UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Optimal Control

Optimal Control of a Bipedal Robot

Professor: **Giuseppe Notarstefano**

Students:
Agatino Ricciardi
Riccardo Selleri
Davide Verri

Academic year 2022/2023

Abstract

In this report we will discuss the implementation of an optimal control algorithm to a simplified walking robot, the so-called compass gait model.

The optimization algorithm required for the generation of the optimal trajectory is the Newton's Method. Once it is implemented, the optimal trajectory needs to be tracked via the use of a Regulator.

The project is divided into four main tasks:

- Discretization of the system's dynamics and computation of the gradients and the hessians.
- Define a step between two chosen configurations, our equilibria. The transition of the robot must be done by exploiting the Newton's Method.
- A further step must be actuated defining a smooth reference trajectory. Once again, the Newton's Method must be applied.
- Lastly, our goal is to track the optimal trajectory generated in the previous task via Regularization.

In conclusion, a simple animation of the two-legged robot that satisfies the request of the trajectory tracking task must be done.

Contents

Task 0 Problem Setup

In this project we had to develop an optimal control strategy for a simplified walking robot, the so-called compass gait model, composed of two spokes, with a pin joint at the hip. This model considers the dynamics of both the stance leg and the swing one. To model the dynamics of swing, point masses to each of the legs are added. For actuation, there is a torque source at the hip, resulting in swing dynamics. [1]

We describe the state of the robot with four state variables: θ_{st}, θ_{sw}, $\dot{\theta}_{st}$ and $\dot{\theta}_{sw}$ where the abbreviation st is shorthand for the stance leg and sw for the swing leg. Denoting $q = [\theta_{sw}, \theta_{st}]^\top$ and $u = \tau$, with τ being the torque.

The dynamics of the compass leg is given by the following formula:

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

with:

$$M(q) = \begin{bmatrix} mb^2 & -mlb\cos(\theta_{st}-\theta_{sw}) \\ -mlb\cos(\theta_{st}-\theta_{sw}) & (m_h+m)ml^2 + ma^2 \end{bmatrix} \quad G(q) = \begin{bmatrix} -mgb\sin\theta_{sw} \\ -(m_h l + ma + ml)g\sin\theta_{st} \end{bmatrix}$$

$$C(q,\dot{q}) = \begin{bmatrix} 0 & mlb\sin(\theta_{st}-\theta_{sw})\dot{\theta}_{st} \\ mlb\sin(\theta_{st}-\theta_{sw})\dot{\theta}_{sw} & 0 \end{bmatrix} \quad B = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Discretization of the dynamics

In this section we discuss how we approached the definition of the discretized dynamics of the bipedal robot and the computation of the gradients and of the hessians with respect to the states and the input torque. The states and input vectors are defined as follows:

$$\mathbf{x} = \begin{bmatrix} \theta_{sw} & \dot{\theta}_{sw} & \theta_{st} & \dot{\theta}_{st} \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix} \quad \mathbf{u} = \tau$$

First thing that we had to do was to implement the dynamics given by the problem, for this purpose a *dynamics.py* file was coded. After setting up the problem's parameters, the idea was to define a function named **statedot**

that computes and discretizes our dynamics, returning the variables $\dot{x}_1$, $\dot{x}_2$, $\dot{x}_3$, $\dot{x}_4$ at the following discretized time step. The discretization was done via Euler method:

$$\dot{x} \simeq \frac{x_{t+1} - x_t}{dt}$$

Where dt is the sampling time, x_{t+1} is the state vector at the next time instant and x_t is the state vector at the current time instant.

The discretized state-space is represented as follows:

$$\begin{aligned}
x_{1,t+1} &= x_{1,t} + dt x_{2,t} \\
x_{2,t+1} &= x_{2,t} + dt\left(\frac{1}{mb^2}\right)\Big\{u - mlb\sin(x_{3,t} - x_{1,t})x_{4,t}^2 - mbg\sin(x_{1,t}) + \\
&\quad + mlb\cos(x_{3,t} - x_{1,t})\frac{1}{(m_h + m)l^2 + ma^2 - ml^2\cos^2(x_{3,t} - x_{1,t})} \\
&\quad \Big[-u - mlb\sin(x_{3,t} - x_{1,t})x_{2,t}^2 - (m_h l + ma + ml)g\sin(x_{3,t}) + \frac{l}{b} \\
&\quad \cos(x_{3,t} - x_{1,t})\Big(u - mlb\sin(x_{3,t} - x_{1,t})x_{4,t}^2 - mbgsin(x_{1,t})\Big)\Big]\Big\} \\
x_{3,t+1} &= x_{3,t} + dt x_{4,t} \\
x_{4,t+1} &= x_{4,t} + dt\frac{1}{(m_h + m)l^2 + ma^2 - ml^2\cos^2(x_{3,t} - x_{1,t})} \\
&\quad \Big[-u - mlb\sin(x_{3,t} - x_{1,t})x_{2,t}^2 - (m_h l + ma + ml)g\sin(x_{3,t}) + \frac{l}{b} \\
&\quad \cos(x_{3,t} - x_{1,t})\Big(u - mlb\sin(x_{3,t} - x_{1,t})x_4^2 - mbgsin(x_{1,t})\Big)\Big]
\end{aligned}$$

Then, through the use of the *SymPy* library, we defined the state variables x_1, x_2, x_3, x_4 and the input variable u as symbols, otherwise we would've ran into computational problems when trying to further derivate and evaluate the hessian matrices.

