SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER, CONTROL AND MANAGEMENT ENGINEERING

Trajectory optimization of an Acrobot swing up using iterative Linear Quadratic Regulator (iLQR)

UNDERACTUATED ROBOTS

Professor:
Leonardo Lanari
Supervisor:
Tommaso Belvedere

Students:
Anna Fumagalli
Catia Romaniello
Vincenzo Maria Vitale

Academic Year 2022/2023

Contents

Abstract

Among the many optimization trajectory techniques used in robotics, Differential Dynamic Programming (DDP) has become successful thanks to its computational efficiency. The trajectory optimizer employed in this project is the iterative Linear Quadratic Regulator (iLQR), a variant of the classic DDP. The project here reported aims to implement the algorithm to the swing-up problem of an Acrobot system. The results of simulations with the PyBullet robotic simulator are proposed to demonstrate the effectiveness of the implementation.

1 Introduction

This problem addresses the swing-up control problem for the Acrobot, a two-link and two-degree-of-freedom planar robot with the first joint passive and a single actuator on the second joint (Fig. 1)[1]. The swing-up control problem is to move the Acrobot

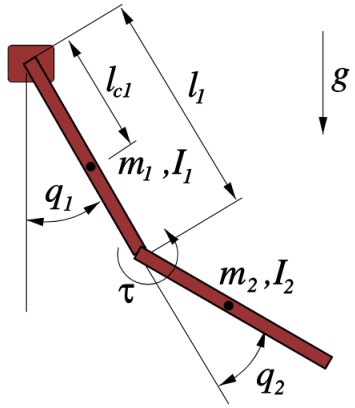

Figure 1

from its stable downward position to its unstable inverted position and balance it vertically[2]. A *.urdf* file specifies the physical characteristics of the robot in this project employed to track its movement and the simulation accordingly.
The aim of the project is to find an optimal trajectory for the swing-up of the robot. It is therefore appropriate to precede the solution approach with an explanation of what is considered an optimized trajectory[3]. To do so, consider discrete-time dynamics to describe the evolution of state $\mathbf{x}$ given control $\mathbf{u}$.

$$\mathbf{x}_{i+1} = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) \tag{1}$$

Let J_0 be defined as the total cost, i.e. the sum of the running costs l and the final cost l_f. These costs are incurred starting from state x_0 and applying the control sequence $\mathbf{U} \equiv \{\mathbf{u}_0, \mathbf{u}_1 ...., \mathbf{u}_{N-1}\}$ up to:

$$J_0(\mathbf{x}, \mathbf{U}) = \sum_{i=0}^{N-1} \ell(\mathbf{x}_i, \mathbf{u}_i) + \ell_f(\mathbf{x}_N).$$

By trajectory optimization, it is here meant to find a $\mathbf{U}^*$ for a particular $\mathbf{x}$ minimizing the control sequence

$$\mathbf{U}^*(x) \equiv \underset{\mathbf{U}}{\operatorname{argmin}} J_0(\mathbf{x}, \mathbf{U})$$

In Fig. 2, a representation of the intermediate cost and the cost-to-go, there to determine the shortest path.

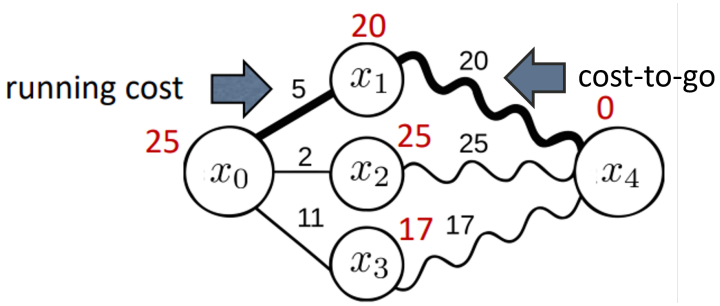

Figure 2

The trajectory optimizer algorithm employed for this purpose is the iterative Linear Quadratic Regulator (ILQR), namely a version of the classic Differential Dynamic Programming (DDP)[1] used for discrete and linearly approximated systems. The main difference between DDP and iLQR is that the latter uses only first rather than second derivatives of the dynamics [2]. The algorithm, in concept, linearises the system around a proposed control and state trajectory and evaluates the performance of a system using a quadratic cost function.

[1]The DDP is the control analogue of the Gauss-Newton method for nonlinear least-squares optimization

[2]Note that the definition is extracted from the paper [3], although the latter refers to iLQG and not iLQR. However, it seemed as fitting as appropriate since iLQG is a stochastic extension of iLQR.

2 Iterative Linear Quadratic Regulator

In this section, the iLQR is dissected theoretically and in its application to this project.

The algorithm, once defined the cost functions (see 3.3), iteratively repeats the following:

1. Linearise the system over an *initial guess* of states and inputs, namely a nominal $(\mathbf{x}, \mathbf{u}, i)$;
2. Runs the *backward pass* to minimize the cost, iterating for decreasing $i = N - 1, ...0$;
3. Applies the new inputs to the nominal model in the *forward pass*;
4. Set the new states and the new inputs obtained running the *forward pass*, as a new initial guess.

To elaborate on what has been summarised so far, consider what was mentioned in the introduction, lets define the cost-to-go J_i as the partial sum of costs from i to N.

$$J_i(\mathbf{x}, \mathbf{U}_i) = \sum_{j=i}^{N-1} \ell(\mathbf{x}_j, \mathbf{u}_j) + \ell_f(\mathbf{x}_N)$$

The cost-to-go obtained by the minimization control sequence is the *Value* and is defined as:

$$V(\mathbf{x}, i) \equiv \min_{\mathbf{U}_i} J_i(\mathbf{x}, \mathbf{U}_i)$$

which, considering to set $V(\mathbf{x}, N) \equiv \ell_f(\mathbf{x}_N)$ and proceeding backwards, assume the form:

$$V(\mathbf{x}, i) = \min_{\mathbf{u}} [\ell(\mathbf{x}, \mathbf{u}) + V(\mathbf{f}(\mathbf{x}, \mathbf{u}), i + 1)]. \tag{2}$$

The equation in 2 can be defined as a function of perturbations around the $i - th$ ($\mathbf{x}$, $\mathbf{u}$) pair, resulting:

$$Q(\delta\mathbf{x}, \delta\mathbf{u}) = \ell(\mathbf{x}+\delta\mathbf{u}, \mathbf{u}+\delta\mathbf{u}, i) - \ell(\mathbf{x}, \mathbf{u}, i) + V(\mathbf{f}(\mathbf{x}+\delta\mathbf{x}, \mathbf{u}+\delta\mathbf{u}), i+1) - V(\mathbf{f}(\mathbf{x}, \mathbf{u}), i+1), \tag{3}$$

of which second order expansion follows:

$$\approx \frac{1}{2} = \begin{bmatrix} 1 \\ \delta\mathbf{x} \\ \delta\mathbf{u} \end{bmatrix}^T \begin{bmatrix} 0 & Q_x^T & Q_u^T \\ Q_x & Q_x x & Q_x u \\ Q_u & Q_u x & Q_u u \end{bmatrix} \begin{bmatrix} 1 \\ \delta\mathbf{x} \\ \delta\mathbf{u} \end{bmatrix}. \tag{4}$$

The coefficients Q_x, Q_u, Q_{xx}, Q_{uu}, and Q_{ux} of 4 are listed down below[3] in 5.

