UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Optimal Control

Course Project 1 - Group 3
Flexible Robotic Arm Assignment

Professor: **Giuseppe Notarstefano**

Tutor: **Marco Falotico**

Students:

Alessandro Letti

Vincenzo Lombardi

Giorgia Bettini

Academic year 2024/2025

Abstract

The assignment we developed over the past few weeks focuses on designing an optimal control strategy for a flexible robotic arm. The system we worked with is defined as a planar two-link structure where only the first joint is actuated. The key tasks to which we dedicated are trajectory generation between certain pre-defined equilibria using Newton's-like optimization algorithms and tracking of them using Linear Quadratic Regulator (LQR) and/or Model Predictive Control (MPC) strategies. Once the control algorithms were implemented, their performances were evaluated under perturbed initial conditions and in the presence of measurement noise. We were able to observe the results of our work through plots and animations, which provided insights into system behavior, trajectory adherence, and controller robustness.

Contents

Introduction

During the implementation of the project, we followed six different assigned tasks:

- **Task 0 - Problem Setup:** discretize the dynamics, write the discrete-time state-space equations and code the related dynamic function.
- **Task 1 - Trajectory generation (I):** compute two equilibria for your system and define a reference curve between the two. Compute the optimal transition to move from one equilibrium to another exploiting the Newton's-like algorithm (in closed-loop version) for optimal control.
- **Task 2 - Trajectory generation (II):** generate a desired (smooth) state-input curve and perform the trajectory generation task (Task 1) on this new desired curve.
- **Task 3 - Trajectory tracking via LQR:** linearizing the robot dynamics about the generated trajectory $(x_{\text{gen}}, u_{\text{gen}})$ computed in Task 2, exploit the LQR algorithm to define the optimal feedback controller to track this reference trajectory. The cost matrices of the regulator are a degree-of-freedom we had.
- **Task 4 - Trajectory tracking via MPC:** linearizing the robot dynamics about the trajectory $(x_{\text{gen}}, u_{\text{gen}})$ computed in Task 2, exploit an MPC algorithm to track this reference trajectory.
- **Task 5 - Animation:** produce a simple animation of the robot executing Task 3.

Chapter 1

Task 0 - Problem setup

1.1 System definition

The system is such that it can be modeled as a flexible arm model as shown in Figure 1.1.

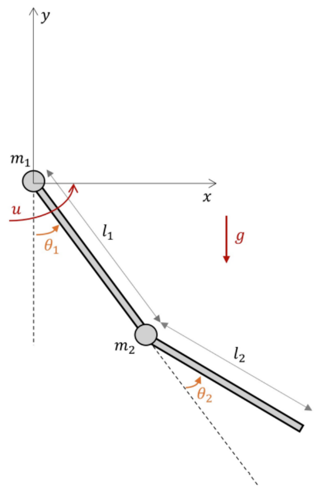


Figure 1.1: flexible arm

The state space consists in:

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

where θ_1 represents the angle of the first link with respect to the vertical direction, θ_2 represents the angle of the second link with respect to the first link, $\dot{\theta}_1$ and $\dot{\theta}_2$ the angular rates of changes associated with θ_1 and θ_2, respectively. The input is the torque u on the first link. The system dynamic is given by the equation:

$$M(\theta_1, \theta_2) \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{bmatrix} + C(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} + F \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} + G(\theta_1, \theta_2) = \begin{bmatrix} u \\ 0 \end{bmatrix} \tag{1.1}$$

where

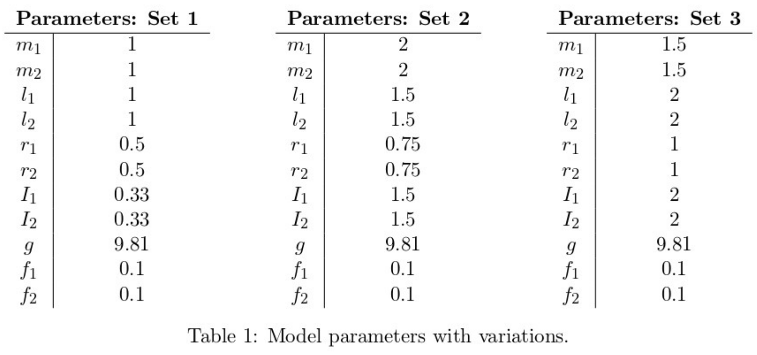
$$M = \begin{bmatrix} I_1 + I_2 + m_1 r_1^2 + m_2(l_1^2 + r_2^2) + 2 m_2 l_1 r_2 \cos(\theta_2) & I_2 + m_2 r_2^2 + m_2 l_1 r_2 \cos(\theta_2) \\ I_2 + m_2 r_2^2 + m_2 l_1 r_2 \cos(\theta_2) & I_2 + m_2 r_2^2 \end{bmatrix}$$

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

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

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

where r_1 and r_2 are the distances between the pivot points of the link and their center of mass, m_1 and m_2 are the respective masses, f_1 and f_2 are the viscous friction coefficients, and g is the gravitational acceleration. All the parameters of the robot are available in Figure 1.2. In particular our project was developed with respect to the parameters shown in the third table of that figure.



Parameters: Set 1	
m_1	1
m_2	1
l_1	1
l_2	1
r_1	0.5
r_2	0.5
I_1	0.33
I_2	0.33
g	9.81
f_1	0.1
f_2	0.1

Parameters: Set 2	
m_1	2
m_2	2
l_1	1.5
l_2	1.5
r_1	0.75
r_2	0.75
I_1	1.5
I_2	1.5
g	9.81
f_1	0.1
f_2	0.1

Parameters: Set 3	
m_1	1.5
m_2	1.5
l_1	2
l_2	2
r_1	1
r_2	1
I_1	2
I_2	2
g	9.81
f_1	0.1
f_2	0.1

Table 1: Model parameters with variations.

Figure 1.2: model parameters with variations

1.2 Computation of the discretized dynamics function

We developed this step by writing the function *discretizedDynamicFRA*() that, as it say, implements the dynamics of the flexible robotic arm (FRA) in a discretized form. In order to define it, we followed the procedure outlined below:

- Firstly, we introduced and defined the Inertia matrix M, the Coriolis forces matrix C, the Gravity forces matrix G and the Friction forces matrix F.
- Then, we proceeded by computing the first derivative of all the elements defined in the previous step.
- Lastly, we coded the actual dynamic in order to compute the state at the next istant of time (alias x_{t+1}) and we also coded the definition of the jacobians of the dynamic itself w.r.t. x and u:

$$\frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial u}$$

.

1.3 Running the dynamic forward in time

In order to be able to simulate the evolution of the dynamic forward in time given a certain initial state and a certain input curve, we defined the function *runDynamicFunction*. This function requires as arguments:

- *discretizedDynamicFunction*: this must be a well defined function that takes as arguments the state and input values at time t, and returns the state value at time $t+1$ (alias $\mathbf{x}_{t+1}$) and all jacobians of the dynamics with respect to state and input (such it is for *discretizedDynamicFRA*), in the following order:

$$\mathbf{x}_{t+1}, \quad \frac{\partial f}{\partial \mathbf{x}}, \quad \frac{\partial f}{\partial \mathbf{u}},$$

- *uu*: the input curve.
- xx_0: the fixed initial state.
- TT: number of time steps (each with duration dt), sufficient to evolve from $t=0$ to $t=T$, where $[0,T]$ is the considered time horizon.

In a nutshell, we are initializing the state trajectory $\mathbf{x}$ with the given initial state $\mathbf{x}_0$ and then iterating a loop through $t = 0$ to $T-1$ relying on the given input $\mathbf{u}$ and on the *discretizedDynamicFunction* to generate the state at each time step.

Chapter 2

Task 1 - Optimal trajectory between two equilibria

2.1 Task description

The first aim of task 1 is to compute two equilibria for our system and define a reference curve between the two. Then we have to proceed by computing the optimal transition to move from one equilibrium to another exploiting the Newton's-like algorithm (in closed-loop version) for optimal control.

2.2 Equilibria computation and definition of the reference curve between them

For what concern the first part of this task, we define two equilibrium points thanks to the function implemented inside the equilibria.py file, $searchFRAInputGivenAnEquilibria$, that has as argument the scalar equilibrium value for x_0 (alias $-x_1$) and returns the scalar input value that generates the requested equilibrium point.
Once the equilibrium points are defined we proceed by defining the reference curve between them. Firstly, we decide on the time instants in which the desired input-space curve will evolve, opting for $T = array([0, 5, 11, 16])$. Then we defined our desired state curve, thanks to the function $generateCurves(xxValues, uuValues, ttValues, curveTypeValues)$, as an exponential spline (generated in the function $exponentialSpline(t_1, t_2, v_1, v_2, dt)$), where the single spline's parameter has been tuned to minimize the variance of the curvature (normalized w.r.t its mean value) of the curve itself. After that, the input curve is generated by considering the angle of the first link of the FRA (at each time instant) and by searching the corresponding input value that leads to the equilibrium in which the first link is placed like that and the second link is pointing downwards (alias at the same angle but with opposite sign).
All of that leds to a behaviour of the following type: from 0 to 5 seconds we have a constant section at the first equilibrium point, then from 5 to 11 seconds we have the evolving section, in which our curve reaches the second equilibrium point, lastly, from 11 to 16 seconds we continue with another constant section. We can conclude that our desired trajectory will vary inside a range that goes from -60 degrees to +60 degrees, for a total swing of 120 degrees. Before we proceed, let's take a minute to analyze the function $generateCurves(xxValues, uuValues, ttValues, curveTypeValues)$. As said before, is used for the generation of curves for states. These curves are generated interpolating

the given points at the given times. Each segment of the curve can be generated by using a sigmoid, a cubic spline or an exponential spline. This function takes as arguments:

- $xxValues$ = ns x p sequence of states values to interpolate (where p is the numberOfPoints)
- $uuValues$ = ni x p sequence of inputs values to interpolate (if None, only the states curve is generated)
- $ttValues$ = 1 x p sequence of times values (ttValues[i] and ttValues[i+1] are the time instants of the i-th segment, with i from 0 to p-1)
- $curveTypeValues$ = 1 x (p-1) sequence of spline types to use for each segment (this value can also be passed as a scalar; in this case the same provided spline type is used for all the segments)

Also, the function returns:

- xx = ns x TT sequence of states values (where TT is the number of time steps and is computed as int(ttValues[-1]/dt))
- uu = ni x TT sequence of inputs values
- t = 1 x TT sequence of time values

2.3 Optimal trajectory via Newton's like algorithm

Once that we complete the first step we can proceed by defining the cost matrices (that define the cost function) for the trajectory tracking optimization problem, this step is needed to represents how accurately we want to track a variable and allow us to apply the Newton's method later on. We choose a matrix Q = diag[12, 12, 12, 12], used for the state variables, that we can describe as a diagonal matrix with 12 as both positions and velocities costs, and R = 0.001*eye(ni), used for the input variable, that we can describe as a diagonal matrix with 0.001 on the diagonal used for the input variable. Notice that R will be a one element matrix since we have just one input. After defining a maximum number of iteration for the newton's method and a tolerance for it, we can now run the Newton's method itself in order to minimize the cost function. In it we are using a regularized approach to avoid computing the Hessians of the dynamics. Let's see now, specifically, the function that implements the Newton's method and all the other functions that are involved within it:

- $runNewtonMethodTrkTrj(xx_{des}, uu_{des}, maxIterations,$ $discretizedDynamicFuntion, tolerance, QQ, RR, QQT = None,$ $fixedStepsize = None)$
  This function contain the overall newton's like method in closed loop version for an optimal control trajectory generation problem. The arguments of this function are:
  - xx_{des} = a column vector state desired curve of dimension ns*TT where ns is the number of states of the system. We can describe its usage as: for t=0 we have the initial state xx_0, then we have the state curve for t from 1 up to T-2, then value for t=T-1 is considered as the state terminal value.

- uu_{des} = a column vector input desired curve of dimension ni*TT where ni is the number of inputs of the system. We can describe its usage as: for t from 0 to T-2, we have the input curve; for t=T-1 we MUST have the input value that MUST be of equilibria for the system dynamic if considered within the terminal state value; it is also important that the state-input couple ad time t=0 is an equilibrium for the system dynamic.
- $maxIterations$ = maximum number of allowed iterations for the method to converge.
- $discretizedDynamicFunction$ = this function implements the discretized dynamics of the system that is being considered, requiring as arguments respectively the state and input values at time t, returning the state value at time t+1 and all the jacobians of the dynamic with respect to state and input in the following order: xxp, dfdx, dfdu.
- $tolerance$ = minimum value that the norm of the descent direction has to reach to consider the optimization as converged and completed.
- QQ = ns*ns stage state cost matrix, if it is time invariant, or ns*ns*TT stage state cost tensor, if it is time variant.
- RR = ni*ni stage input cost matrix (if time invariant) or ni*ni*TT stage input cost tensor (if time variant).
- QQT = ns*ns terminal state cost matrix. Note that if None, the solution of the ARE (Algebric Riccati Equation) at t=T-1 is used for the terminal cost.
- $fixedStepsize$ = fixed stepsize to use for the optimization, if none option is present, the Armijo's rule is exploited to compute the stepsize.
- $generateNicePlots$ = flag to enable the generation of nice plots for the Armijo's rule (if True, the method generates the plots)

For what concern the elements that it returns, we have:

- xx = column vector state of a feasible trajectory obtained through the optimization. It is coupled with uu in order to return a state-input trajectory.
- uu = column vector input of a feasible trajectory obtained through the optimization. It is coupled with xx in order to return a state-input trajectory.

- $solveCostateEquation(xx, uu, xx_{des}, uu_{des}, discretizedDynamicFuntion, stageCostFunction, termCostFunction, TT)$ = implementation of the backwards-in-time solution of the costate equation of an Optimal Control Problem. The arguments of this function are:
  - xx, uu, xx_des, uu_des, $discretizedDynamicFunction$ all previously defined.
  - $stageCostFunction$ = this is the function that computes the cost associated with each stage of the process. In an optimal control problem, each time step or stage has a cost that depends on the current state and the control input. Thus, this function, represents the cost at each time step of the optimization process.
  - $termCostFunction$ = Instead, this function computes the terminal cost, which is the cost associated with the final stage of the optimization problem. In optimal control problems, there is often a terminal cost that depends only on the final state of the system. Therefore, it takes the final state xx_{des} as input and returns the terminal cost associated with that state.

