UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Optimal Control

OPTIMAL CONTROL OF A QUADROTOR WITH A SUSPENDED LOAD

Professor: **Giuseppe Notarstefano**

Students:
Davide Corroppoli
Andrea Perna
Riccardo Marras

Academic year 2023/2024

Abstract

The main objective of this project deals with the development of an optimal feedback law for a quadrotor with a suspended load. At a first glance, a reference curve in space will be defined between two hovering equilibria, either by means of a step or a smooth poly5 function. Then, the optimal state-input trajectory will be computed by a Newton Method's algorithm, by taking into account the quadrotor's dynamics and the weights on the cost chosen a priori. From the optimal result of the algorithm, two different approaches for the computation of an optimal controller will be finally presented: Linear Quadratic Regulator and Model Predictive Control.

Contents

Introduction

The system under analysis can be modeled as a planar quadrotor with a downward pendulum attached at its center of mass, representing the load. The model is represented in the following figure:

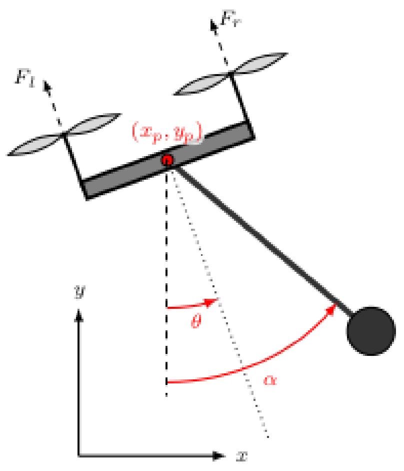

Figure 1: Model of a planar quadrotor with a suspended load

The state space consists of $x = [x_p, y_p, \alpha, \theta, v_x, v_y, \omega_\alpha, \omega_\theta]^T$, where $(x_p, y_p) \in \mathbb{R}^2$ represents the position of the center of mass, θ is the roll angle of the quadrotor, α is the angle of the pendulum with respect to the inertial frame (cf. Figure 1), (v_x, v_y) is the velocity of the center of mass, and ω_α and ω_θ are angular rates of changes associated with α and θ, respectively. The input is $u = [F_s, F_d]^T$, where $F_s = F_l + F_r$, $F_d = F_r - F_l$, and F_l, F_r are the forces generated by the propellers.

The dynamical model of the quadrotor is:

$$\begin{aligned}
(M+m)\dot{v}_x &= mL\omega_\alpha^2\sin(\alpha) - F_s\left(\sin(\theta) - \frac{m}{M}\sin(\alpha-\theta)\cos(\alpha)\right)\\
(M+m)\dot{v}_y &= -mL\omega_\alpha^2\cos(\alpha) + F_s\left(\cos(\theta) + \frac{m}{M}\sin(\alpha-\theta)\sin(\alpha)\right) - (M+m)g\\
ML\dot{\omega}_\alpha &= -F_s\sin(\alpha-\theta)\\
J\dot{\omega}_\theta &= \ell F_d
\end{aligned}$$

All the parameters of the quadrotor are available in the following table:

Parameters:	
M	0.028
m	0.04
J	0.001
g	9.81
L	0.2
ℓ	0.05

Table 1: Parameter values

The development of an optimal control algorithm for generating and tracking optimal trajectories for such an aerial system will be discussed in detail in the next chapters.

Chapter 1

Quadrotor dynamics

1.1 Discretization of the dynamics

From the continuous dynamics we get the discretized dynamics. The continuous state space is $[x_p, y_p, \alpha, \theta, v_x, v_y, \omega_\alpha, \omega_\theta]^T$, whereas the discretized one is $[x_{p,t}, y_{p,t}, \alpha_t, \theta_t, v_{x,t}, v_{y,t}, \omega_{\alpha,t}, \omega_{\theta,t}]^T$ at time t. From the continuous dynamics, considering a step of time δ, the discrete-time dynamics is provided according to the Euler method:

$$
\begin{aligned}
&x_{p,t+1} = x_{p,t} + \delta \cdot v_{x,t} \quad y_{p,t+1} = y_{p,t} + \delta \cdot v_{y,t} \\
&\alpha_{t+1} = \alpha_t + \delta \cdot \omega_{\alpha,t} \quad \theta_{t+1} = \theta_t + \delta \cdot \omega_{\theta,t} \\
&v_{x,t+1} = v_{x,t} + \delta \cdot (mL\omega_{\alpha,t}^2 \sin(\alpha_t) + F_{s,t}(\sin(\theta_t) - \frac{m}{M}\sin(\alpha_t - \theta_t)\cos(\alpha_t)/(M+m) \\
&v_{y,t+1} = v_{y,t} + \delta \cdot (-mL\omega_{\alpha,t}^2 \cos(\alpha_t) + F_{s,t}(\cos(\theta_t) - \frac{m}{M}\sin(\alpha_t - \theta_t)\sin(\alpha_t) - g(M+m))/(M+m) \\
&\omega_{\alpha,t+1} = \omega_{\alpha,t} - \delta \cdot F_{s,t}\frac{\sin(\alpha_t - \theta_t)}{ML} \\
&\omega_{\theta,t+1} = \omega_{\theta,t} + \delta \cdot \frac{F_{d,t}l}{J}
\end{aligned}
\tag{1.1}
$$

Such computations are properly handled inside the *Dynamics.py* file, from the function *dynamics()*. This latter takes as an input the current state space x at time t ($x(t)$) and the current input applied u at time t ($u(t)$), and return the state x at the next iteration $x(t+1)$.

1.2 Equilibrium points

The computation of equilibrium points is still managed in the *Dynamics.py* file, by the function *hovering_equilibria()*. Such equilibria are going to be connected with the proper interpolation function in the main file according to the specific needs of the user. In particular, we have exploited the computation of the hovering equilibria, i.e., those state and input values for which the quadrotor is able to assume a stationary position in air, while maintaining at a costant altitude relative to the ground and a steady position of the suspended load. The equilibrium conditions lead to a null accelleration, that is, a condition for which $X_{t+1} = X_t$, and the same for the inputs. Hence, we need to solve the following system of equations:

$$\begin{aligned}
&0 = \delta \cdot v_{x,t} \quad 0 = \delta \cdot v_{y,t} \\
&0 = \delta \cdot \omega_{\alpha,t} \quad 0 = \delta \cdot \omega_{\theta,t} \\
&0 = \delta \cdot (mL\omega_{\alpha,t}^2 \sin(\alpha_t) + F_{s,t}(\sin(\theta_t) - \frac{m}{M} \sin(\alpha_t - \theta_t) \cos(\alpha_t)/(M+m) \\
&0 = \delta \cdot (-mL\omega_{\alpha,t}^2 \cos(\alpha_t) + F_{s,t}(\cos(\theta_t) - \frac{m}{M} \sin(\alpha_t - \theta_t) \sin(\alpha_t) - g(M+m))/(M+m) \\
&0 = \omega_{\alpha,t+1} = \omega_{\alpha,t} - \delta \cdot F_{s,t} \frac{\sin(\alpha_t - \theta_t)}{ML} \\
&0 = \delta \cdot \frac{F_{d,t} l}{J}
\end{aligned} \tag{1.2}$$

whose solutions turn out to be the hovering equilibria:

$$x_{p,t} = x_{p,des} \quad y_{p,t} = y_{p,des} \tag{1.3}$$
$$v_{x,t} = 0 \quad v_{y,t} = 0 \tag{1.4}$$
$$\omega_{\alpha,t} = 0 \quad \omega_{\theta,t} = 0 \tag{1.5}$$
$$F_{s,t} = (M+m) \cdot g \tag{1.6}$$
$$F_{d,t} = 0 \tag{1.7}$$

1.3 Jacobian matrices computation

In addition to the discretized dynamics, the *Dynamics.py* file takes also into account the computation of the Jacobian matrices, which are going to be used throughout the optimization procedure. They are specifically evaluated on the state-input sequence at each iteration, and returned to the main program at the end of the routine. These linearization matrices, square by construction, are computed as the transposition of the gradient matrices, evaluated with respect to the state and input sequences as follows:

$$\begin{aligned}
\nabla f_x(x,u)^T &= \begin{bmatrix} \frac{\partial}{\partial x_1} f_1 \cdots \frac{\partial}{\partial x_1} f_m \\ \vdots \ddots \vdots \\ \frac{\partial}{\partial x_n} f_1 \cdots \frac{\partial}{\partial x_n} f_m \end{bmatrix}^T \\
\nabla f_u(x,u)^T &= \begin{bmatrix} \frac{\partial}{\partial u_1} f_1 \cdots \frac{\partial}{\partial u_1} f_n \\ \vdots \ddots \vdots \\ \frac{\partial}{\partial u_k} f_1 \cdots \frac{\partial}{\partial u_k} f_n \end{bmatrix}^T
\end{aligned} \tag{1.8}$$

Chapter 2

Trajectory generation

The core of the trajectory generation is the function that allows us to create fifth order polynomial reference curves, namely *ref_curve()*. Once defined, it is possible to exploit it in order to create more complex functions such a double S, or even more sophisticated curves, to interpolate equilibria. Hence, by properly concatenating multiple curves obtained with this function, different kind of trajectories can be developed. With a fifth order equation of the form $p = k_5x^5 + k_4x^4 + k_3x^3 + k_2x^2 + k_1x_1 + k_0$ we can satisfy constraints for both initial and final position, speed and acceleration. The main difference with respect to a third order polynomial is that we can satisfy acceleration constraints. Our inputs affect the accelerations, but actually in our tasks we don't have explicit constraints for the acceleration.