[3]Note that $V' \equiv V(i + 1)$

$$Q_x = \ell_{\mathbf{x}} + \mathbf{f}_{\mathbf{x}}^T V'_{\mathbf{x}} \tag{5a}$$
$$Q_u = \ell_{\mathbf{u}} + \mathbf{f}_{\mathbf{u}}^T V'_{\mathbf{x}} \tag{5b}$$
$$Q_{xx} = \ell_{\mathbf{xx}} + \mathbf{f}_{\mathbf{x}}^T V'_{\mathbf{xx}} \mathbf{f}_{\mathbf{x}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{xx}} \tag{5c}$$
$$Q_{uu} = \ell_{\mathbf{uu}} + \mathbf{f}_{\mathbf{x}}^T V'_{\mathbf{xx}} \mathbf{f}_{\mathbf{u}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{uu}} \tag{5d}$$
$$Q_{ux} = \ell_{\mathbf{ux}} + \mathbf{f}_{\mathbf{u}}^T V'_{\mathbf{xx}} \mathbf{f}_{\mathbf{x}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{ux}}. \tag{5e}$$

As mentioned in the introduction, the difference between DDP and iLQR lies exactly in whether or not the second derivates of the dynamics are taken into account. For this reason, the last terms of the equations 5c, 5d, 5e can be considered negligible.

Minimizing the equation 3 with respect to the input perturbation $\delta\mathbf{u}$, it is obtained:

$$\delta\mathbf{u}^* = \underset{\delta\mathbf{u}}{\operatorname{argmin}} Q(\delta\mathbf{x}, \delta\mathbf{u}) = -Q_{\mathbf{uu}}^{-1}(Q_{\mathbf{u}} + Q_{\mathbf{ux}}\delta\mathbf{x}), \tag{6}$$

which can also be rewritten as

$$\delta\mathbf{u}^* = \mathbf{k} + \mathbf{K}\delta\mathbf{x}, \tag{7}$$

where thus

- $\mathbf{k} \triangleq -Q_{\mathbf{uu}}^{-1} Q_{\mathbf{u}}$ and represents the *open-loop gain* term which is a corrective one;
- $\mathbf{K} \triangleq -Q_{\mathbf{uu}}^{-1} Q_{\mathbf{ux}}$ and represents the *feedback gain* term.

It is now possible to substitute the optimal policy $\delta\mathbf{u}^*$ in 6 via 7, in 4 obtaining the estimate of the *Value* quadratic model [4]:

$$\Delta V = -\frac{1}{2}\mathbf{k}^{\mathbf{T}} Q_{\mathbf{uu}} \mathbf{k} \tag{8a}$$
$$V_{\mathbf{x}} = Q_{\mathbf{x}} - \mathbf{K}^{\mathbf{T}} Q_{\mathbf{uu}} \mathbf{k} \tag{8b}$$
$$V_{\mathbf{xx}} = Q_{\mathbf{xx}} - \mathbf{K}^{\mathbf{T}} Q_{\mathbf{uu}} \mathbf{K} \tag{8c}$$

Computing recursively the i_{th} *Value* function V, initialising it to the terminal cost and its derivatives $V_N = \ell(\mathbf{x}_N)$, constitutes the *backward pass*. As mentioned at the beginning of this section, the following step in the iLQR is the *forward pass* which computes a new trajectory achieved through:

$$\hat{\mathbf{x}}(0) = \mathbf{x}(0) \tag{9a}$$
$$\hat{\mathbf{u}}(i) = \mathbf{u}(i) + \mathbf{k}(i) + \mathbf{K}(i)(\hat{\mathbf{x}}(i) - \mathbf{x}(i)) \tag{9b}$$
$$\hat{\mathbf{x}}(i+1) = \mathbf{f}(\hat{\mathbf{x}}(i), \hat{\mathbf{u}}(i)). \tag{9c}$$

Departing from the basic algorithm, one can refer to improvements made in the proposed work of Tassa et al. 2012[3], in which it was introduced a regularization term

μ to harden the algorithm and a scaling factor $\alpha \in (0, 1]$ to determine the decreasing step of the cost.

The regularization factor causes the addition of a diagonal term to the local control-cost Hessian:

$$\tilde{Q}_{uu} = Q_{uu} + \mu \mathbf{I}_m.$$

This variation consists in adding a quadratic cost around the current control-sequence, making the steps more conservative, as the Levenberg-Marquardt parameter does in the Gauss–Newton. In the reference this modification was considered as to penalizes deviation from the states as follows:

$$\tilde{Q}_{\mathbf{uu}} = \ell_{\mathbf{uu}} + \mathbf{f}_{\mathbf{u}}^{\mathbf{T}}(V'_{xx} + \mu \mathbf{I}_n)\mathbf{f}_{\mathbf{u}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{uu}} \tag{10a}$$

$$\tilde{Q}_{ux} = \ell_{\mathbf{ux}} + \mathbf{f}_{\mathbf{u}}^{\mathbf{T}}(V'_{xx} + \mu \mathbf{I}_n)\mathbf{f}_{\mathbf{x}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{ux}} \tag{10b}$$

$$\mathbf{k} = -\tilde{Q}_{\mathbf{uu}}^{-1}\tilde{Q}_{\mathbf{u}} \tag{10c}$$

$$\mathbf{K} = -\tilde{Q}_{\mathbf{uu}}^{-1}\tilde{Q}_{\mathbf{ux}} \tag{10d}$$

Thanks to this adding, robustness should improve, since the feedback gains do not vanish as $\mu \to \infty$, but rather force the new trajectory closer to the old one. Moreover, considering what has been obtained in equations 10 and the several cancellations of $Q_{\mathbf{uu}}$ the *Value* evolution 11 updated results as follows:

$$\Delta V = +\frac{1}{2}\mathbf{k}^{\mathbf{T}}\tilde{Q}_{\mathbf{uu}}\mathbf{k} + \mathbf{k}^{\mathbf{T}}Q_{\mathbf{u}} \tag{11a}$$

$$V_{\mathbf{x}} = Q_{\mathbf{x}} + \mathbf{K}^{\mathbf{T}}\tilde{Q}_{\mathbf{uu}}\mathbf{k} + \mathbf{K}^{\mathbf{T}}Q_{\mathbf{u}} + \tilde{Q}_{\mathbf{ux}}^{\mathbf{T}}\mathbf{k} \tag{11b}$$

$$V_{\mathbf{xx}} = Q_{\mathbf{xx}} + \mathbf{K}^{\mathbf{T}}\tilde{Q}_{\mathbf{uu}}\mathbf{K} + \mathbf{K}^{\mathbf{T}}\tilde{Q}_{\mathbf{ux}} + \tilde{Q}_{\mathbf{ux}}^{\mathbf{T}}\mathbf{K}. \tag{11c}$$