- TT = is the time horizon of our problem.

This function returns:

- $lmbda$ = the costate trajectory, alias the solution of the costate equation (lmbda)
- AA, BB = the jacobians of the dynamic w.r.t. x and w.r.t. u at x,u at each time step (respectively AA and BB)
- $QQdyn, SSdyn, RRdyn$ = the transposed hessians of the dynamic w.r.t. x and w.r.t. u at x,u at each time step (Qdyn as d2fdxdx, Sdyn as d2fdxdu, Rdyn as d2fdudu)
- qq, rr = the transposed jacobians of the stage cost w.r.t. x and w.r.t. u at x,u at each time step (respectively q and r)
- qqT = the transposed jacobian of the terminal cost w.r.t. x at the terminal state value (alias qT)
- $QQtilde$, $RRtilde$, $SStilde$ = the [transposed] hessians of the stage cost w.r.t. x and w.r.t. u at x,u at each time step (Qtilde as d2lldxdx, Stilde as d2lldxdu, Rtilde as d2lldudu)
- $grdJdu$ = the gradient of the cost function (expressed as only a function of the input) at xx,uu at each time step (alias grdJdu)
- ll = the acutal cost associated to the given trajectory xx,uu having xx_des, uu_des as desired curves (alias l)

- $solveAffineLQP(AA, BB, QQ, RR, SS, QQT, TT, xx_0, qq, rr, qqT)$
  Implementation of the solution of an affine linear quadratic optimal control problem solver.

- $solveLQP(AA, BB, QQ, RR, QQT, TT, xx0)$
  Implementation of the solution of a linear quadratic optimal control problem solver.

- $armijoStepSize(uu, xx, xx_{des}, uu_{des}, ll, direction, grdJdu, KK, sigma,$ $TT, discretizedDynamicFuntion, stageCostFunction, terminalCostFunction,$ $stepsizeInitialGuess = None)$
  Armijo's rule for step size selection, this is used to find an appropriate step size that ensures sufficient decrease in the cost function.The loop terminates when a satisfactory step size is found, or after a maximum number of iterations. If no satisfactory step size is found, the function selects the best step size from those tested. The arguments of our function are:

  - $uu, xx, xx_{des}, uu_{des}, ll, grdJdu, KK, sigma, TT,$ $discretizedDynamicFuntion, stageCostFunction, terminalCostFunction$ all previously defined.
  - $direction$ = a direction vector representing the direction in which to perturb the current control trajectory (uu) in order to minimize the cost.
  - $stepsizeInitialGuess$ = an optional initial guess for the step size. If not provided, a default value of 1 is used.

  The function returns:

  - $stepsize$ = the selected step size based on Armijo's rule. This is the final step size that satisfies the Armijo condition for sufficient decrease in the cost function.

  - $armijoStepsizes$ = a list of the stepsizes that were tested during the Armijo line search process.
  - $armijoCosts$ = a list of the corresponding costs evaluated at each tested step size.

- $solveARE(A, B, Q, R, S)$:
  Used to solve the Algebraic Riccati Equation (ARE), which is a key equation in optimal control theory, particularly in the Linear Quadratic Regulator (LQR) and other control problems. The arguments of this function are:

  - A = state transition matrix
  - B = control input matrix
  - Q = state cost weight matrix in the quadratic cost function
  - R = control cost weight matrix in the quadratic cost function
  - S = additional matrix that is included in the augmented system. It is used to account for additional cross-coupling between state and control inputs. If S is provided and contains non-zero values, the system is augmented and solved accordingly.

  The function returns the solution to the Algebraic Riccati Equation (ARE) or the augmented ARE, which is the optimal cost matrix P.

- $updateInputStateTrajectory(ns, ni, xx_0, uu_{old}, xx_{old}, stepsize, deltau,$
  $KK, sigma, TT, discretizedDynamicFuntion)$:
  This function updates the control inputs and state trajectory for a given system using a step size and perturbations to the control input. It handles both open-loop and closed-loop versions depending on the availability of feedback control parameters (K, sigma). Its arguments are:

  - ns, ni = number of states and inputs of the system
  - $xx0$ = initial state vector
  - xx_{old}, uu_{old} = previous state and control trajectory over time, which represents the state and control inputs at each time step.
  - $stepsize$ = The step size used to scale the control perturbation
  - $deltau$ = The perturbation (change) to the control inputs, typically a matrix of size (ni, T), which indicates how the control inputs will be adjusted.
  - $KK, sigma, TT, discretizedDynamicFunction$ = all previously defined.

  The function returns:

  - uu_{new}, xx_{new} = the updated control and state trajectory over time.

In addition, we define the class $TrjTrkOCPData$ containing the following methods: $setEndingTime$, $getElapsedTime$, $getOptimalTrajectory$, $getOptimalCostGradient$, $getOptimalTrajectoryErrorsAtFinalTime$.

Now we can look at the required plot obtained from all this work.

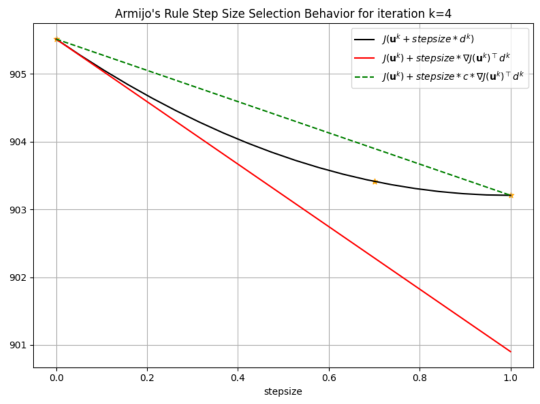


Figure 2.1: Armijo's rule for k = 4

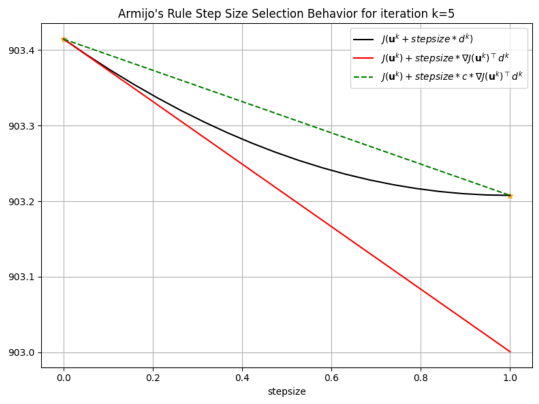


Figure 2.2: Armijo's rule for k = 5

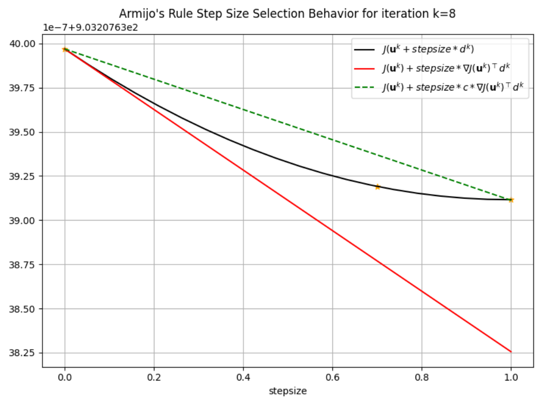


Figure 2.3: Armijo's rule for k = 8

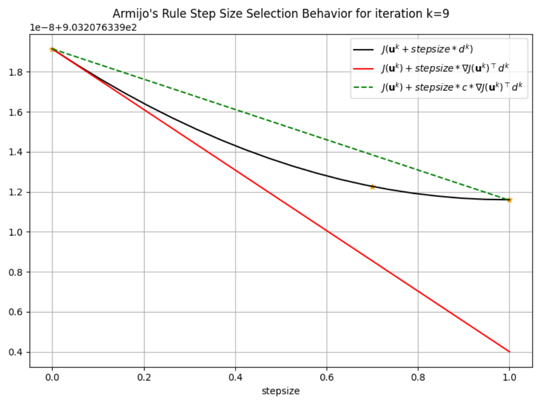


Figure 2.4: Armijo's rule for k = 9

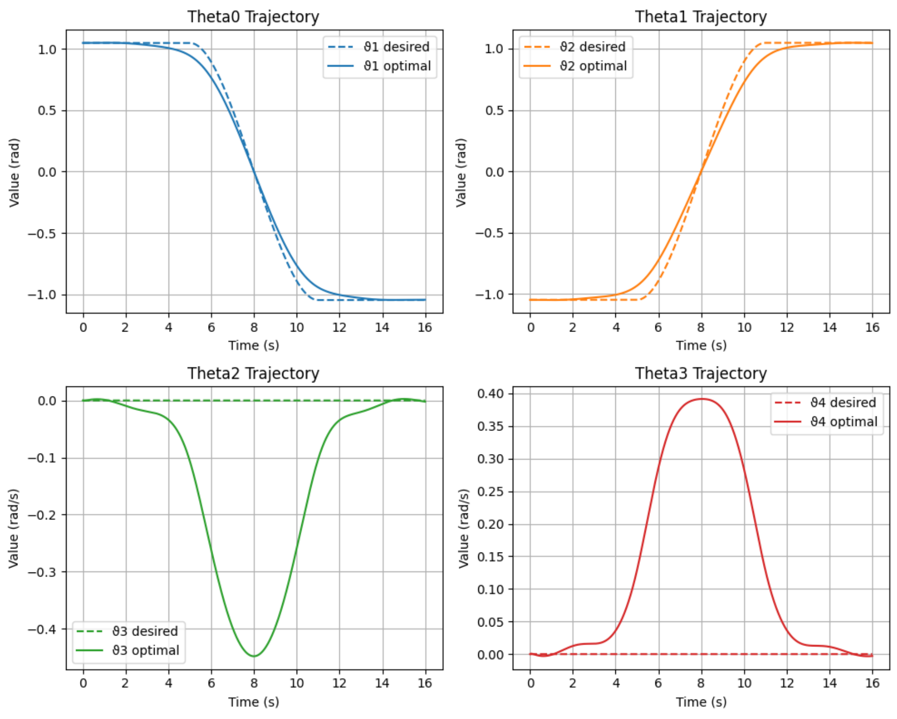


Figure 2.5: Desired and optimal trajectories

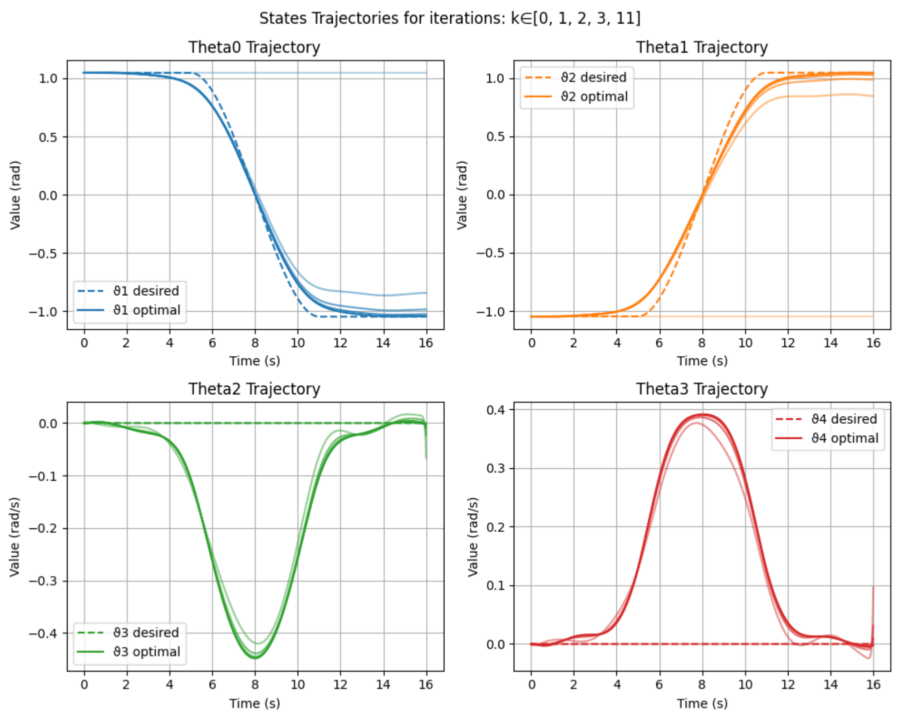
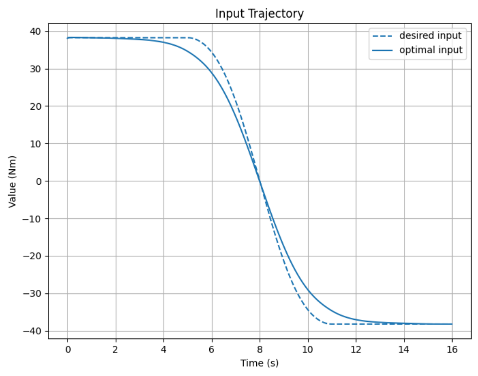


Figure 2.6: Intermediate iteration to obtain the optimal trajectories



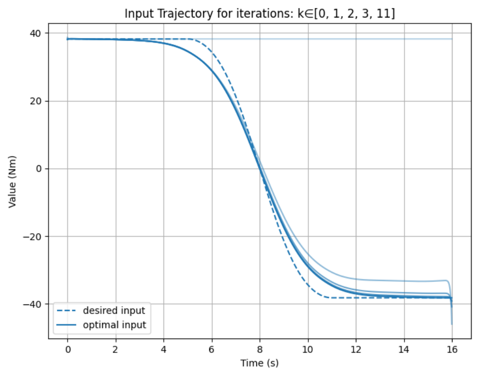


Figure 2.7: Desired and optimal input trajectories

Figure 2.8: Intermediate iteration to obtain the optimal input trajectory

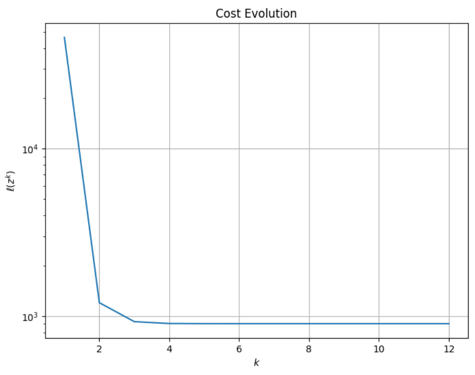


Figure 2.9: Obtained decrease for the cost function evolution

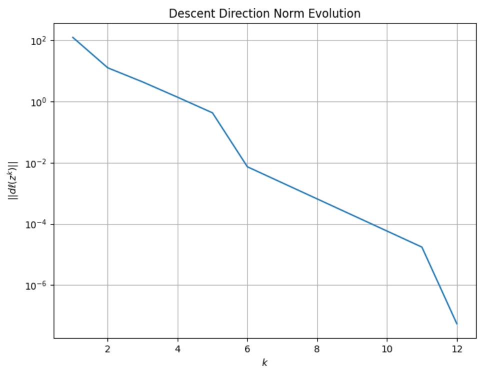


Figure 2.10: Obtained decrease for the norm of descent direction

2.4 Conclusions

The results demonstrate the effectiveness of using a Newton-like algorithm in a closed-loop framework to compute the optimal control trajectories for transitioning between two equilibria of our system. All the plots are a demonstration of what just said, let's focus on them individually. The descent direction norm evolution plot confirms the rapid convergence of the optimization algorithm within 10 iterations, indicating efficient optimization. The cost evolution plot further proves this efficiency, as the cost function quickly stabilizes after the initial iterations. The intermediate trajectory plots for the system variables ($theta0, theta1, theta2, theta3$) show that the desired and optimal trajectories align closely, ensuring smooth transitions between the equilibria while respecting the constraints. Additionally, the input trajectories demonstrate precise control inputs that facilitate the desired state transition without excessive energy usage. Finally, the Armijo plots across iterations validate the step size selection strategy, ensuring that the optimization progresses steadily. These results highlight the algorithm's capability to achieve optimal state transitions with high accuracy, stability, and computational efficiency. While working with the Newton's method we face the importance of the cost matrices tuning, in particular for what concern R since it allow us to balance the trade-off between minimizing control effort and achieving the desired state trajectories, ensuring smooth trajectories.

