crucial. The use of a step reference may exacerbate this problem due to its lack of continuity.

In Task 2 a smooth reference curve was employed. In this case, the Armijo plots appeared accurate, with the descent direction consistently tangent to the cost function. Furthermore, the plot's scale indicated that the cost function's slope decreased over time, resulting in the Armijo method taking more steps as expected when nearing convergence.

The trajectory tracking task using Linear Quadratic Regulator (LQR) control was successfully implemented to design an optimal feedback control. The results demonstrated that the controller could rapidly eliminate disturbances applied at the initial condition. Since the inputs were not constrained, the control was highly aggressive initially, quickly removing oscillations. However, when disturbances increased, some oscillations were not entirely eliminated.

Trajectory tracking using Model Predictive Control (MPC) demonstrated robustness to both state and input perturbations. In cases 1 and 2, the same disturbance values were applied at the initial condition as in cases 2 and 3 of the LQR. A comparison revealed that LQR quickly mitigated perturbations but struggled with larger disturbances. In contrast, MPC effectively compensated for even larger disturbances, as observed in case 3. Additionally, MPC managed Gaussian noise in both sensor readings and actuation effectively while adhering to constraints on input and

state variations. However, as noise levels increased, noticeable oscillations in actuation arose, a result of the controller's aggressive behavior. These oscillations could potentially be mitigated by imposing constraints on input variation over time. Another drawback of the MPC was the limited prediction window. While this constraint enables real-time applications, it reduces the controller's overall effectiveness.

The animations provided demonstrated satisfactory results across all tasks, as the references were closely followed and oscillations were eliminated through actuation.