The application of the regularization term in the reference paper[3], makes use of a quadratic modification schedule, defining a minimal value μ_{min} and a minimal modification factor Δ_0, as to adjust μ as follows:
increase μ:

$$\Delta \leftarrow max(\Delta_0, \Delta \cdot \Delta_0)$$

$$\mu \leftarrow min(\frac{1}{\Delta_0}, \frac{\Delta}{\Delta_0})$$

decrease μ:

$$\mu \leftarrow \begin{cases} \mu \cdot \Delta & \text{if } \mu \cdot \Delta > \mu_{min}, \\ 0 & \text{if } \mu \cdot \Delta < \mu_{min}. \end{cases} \tag{12}$$

On the other hand, the aforementioned scaling factor α constitutes an other attempt to improve the optimizer. This value has its effect in the *forward pass* phase. The latter, based on 9, is a key phase for the fast convergence. There, is introduced $0 < \alpha \geq 1$, as a backtracking line-search parameter, employed in the following way:

$$\hat{\mathbf{u}}(i) = \mathbf{u}(i) + \alpha\mathbf{k}(i) + \mathbf{K}(i)(\hat{\mathbf{x}}(i) + \mathbf{x}(i)). \tag{13}$$

The solution proposed allows to determine whether to accept the variations obtained at a specific iteration in base on the expected reduction of cost, modifying the α factor. Considering what has been achieved so far, it can be derived a better estimate of the expected cost reduction just mentioned, via 11a, obtaining:

$$\Delta J(\alpha) = \alpha \sum_{i=1}^{N-1} \mathbf{k}(i)^{\mathbf{T}} Q_{\mathbf{u}}(i) + \frac{\alpha^2}{2} \sum_{i=1}^{N-1} \mathbf{k}(i)^{\mathbf{T}} \tilde{Q}_{uu}(i) \mathbf{k}(i). \tag{14}$$

When comparing the actual and expected reductions,

$$z = [J(\mathbf{u}_{1\ldots N-1} - J(\hat{\mathbf{u}}_{1\ldots N-1)}]/\Delta J(\alpha)$$

it can be chosen- as just mentioned- whether to accept or not the iteration. The latter would be accepted only if:

$$0 < c1 < z.$$

To summarize, the α factor is initialized at 1 at the beginning of the *forward pass*. The steps in 6 and 9c are iterated computing a new nominal initial guess and, if the integration diverged or does not meet the condition defined in 7, α is decreased and the *forward pass* restarts

3 Implementation

The specific problem of the Acrobot swing-up means to consider a change of its state $x = [q_1 \quad q_2 \quad \dot{q_1} \quad \dot{q_2}]$, from the initial configuration $x_0 = [-\pi \quad 0 \quad 0 \quad 0]$ to the final configuration $x_N = [0 \;\; 0 \;\; 0 \;\; 0]$.

3.1 Libraries

In view of the aforementioned purpose, *Pinocchio* and *CasADi* libraries have been implemented. The first one has been used for an efficient computation of the dynamics and derivatives of a robot model [5]: in the rollout and in the forward phase to determine the Acrobot dynamics, while in the backward its derivatives.
On the other hand, with regard to the *CasADi* library, it is an open-source tool for nonlinear optimization and algorithmic differentiation [6]. In the specific case, its functionality helped in the backward phase to define the cost functions.

3.2 Acrobot dynamics

As mentioned in the introduction to this report, the characteristics of the reference robot are defined in a file of type *.urdf*, from which a function of the Pinocchio library can deduce the dynamics. This specific function is `aba`, which returns the direct dynamics $\ddot{q}$, necessary but not already structured as to define the dynamics in the state-input form. Indeed, what can directly be obtained from `aba` is $\ddot{q} = (\ddot{q}_1, \ddot{q}_2) = \mathbf{M}^{-1}(\mathbf{u} - c(\mathbf{x}) - g(\mathbf{x}_{1,2}))$, where:

- $\mathbf{u} = (0, \tau)$;
- $\mathbf{M}(q) = \begin{bmatrix} a_1 + 2a_2c_2 & a_3 + a_2c_2 \\ a_3 + a_2c_2 & a_3 \end{bmatrix}$;
- $\mathbf{c}(q, \dot{q}) = \begin{bmatrix} -a_2s_2\dot{q}_2(2\dot{q}_1 + \dot{q}_2) \\ a_2s_2\dot{q}_1^2 \end{bmatrix}$;
- $\mathbf{g}(q) = \begin{bmatrix} a_4s_1 + a_5s_{12} \\ a_5s_{12} \end{bmatrix}$;

where, in turn:

- $a_1 = \mathbf{I}_1 + m_1\ell c_1^2 + \mathbf{I}_2 + m_2(\ell_1^2 + \ell c_2^2)$
- $a_2 = m_2\ell_1\ell c_2$
- $a_3 = \mathbf{I}_2 + m_2\ell c_2^2$

- $a_4 = \mathbf{g}(m_1 \ell c_1 + m_2 \ell_1)$
- $a_5 = g m_2 \ell c_2$

while the input-state form requires:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\mathbf{x}}_1 \\ \dot{\mathbf{x}}_2 \\ \dot{\mathbf{x}}_3 \\ \dot{\mathbf{x}}_4 \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{q}}_1 \\ \dot{\mathbf{q}}_2 \\ \ddot{\mathbf{q}}_1 \\ \ddot{\mathbf{q}}_2 \end{bmatrix}^{\mathbf{T}} = \begin{bmatrix} \mathbf{x}_3 \\ \mathbf{x}_4 \\ \mathbf{M}^{-1}(\mathbf{u} - c(\mathbf{x}) - g(\mathbf{x}_{1,2})) \end{bmatrix} \tag{15}$$

Thus, `aba` misses to structure it with $\dot{\mathbf{x}}_1$ and $\dot{\mathbf{x}}_2$, thus it has been then reconstructed accordingly.

The derivatives of the dynamics - which are required in the *backward pass* - can also be calculated using a function of the Pinocchio library, specifically `computeABADerivate`. Again, these were then reconstructed to have a form consistent with the input-state form mentioned above.

The conversion of the dynamics into the input-state structure results from iLQR's election of this mode. Furthermore, the algorithm requires it to be linearized and discretized in the form in 1.

In the case of the project under analysis, the desired discretization was obtained by means of Euler integration[4], resulting in:

$$\begin{pmatrix} q_1(k+1) \\ q_2(k+1) \\ \dot{q}_1(k+1) \\ \dot{q}_2(k+1) \end{pmatrix} = \begin{pmatrix} q_1(k) + \dot{q}_1(k)\delta_t \\ q_2(k) + \dot{q}_2(k)\delta_t \\ \dot{q}_1(k) + \ddot{q}_1(k)\delta_t \\ \dot{q}_2(k) + \ddot{q}_2(k)\delta_t \end{pmatrix}, \tag{16}$$

where $\delta_t = 0.005s$ has been used for simulations.

The intermediate steps leading to the linearised and discretized process follow.