Chapter 3

Task 2 - Newton's method on smooth trajectory

3.1 Task description

The goal we have to accomplish for this second task of our project is to generate a desired (smooth) state input curve and perform the trajectory generation task (Task 1) on this new desired curve. In Task 1, the goal was to achieve a transition between two equilibrium points, without the need of following a trajectory perfectly. For this reason, we used a step-like fitting, but made smoother through an exponential function. In addition, we assigned the same weight to the costs for position and velocity. This allowed Newton's method to develop an optimal solution by considering position and velocity with equal importance, avoiding favoring one component at the expense of the other. In Task 2, however, the goal is different, in particular we have to define and follow a more complex trajectory in position. To achieve this, we introduce one main change from Task 1.

- Cost balancing: we proceed by assigning significantly higher costs to positions than to velocities. This allowed us to focus the optimization on good trajectory tracking in position, achieving the main objective of Task 2.

In summary, recalibrating costs is the key steps to achieve the desired outcome in Task 2, in contrast to the focus on simply linking equilibria in Task 1.

3.2 Equilibria computation

We define the reference trajectory, with the same procedure we adopted for Task 1, so we need to define the time instants in which the desired input-space curve should evolve, opting for $T = array([0, 5, 9, 13, 17, 21, 26])$. Then we define our desired input-state curve, thanks to the function $generateCurves(xxValues, uuValues, ttValues, curveTypeValues)$ as a series of exponential-spline-junctions between a set of equilibrium points, that is generated by considering the angle of the first link of the FRA (at each time instant) and by searching the corresponding input value that leads to the equilibrium in which the second link is pointing downwards (alias at the same angle but with opposite sign). This lead us to a behaviour of the following type: from 0 to 5 we start from the first equilibrium point and we have a constant section, then from 4 to 21 we have multiple evolving sections through the equilibrium points, lastly, from 21 to 26 we end with another constant section. We can conclude that our desired trajectory will vary inside a range that goes from -30 degrees to +90 degrees, for a total maximum swing of 120 degrees.

3.3 Newton's method on smooth curve

We can proceed now, as we did for Task 1, therefore, once that we complete the first step we can proceed by defining the cost matrices, and in particular a diagonal matrix Q = diag([16, 16, 6, 6]) to describe the state variables and a diagonal matrix R = 0.001*eye(ni) used for the input variable, we can notice that RR will be a one element matrix since we have just one input. As said previously, we introduce these matrices to define the cost function, for the trajectory tracking optimization problem, in fact this step is needed to represent how accurately we want to track a variable and allow us to apply the Newton's method after the definition of a maximum number of iterations and a tolerance. The application of this method enable us to minimize the cost function. This minimization is of main importance for the computation of the descent direction, in which we can avoid to compute the Hessian of the dynamics by solving the costate equations and solving an affine linear quadratic problem. Inside paragraph 2.3 we described all the functions we had to develop to compute the Newton's method and other procedures mentioned above, since nothing changes from the previous task except for what already mentioned, we won't report and describe all the functions again. However we will obviously obtain different plots, therefore, I will report them here.

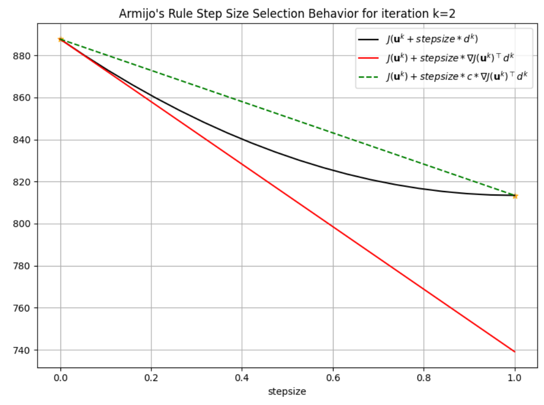


Figure 3.1: Armijo's rule for k = 2

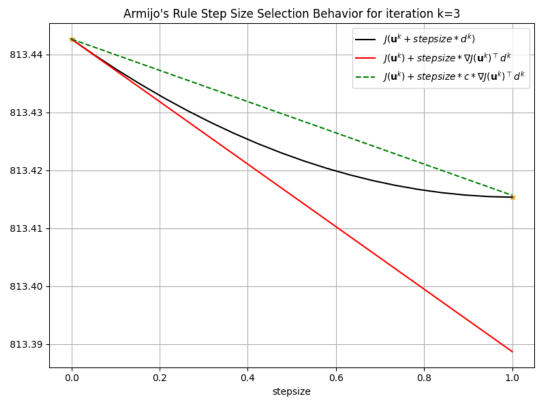


Figure 3.2: Armijo's rule for k = 3

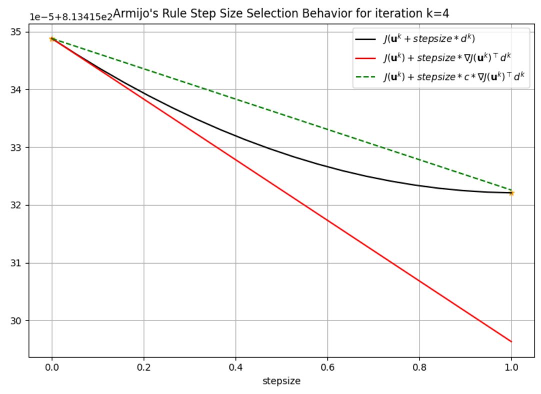


Figure 3.3: Armijo's rule for k = 4

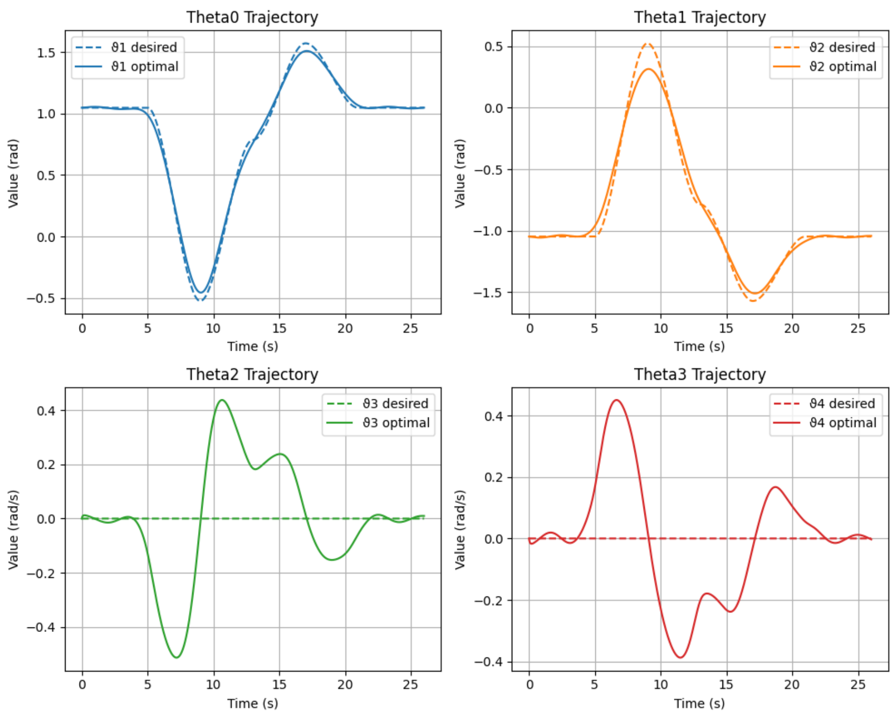


Figure 3.4: Desired and optimal trajectories

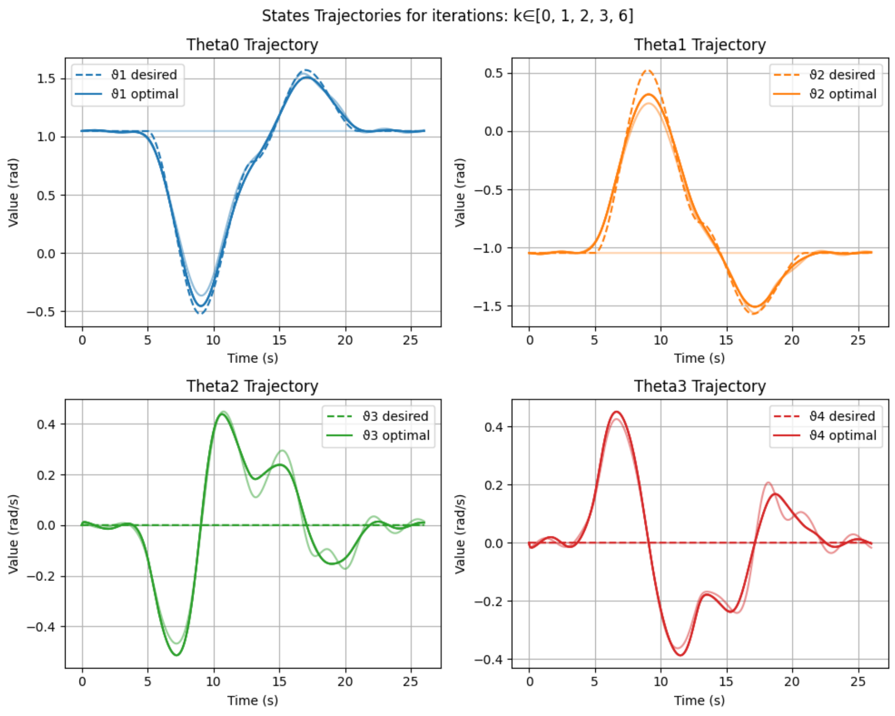


Figure 3.5: Intermediate iterations to obtains the optimal trajectories

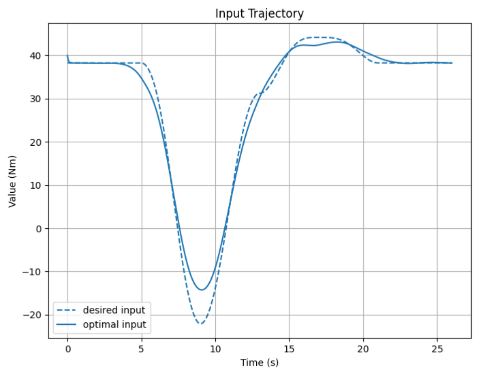


Figure 3.6: Desired ad optimal input trajectories

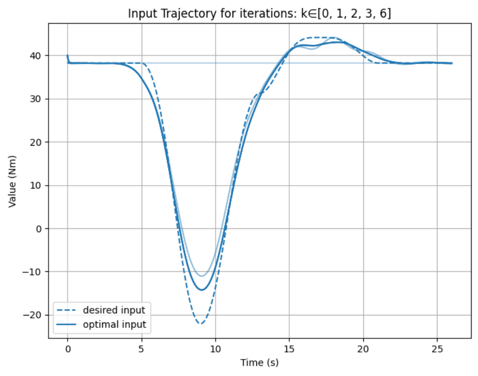


Figure 3.7: Intermediate iterations to obtains the optimal input trajectories

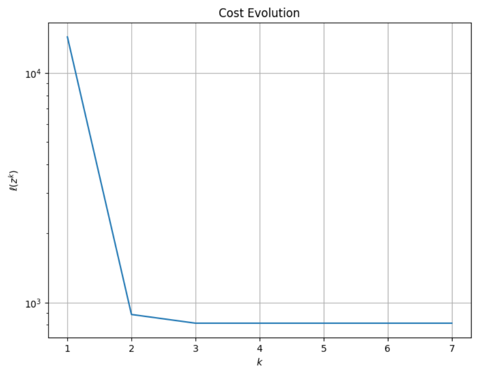


Figure 3.8: Obtained decrease for the cost function evolution

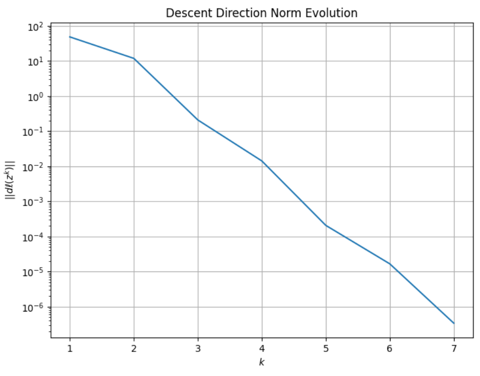


Figure 3.9: Obtained decrease for the norm of descent direction

3.4 Conclusions

The results showcase the effectiveness of employing a Newton's like algorithm within a closed-loop framework to compute optimal control trajectories for transitioning between two equilibria of the system. Each plot provides clear evidence supporting this conclusion. The descent direction norm evolution plot highlights the rapid convergence of the algorithm, achieving optimization within just 6 iterations, demonstrating its robustness and efficiency. Similarly, the cost evolution plot confirms this efficiency, as the cost function stabilizes quickly after the initial iterations. The intermediate trajectory plots for the system variables $(theta0, theta1, theta2, theta3)$ reveal a close alignment between the desired and optimal trajectories, ensuring smooth transitions between equilibria while adhering to constraints. The input trajectories further validate the precision of the control inputs, enabling the desired state transition with minimal energy expenditure. Moreover, the Armijo plots across iterations confirm the effectiveness of the step size selection strategy, facilitating steady and consistent optimization progress. These findings underline the algorithm's ability to achieve accurate, stable, and computationally efficient state transitions. Finally, the results emphasize the critical role of tuning the cost matrices, particularly R, to balance the trade-off between minimizing control effort and achieving smooth, desired state trajectories.