2.1 Implementation of fifth order references

The function *ref_curve()* takes as input parameters the initial and final time's constraints, namely (t_0, y_0, v_0, a_0) and (t_f, y_f, v_f, a_f). First of all in the function are defined the difference vectors $t_1 = t_f - t_0, \quad y_1 = y_f - y_0, \quad v_1 = v_f - v_0, \quad a_1 = a_f - a_0$. They allow us to build up general functions, which can be used in multiple circumstances, i.e., not only when initial positions, velocities and accelerations are zeros. The final goal of the function is to interpolate two distinct points within a time interval $t_1 = t_f - t_0$. Hence, the general fifth order polynomial becomes $y(t) = k_5t^5 + k_4t^4 + k_3t^3 + k_2t^2 + k_1t + k_0$, where $t = 0$ at $T = t_0$. Therefore, we can compute the coefficients by imposing in time the condition we want to fulfill. To start, when $t = 0$ we get $k_0 = y(0) = y_0$. Then, in the same way we can find k_1 and k_2: $k_1 = v_0 = \dot{y}(0)$ and $k_2 = \frac{a_0}{2} = \ddot{y}(0)$. The coefficients k_3, k_4, k_5 are computed thanks to three auxiliary terms for acceleration, velocity and position, respectively:

$$b_1 = y_1 - (k_2t_1^2 + v_0t_1) \quad b_2 = v_f - (a_0t + v_0) \quad b_3 = a_1 = a_f - a_0$$

Finally the last three coefficients are computed, that can be seen as the jerk, the snap and the crackle at time 0:

$$k_3 = \frac{b_3 t_1^2 - 8b_2 t_1 + 20b_1}{(2t_1^3)} \qquad k_4 = \frac{-b_3 t_1^2 + 7b_2 t_1 - 15b_1}{t_1^4} \qquad k_5 = \frac{b_3 t_1^2 - 6b_2 t_1 + 12b_1}{2t_1^5} \tag{2.1}$$

Here below an example of poly 5 is presented, it is used as a reference in task 2 for (xp, yp) position of the center of mass of the drone in plane:

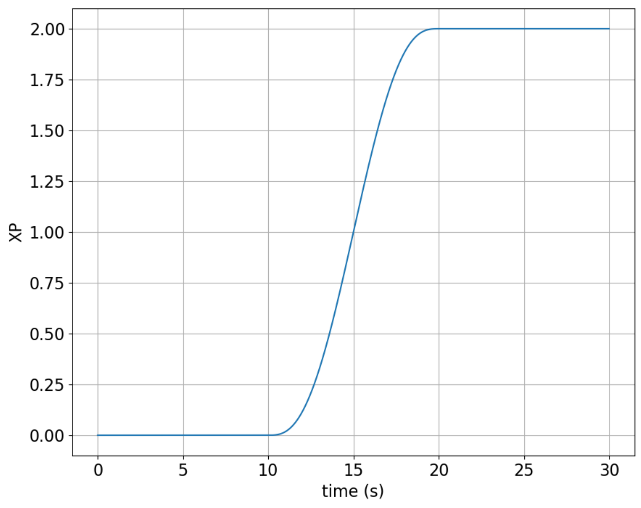

Figure 2.1: Position xp of the drone's center of mass - Task2

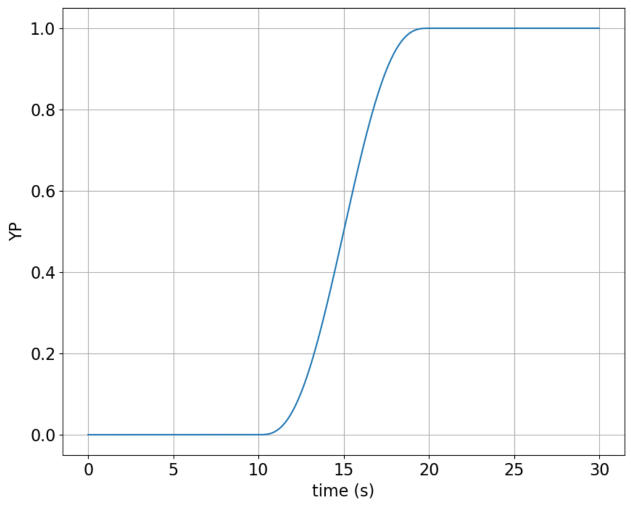

Figure 2.2: Position yp of the drone's center of mass - Task2

2.2 Implementation of step references

The step reference can be seen as a particular case of the smooth reference curve. In fact, with a poly5 curve it is possible to build constant reference curves by imposing all the accelerations, initial and final velocities to zero, and the starting position equal to the final one. In this situation, the time imposition becomes just the time for what we want the system to maintain that position. By concatenating a couple of them relative to two different positions in space, it is possible to create the step reference curve. Here below an example is presented. The following figures represent the reference curves for the (xp, yp) position of the center of mass of the drone in plane, that have been used in the Newton's Method for the implementation of the first task:

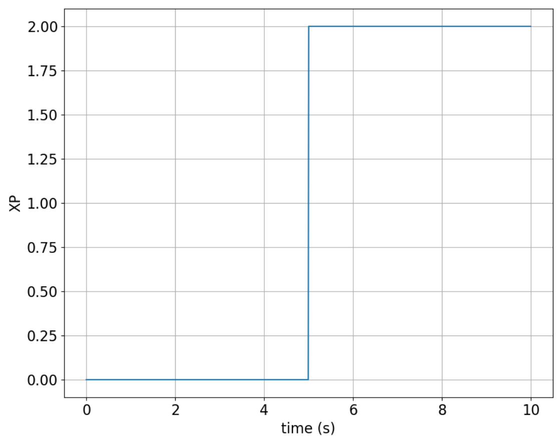

Figure 2.3: Position xp of the drone's center of mass - Task1

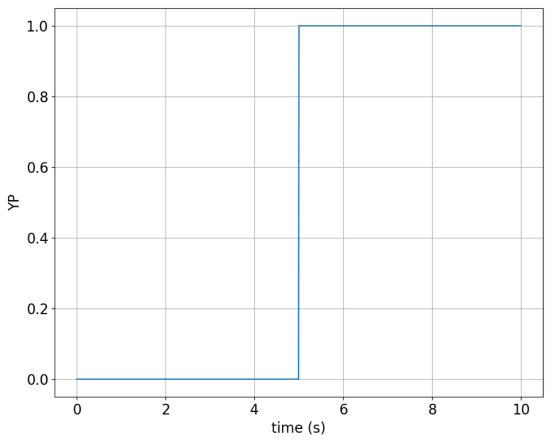

Figure 2.4: Position yp of the drone's center of mass - Task1

2.3 Implementation of Double S references

An easy, but very common trajectory we can build with poly5 functions is the so-called double S. Basically, a double S is characterized by 3 phases: rise, dwell and fall. Rise and fall are specular, in those phases we pass from an equilibrium point to another. Dwell, instead, is the constant phase in the middle of the two, i.e., where a certain kind of action is usually performed by the system. The following figure represents the double S reference curves for the (xp, yp) position of the center of mass of the drone in plane:

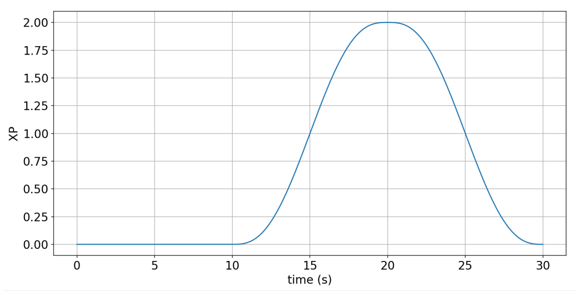

Figure 2.5: Position xp of the drone's center of mass - Double S

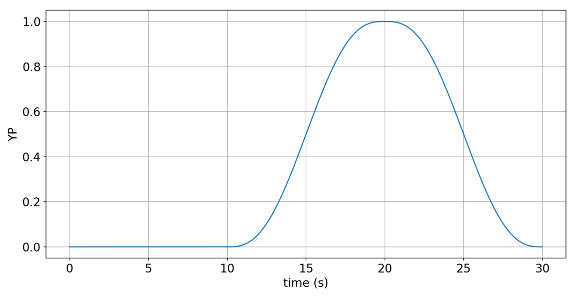

Figure 2.6: Position yp of the drone's center of mass - Double S

Chapter 3

Newton's method

The core of the project is undoubtedly represented by the Newton's Method, a powerful optimization algorithm used to efficiently compute the optimal control inputs that enable the quadrotor to follow the desired reference curve, while minimizing a specific cost function. Hence, it is used in order to evaluate the optimal trajectory for the quadrotor. In general, Newton's method starts with an initial guess for the optimal control inputs and states and then iteratively update them by using the first and second derivatives of the cost function with respect to the control inputs, i.e., the Hessian and the Gradient of the cost function. In this project, a different approach has been used for the computation of the descent direction, that is going to be explained in the following sections. In a nutshell, Newton's Method is able to converge to a local optimum through successive iterations, providing the control inputs that optimize the quadrotor's behaviour according to the defined criteria.

3.1 Initial guess

As as initial guess for the trajectory, i.e., the starting point of the optimization algorithm, a constant trajectory at the first equilibrium has been chosen. In fact, the starting condition of the algorithm represents a crucial role for the sake of algorithm's convergence, i.e., an infeasible initial guess of the trajectory may lead to divergence or other serious issues. The following plots show some instances of the initial state-input trajectory:

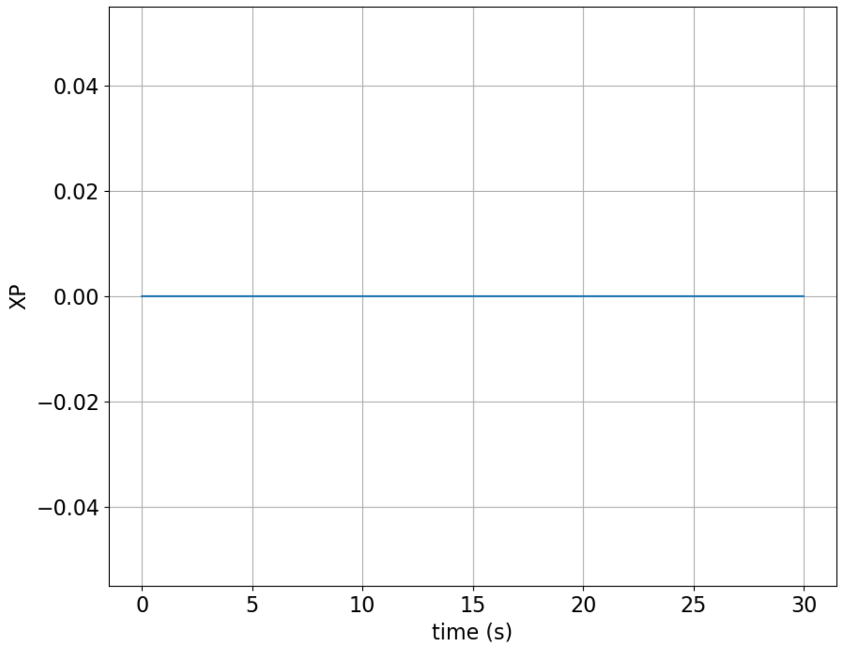

Figure 3.1: Position yp of the quadrotor's center of mass - Initial guess

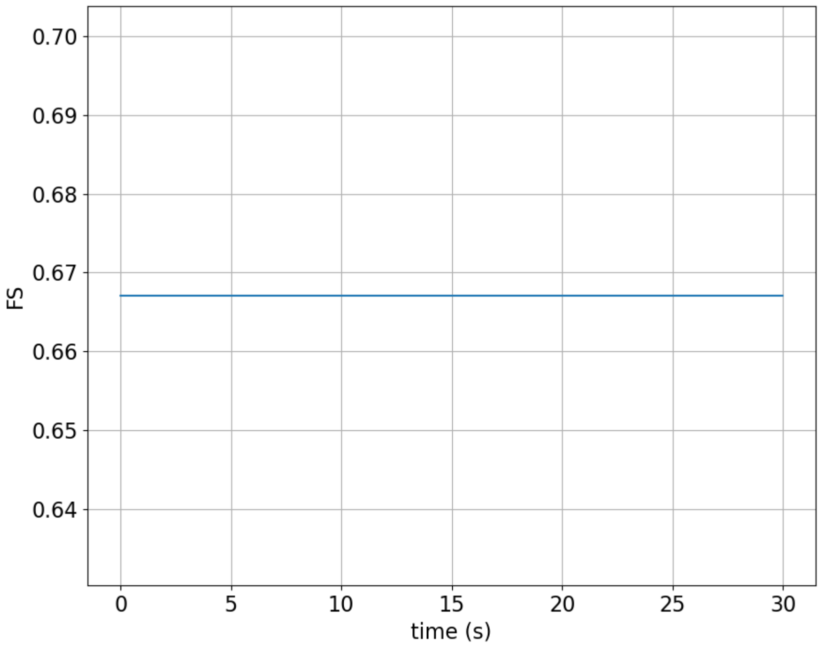

Figure 3.2: Force Fs of the quadrotor - Initial guess

3.2 Cost function computation

The cost function of the optimal control problem has been chosen to be quadratic both for the states and the inputs. Basically, it defines the overall cost as related to the square of the difference between each state-input approximation of the optimal trajectory and and the reference values, i.e., the 2-norm of the error. The overall optimal control problem reads:

$$\begin{aligned} \min_{x_t, u_t} \sum_{t=0}^{T-1} & x_t^T Q_t x_t + u_t^T R_t u_t + x_T^T Q_T x_T \\ \text{subject to } & x_{t+1} = f\left(x_t, u_t\right) \quad t = 0, \ldots, T-1 \\ & x_0 = x_{\text{init}} \end{aligned} \tag{3.1}$$

A crucial aspect for the cost is represented by the weight matrices, which give the relative importance to each component, concerning both inputs and outputs. They represent a sort of trade-off, i.e., the higher is the cost in control, the lower should be the cost for the actuation and vice-versa. The standard Newton's Method algorithm expects the following matrices:

$$Q_t = \nabla^2 x_t x_t l(x_t, u_t) + \nabla^2 x_t x_t f(x_t, u_t) \lambda_{t+1} \tag{3.2}$$

$$R_t = \nabla^2 u_t u_t l(x_t, u_t) + \nabla^2 x_t x_t f(x_t, u_t) \lambda_{t+1} \tag{3.3}$$

$$S_t = \nabla^2 x_t u_t l(x_t, u_t) + \nabla^2 x_t u_t f(x_t, u_t) \lambda_{t+1} \tag{3.4}$$

Since it is not always possible to guarantee positive semi-definiteness of the matrices ($Q \geq 0$ and $R > 0$), then a regularized version of those matrices have been chosen. Basically, the approximation of the matrices is based on the Hessian of the cost function only, while the Hessian of the dynamics is neglected. This approach makes sure that the LQR problem under analysis is solvable. Q and R matrices' computation reads as follows:

$$\begin{aligned} \tilde{Q}_t &= \nabla_{x_t x_t} l\left(x_t, u_t\right) \\ \tilde{R}_t &= \nabla_{u_t, u_t} l\left(x_t, u_t\right) \\ \tilde{S}_t &= 0 \end{aligned}$$

which lets us be able to set up a regularized optimal control problem.