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}), \tag{17a}$$

$$\mathbf{x}_{k+1} = f_d(\mathbf{x}_k, \mathbf{u}_k), \tag{17b}$$

$$\tag{17c}$$

where, deriving the latter w.r.t. x_k,

$$\mathbf{x}_{k+1} \approx \mathbf{x}_k + \delta f(\mathbf{x}_k, \mathbf{u}_k), \tag{18a}$$

$$\mathbf{x}_{k+1} = I + \delta \cdot \frac{\partial f}{\partial \mathbf{x}_k}, \tag{18b}$$

[4]The Euler method is a basic explicit first-order numerical method for solving ordinary differential equations. This method was chosen, but more accurate methods could have been used, such as that of Runge-Kutta

defining $\mathbf{A} = \frac{\partial f}{\partial \mathbf{x}_k}$,
while, deriving w.r.t. $\mathbf{u}$ it results in $\delta\frac{\partial f}{\partial \mathbf{u}} = \delta\mathbf{B}$. It follows that

$$f_{\mathbf{x}} = I + \delta\mathbf{A}, \tag{19a}$$

$$f_{\mathbf{u}} = \delta\mathbf{B}. \tag{19b}$$

3.3 Cost functions

As in most cases, the cost function employed in the optimiser is quadratic, following the formula below:

$$\ell(\mathbf{x}_i, \mathbf{u}_i) = e_i^{\mathbf{T}} \cdot Q \cdot e_i + \mathbf{u}_i^{\mathbf{T}} \cdot R \cdot \mathbf{u}_i$$

$$\ell_f(x_N) = e_N^{\mathbf{T}} \cdot Q_N \cdot e_N$$

where,

- $\ell(\mathbf{x}_i, \mathbf{u}_i)$ is the cost-to go at that iteration;
- Q is a 4×4[5] diagonal matrix of state weights;
- R is a scalar value [6] which regulates the weight of the input;
- the final state weights matrix Q_N, which is conceptually the same as Q, but with an increased weight to force the optimiser to direct the trajectory to the desired final position;
- e which is the difference between the current state and the target;
- e_N which is the difference between the final current state and the target.

It should be noted that the optimizer performance largely depends on the cost function and on the chosen horizon length (i.e. how *far we look into the future*). In fact, even the very choice of weights - especially belonging to the Q-matrix - greatly conditioned the number of iterations or could even result in failure to achieve the desired target convergence. The employment of a Q_N (see 2) matrix with low values did not guarantee the optimisation of the trajectory: indeed, the correction of the latter depends strictly on the derivatives of the costs by means of the gain term k. Conversely, if the Q_N matrix were made up of high values, the number of iterations to reach the desired final position would consequently increase: the values mentioned would in fact be representative of high relative costs, reflecting a high error, namely, a large distance from the target. When the latter was the case, the computational algorithmic effort inevitably increased in the often unsuccessful attempt at convergence at the destination.

[5] Q has a $n \times n$ dimension, with n = number of states, which in our case is four.

[6] R has a $m \times m$ dimension, with m = number of inputs, which in our case is just one, since we only have an active joint.

With regard to the scalar value R, it was empirically determined that assigning a high value - and therefore a heavier weight to the input - was advantageous to the optimisation of the trajectory with the iLQR.

The tuning of the parameters Q, Q_N, and R was finalised following several tests, resulting in:

- $Q_N = 1000 \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$;
- $Q = \begin{bmatrix} 0.1 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix}$;
- R=100;

3.4 Angles Wrapping

Angle *wrap* or *normalisation* is required to ensure that angles remain within a specific range and can be represented consistently. Operations that may require a wrapping function include, among others, algebraic sum operations. These are, for example, in the definition of above cost functions or, more generally, in the calculation of the error which is, indeed, the result of a difference. There are several techniques for wrapping angles. After experimenting with more than one of them, the method chosen is to compute the *sin* and *cosin* of the candidate angle and then their unique arcotangent returned by the function `math.atan2`[7]. The function follows:

$$\begin{aligned} c &= cos(\beta), \\ s &= sin(\beta), \\ \beta &= \arctan\left(\frac{s}{c}\right), \end{aligned} \tag{20}$$

where the β employed to compute sin and *cosin* is the angle that must be wrapped.

3.5 Initial Guess Rollout

The rollout of a trajectory for the oscillation of an Acrobot refers to the execution of system dynamics using a specific input trajectory. This input trajectory is usually obtained from an initial input hypothesis. During the rollout, the system is evolved

[7]It is unique since the python function returns a limited value in the range between π and $-\pi$

forward in time using the given input, and the resulting state trajectory is observed. In the iLQR, the *initial guess*is crucial since the optimisation process starts from the first trajectory.

Two initial input assumptions were mainly evaluated: a null and a sinusoidal *initial guess*.

Choosing an initial null hypothesis means starting from a point of equilibrium, in the present case with $\mathbf{x}_0 = [-\pi \;\; 0 \;\; 0 \;\; 0]$, without generating motion. This leads to the need for a greater number of iterations, as the optimiser has to *strive* more to define the final trajectory. In the case presented, `821` iterations were required to obtain the best result.

The idea of an *initial guess* sinusoidal comes from observing the movement of the swing up, which begins by moving like a pendulum, reminiscent of a sinusoidal motion. Applying a sinusoidal input enabled convergence to be achieved with approximately two hundred iterations less.

The use of a null initial input was finally chosen, but tests were carried out with both sinusoidal and zero input.

3.6 Backward and Forward Pass

The formulae defined in 2 were used for the backward and forward pass, except for the regularisation factor μ as it did not seem to lead to improved results. In fact, the regularisation factor never updated according to 12, instead always remaining equal to zero and therefore having no effect.

Plots depicting the optimised trajectories obtained at iLQR termination are shown below. Fig. 3 shows a comparison of the results obtained from an initial null input and an initial sinusoidal input.