Chapter 4

Task 3 - Trajectory tracking via LQR

4.1 Task description

In this third task what was asked was to linearize the robot dynamics about the generated trajectory $(x^{\text{gen}}, u^{\text{gen}})$ computed in Task 2, exploit the LQR algorithm to define the optimal feedback controller to track this reference trajectory and in particular, to solve the LQ Problem:

$$\begin{aligned} \min_{\Delta x_1,\ldots,\Delta x_T,\Delta u_0,\ldots,\Delta u_{T-1}} \quad & \sum_{t=0}^{T-1} \Delta x_t^\top Q^{\text{reg}} \Delta x_t + \Delta u_t^\top R^{\text{reg}} \Delta u_t + \Delta x_T^\top Q_T^{\text{reg}} \Delta x_T \\ \text{subject to} \quad & \Delta x_{t+1} = A_t^{\text{gen}} \Delta x_t + B_t^{\text{gen}} \Delta u_t, \quad t = 0,\ldots,T-1 \\ & \mathrm{x}_0 = 0 \end{aligned}$$

where $A_t^{\text{gen}}, B_t^{\text{gen}}$ represent the linearization of the (nonlinear) system about the optimal trajectory. Considering the fact that the cost matrices of the regulator are our degree-of-freedom. In summary, our goal was to design a linear quadratic regulator (LQR) able to follow the smooth optional trajectory generated in the previous task.

4.2 Description of the implemented solution

We start to develop this task by linearizing the system dynamics computed in task 0, about the smooth optimal trajectory that we computed in task 2 through the functions *computeLocalLinearization* contained in our dynamics file. It is possible to notice that we import the trajectory thanks to the $loadDataFromFIle()$ function, that avoid us the burden to recompute it for every task. Once obtained the matrix A_{lin}, B_{lin} from the linearization we compute the gain K as the solution of the LQP (linear quadratic optimization problem) given by the function *solveLQP*. In order to implement this function we use as QQT, therefore as terminal cost matrix, the ARE (Algebraic Riccati Equation) solution implemented in the function *solveARE* and as Q and R the matrices defined in task 2. Concluded these steps we use the obtained K to design the desired LQR (Linear Quadratic Regulator) defined in the function *runLQRController*. At this point, we test it with different noise levels, in particular we choose $xx0noiseLevels = [0.0, 0.2, 0.4]$ that will be used by the function *generateInitialStateNoise*, to see how it reacts to perturbations of the initial state while asked to follow the optimal trajectory. Let's now analyze, in detail, all the function mentioned.

- $computeLocalLinearization(xx_traj, uu_traj)$ = given a feasible state-input trajectory of states and inputs, this function computes the local linearization of the dynamics around that trajectory. It takes as arguments:
  - $xx_{traj} = ns * TT$ column vector state trajectory
  - $uu_{traj} = ni * TT$ column vector input trajectory

  While it returns:
  - $AA = ns * ns * TT$ tensor of jacobians of the dynamics w.r.t. the state at each time instant
  - $BB = ns * ni * TT$ tensor of jacobians of the dynamics w.r.t. the input at each time instant
- $solveLQP$ = this function solves a time-varying linear quadratic optimal control problem. It computes the optimal control and state trajectory for a discrete-time linear system with time-varying dynamics and quadratic cost function. This involves solving the backward Riccati Equation to obtain the cost matrices and gain matrices, followed by a forward simulation to calculate the optimal state and control trajectories. The algorithm is well-suited for trajectory tracking and optimal control of linearized systems. This because, it ensures efficient computation of control laws while minimizing a trade-off between control effort and state trajectory deviation. Its arguments are:
  - AA, BB, QQ, RR, QQT, TT, xx_0 = all previously defined

  Instead, it returns:
  - KK = array of optimal feedback gain matrices for every time step
  - PP = array of cost matrices computed using the Riccati Equation
  - xx_{out} = array of the optimal state trajectory over time
  - uu_{out} = array of the optimal control inputs over time
- $solveARE(A, B, Q, R, S)$:
  Used to solve the Algebraic Riccati Equation (ARE), which is a key equation in optimal control theory, particularly in the Linear Quadratic Regulator (LQR) and other control problems. The arguments of this function are:
  - A = state transition matrix
  - B = control input matrix
  - Q = state cost weighting matrix in the quadratic cost function
  - R = control cost weighting matrix in the quadratic cost function
  - S =additional matrix that is included in the augmented system. It is used to account for additional cross-coupling between state and control inputs. If S is provided and contains non-zero values, the system is augmented and solved accordingly.

  The function returns the solution to the Algebraic Riccati Equation (ARE) or the augmented ARE, which is the optimal cost matrix P.

- $runLQRController(xx_{traj}, uu_{traj}, KK, discretizedDynamicFunction,$ $xx0Noise = None, includeMeasureNoises = False)$ = this function run the LQR controller, using the given feedback gain KK, on the given trajectory. The arguments of this function are:
  - xx_{traj}, uu_{traj} = the state and input reference trajectory
  - KK = the feedback gain matrix related to the LQR
  - = $discretizedDynamicFunction$ = a function that represents the discretized dynamic of the system
  - $xx0noise$ = this is a noise to be eventually added to the initial stare
  - $includeMeasureNoises$ = a boolean flag that indicates if the measure noises should be included in the simulation. In case, noises are added to the state trajectory at each time instant, where for each state the noise is generated by taking samples from a N(0,1) normal distribution scaled by a percentage of the maximum value assumed by the state itself along the trajectory.

  This function returns:

  - xx_{track}, uu_{track} = the state and input trajectory, respectively tracked and applied by the LQR controller.

- $generateInitialStateNoise(xx, noiseStdPercentage, gainK = 2,$ $randomNumberGenerator = None)$ = This function takes care of the generation of a noise to be added to the initial state of the system. That noise in generated (for each single state) by taking samples from a N(0,1) normal distribution scaled (in its standard deviation by the percentage of the standard deviation of the state itself). Its arguments are:
  - xx = this is the state trajectory matrix where each row represents a different state variable over time. The function computes the standard deviation of each state variable (row-wise) to determine the noise scale.
  - $noiseStdPercentage$ = a percentage (expressed as a fraction) that scales the noise standard deviation relative to the standard deviation of the state variables. If set to None or ¡= 0, the function will return a zero vector.
  - $gainK$ = a multiplicative gain applied to further scale the noise standard deviation. Default is 2
  - $randomNumberGenerator$ = A random number generator instance for producing reproducible noise. If not provided, a default generator seeded with 2828 is used.

  It returns:

  - noise = a 1D array of random Gaussian noise with the same size as the number of state variables (ns). The noise is scaled based on the computed standard deviations of the state variables and the user-specified parameters (noiseStd-Percentage and gainK).

4.3 Results

Let's see the plot that we obtained from the procedure that we just explained.