3.3 Descent direction computation

In Newton's Method, the descent direction needs to be computed by taking the opposite direction with respect to the gradient of the cost function, multiplied by the inverse of the Hessian of that cost. For the sake of this project, instead, the Affine LQR approach has been deviced in order to compute the descent direction. In fact, it can be proven that the optimal solution of the quadratic approximation of the cost function at a given point X_k turns out to be the descent direction itself. Namely, we need to solve the following linear quadratic problem:

$$\begin{aligned} &\min_{\Delta x_t, \Delta u_t} \sum_{t=0}^{T-1} \begin{bmatrix} q_t \\ r_t \end{bmatrix}^T \begin{bmatrix} \Delta x_t \\ \Delta u_t \end{bmatrix} + \begin{bmatrix} \Delta x_t \\ \Delta u_t \end{bmatrix}^T \begin{bmatrix} Q_t & 0 \\ 0 & R_t \end{bmatrix} \begin{bmatrix} \Delta x_t \\ \Delta u_t \end{bmatrix} \\ &\quad + q_T^T X_T + \frac{1}{2} \Delta X_T^T Q_T \Delta X_T \\ &\text{subject to } \Delta x_{t+1} = A_t \Delta x_t + B_t \Delta u_t \quad t = 0, \ldots, T-1 \\ &\text{where } q_t = \nabla_{x_t} l\left(x_t, u_t\right), r_t = \nabla_{u_t} l\left(x_t, u_t\right), A_t^T = \nabla_{x_t} f\left(x_t, u_t\right) \text{ and } B_t^T = \\ &\nabla_{u_t} f\left(x_t, u_t\right) \end{aligned} \tag{3.5}$$

The quadratic optimization program previously shown can be easily solved by using an augmented state:

$$\delta \tilde{x}_t = \begin{bmatrix} 1 \\ \hline \Delta x_t \end{bmatrix} \tag{3.6}$$

The optimal solution of the problem turns out to be the Affine LQR, it reads:

$$\begin{aligned} \Delta u_t^* &= K_t^* \Delta x_t^* + \sigma_t^* \\ \Delta x_{t+1}^* &= A_t \Delta x_t^* + B_t \Delta u_t^* \end{aligned} \qquad t = 0, \ldots, T-1 \tag{3.7}$$

where the expressions of the feedback gains K_t and $\sigma_t$are determined by running backward two Difference Riccati Equations for the matrices P_t and p_t, as follows:

$$\begin{aligned} K_t^* &= -\left(R_t + B_t^T P_{t+1} B_t\right)\left(S_t + B_t P_{t+1} A_t\right) \\ \sigma_t^* &= -\left(R_t + B_t^T P_{t+1} B_t\right)^{-1}\left(r_t + B_t^T p_{t+1}\right) \\ p_t &= q_t + A_t^T p_{t+1} - K_t^*\left(R_t + B_t^T P_{t+1} B_t\right) \sigma_t^* \\ P_t &= Q_t + A_t^T P_{t+1} A_t - K_t^*\left(R_t + B_t^T P_{t+1} B_t\right) K_t^* \end{aligned} \tag{3.8}$$

where: $p_T = q_T$ and $P_t = Q_T$.

3.4 Stepsize selection

For the sake of stepsize's selection, i.e., the computation of the optimal γ, the Armijo's Backtracking rule has been deviced. In the context of optimal control algorithms, such as Newton's Method, determining an appropriate step-size for updating the control inputs can significantly impact the convergence and the computational efficiency of the algorithm itself. Hence, Armijo's selection rule provides a pragmatic and efficient way to dynamically adjust the step-size during each iteration of the algorithm. Its main rationale involves iteratively adjusting the step size, starting from an initial value of the step $\gamma_1 = 1$, then reducing it until a specific sufficient decrease in the cost function is observed. Here below some examples:

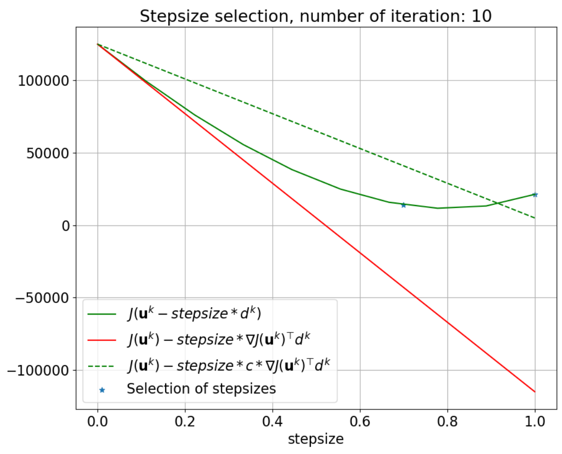

Figure 3.3: Armijo's initial plot

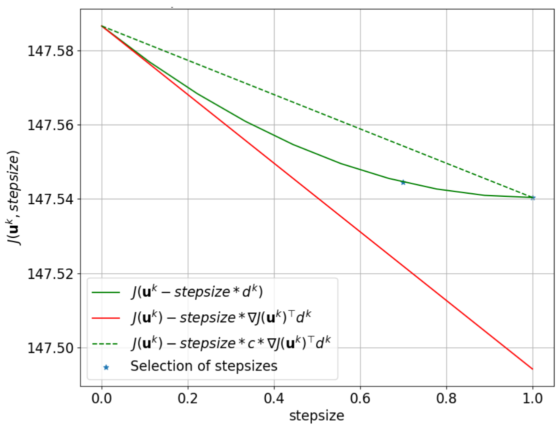

Figure 3.4: Armijo's final plot

3.5 Trajectory update

Once descent direction and optimal stepsize have been properly determined, then a closed-loop version of the update is exploited:

$$
\begin{aligned}
u_t^{k+1} &= u_t^k + \gamma^k \left(K_t^k \left(x_t^{k+1} - x_t^k \right) + \sigma_t^k \right) \\
x_{t+1}^{k+1} &= f\left(x_t^{k+1}, u_t^{k+1} \right)
\end{aligned}
\tag{3.9}
$$

Trajectory update step plays a pivotal role in the Newton's method algorithm, as it dynamically refines the trajectory based on local optimization, ensuring the algorithm converges towards an optimal solution. Also, it leads to a substantial decrease both in the cost and in the descent direction.

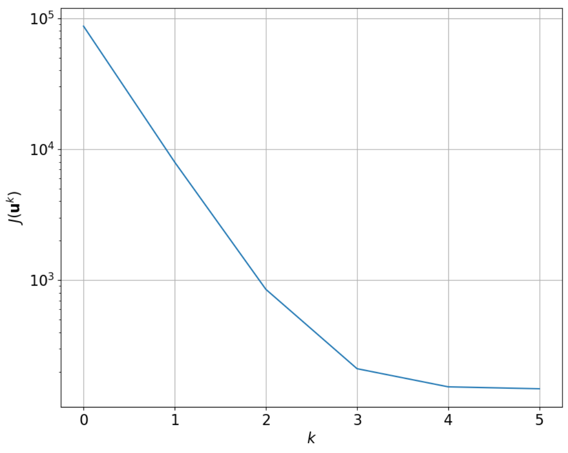

Figure 3.5: Plot of the cost function

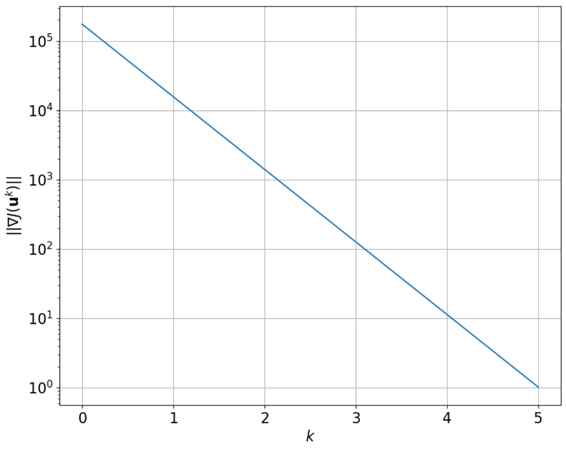

Figure 3.6: Plot of the descent direction

3.6 Algorithm's results

In this brief section we would like to present some results of the Newton's method optimal algorithm. In particular, the optimal trajectories obtained at the last iteration of the algorithm are shown, in order to highlight its convergence capabilities with respect to the reference curves. Therefore, both the requests of task 1 and 2, respectively step and smooth references, have been properly satisfied.