Computation of the gradients

In order to linearize the system, the gradients must be computed:

$$\nabla_x f(x,u) = \begin{bmatrix}\nabla_x f_1(x,u) & \nabla_x f_2(x,u) & \nabla_x f_3(x,u) & \nabla_x f_4(x,u)\end{bmatrix} = \begin{bmatrix}\frac{\partial f_1(x,u)}{\partial x_1} & \cdots & \frac{\partial f_4(x,u)}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_1(x,u)}{\partial x_4} & \cdots & \frac{\partial f_4(x,u)}{\partial x_4}\end{bmatrix}$$

$$\nabla_u f(x,u) = \begin{bmatrix} \nabla_u f_1(x,u) & \nabla_u f_2(x,u) & \nabla_u f_3(x,u) & \nabla_u f_4(x,u) \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1(x,u)}{\partial u} & \dots & \frac{\partial f_4(x,u)}{\partial u} \end{bmatrix}$$

To compute the gradients we thought of defining two empty lists, one committed to derivate the symbolic values of our variables, leaving it generic, while the other one, through the use of the *lambdify* function, is dedicated to the substitution of the values of interests (the equilibria) inside the variables.

This is done through a double *for cycle* (in the case of the **gradient of x**, for the **gradient of u** a single one is enough), with the nested cycle differentiating as requested and substituting with the *lambdify* function, while the external fills the two lists described above, two big matrices, the generic one ready to be derivated the obtain the hessian, and the other ready to receive the numerical values that we will embed into it.

Computation of the hessians

$$\nabla^1_{xx} f(x,u) = \begin{bmatrix} \frac{\partial^2 f_1(x,u)}{\partial^2 x_1} & \frac{\partial^2 f_1(x,u)}{\partial x_1 \partial x_2} & \dots & \frac{\partial^2 f_1(x,u)}{\partial x_1 \partial x_4} \\ \frac{\partial^2 f_1(x,u)}{\partial x_2 \partial x_1} & \frac{\partial^2 f_1(x,u)}{\partial^2 x_2} & \dots & \frac{\partial^2 f_1(x,u)}{\partial x_2 \partial x_4} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f_1(x,u)}{\partial x_4 \partial x_1} & \frac{\partial^2 f_1(x,u)}{\partial x_4 \partial x_2} & \dots & \frac{\partial^2 f_1(x,u)}{\partial^2 x_4} \end{bmatrix} \quad \nabla_{xx} f(x,u) = \begin{bmatrix} \nabla^1_{xx} f(x,u) \\ \nabla^2_{xx} f(x,u) \\ \nabla^3_{xx} f(x,u) \\ \nabla^4_{xx} f(x,u) \end{bmatrix}$$

$$\nabla_{uu} f(x,u) = \begin{bmatrix} \nabla_{uu} f_1(x,u) & \dots & \nabla_{uu} f_4(x,u) \end{bmatrix} = \begin{bmatrix} \frac{\partial^2 f_1(x,u)}{\partial^2 u} & \dots & \frac{\partial^2 f_4(x,u)}{\partial^2 u} \end{bmatrix}$$

$$\nabla_{xu} f(x,u) = \begin{bmatrix} \frac{\partial^2 f_1(x,u)}{\partial x_1 \partial u} & \frac{\partial^2 f_1(x,u)}{\partial x_2 \partial u} & \dots & \frac{\partial^2 f_1(x,u)}{\partial x_4 \partial u} \\ \frac{\partial^2 f_2(x,u)}{\partial x_1 \partial u} & \frac{\partial^2 f_2(x,u)}{\partial x_2 \partial u} & \dots & \frac{\partial^2 f_2(x,u)}{\partial x_4 \partial u} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f_4(x,u)}{\partial x_1 \partial u} & \frac{\partial^2 f_4(x,u)}{\partial x_2 \partial u} & \dots & \frac{\partial^2 f_4(x,u)}{\partial x_4 \partial u} \end{bmatrix}$$

The same process is employed in the hessians computation, adding an extra cycle for the **hessian of x**, since our idea was to build this 16×4 matrix as a block of four 4×4 matrices.

Substitution of the values inside gradients and hessians

Once we have the structures ready to be filled, we define a function named **dynamics**. Since we previously used the *lambdify* function on the symbolic functions that expressed our dynamics, we can now use the **dynamics** function in order to substitute our values of interests, both the states and the input, inside our gradients and hessians.

Task 1 Trajectory exploration: step ahead

In this section, we discuss the evaluation and computation of the optimal transition between two equlibria, as a result of the step given from one equilibrium to the other one. The reference used for this task is a **step function**.

In particular, we asked our robot to move a leg up to 40° in 3.5 seconds. At first, we define the equilibrium points as:

$$x_{1,e} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \quad x_{2,e} = \begin{bmatrix} 0 & 40 & 0 & 0 \end{bmatrix}$$

After this choice we applied a step between the two equilibria. The optimal transition from one equilibrium state to the other is found by exploiting the Newton's Algorithm for optimal control.

Newton's algorithm

The Newton's algorithm is defined as:

$$u^{k+1} = u^k - \gamma^k \Delta u^k$$

where

$$\Delta u^k = \arg \min_{\Delta u} \nabla J(u^k)^T \Delta u + \Delta u^T \nabla^2 J(u^k) \Delta u$$

In order to compute the different parts of the equations, for each iteration k we have to evaluate:

$$\nabla_{x_t} f(x_t^k, u_t^k), \nabla_{u_t} f(x_t^k, u_t^k), \nabla_{x_t} \ell_t(x_t^k, u_t^k), \nabla_{u_t} \ell_t(x_t^k, u_t^k), \nabla_{x_T} f(x_T^k)$$