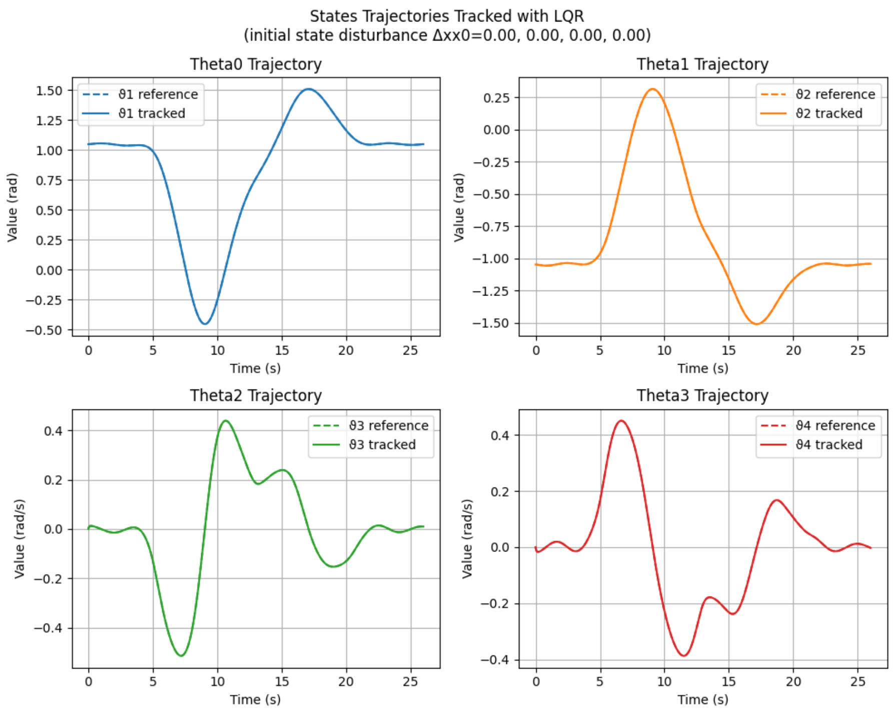


Figure 4.1: States trajectories tracked with LQR and no disturbance

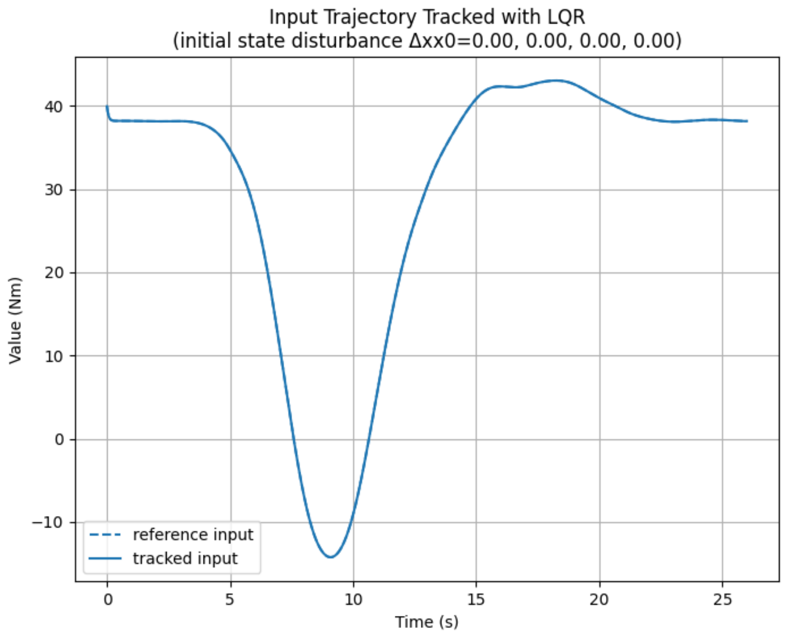


Figure 4.2: Input trajectory tracked with LQR and no disturbance

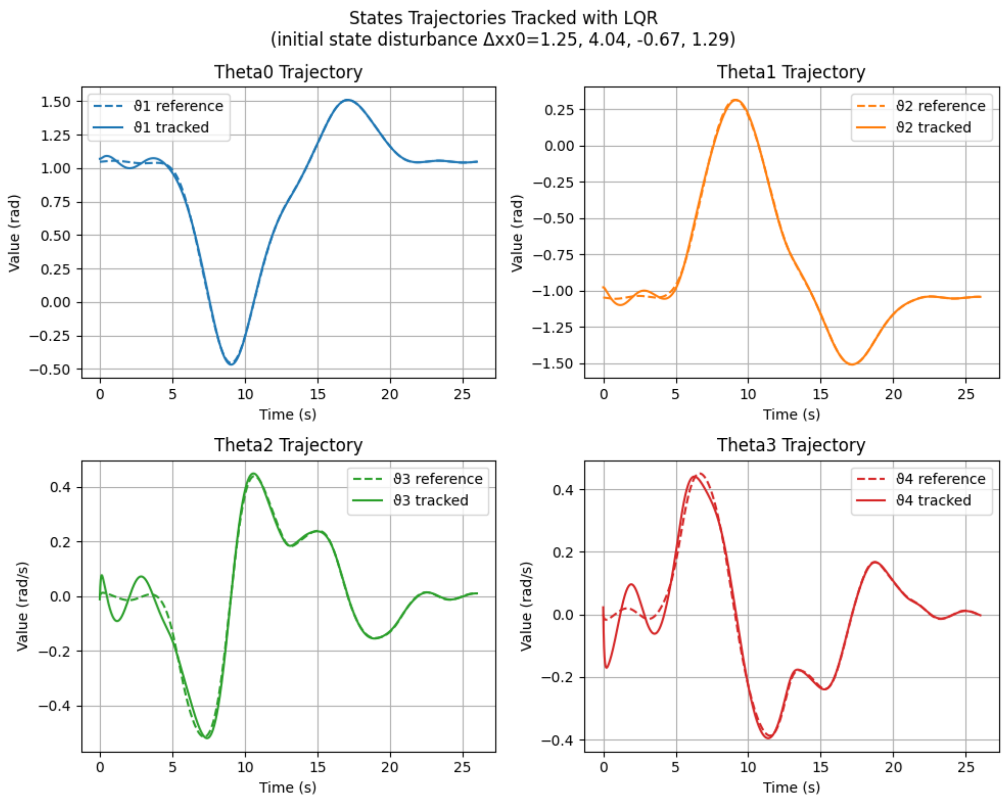


Figure 4.3: States trajectories tracked with LQR and smaller disturbance

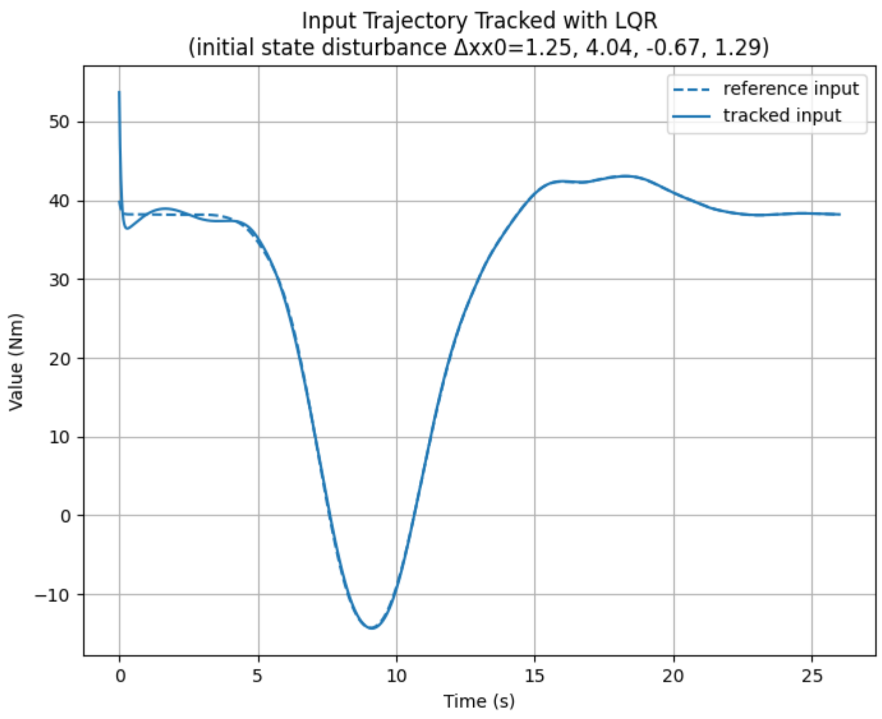


Figure 4.4: Input trajectories tracked with LQR and smaller disturbance

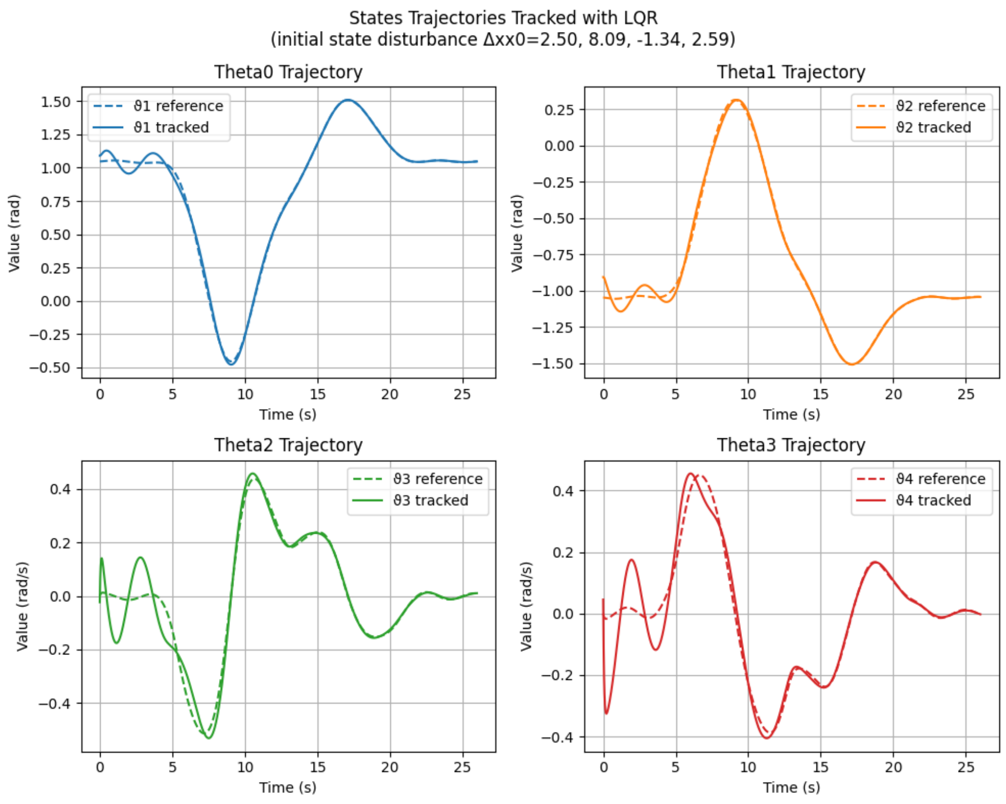


Figure 4.5: States trajectories tracked with LQR and bigger disturbance

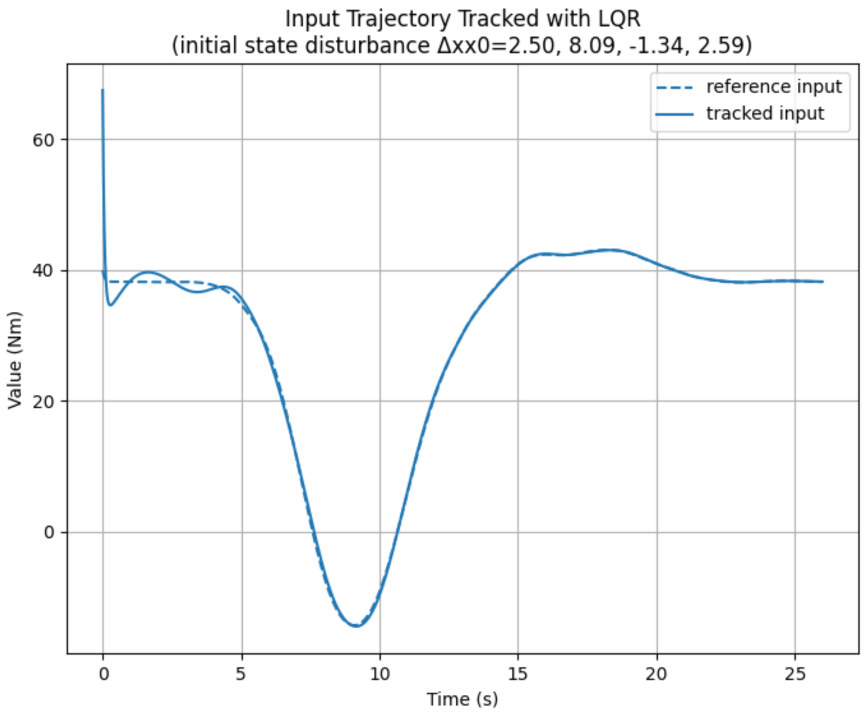


Figure 4.6: Input trajectories tracked with LQR and bigger disturbance

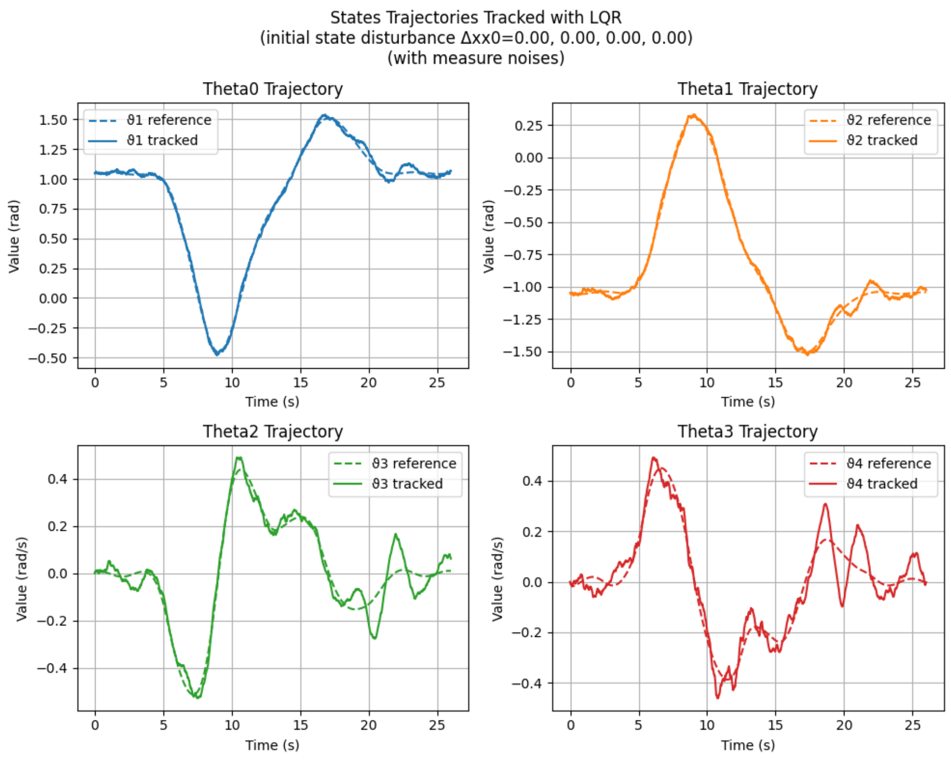


Figure 4.7: States trajectories tracked with LQR and no disturbance, but with measure noise

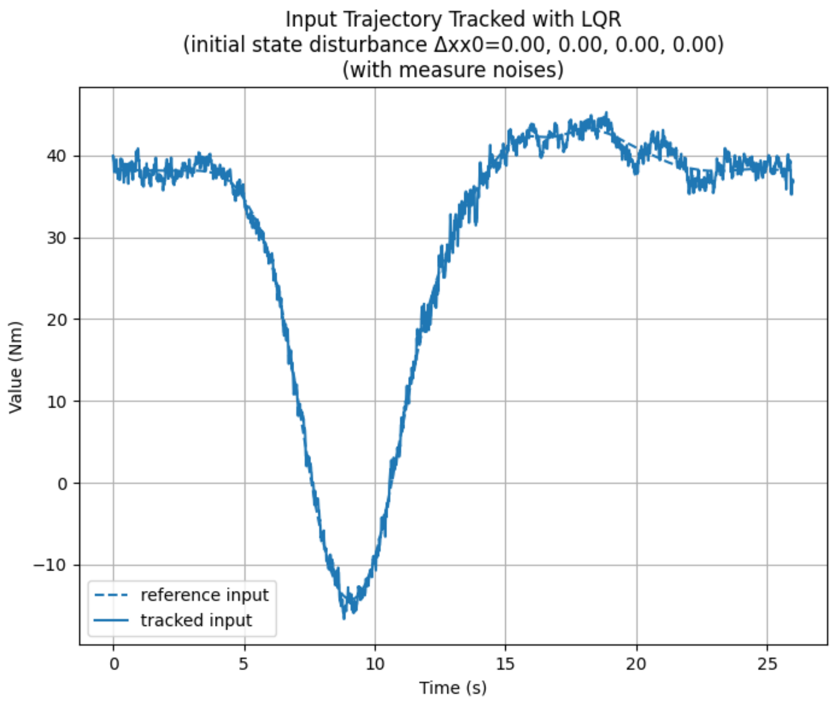


Figure 4.8: Input trajectories tracked with LQR, no disturbance, but with measure noise

4.4 Conclusions

Based on the results obtained from the implementation of the LQR controller, we can conclude that the designed feedback control law effectively tracks the reference trajectory generated in Task 2. The state and input trajectories shown in the plots illustrate the system's ability to follow the optimal path with minimal deviation. Furthermore, testing the controller under different noise levels demonstrates its robustness, as it successfully mitigates disturbances and maintains trajectory tracking performance. Higher noise levels introduce some deviations, as expected, but the control strategy ensures stability and convergence. Overall, the LQR-based approach provides a reliable and efficient solution for trajectory tracking in this robotic system, validating the theoretical framework and implementation.

Chapter 5

Task 4 - Trajectory tracking via MPC

5.1 Task description

In this task we are asked to linearize the robot dynamics about the trajectory (x_{gen}, u_{gen}) computed in Task 2, exploiting an MPC algorithm to track this reference trajectory.

5.2 Description of the solution implemented

The procedure we will follow is really similar to what we did to achieve our goal in task 3. In fact we begin this task by linearizing the system dynamics obtained in Task 0 around the smooth optimal trajectory generated in Task 2. This is done using the computeLocalLinearization function from our dynamics file. Notably, we retrieve the trajectory using the loadDataFromFile() function, which eliminates the need to recompute it for each task, simplifying the process. Once obtained AAlin e BBlin from the linearization, we define the prediction time horizon for the MPC, MPC_{TT} and the cost matrices $Q = diag([16.0, 16.0, 6.0, 6.0])$ and $R = 0.0001 * eye(ni)$. Once developed the MPC controller through the function $runMPCController$, we proceed choosing with which noise levels perturbate our system in order to study how it reacts to disturbances while asked to follow the optimal trajectory. Aside from this, we also try to analyze our system under measure noises and additional constraints (e.i. input saturation). Let's see now the new functions needed in this task to achieve our goal.

- $runMPCController(xx_traj, uu_traj, AA, BB, QQ, RR, MPC_TT$ $, discretizedDynamicFunction, xx0Disturbance = None,$ $generateMeasureNoises = False, useCVXSolver = True,$ $considerAdditionalConstraints = True)$ = this is the function that run the MPC controller on the given trajectory. it takes as arguments:
  - xx_{traj}, uu_{traj} = the state and input reference trajectories
  - AA, BB = matrix A and B of the linearized dynamics around the reference trajectory
  - QQ, RR, QQT = cost matrices that are supposed to be time invariant
  - MPC_TT = prediction time horizon for the MPC.
  - $discretizedDynamicFunction$ = previously descripted

- $xx0Disturbance$ = a disturbance to be eventually added to the initial state
- $generateMeasureNoises$ = a boolean flag that indicates if already-set-up additional constraints should be considered. In case, the cvxpy library is used (instead of the analytic solver) to solve the LQ problems involved in the MPC action (in order to make it possible to take into account the additional constraints).

Our function returns:

- xx_{traj}, uu_{traj} = the state and input respectively tracked and applied by the MPC controller

• $cvxpyProblemSetup(ns, ni, QQ, RR, QQT, MPC_TT, considerAdditionalConstraints)$ = this function sets up the CVXPY problem for the MPC controller (if needed). Its arguments are:

- ns, ni = number of states and inputs
- QQ, RR = States and input cost matrices (supposed to be time invariant)
- MPC_{TT} = already defined
- $considerAdditionalConstraints$ = A boolean flag that indicates if already-set-up additional constraints should be considered.