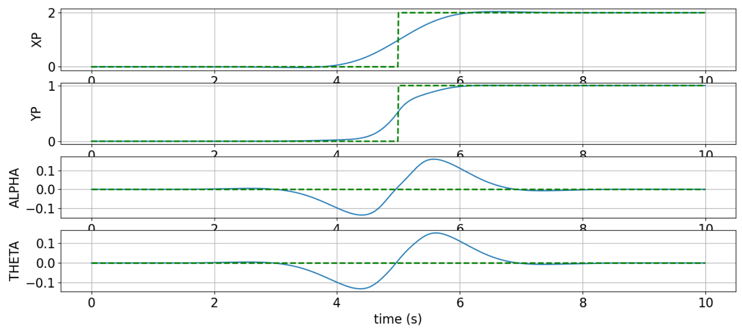

Figure 3.7: First four optimal state trajectories - Task1

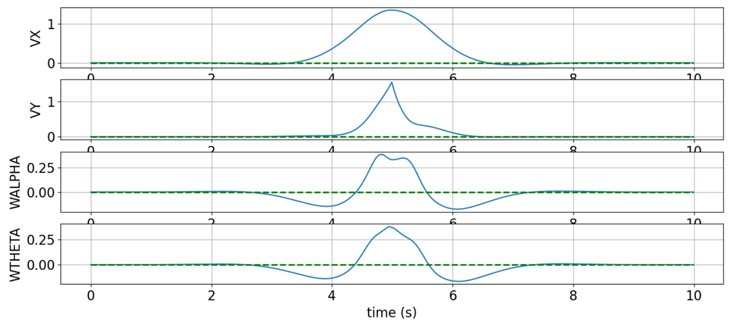

Figure 3.8: Last four optimal state trajectories - Task1

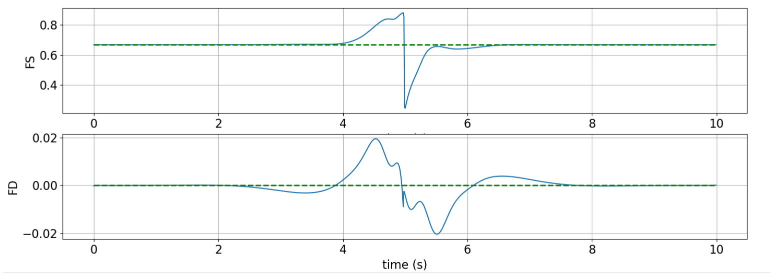

Figure 3.9: Optimal input trajectories - Task1

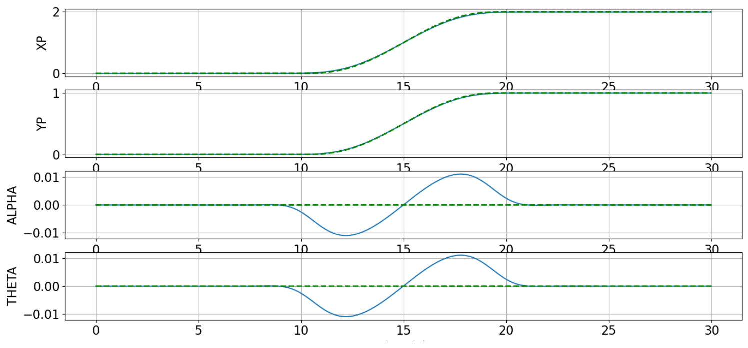

Figure 3.10: First four optimal state trajectories - Task2

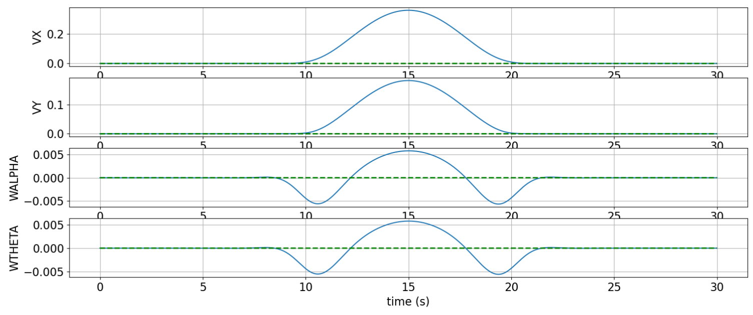

Figure 3.11: Last four optimal state trajectories - Task2

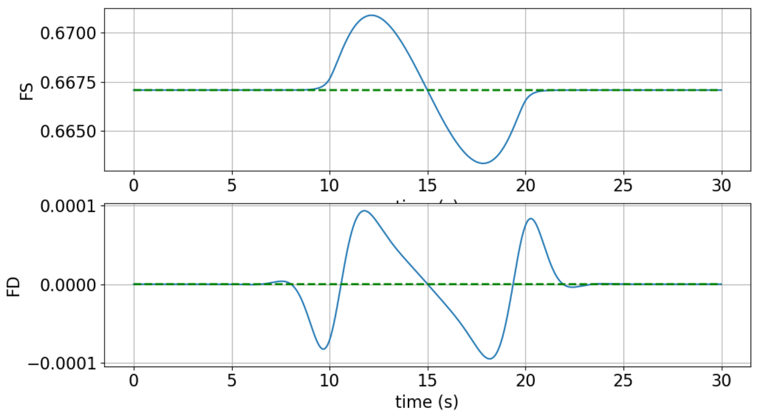

Figure 3.12: Optimal input trajectories - Task2

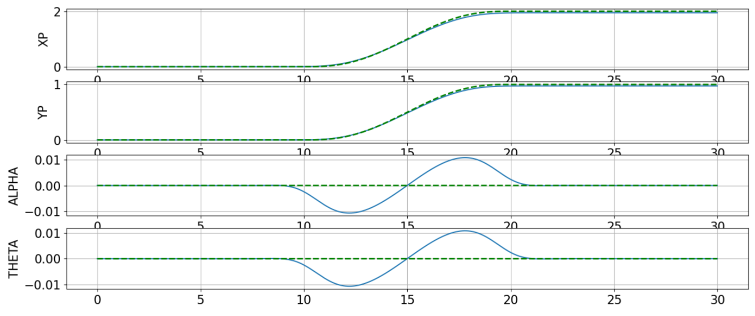

Figure 3.13: First four intermediate state trajectories - Task2

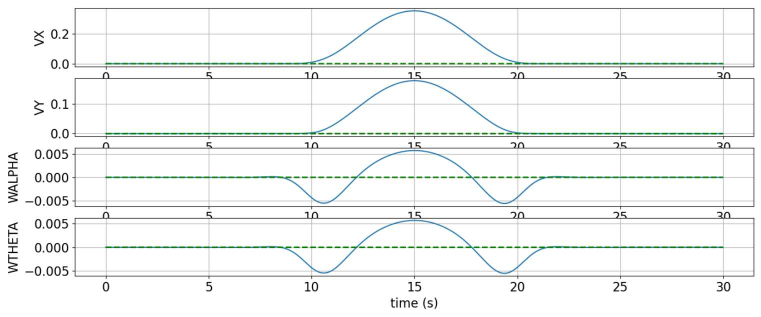

Figure 3.14: Last four intermediate state trajectories - Task2

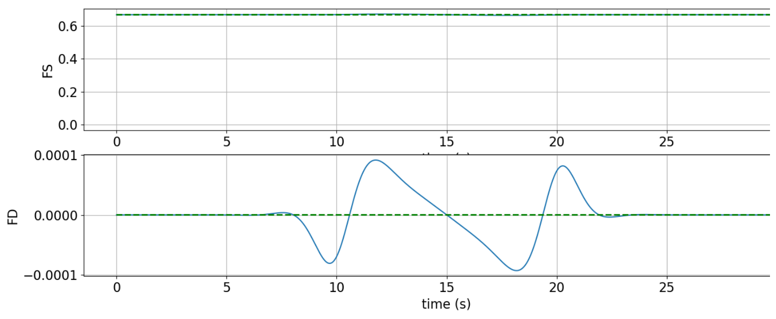

Figure 3.15: Intermediate input trajectories - Task2

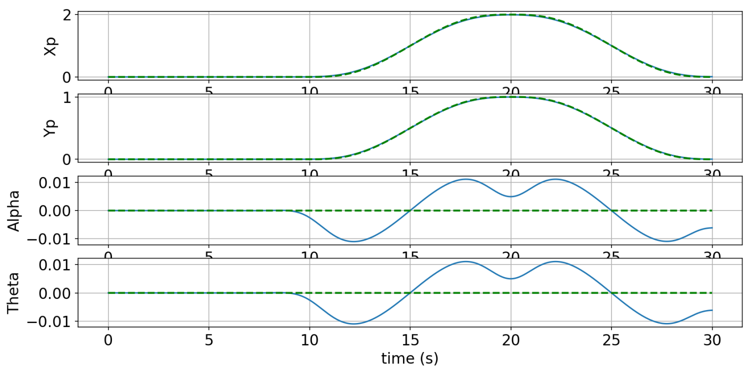

Figure 3.16: First four optimal state trajectories - Double S

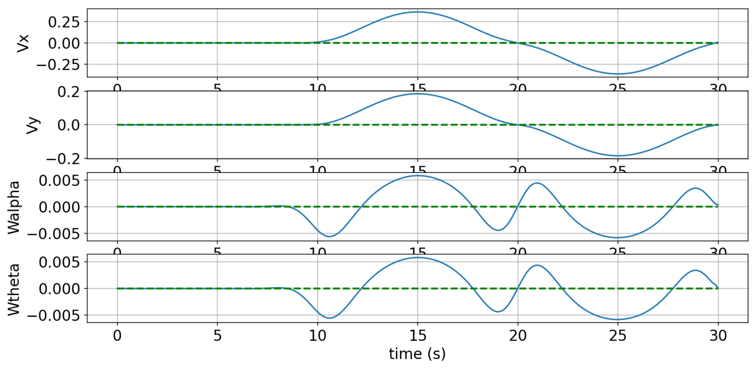

Figure 3.17: Last four optimal state trajectories - Double S

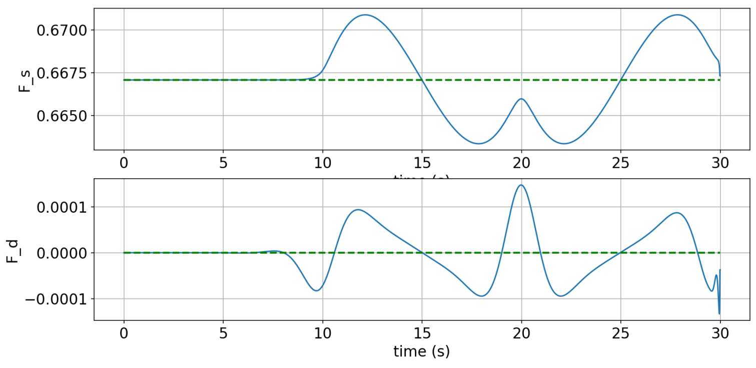

Figure 3.18: Optimal input trajectories - Double S

Chapter 4

Trajectory Tracking

4.1 LQR implementation

Once the optimal trajectory is computed from the Newton's algorithm, then tracking via Linear Quadratic Regulator is addressed. At a first glance, a linearized version of the non-linear system is computed around the optimal trajectory:

$$\Delta x_{t+1} = A_t^{opt} \Delta x_t + B_t^{opt} \Delta u_t$$

where matrices A_t^{opt} and B_t^{opt} are the Jacobian matrices, which are needed in order to evaluate the linear approximation of the system. This latter, however, turns out to be only valid in a neighborhood of the optimal trajectory.

$$\begin{aligned} A_t^{opt} &:= \nabla_{x_t} f\left(x_t^{opt}, u_t^{opt}\right)^T \\ B_t^{opt} &:= \nabla_{u_t} f\left(x_t^{opt}, u_t^{opt}\right)^T \end{aligned}$$

the Jacobian matrices are evaluated by means of the function *dynamics()* contained in the *Dynamics.py* file. Once the linearized version of the discrete-time system is computed around the optimal trajectory, it is finally possible to determine the actual expression of the optimal controller, which aims to stabilize the system around the optimal trajectory. The following linear quadratic program needs to be solved in order to find the optimal controller's expression:

$$\begin{aligned} \min_{\Delta x_t, \Delta u_t} & \sum_{t=0}^{T-1} \Delta x_t^T Q_t^{reg} \Delta x_t + \Delta u_t^T R_t^{reg} \Delta u_t + \Delta x_T^T Q_T^{reg} \Delta x_T \\ \text{subject to } & \Delta x_{t+1} = A_t^{opt} \Delta x_t + B_t^{opt} \Delta u_t \quad t = 0, \ldots, T-1 \\ & \Delta_{x_0} = 0 \end{aligned} \tag{4.1}$$

with $Q_{\mathrm{t}^{\mathrm{reg}}} \geq 0$ and $R_{\mathrm{t}^{\mathrm{reg}}} > 0$

The optimal controller's gain of the LQR solution requires the preliminary computation of the matrices P_t, which are evaluated by backward iterating $(t = T-1, \ldots, 0)$ the Difference Riccati Equation, starting from $P_T^{\text{reg}} = Q^{\text{reg}}$ such as:

$$\begin{aligned} P_t =& Q_t^{reg} + A_t^{opt,T} P_{t+1} A_t^{opt} - \left(A_t^{opt,T} P_{t+1} B_t^{opt}\right) \left(R_t^{reg} + B_t^{opt,T} P_{t+1} B_t^{opt}\right)^{-1} \cdot \\ & \left(B_t^{opt,T} P_{t+1} A_t^{opt}\right) \end{aligned} \tag{4.2}$$

Hence, for $(t = 0, \ldots, T-1)$ it is possible to compute the feedback gains $K_{t_{\text{reg}}}$ as follows:

$$K_t^{reg} = -\left(R_t^{reg} + B_t^{opt,T} P_{t+1} B_t^{opt}\right)^{-1} \left(B_t^{opt,T} P_{t+1} A_t^{opt}\right) \tag{4.3}$$

Finally, the feedback controller on the linearization of the non linear system can be computed, in order to track the optimal trajectory $(x_{\text{opt}}, u_{\text{opt}})$ which has been computed in the former step.

$$\begin{gathered} u_t^{reg} = u_t^{\text{opt}} + K_t^{reg}\left(x_t^{reg} - x_t^{\text{opt}}\right) \\ x_{t+1}^{reg} = f\left(x_t^{reg}, u_t^{reg}\right) \end{gathered} \tag{4.4}$$

Notice that the system's linearization is valid only around a neighborhood of the optimal trajectory, i.e., we could be able to track the desired reference even by changing initial conditions in a reasonable manner. It turns out that the linear closed-loop system

$$\Delta x_{t+1} = (A_t^{\text{opt}} + B_t^{\text{opt}} K_t^{\text{reg}})\Delta x_t \tag{4.5}$$

is globally exponentially stable (GAS) if and only if x^{opt} is locally exponentially stable (LAS) for the closed-loop non linear system.

In order to test the tracking capabilities of the controller, some disturbances have been imposed to initial states:

$$\alpha(0) = 0.3 \quad \theta(0) = 0.25 \tag{4.6}$$

The following plots show how the LQR controller is able to react to the initial disturbances and settle to the true reference in few steps.