Then we have to solve the co-state equation:

$$\lambda_t^k = \nabla_{x_t} f(x_t^k, u_t^k) \lambda_{t+1}^k \nabla_{x_t} \ell_t(x_t^k, u_t^k) \quad t = T-1, \cdots, 1$$

$$\text{with } \lambda_T^k = \nabla \ell_T(x_T^k)$$

After doing so, we must compute the following matrices for all $t = 0, \cdots, T-1$:

$$\begin{aligned} Q_t^k &:= \nabla_{x_t x_t} \ell_t(x_t^k, u_t^k) + \nabla_{x_t x_t} f_t(x_t^k, u_t^k) \cdot \lambda_{t+1}^k \\ R_t^k &:= \nabla_{u_t u_t} \ell_t(x_t^k, u_t^k) + \nabla_{u_t u_t} f_t(x_t^k, u_t^k) \cdot \lambda_{t+1}^k \\ S_t^k &:= \nabla_{x_t u_t} \ell_t(x_t^k, u_t^k) + \nabla_{x_t u_t} f_t(x_t^k, u_t^k) \cdot \lambda_{t+1}^k \end{aligned}$$

$$\text{and } Q_T^k := \nabla_{x_T x_T} \ell_T(x_T^k).$$

LQR problem with affine dynamics

We can now consider a LQR problem with affine dynamics:

$$\begin{aligned} \min_{\Delta \mathbf{x}, \Delta \mathbf{u}} \quad & \sum_{t=0}^{T-1} \begin{bmatrix} q_t \\ r_t \end{bmatrix}^T \begin{bmatrix} \Delta x_t \\ \Delta u_t \end{bmatrix}^T + \frac{1}{2} \begin{bmatrix} \Delta x_t \\ \Delta u_t \end{bmatrix}^T \begin{bmatrix} Q_t & S_t^T \\ S_t & R_t \end{bmatrix} \begin{bmatrix} \Delta x_t \\ \Delta u_t \end{bmatrix} + q_T^T x_T + \frac{1}{2} \Delta x_T^T Q_T \Delta x_T \\ \text{subj.to} \quad & \Delta x_{t+1} = A_t \Delta x_t + B_t \Delta u_t + c_t \quad t = 0, \cdots, T-1 \end{aligned} \tag{1}$$

This problem can be conveniently solved by augmenting the state as:

$$\Delta \tilde{x_t} := \begin{bmatrix} 1 \\ \Delta x_t \end{bmatrix}$$

We can rewrite the cost and system matrices as:

$$\begin{gathered} \tilde{Q_t} := \begin{bmatrix} 0 & q_t^T \\ q_t & Q_t \end{bmatrix}, \; \tilde{S_t} := \begin{bmatrix} r_t & S_t \end{bmatrix}, \; \tilde{R_t} := R_t, \\ \tilde{A_t} := \begin{bmatrix} 1 & 0 \\ c_t & A_t \end{bmatrix}, \; \tilde{B_t} := \begin{bmatrix} 0 \\ B_t \end{bmatrix} \end{gathered}$$

The associated LQR problem can finally be solved:

$$\begin{aligned} \min_{\Delta \mathbf{x}, \Delta \mathbf{u}} \quad & \frac{1}{2} \sum_{t=0}^{T-1} \begin{bmatrix} \Delta \tilde{x_t} \\ \Delta u_t \end{bmatrix}^T \begin{bmatrix} \tilde{Q_t} & \tilde{S_t^T} \\ \tilde{S_t} & \tilde{R_t} \end{bmatrix} \begin{bmatrix} \Delta \tilde{x_t} \\ \Delta u_t \end{bmatrix} + \frac{1}{2} \Delta \tilde{x_T}^T Q_T \Delta \tilde{x_T} \\ \text{subj.to} \quad & \Delta x_{t+1} = A_t \Delta x_t + B_t \Delta u_t + c_t \quad t = 0, \cdots, T-1 \\ & \Delta \tilde{x_0} = x_{init} \end{aligned} \tag{2}$$

These matrices are very important since every decision variable contribution to the cost depends on the cost matrices.

- Matrix Q is the hessian of the cost function with respect to x;
- Matrix R is the hessian of the cost function with respect to u;
- Matrix S is the hessian of the cost function with respect to x and u.

We define the regularized cost matrices as:

$$Q = \begin{bmatrix} 1000 & 0 & 0 & 0 \\ 0 & 100 & 0 & 0 \\ 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 100 \end{bmatrix} \quad R = \begin{bmatrix} 10 \end{bmatrix} \quad Q_T = 10 \cdot Q = \begin{bmatrix} 10000 & 0 & 0 & 0 \\ 0 & 1000 & 0 & 0 \\ 0 & 0 & 10000 & 0 \\ 0 & 0 & 0 & 1000 \end{bmatrix}$$

By doing this, more weight is given to the last iterations with respect to the previous ones, since the more iterations are performed by the algorithm, the more precise the solution becomes.

Since we used the augmented formulation, the solution is given by:

$$\Delta u = \tilde{K} \Delta x$$
$$\Delta x_{t+1} = \tilde{A}_t \Delta x_t + \tilde{B} \Delta u_t$$

Where the the value of K is computed at each instant of time using the **Riccati equation**.

All this process is computed by the function *ltv_LQR*

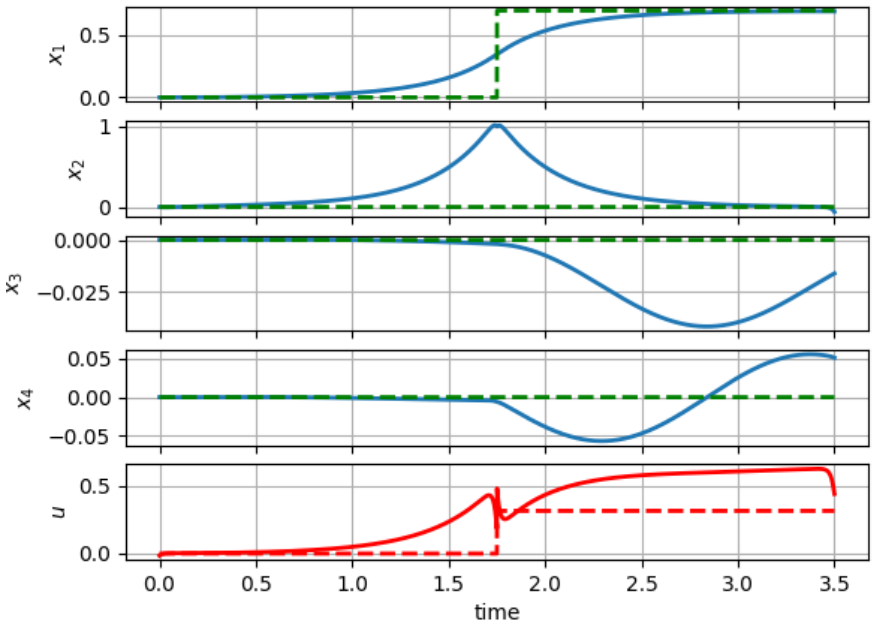

Figure 1: Optimal following of the step reference

The trajectories with a dashed line show the ideal state and input references (step function) while the trajectories with continuous line show the feasible optimal trajectories that are generated by the Newton's algorithm, taking into account the dynamics of the system.