The function returns:

- problem = the set-up CVXPY problem
- xx_traj, uu_traj = The state and input CVXPY Variables (that correspond to the prediction that the MPC will compute at each instant of time)
- xx_traj, uu_traj = The CVXPY Parameters related to the reference trajectory (that the MPC aims to track)
- $xx0_{real}$: The initial state CVXPY Parameter (that correspond to the actual real value of the state at current istant of time, beginning of the prediction horizon)
- AA, BB = The list of CVXPY Parameters related to the local linearization matrices (on the prediction horizon)

• $cvxpyProblemSolver(problem, xx_real, uu_real, xx0_real, AA, BB, xx_traj, uu_traj, xx0_real_val, AAval, B$ = this function update CVXPY parameters and solve the problem. It takes as arguments:

- $problem, xx_{real}, uu_{real}, xx0_{real}, AA, BB, xx_{traj}, uu_{traj}$ = all previously defined.
- $xx0_{real_val}, AAval, BBval, xx_{traj_val}, uu_{traj_val}$ = correspond to the previously defined elements but thought for validation.

the function returns:

- $optimal_x x, optimal_u u$ = the optimized state and input trajectories.
- status = a flag indicating whether the solver successfully found a solution
- cost = the objective function value at the optimal solution
- $validating_results$ = performance metrics or validation results using the test trajectory.

5.3 Results

Now we can look at the plot we obtained from the procedure just described. In particular for what concern the last four plots we analyze a possible solution we achieved by considering an hypothetic additional measure noise.