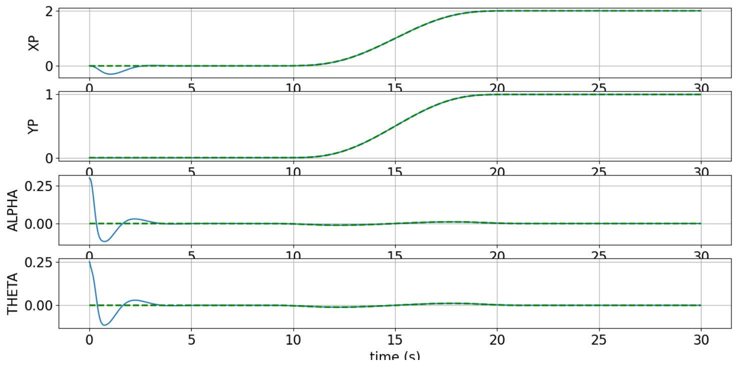

Figure 4.1: First four optimal state trajectories - LQR

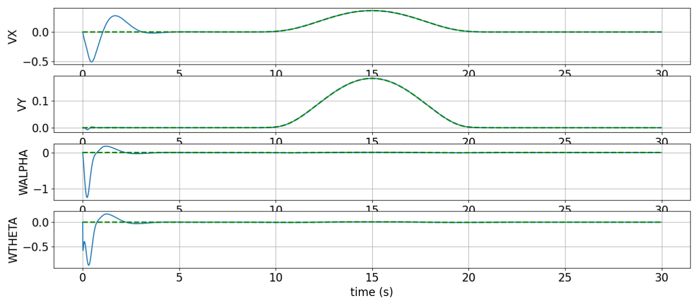

Figure 4.2: Last four optimal state trajectories - LQR

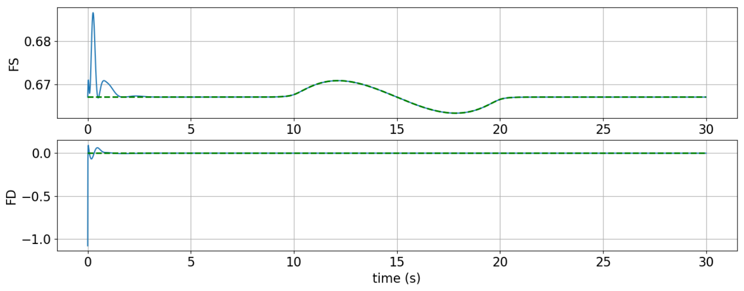

Figure 4.3: Optimal input trajectories - LQR

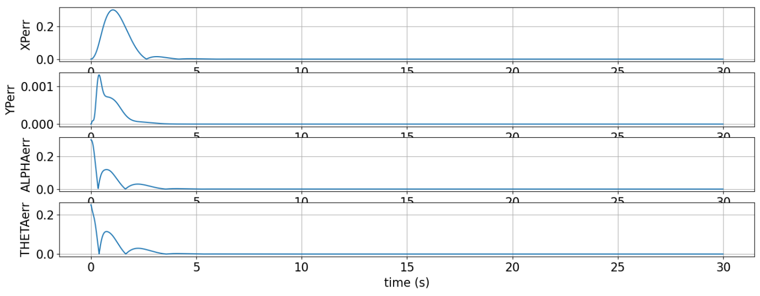

Figure 4.4: First four states tracking errors - LQR

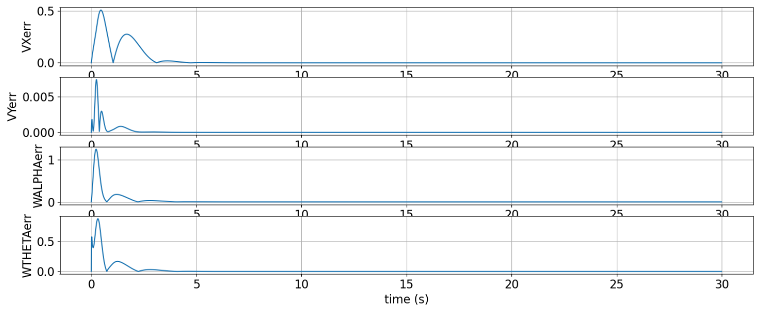

Figure 4.5: Last four state tracking errors - LQR

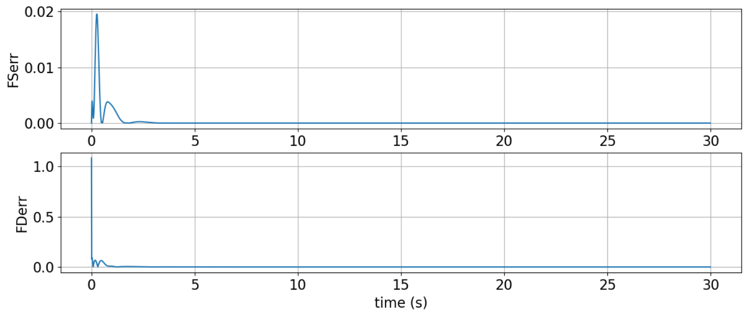

Figure 4.6: Input tracking errors - LQR

4.2 MPC implementation

Model Predictive Control (MPC) is a powerful and widely applied approach in solving optimal control problems, as trajectory tracking of optimal trajectories. As its core, it formulates the optimal control problem as an optimization task, aiming to find an optimal sequence of control inputs over a finite prediction horizon, while considering system dynamics and constraints. At each time step, the optimization problem is solved, generating a control sequence. Only the first control input is applied to the system, and the process repeats, incorporating new measurements and refining the control sequence at each iteration. To settle down the optimal control problem, a linearized version of the system needs to be computed around the optimal trajectory, which has been previously generated by the Newton's method. The following Jacobian matrices are evaluated:

$$\begin{aligned} A_t^{opt} &:= \nabla_{x_t} f\left(x_t^{opt}, u_t^{opt}\right)^T \\ B_t^{opt} &:= \nabla_{u_t} f\left(x_t^{opt}, u_t^{opt}\right)^T \end{aligned} \tag{4.7}$$

In addition, positive definite weight matrices Q_t, R_t and Q_T are chosen to build up a feasible problem, as follows:

$$\begin{aligned} \min_{x_t,\ldots,x_{t+T},u_t,\ldots,u_{t+T-1}} \quad & \sum_{\tau=t}^{t+T-1} \ell_\tau(x_\tau, u_\tau) + \ell_{t+T}(x_{t+T}) \\ \text{subject to} \quad & x_{\tau+1} = f(x_\tau, u_\tau), \quad \forall \tau = t, \ldots, t+T-1 \\ & x_\tau \in X, \quad u_\tau \in U, \quad \forall \tau = t, \ldots, t+T \\ & x_\tau = x_{\text{meas}}, \end{aligned} \tag{4.8}$$

While the problem's setting is handled in the main file, a specific file called *MPC_Solver* is called in order to solve the optimal control problem at each iteration. Minimization is handled via a convex optimizer from the *cvxpy* library, by taking as parameters the cost to be minimized and an ordered set of path constraints. At each iteration, the algorithm finds the optimal state-input trajectory within the selected time horizon, whose length is specified by the T_{pred} variable. The graphs below show that MPC is a stable and accurate method for tracking optimal trajectory and managing multiple constraints simultaneously. PC's results would be identical to those of the LQR, in case no additional constraints in addition to the dynamics were considered by the algorithm. One of the main limits of the method, however, can be found in its high computational burden. Despite a longer convergence time than the LQR counterpart, it remains a viable and solid technique for optimal control applications. The following path constraints are applied: $-10 <= Fs <= 10$, $-0.5 <= Fd <= 0.5$, $0.1 <= Xp <= 2.1$, $-0.1 <= Yp <= 1.1$, $-0.5 <= \alpha <= 0.5$, $-0.5 <= \theta <= 0.5$.