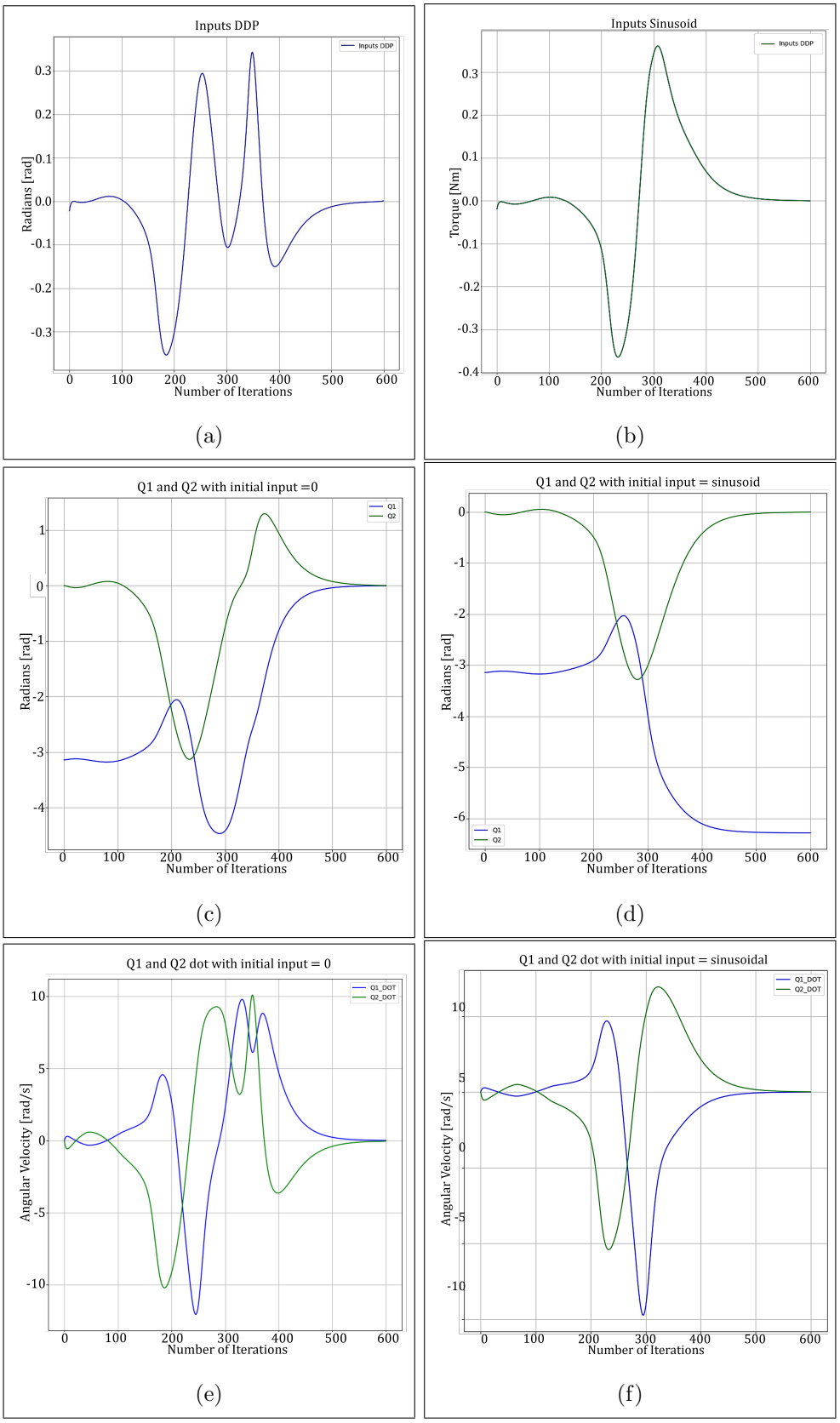

Figure 3: In the present figure, the plot of the inputs and the ILQR obtained by the optimized trajectory generated by iLQR is reported. Specifically, (a) and (b) show the inputs beginning from a null initial guess and a sinusoidal one. On the other hand, (c) and (e) show the trajectory converging to the desired end position and the relative velocities converging to zero, with a null initial input, while (d) and (f) show the trajectory converging to the desired end position and the relative velocities converging to zero, with a sinusoidal initial input.

4 Simulation

Once obtained the points constituting the optimised trajectory and the inputs from which they derive, they are passed to the Pybullet simulator.

PyBullet is a fast and easy to use Python module for robotics simulation, with a focus on sim-to-real transfer. With this simulator, you can load articulated bodies from *urdf* files, as in this case or also with *sdf*, *mjcf* and other file formats and provides: forward dynamics simulation, inverse dynamics computation, forward and inverse kinematics, collision detection and ray intersection queries. [7]

The optimised points and inputs passed to the simulator, mentioned earlier, are computed by the iLQR. However, it should be specified that what is obtained with the algorithm does not (necessarily and in our case) correspond to what happens in the simulation phase. Among the hypotheses of the reasons for this divergence, one undoubtedly finds the presence or absence of friction within the simulation scene or, again, the type of integration employed for the discretization. In support of this, comparisons should be made using videos and plots.

It is necessary at this stage to specify that in the plots, a comparison will also be made with data obtained dependent on a *K feedback* or not. By this, it is meant whether states and inputs were passed by the iLQR exactly as calculated by the optimiser or whether a controller was added. This second case deserves a more specific explanation. The iLQR is an optimizer algorithm that, as already mentioned, can find the optimal trajectory bringing the robot to the target state and controlling it through the optimal computed inputs. In the case of the swing-up motion, it ends in an unstable equilibrium point, indeed even reaching the target state, for small perturbations, the Acrobot can fall. To stabilize it around the upright position, an LQR controller was used during the simulation. In this case, the last K computed[8] corresponds to the control of the simulation robot by a single iteration of iLQR (LQR), making the case of starting from the penultimate trajectory obtained from the iLQR, ensuring stabilization around equilibrium points. The inputs of the simulated Acrobot are controlled by adding to the iLQR input the feedback term determined by the difference between the current simulation state and the relative state returned by the ILQR, multiplied by the LQR gain (referred to as *K feedback*), as follows:

$$u_{simulation,i} = u(i) + K_{feedback}(i)(\hat{x}(i) - x(i)) \tag{21}$$

The plots of the results obtained from the iLQR compared to the simulations' now follow. In the Figure 4 it can be seen, without too many surprises, that the inputs obtained from the iLQR and the simulation coincide perfectly if the gain factor K is not taken into account, which is not the case when the simulation is carried out

[8]With this we refer to the K in the section 2

considering the correction factor K.