Armijo stepsize and descent direction computation

In order to achieve a faster convergence, we implemented the **Armijo stepsize** selection rule. The stepsize is selected by following the procedure:

- Until the following inequality holds:

$$\ell(x^k) + \bar{\gamma}^i d^k \leq \ell(x^k) + c\bar{\gamma}^i \nabla \ell(x^k)^T d^k$$

- We perform the update of the stepsize γ as:

$$\bar{\gamma}^{i+1} = \beta \bar{\gamma}^i$$

The basic idea behind this method is to select the value γ^k such that the cost is reduced, while the algorithm goes further.

The other important parameter to be defined is the **descent direction**:

$$d^k = (\nabla_{u_t} \ell(x^k, u^k) + B_t^T \cdot \lambda_t^k)^T \Delta u_t$$

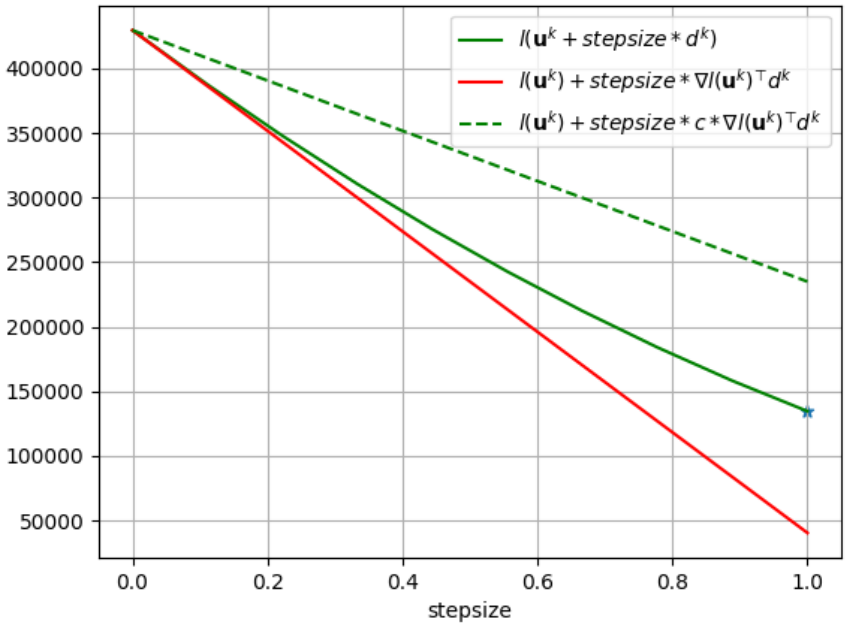

Figure 2: Plot of the Armijo stepsize

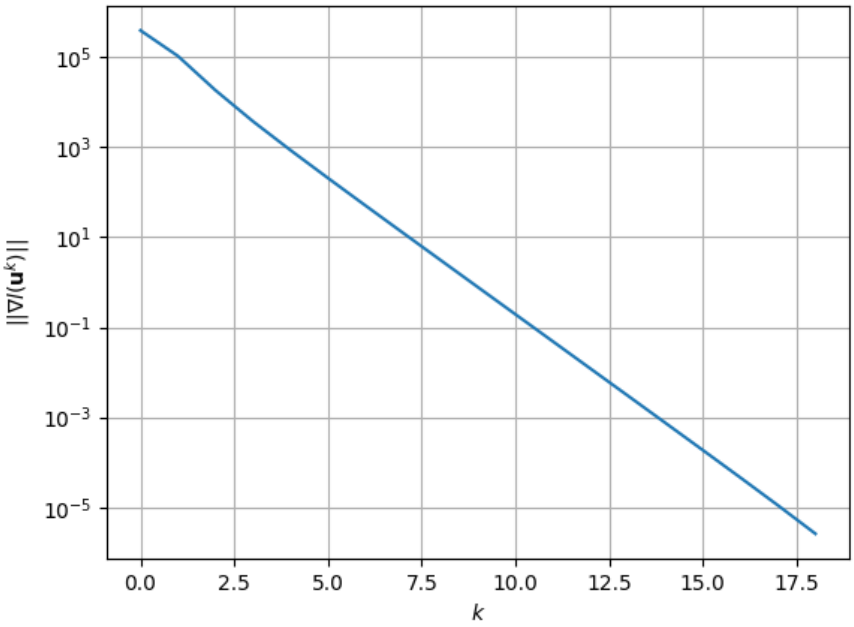

Figure 3: Descent direction

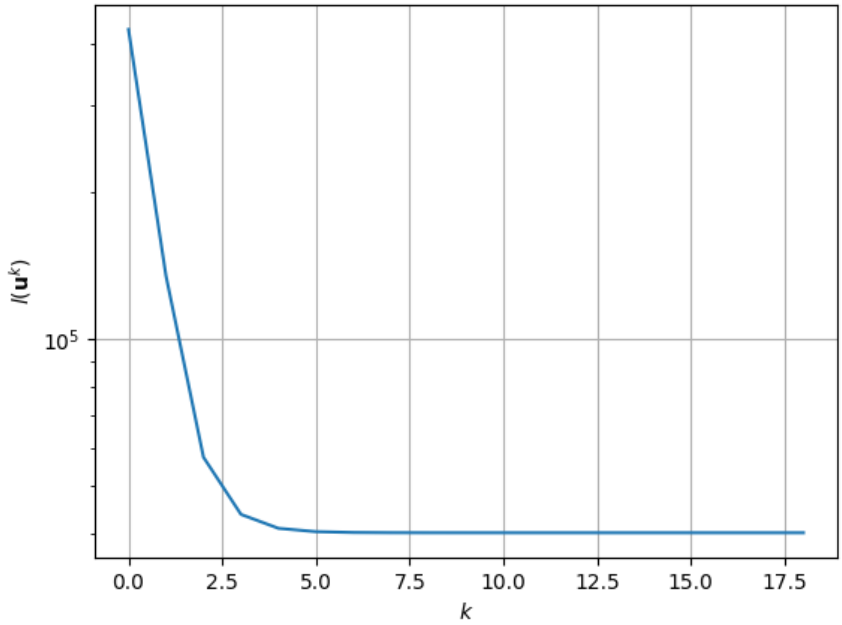

Figure 4: Cost function

Task 2 Trajectory optimization: smooth trajectory

In this section we discuss how we optimize the trajectory between two equilibria by doing a step ahead, this time by following a smoother trajectory with respect to the simpler one previously used in the first task.

While the simple reference trajectory was a step in the angle θ_{sw} related to the swing leg, in this task we choose a **sigmoid function**, in order to get a smoother trajectory.

Sigmoid function reference

The sigmoid is set to start in correspondence of half of the time period T, giving a smoother transition with respect to the step from the initial position p_0 to the final one p_T.

We divided the exponent of the sigmoid by a large constant number, in order to make the transition less steep and avoid some overflow problems that we faced before while running the code.