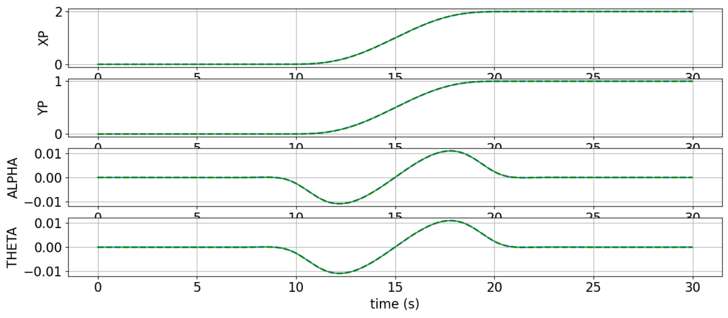

Figure 4.7: First four optimal state trajectories - MPC

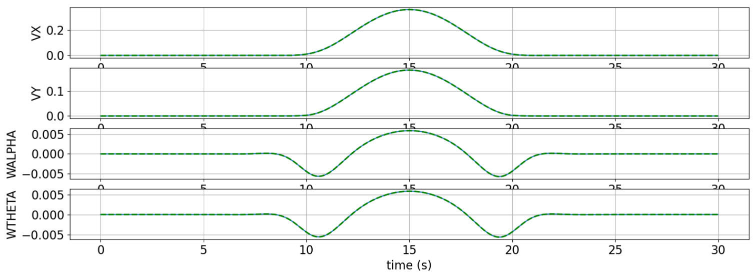

Figure 4.8: Last four optimal state trajectories - MPC

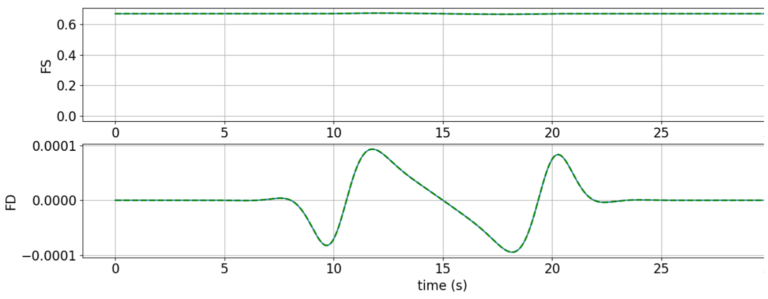

Figure 4.9: Optimal input trajectories - MPC

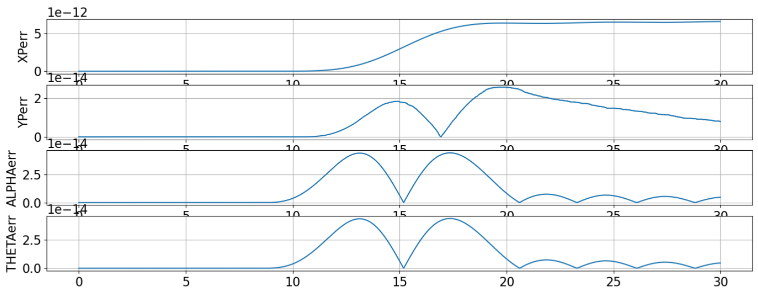

Figure 4.10: First four state tracking errors - MPC

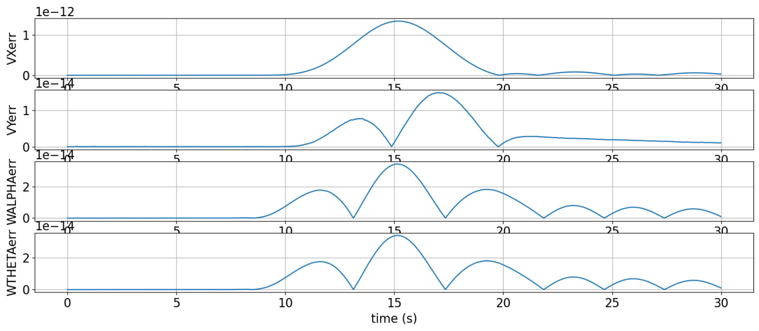

Figure 4.11: Last four state tracking errors - MPC

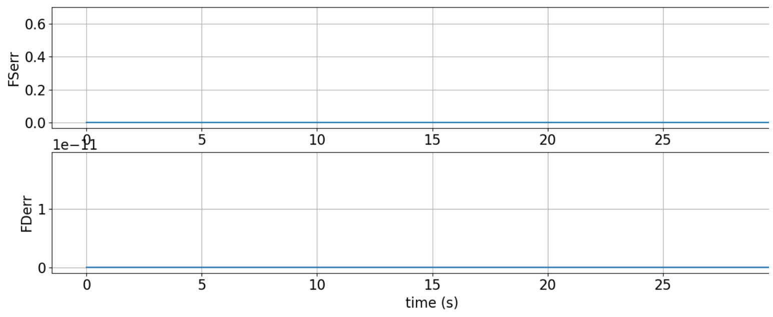

Figure 4.12: Input tracking errors - MPC

Chapter 5

Visualization

5.1 Animation setup

The visualization is implemented by animating some geometrical forms that aim to represent a quadrotor with the *Matloplotlib.Animation* tools. We imported some patches from the *Matplotlib.patches* library, in particular a rectangle to represent the body of the quadrotor and three circles representing respectively the suspended load and the two rotor (it's a 2D representation). Then we connect these circles with the rectangular body with three lines. To effectively animate them, we need to assign them the right coordinate of our code's variable. The size of these object (i.e., height and width of the rectangle, radius of the circles and length of the lines) is voluntarily enlarged to show the quadrotor's movements better. The rectangle is animated in both position and orientation. The following figure shows a screenshot of the animation, with the elements mentioned above:

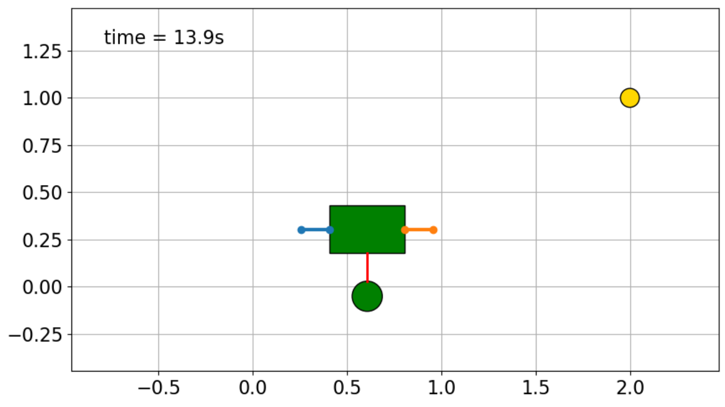

Figure 5.1: A screenshot of the animation

5.2 Animation sequence

After the setup of the sizes and the connection betweeen the pieces mentioned before, we have to animate them, i.e. associate to each of the component some coordinate that change in time. The coordinates will be, of course, defined by the state space variable. We build the animation with the values in the vectors. The rectangle will always be centered in position (x_p, y_p), and will rotate with respect to the horizontal line by an angle θ. The arms move solidal with the rectangle in x and y with a certain offest that depends on their length and the angle θ and they also rotate of an angle θ when the quadrotor rolls. Then, the joint of the load moves attached to the rectangle in x and y and rotates of an angle α. In the end there's the load that is at a distance $(l \sin \alpha, -l \cos \alpha)$ (where l length of the joint) from the attachment of the joint to the rectangle.

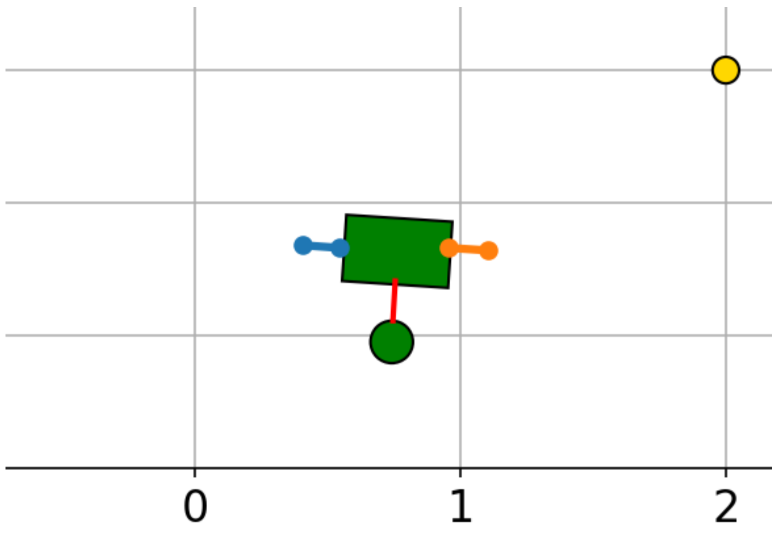

Figure 5.2: A frame of the animation with the quadrotor in motion

Conclusions

In conclusion, the results obtained confirm the effectiveness of the optimal control feedback laws that have been specifically designed for the quadrotor with the suspended load. The graphs shown throughout the report, in fact, underline the remarkable speed of Newton's algorithm in finding the optimal trajectories for the system, while minimizing the cost of states and inputs with respect to the reference curves. We would like to highlight some additional features of the application that were developed during the design phase, which could be valuable resources for future improvements of the algorithm.

- In *Dynamics.py* file, a function named *dynamics_equilibrium()* has been developed, which allows for the calculation of equilibrium points distinct from hovering. In particular, it uses the Python's routine *scipy.optimize.fsolve()* to calculate the equilibria for the quadrotor.
- Within the *RefCurve.py* file, a specific function has been designed to generate a Double S trajectory for the quadrotor between two equilibrium points. It represents a more complex and sophisticated way of interpolating the equilibrium points, constituting a solid starting point for future improvements.

A critical point during the development of the project and in the refinement of the optimal control algorithm was found in the choice of cost matrices. Depending on the type of reference curve considered, the values of these matrices have been chosen ad-hoc. In this regard, possible future investigations can deepen this point, finding better values for such crucial matrices.