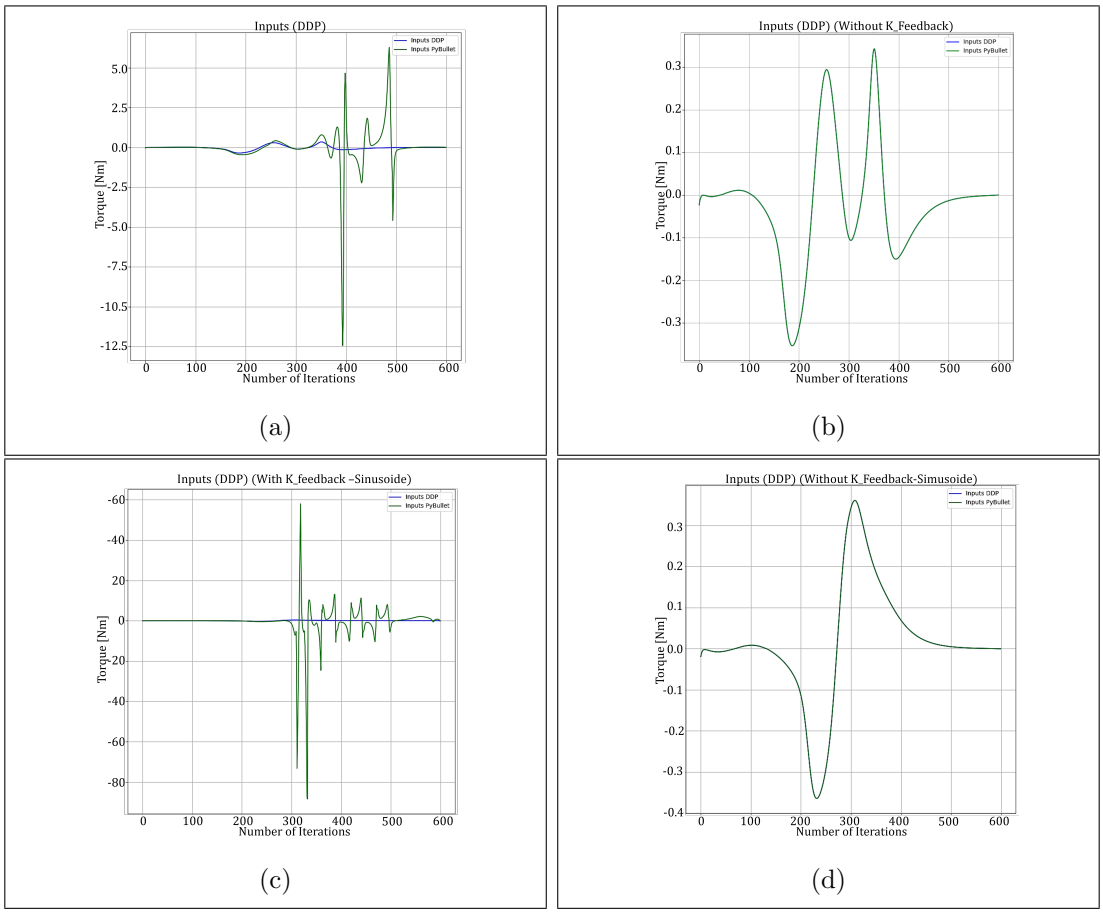

Figure 4: In the present figure, the inputs used for simulation are shown. Specifically, (a) and (b) represent the inputs with a null initial guess, respectively with the K *feedback* term and without. While (c) and (d) represent the inputs with a sinusoidal initial guess, respectively with the K *feedback* term and without.

In the Figure 5 the trajectory of the two constituent joints in simulation phase are compared, taking into consideration the mentioned two different initial guesses and the presence or absence of the gain term K.

The Figures 6, and 7 are probably more explanatory and relevant than the previous ones. In them, one can appreciate the divergence between what one gets as an optimised iLQR trajectory and the simulation trajectory. This is shown in the two figures separately for the two joints.

Going into more detail, in the figures we can notice that the simulation trajectory of the joints follow best the one generated by iLQR when considering the K *feedback* term, both starting with a null input on a sinusoidal one. It can also be seen that the detachment between the simulated trajectories and the algorithmically computed ones occur earlier - in terms of numbers of iterations - when employing an initial sinusoidal input, and thus occurs later when considering the initial null input. This could also be related to the aforementioned fact that with a sinusoidal *initial guess*, the computed

convergence is achieved with fewer iterations than with a null *initial guess*.

To summarise the initial input combinations used (null or sinusoidal) along with the use or non-use of the feedback term K in the simulation phase, representative frames of the relative motions obtained with Pybullet are shown below in Fig. 8, 9, 10, 11. The images are shown to depict pratically what described through the plots. The set of figures 8 shows the best result of this project. Indeed, as is visible in the last picture, the Acrobot reaches the upright position, even if it does some uncontrollable movements in the middle of the simulation. Despite the addition of the LQR controller, the robot is not able to maintain the target state. As a consequence, it would be better to apply another type of controller, such as the MPC, as better explained in the conclusions 5.

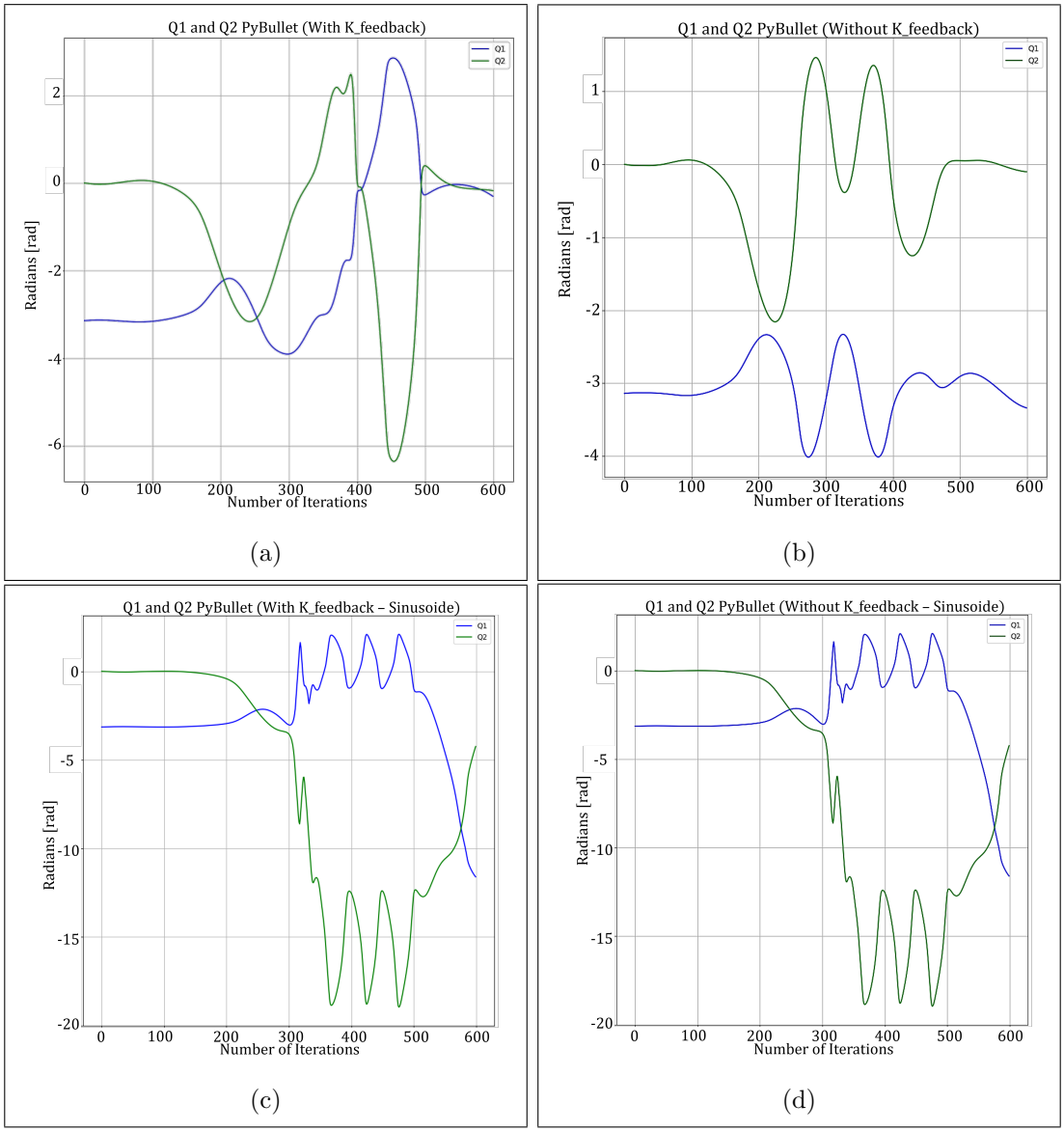

Figure 5: In the present figure, the trajectories of the two joints in the simulation are compared (Q_1, Q_2). Specifically in (a) and (b), the plots show the simulation trajectories of Q_1, Q_2 with a null initial guess, respectively with the K *feedback* term and without, while in (c) and (d), the plots show the simulation trajectories of Q_1, Q_2 with a sinusoidal initial guess, respectively with the K *feedback* term and without.