$$p(t) = p_0 + \sigma(t - T/2)(p_T - p_0)$$

$$\sigma(t) = 1/(1 + e^{-t/100})$$

The stance leg follows a null trajectory as reference, like in the previous case.

The reference angular velocity related to the state variable x_2 follows the derivative of the sigmoid function, while for x_4 we still set a null trajectory.

$$v(t) = \frac{d\sigma(t)}{dt}(p_T - p_0)$$

$$\frac{d\sigma(t)}{dt} = \sigma(t)(1 - \sigma(t))$$

The reference for the input is still set as a sinusoidal function of the state. The successive steps for the smooth trajectory following are exactly the same of the ones described before in the previous task, but the final optimal trajectory turns out different due to the change in the initial reference function. The cost matrices used for this task are the same of the ones used in the previous task, defined at the beginning of the *cost.py* file.

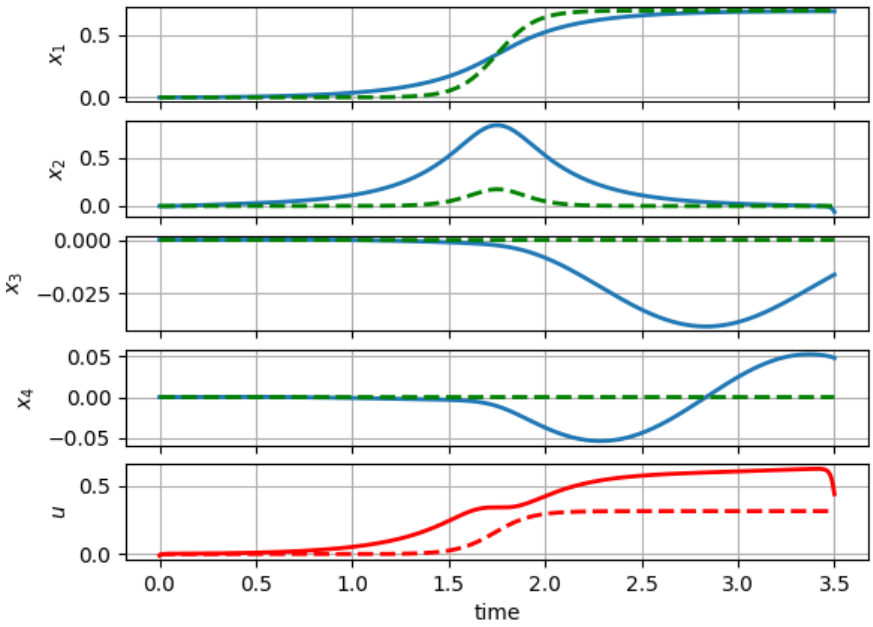

Figure 5: Optimal following of the sigmoid reference

The trajectories with a dashed line show the ideal state and input references (sigmoid function) while the trajectories with continuous line show the feasible optimal trajectories that are generated by the Newton's algorithm, taking into account the dynamics of the system. These optimal feasible trajectories will be used in the following task, in order to track them.

Armijo stepsize and descent direction computation

The stepsize of the Newton's method update is computed with the Armijo rule as described in the previous task. The descent direction continuously decreases with the iterations, up to 10^{-6} that it is used as **stopping condition** for the algorithm. The cost continuously decreases, but tends to stabilize when low values of the descent direction are reached.