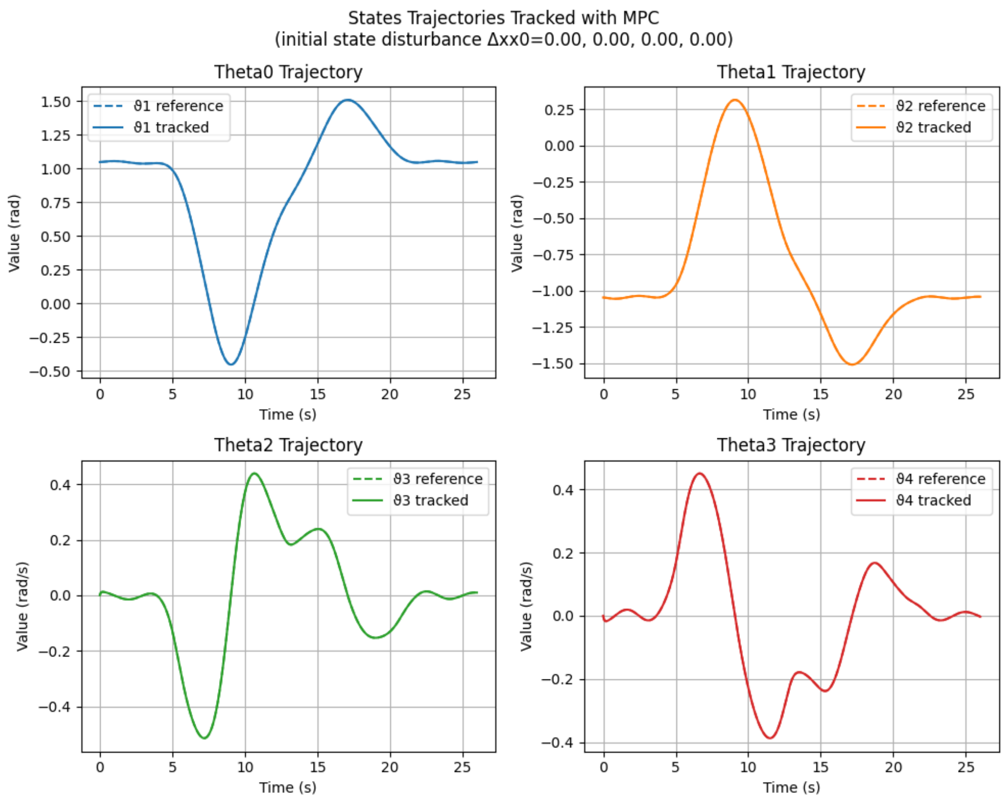


Figure 5.1: States trajectories tracked with MPC and no disturbance

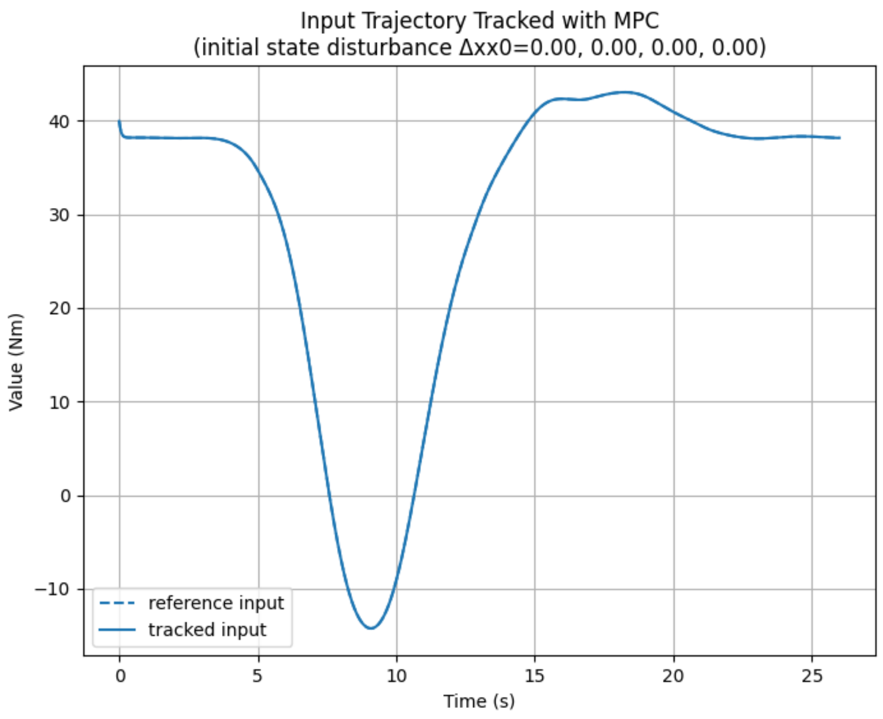


Figure 5.2: Input trajectories tracked with MPC and no disturbance

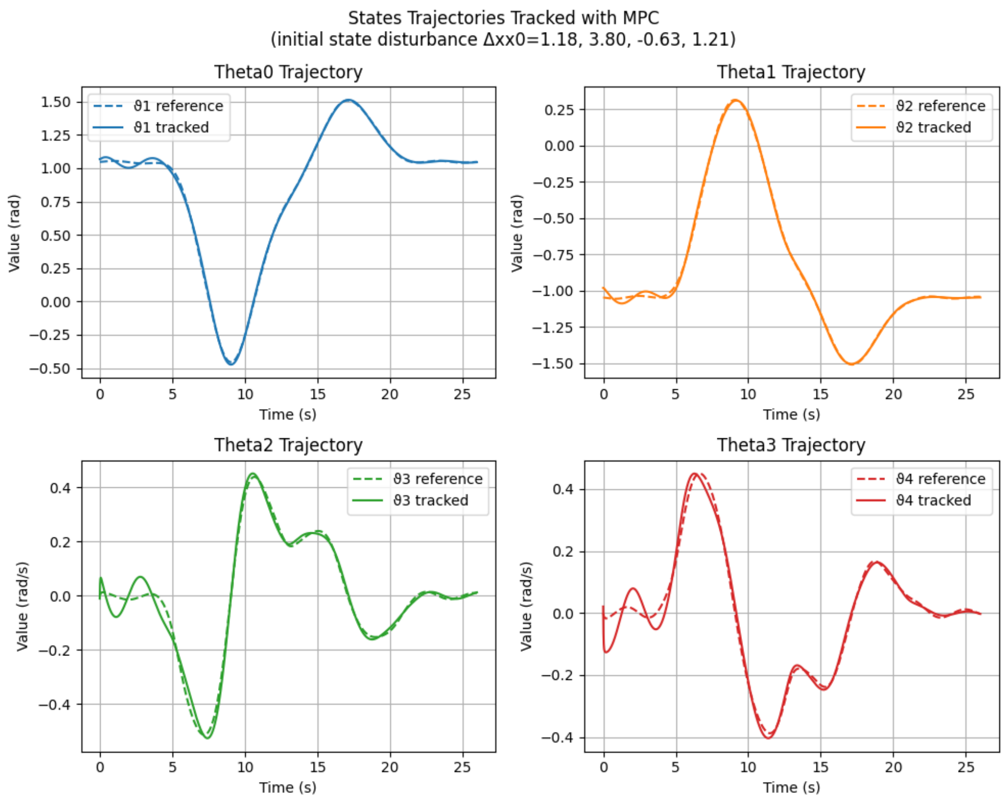


Figure 5.3: States trajectories tracked with MPC and smaller disturbance

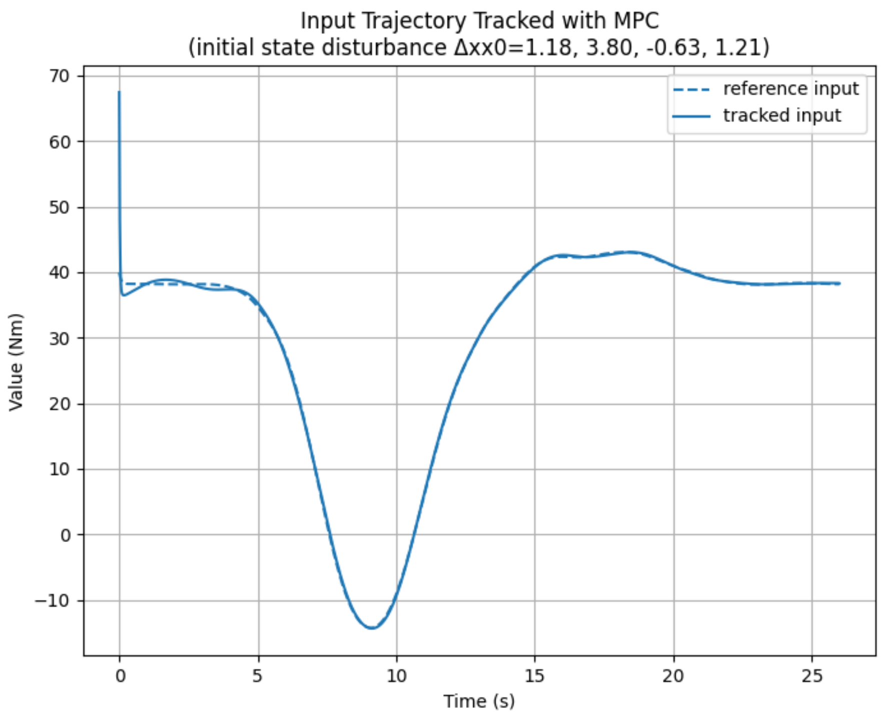


Figure 5.4: Input trajectories tracked with MPC and smaller disturbance

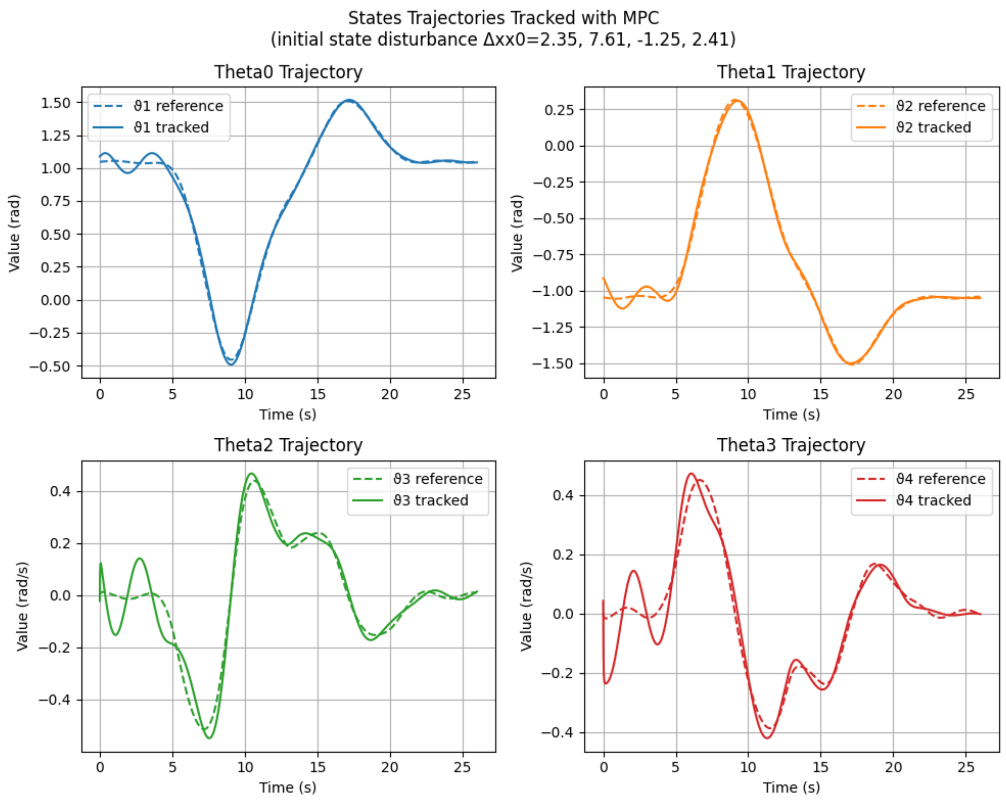


Figure 5.5: States trajectories tracked with MPC and bigger disturbance

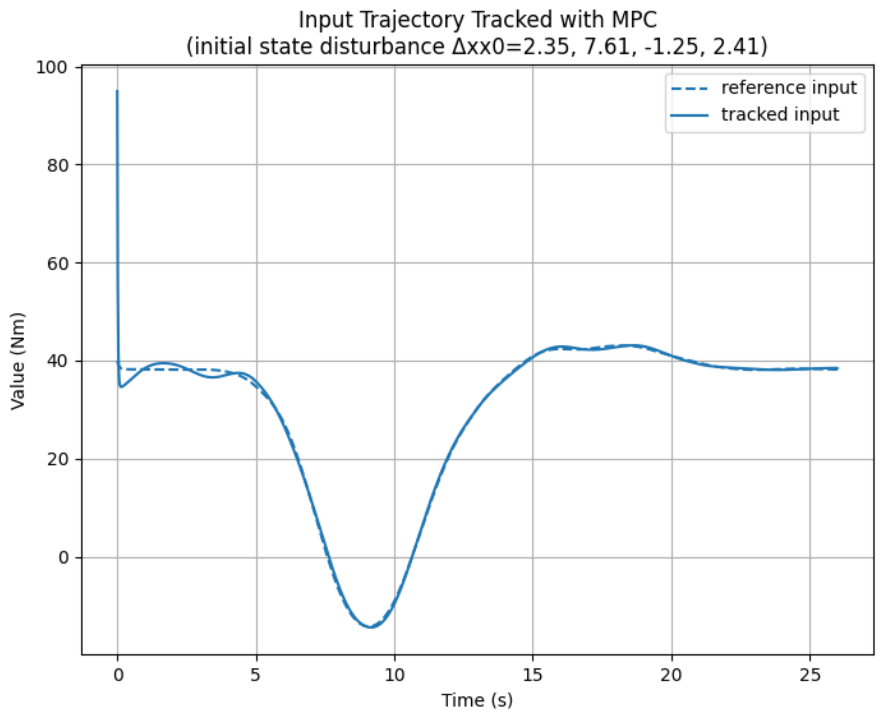


Figure 5.6: Input trajectories tracked with MPC and bigger disturbance

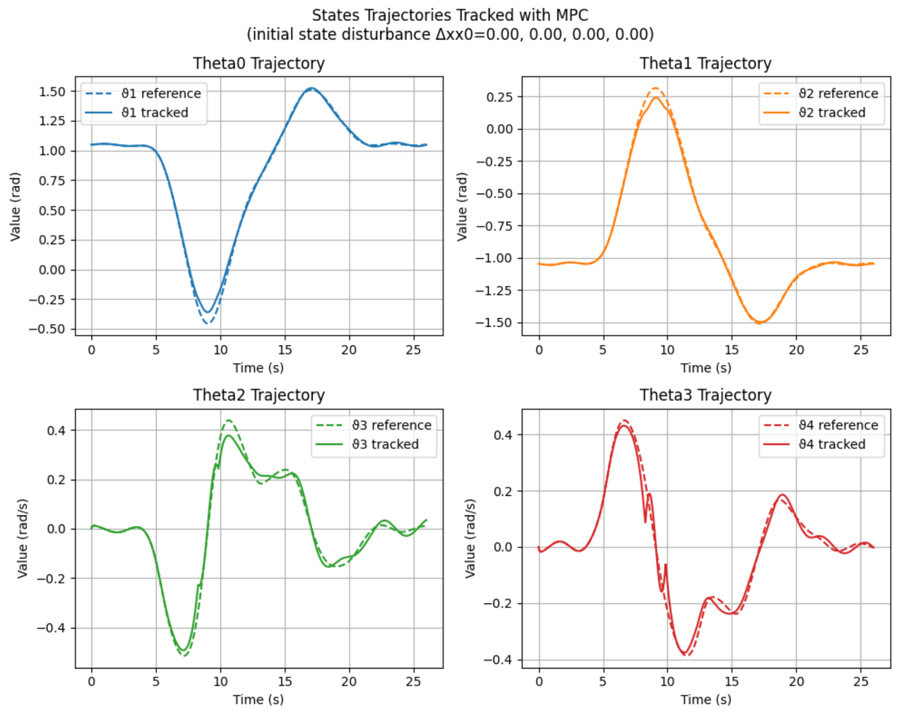


Figure 5.7: States trajectories tracked with MPC and no disturbance, but with input saturation

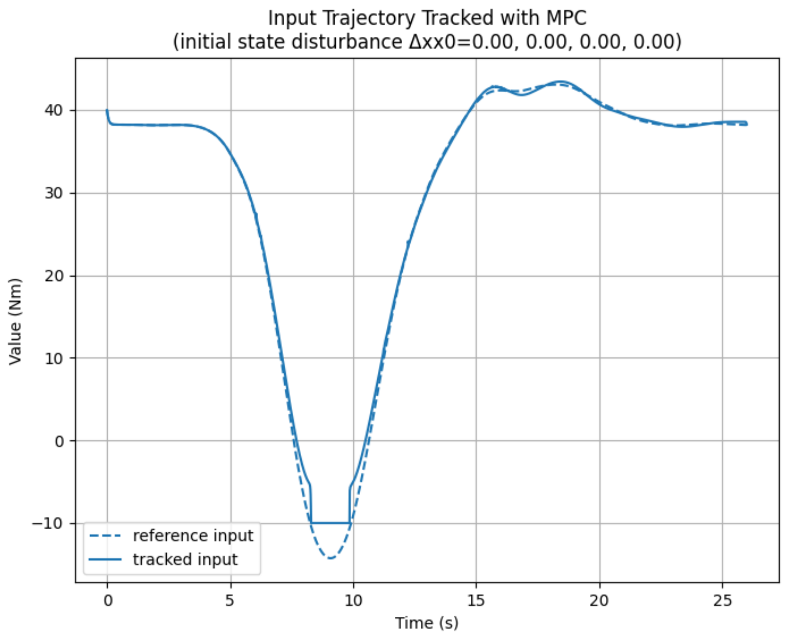


Figure 5.8: Input trajectories tracked with MPC and no disturbance, but with input saturation

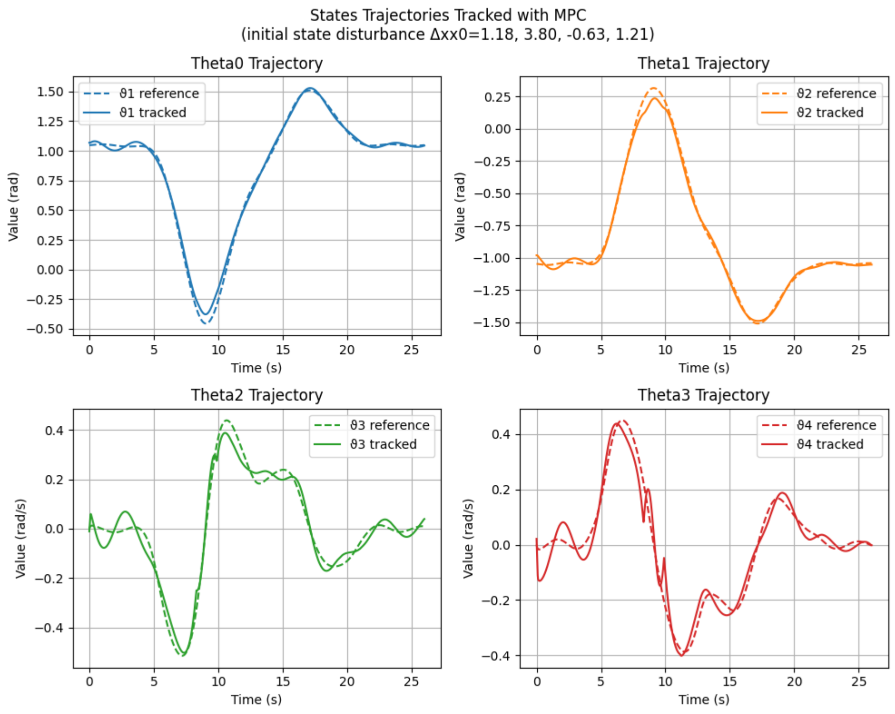


Figure 5.9: States trajectories tracked with MPC and smaller disturbance, with input saturation

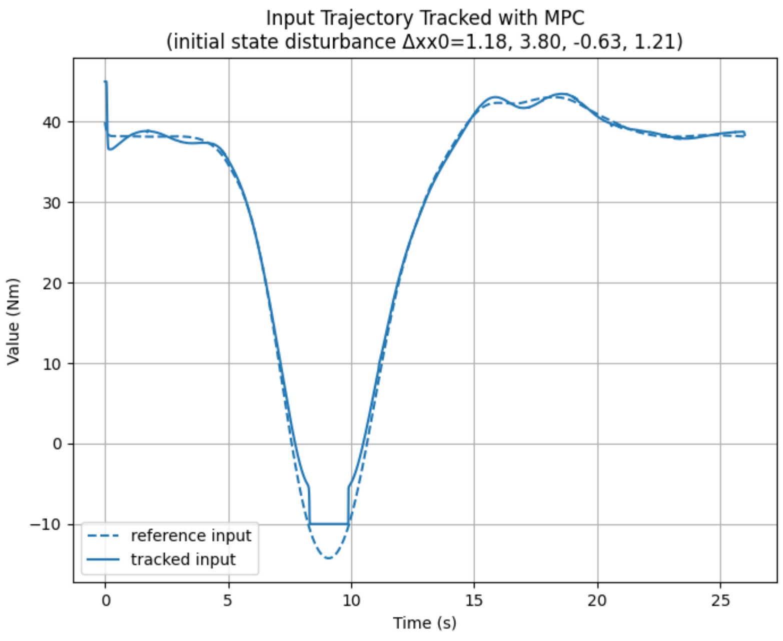


Figure 5.10: Input trajectories tracked with MPC and smaller disturbance, with input saturation

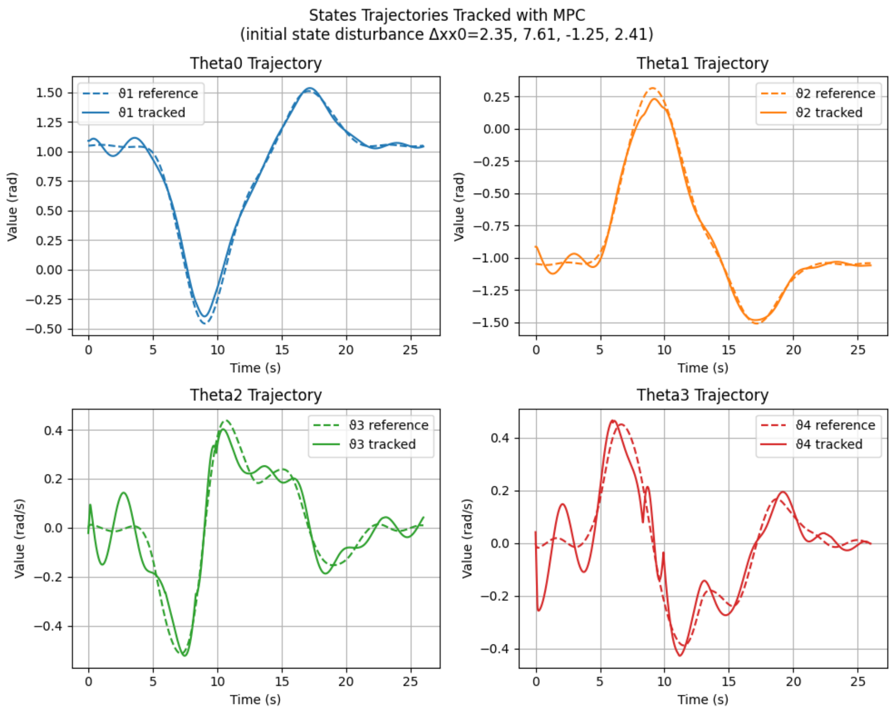


Figure 5.11: States trajectories tracked with MPC and bigger disturbance, with input saturation

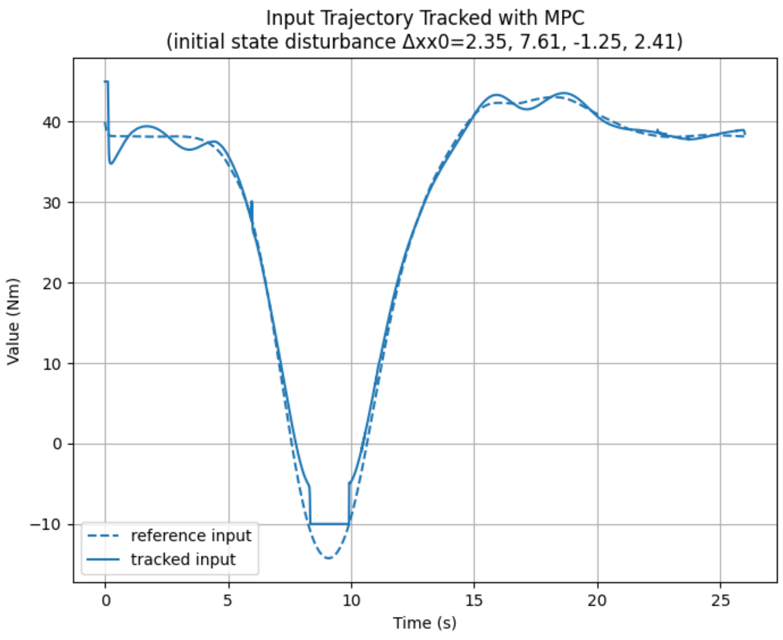


Figure 5.12: Input trajectories tracked with MPC and bigger disturbance, with input saturation

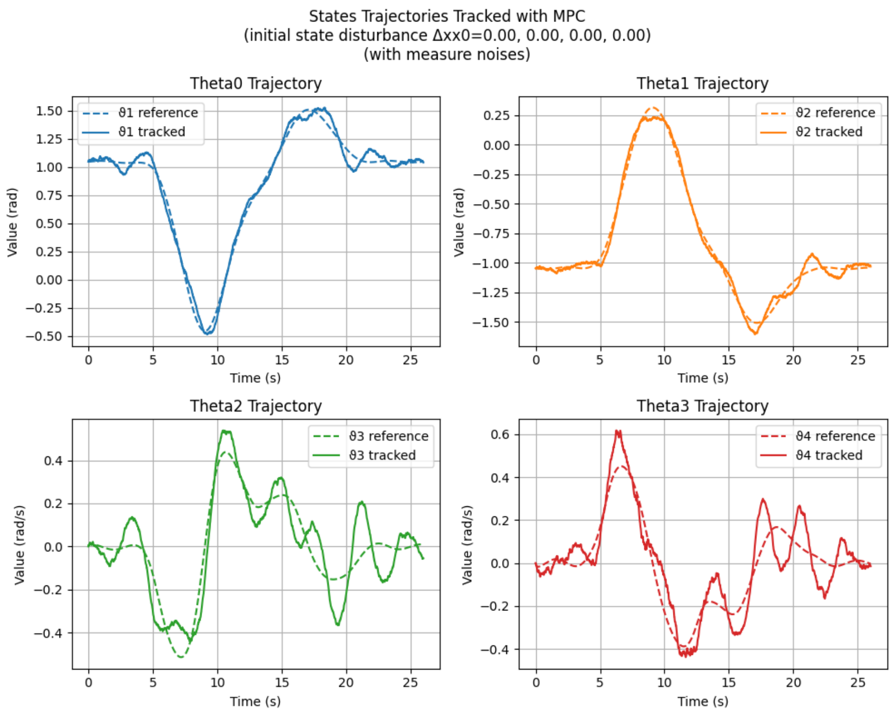


Figure 5.13: States trajectories tracked with MPC and no disturbance, but with measure noise