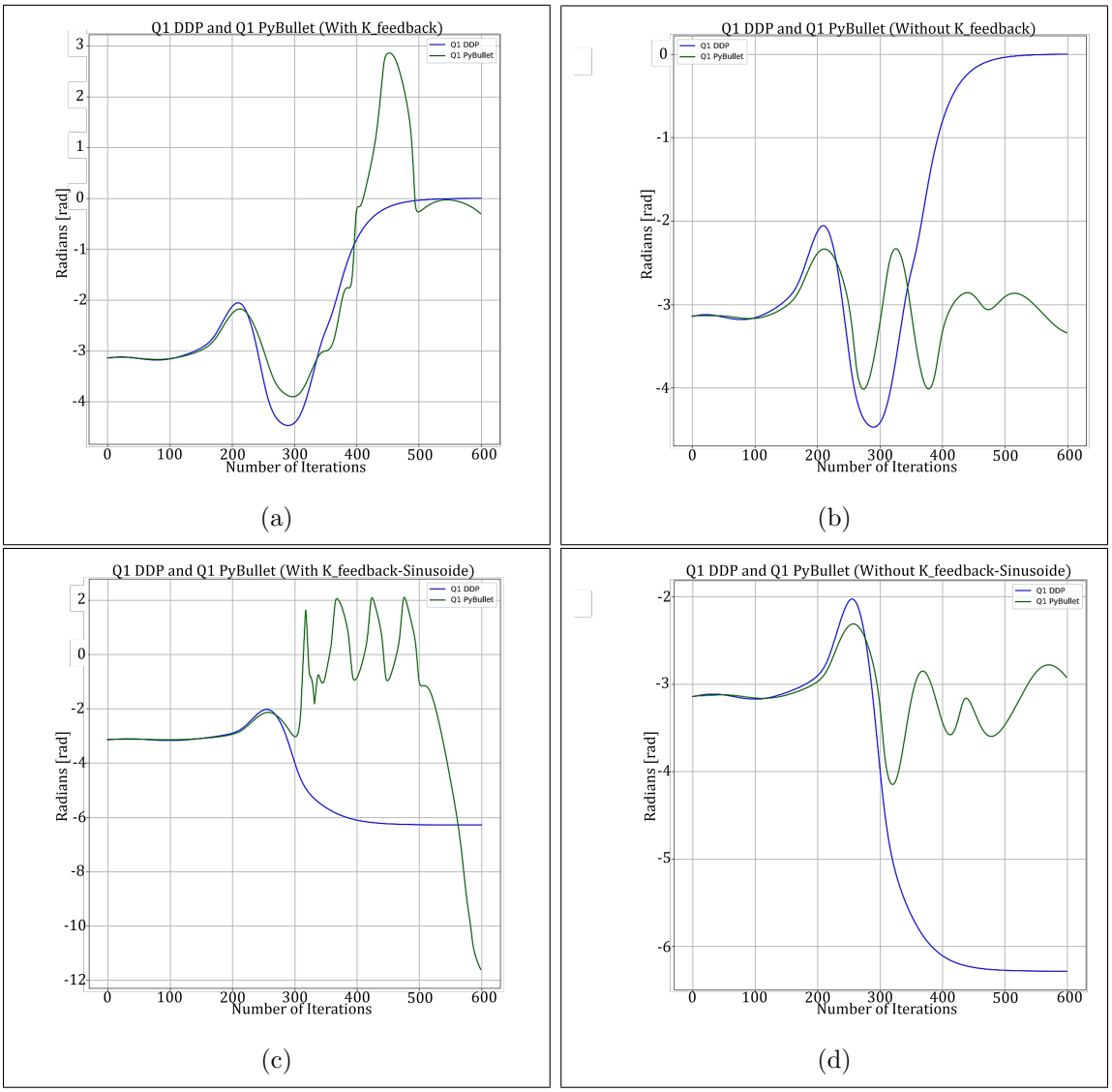

Figure 6: In the present figure, the trajectory of the first joint in the simulation is shown. Specifically in (a) and (b), the plots show respectively the simulation trajectory of Q_1 with a null initial guess, respectively with the *K feedback* term and without, while in (c) and (d), the plots show the simulation trajectory of Q_1 with a sinusoidal initial guess, respectively with the *K feedback* term and without.