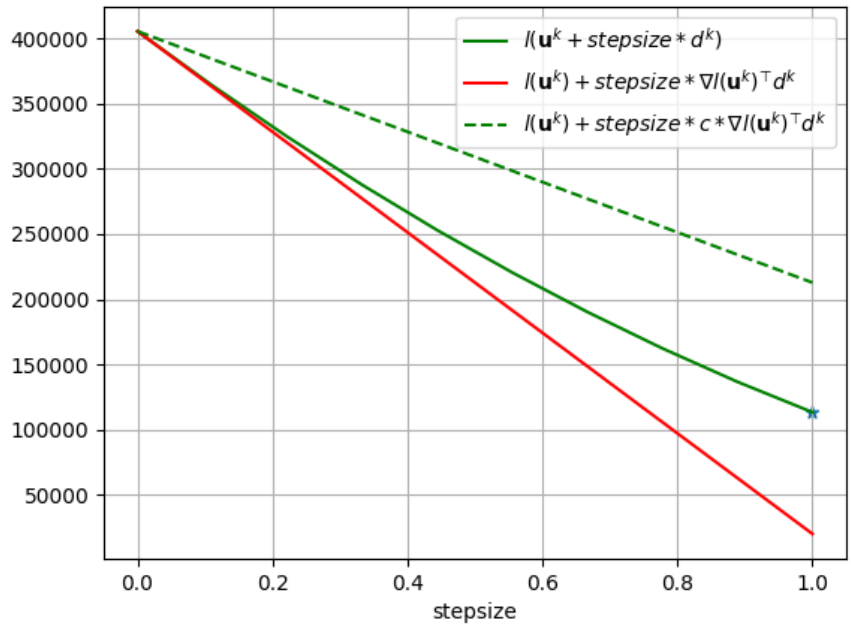

Figure 6: Plot of the Armijo stepsize

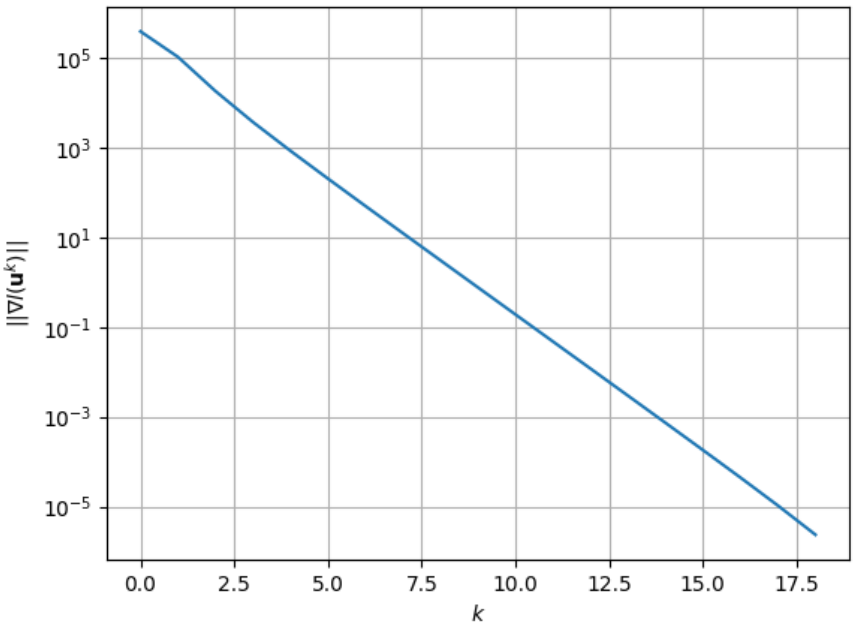

Figure 7: Descent direction

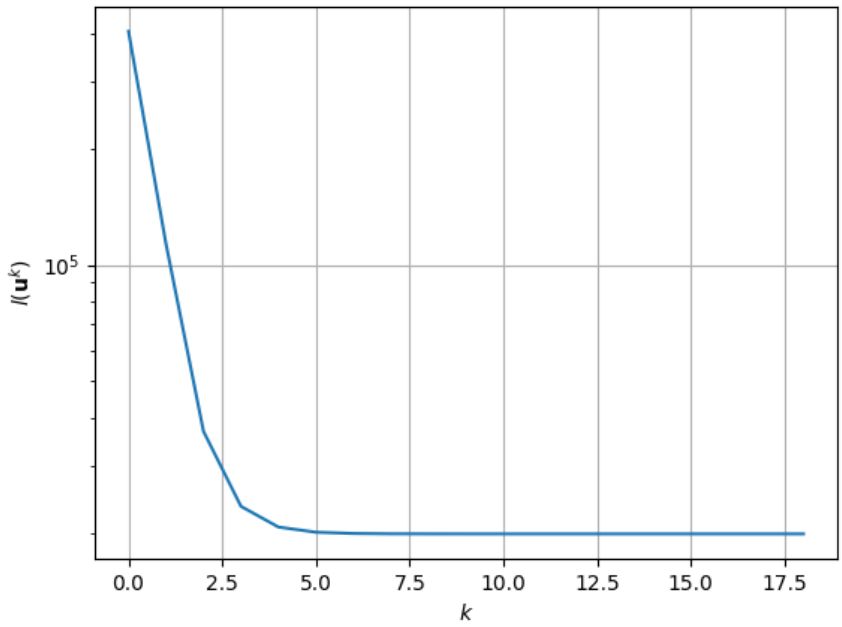

Figure 8: Cost function

Task 3 Trajectory tracking

In this section, we discuss how we approached the resolution of trajectory tracking task, which required to define the **optimal feedback controller** to track the optimal reference trajectory found in the previous task by exploiting the **LQR algorithm**.

Linearization about the optimal trajectory

We firstly linearized the system dynamics about the optimal trajectory (**x***,**u***) obtaining the trajectory of the linearization:

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

$$A_t = \nabla_{x_t} f(x_t^*, u_t^*)^T \quad B_t = \nabla_{u_t} f(x_t^*, u_t^*)^T$$

Computation of the gain K

The next step is finding the **feedback gain** K of the optimal controller that stabilizes the system which corresponds to the gain that stabilizes the system linearization about the optimal trajectory. That gain is obtained by solving the following LQ problem:

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

$$subj\ to\ \Delta x_{t+1} = A_t \Delta x_t + B_t \Delta u_t$$

$$\Delta x_0 = 0$$

In this optimization problem the matrices $Q_t^{reg} \geq 0$, $Q_T^{reg} \geq 0$ and $R_t^{reg} > 0$ are the degrees of freedom used to stabilize the system and are used to compute the gain K, which is obtained by the expression:

$$K_t^{reg} = -(R_t^{reg} + B_t^T P_{t+1} B_t)^{-1} (B_t^T P_{t+1} A_t)$$

Where P_t elements are obtained by iterating backward in time the **Difference Riccati equation**, starting from $P_T = Q_T^{reg}$

The matrices that we used are written in the *cost.py* file and we chose their elements as follows:

$$Q_t^{reg} = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 100 \end{bmatrix}$$

$$Q_T^{reg} = \begin{bmatrix} 500000 & 0 & 0 & 0 \\ 0 & 50000 & 0 & 0 \\ 0 & 0 & 500000 & 0 \\ 0 & 0 & 0 & 50000 \end{bmatrix}$$

$$R_t^{reg} = 1$$

Computation of the input and integration of the dynamics

The last step consists in the computation of the vector of inputs to apply, given the optimal one and the gain of the regulator K. That input is used to control the original non linear system, so the dynamics must be integrated. Given x_0 initial condition, for all $t = 0, \cdots, T-1$ we compute:

$$u_t = u_t^* + K_t^{reg}(x_t - x_t^*)$$

$$x_{t+1} = f(x_t, u_t)$$

Under suitable assumptions, it can be shown that a trajectory (on an infinite horizon) of a nonlinear system is (locally) exponentially stable if and only if the system linearization about the trajectory is exponentially stable.