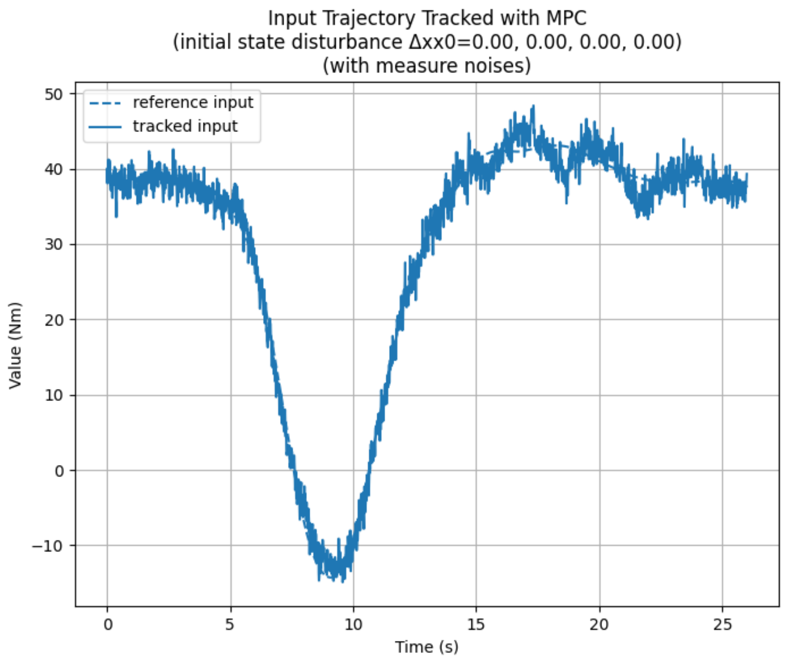


Figure 5.14: Input trajectories tracked with MPC and no disturbance, but with measure noise

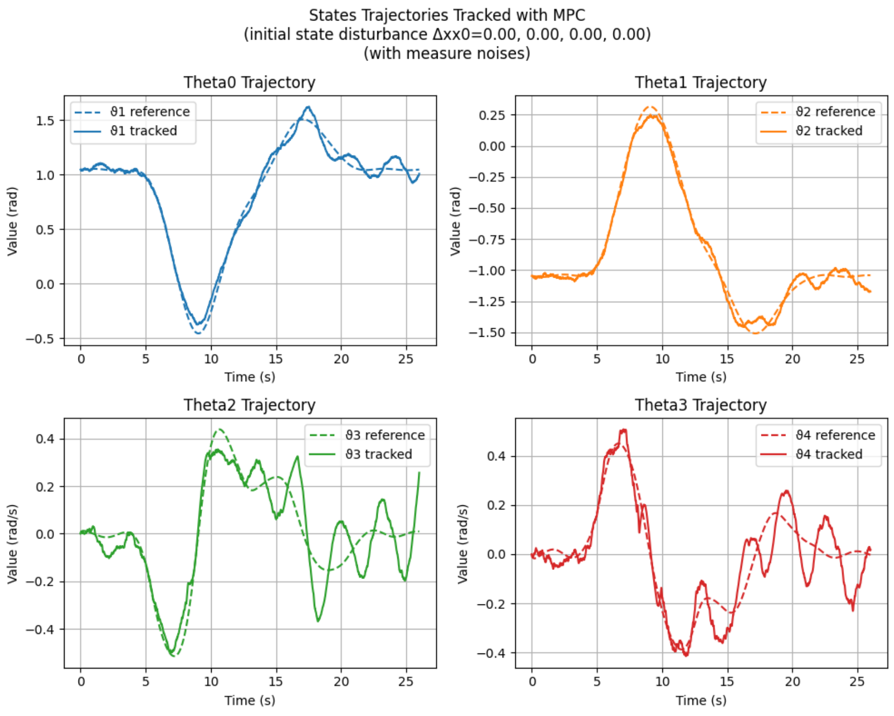


Figure 5.15: States trajectories tracked with MPC and no disturbance, with both input saturation and measure noise

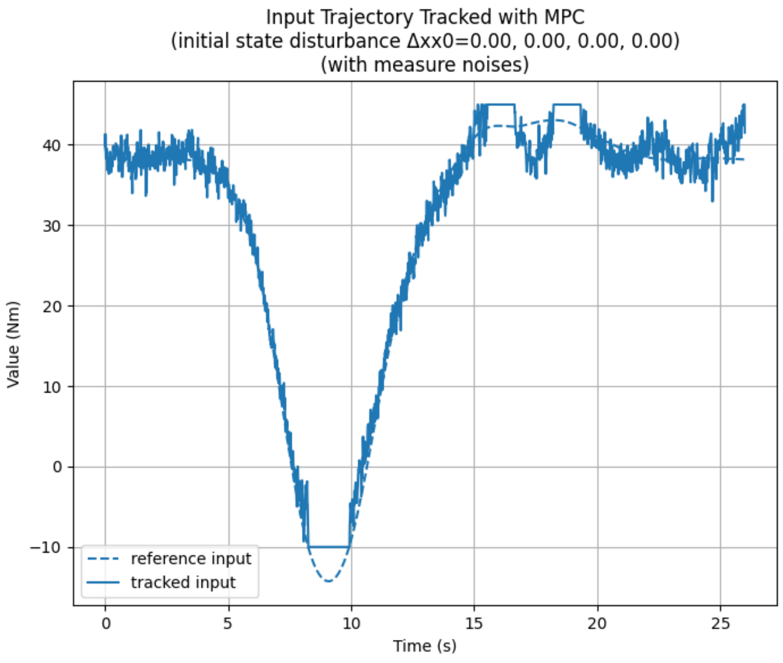


Figure 5.16: Input trajectories tracked with MPC and no disturbance, with both input saturation and measure noise

5.4 Conclusions

Chapter Four has examined trajectory tracking via MPC, highlighting the critical role of disturbances presence. Through an in-depth analysis of the methods and functions we described, this chapter has demonstrated how the trajectory tracking is possible and robust despite the various types of disturbance. In particular we applied firstly some initial noises and then we tried to see what would happen combining measure noises and an additional constraints, that in our case was an imput saturation. In the end, we were able to see that our MPC controller was robust enough to follow the trajectory in a satisfying way.

Chapter 6

Task 5 - Animation

6.1 Task description

In this final task we have to produce a simple animation of the robot executing Task 3.

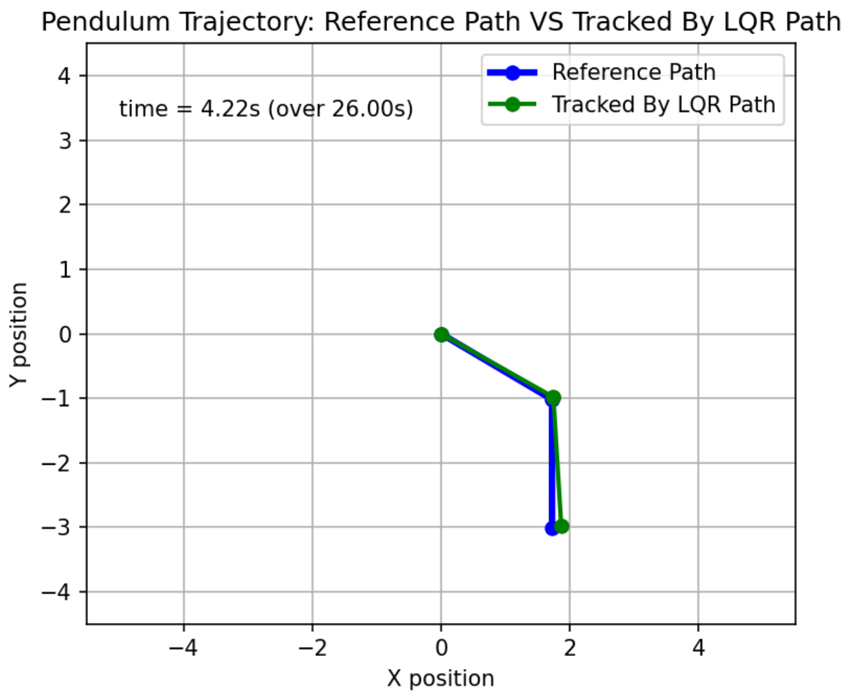


Figure 6.1: first frame

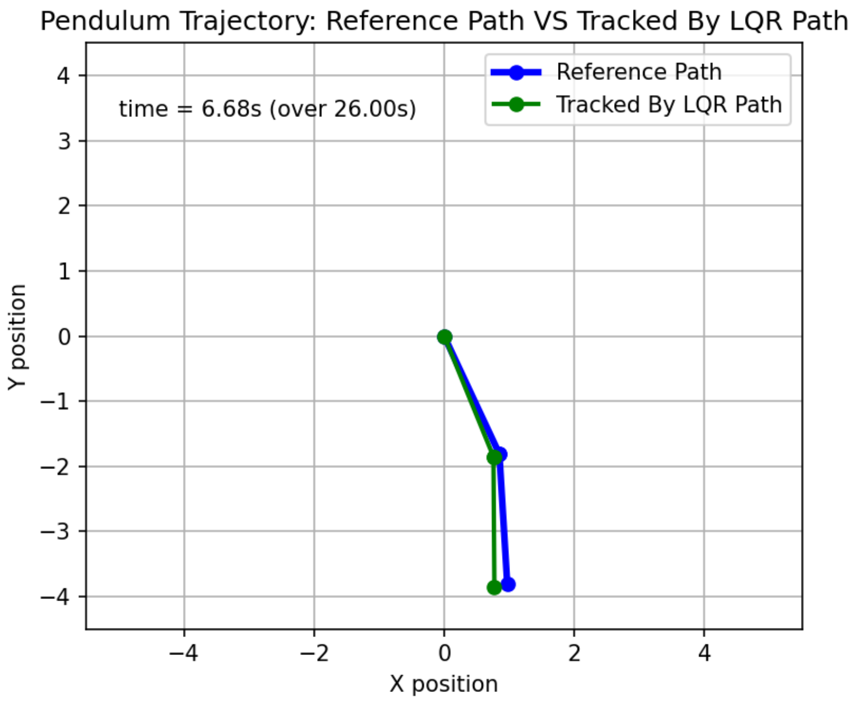


Figure 6.2: second frame

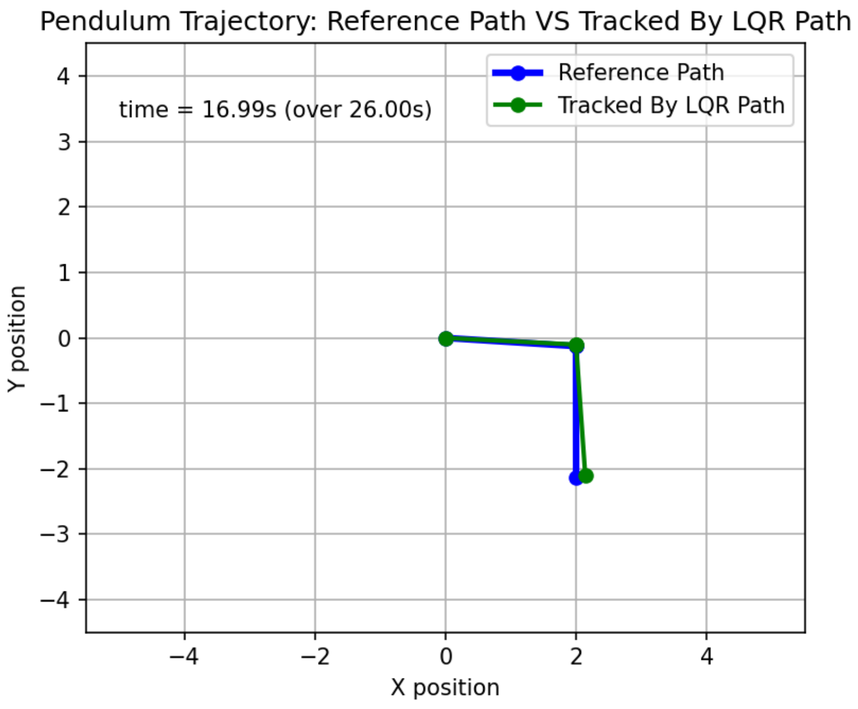


Figure 6.3: third frame

Conclusions

This project successfully demonstrated the design and implementation of optimal control strategies for a flexible robotic arm. Through the completion of six distinct tasks, we explored trajectory generation, optimization, and tracking using advanced control techniques.

The Newton-like algorithm proved effective for trajectory generation, enabling optimal transitions between equilibria with rapid convergence and high accuracy. Transitioning to smooth trajectories highlighted the significance of cost matrix tuning, particularly in balancing position and velocity costs to achieve optimal trajectory adherence.

The trajectory tracking methods using LQR and MPC controllers validated their robustness and efficiency under various conditions.
The LQR controller exhibited strong tracking performance, maintaining stability even with disturbances and measurement noise.
The MPC approach, on the other hand, introduced additional flexibility by being an on-line controller and being able to handle constraints such as input saturation, demonstrating its robustness in managing complex scenarios while maintaining satisfactory trajectory tracking.

The results also emphasized the importance of noise and disturbance management in practical control systems. The comparative analysis of the two control strategies under different perturbations provided valuable insights into their respective advantages and limitations.

Finally, the produced animations allowed for an intuitive visualization of the robot's performance.

In summary, the project not only validated the theoretical framework and implementation of advanced control strategies but also highlighted the challenges and intricacies associated with the control of flexible robotic systems. The findings provide a solid foundation for further exploration and application in real-world scenarios, where precision and adaptability are critical.