The following plots show that even if an error is applied to the initial condition, after a while the regulator manages to make the system follow the optimal trajectory previously computed and to make it reach the final desired position.

We decided to compare both tracking results, the one with the trajectory obtained in the first task with the step reference and the one with the trajectory obtained in the second task with the sigmoid reference.

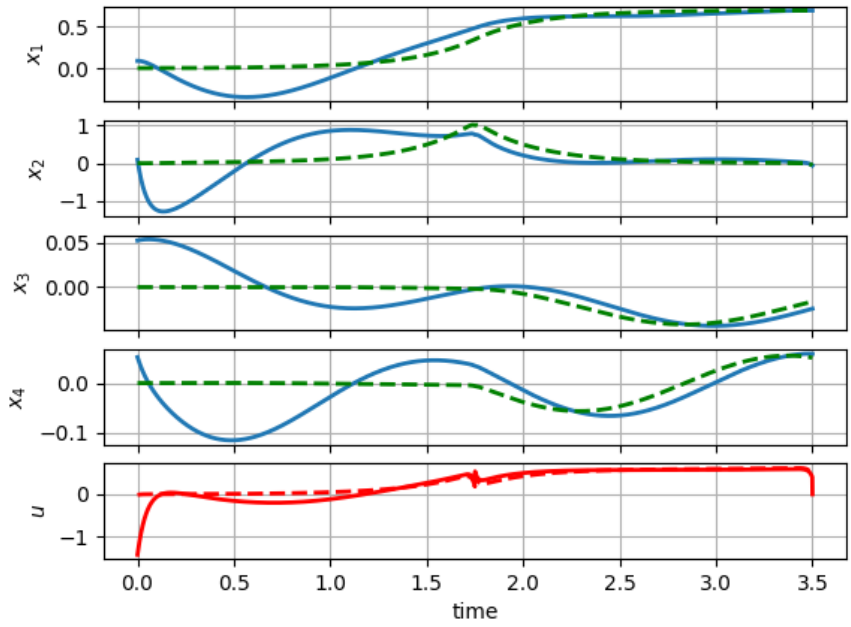

Figure 9: Plot of the step reference trajectory tracking

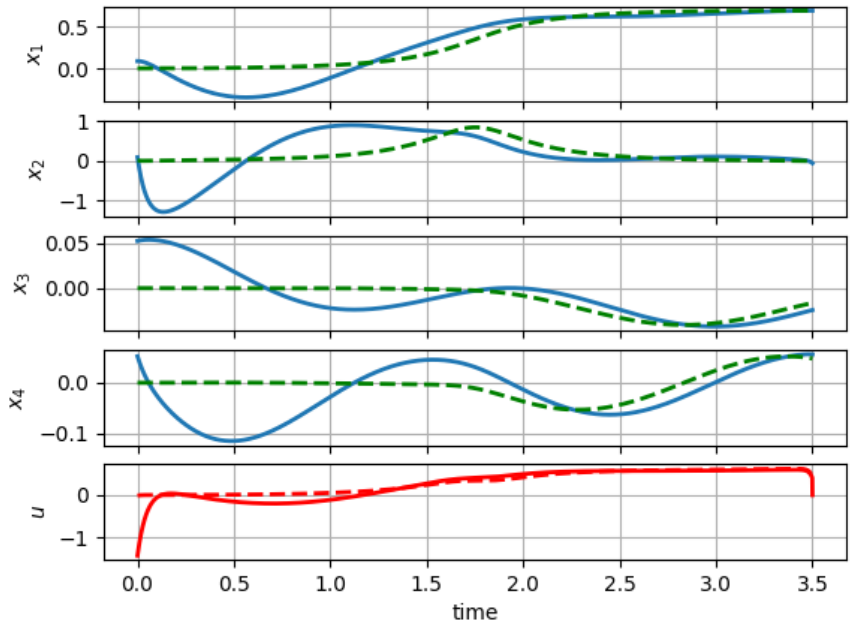

Figure 10: Plot of the sigmoid reference trajectory tracking

Task 4 Animation

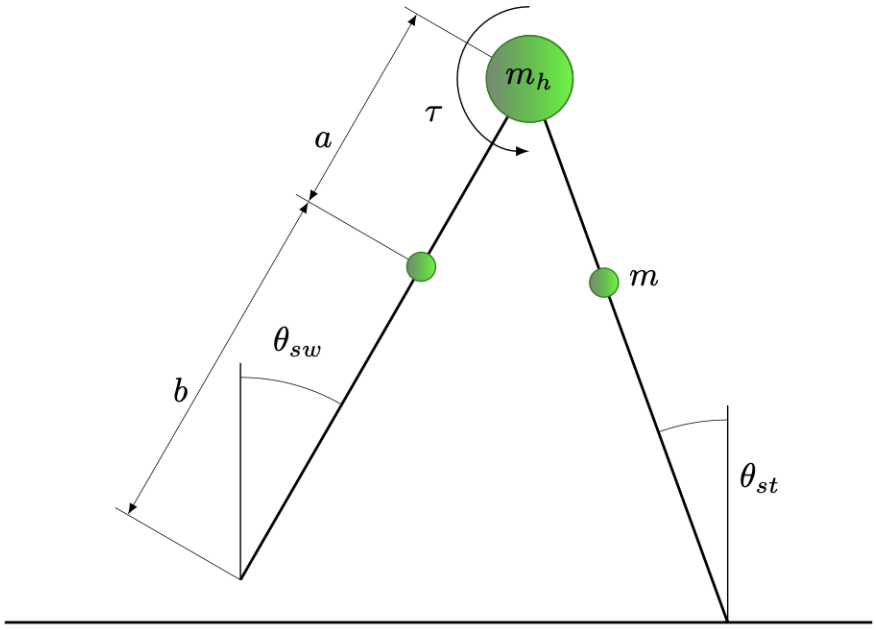

Figure 11: Compass gait model

To implement an animation of the bipedal robot moving and executing the trajectory tracking task, we used the function **FuncAnimation**, found inside the *matplotlib.animation* library.

Starting from the template of the pendulum animation, we designed a second stick to have both legs of the robot, then we added the point masses on each leg. We start with both legs in a vertical position, the animation shows how the swing leg moves forward.

Some frames of the animation can be seen below:

It is shown from the previous plots how the tracking becomes smoother when using the sigmoid reference, achieving a more natural transition while the animation is running.

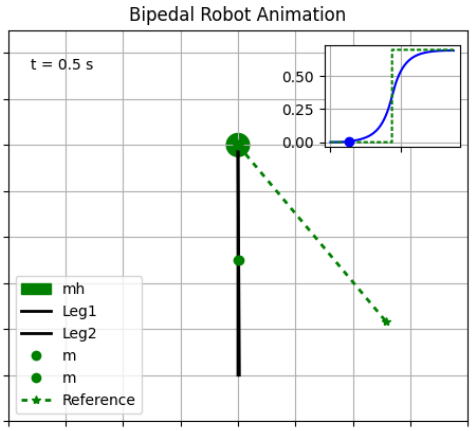

Figure 12: Animation of the robot following the step reference (1)

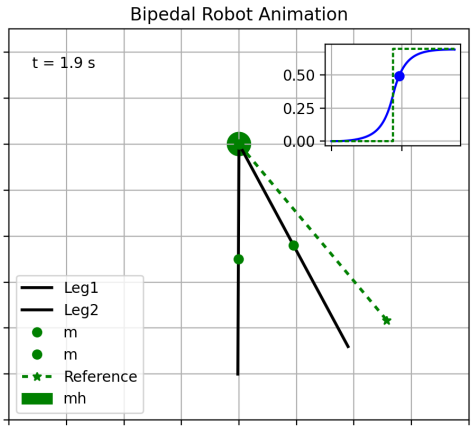

Figure 13: Animation of the robot following the step reference (2)

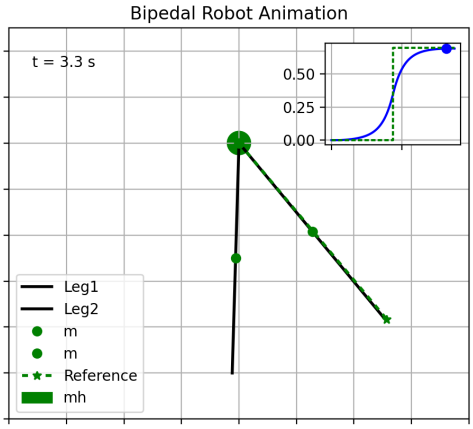

Figure 14: Animation of the robot following the step reference (3)

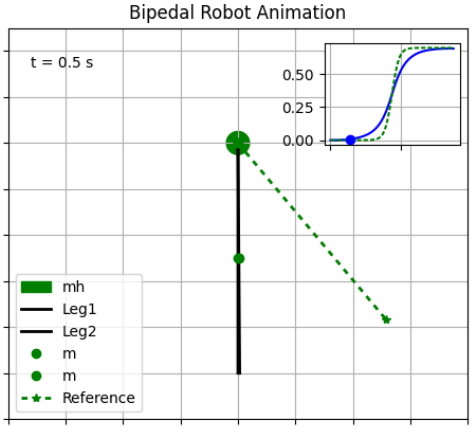

Figure 15: Animation of the robot following the sigmoid reference (1)

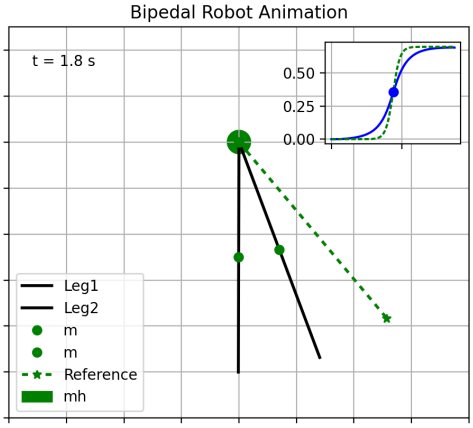

Figure 16: Animation of the robot following the sigmoid reference (2)

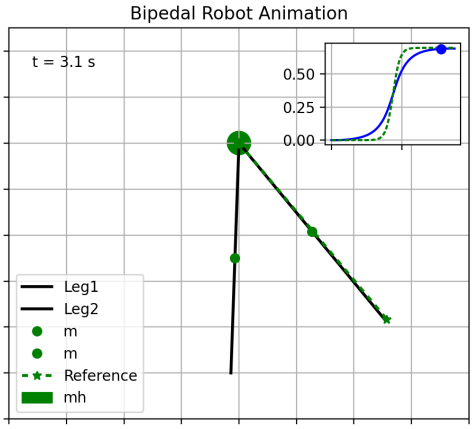

Figure 17: Animation of the robot following the sigmoid reference (3)

Conclusions

In conclusion, in this project we have shown how to generate an optimal trajectory and how to track it, starting from a system dynamics that we had to linearize. The tricky part was understanding how to implement the algorithms learned in class and make them into a functional *Python* code.

Working on the project, while attending classes, was helpful to understand the main topics of the Optimal Control course.

Bibliography

[1] Dimitri P Bertsekas. Nonlinear programming. *Journal of the Operational Research Society*, 48(3):334–334, 1997.