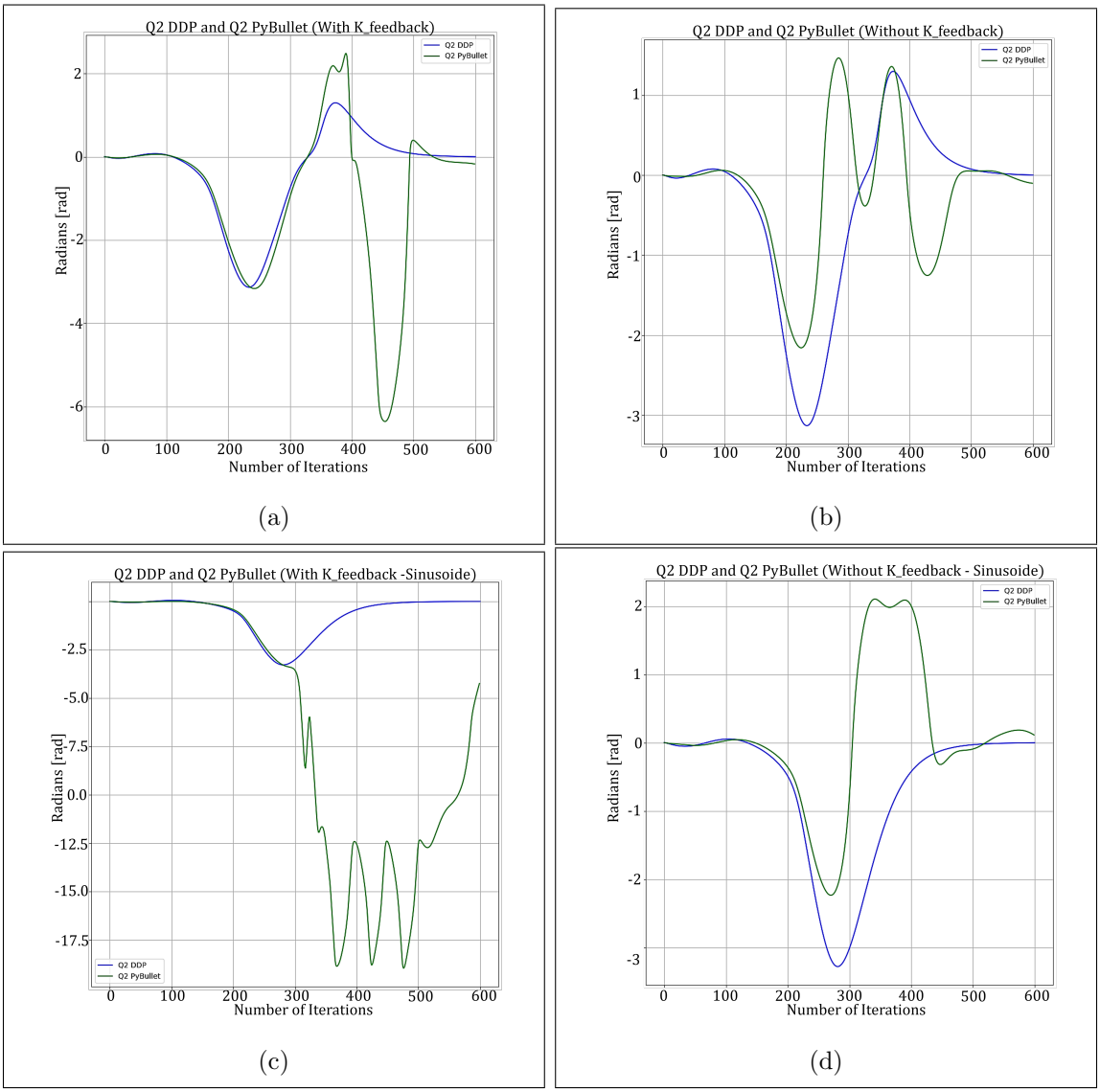

Figure 7: In the present figure, the trajectory of the second joint in the simulation is shown. Specifically in (a) and (b), the plots show respectively the simulation trajectory of Q_2 with a null initial guess, respectively with the *K feedback* term and without, while in (c) and (d), the plots show the simulation trajectory of Q_2 with a sinusoidal initial guess, respectively with the *K feedback* term and without.

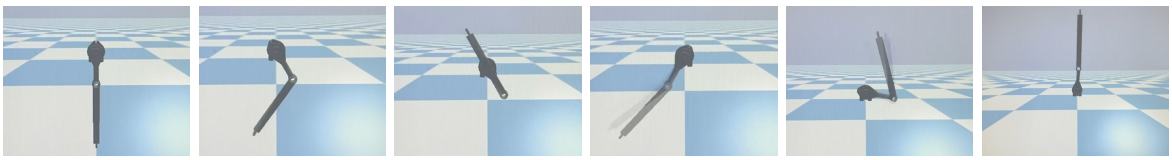

Figure 8: Sequence of images representing the simulation movement in case of null initial input and a *K Feedback* term

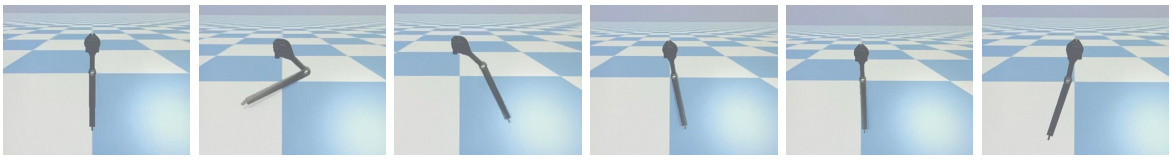

Figure 9: Sequence of images representing the simulation movement in case of null initial input and NO *K Feedback* term

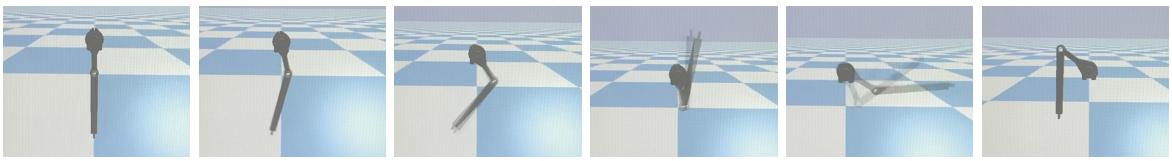

Figure 10: Sequence of images representing the simulation movement in case of a sinusoidal initial input and a *K Feedback* term

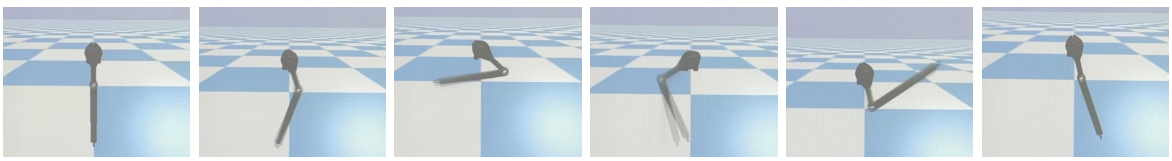

Figure 11: Sequence of images representing the simulation movement in case of a sinusoidal initial input and NO *K Feedback* term

5 Conclusions and Future Works

The use of DDP and, in particular, its variant iLQR, allows convergence to be achieved through an optimised trajectory, and the method is powerful and efficient. Indeed, as can be seen from the joint plots obtained through the iterations of iLQR, an optimised trajectory is achieved that satisfies the design requirements. However, some limitations of this algorithm must be pointed out, like any optimisation method based on cost chaining, it is highly dependent on cost functions, the definition of which represented the first major obstacle. Furthermore, parameter tuning could determine not only the number of iterations required for convergence but also whether or not the objective is reached. Another major limitation of this algorithm is the sub-optimal handling of complex dynamics, such as the one that in this project that can be assumed to be used by PyBullet in the simulation phase. There is an obvious discrepancy between the convergence obtained by the algorithm in the running phase, as demonstrated by its plots, and the animation that is then simulated by the PyBullet environment. To overcome this problem, one could think of future implementations where the DDP is flanked by a controller such as the Model Predictive Control (MPC). The idea would then be to divide optimisation into two distinct phases. In the first phase, the DDP is applied, trying to bring the system to convergence, and in the second phase, the MPC is inserted to regularise, optimise and refine the system's behaviour in the simulation phase. The MPC is known for its ability to anticipate and respond to dynamic variations in the system, thus playing a crucial role in the optimisation phase. Making the two algorithms work in synergy would exploit both the precision of the DDP's local feedback and the predictive capacity of the MPC, which would optimise the system in the presence of more complex dynamics requiring greater precision.

References

[1] Russ Tedrake. *Underactuated Robotics*. 2023.

[2] M.W. Spong. The swing up control problem for the acrobot. *IEEE Control Systems Magazine*, 15(1):49–55, 1995.

[3] Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4906–4913, 2012.

[4] Yuval Tassa, Nicolas Mansard, and Emo Todorov. Control-limited differential dynamic programming. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1168–1175, 2014.

[5] Justin Carpentier, Joseph Mirabel, Nicolas Mansard, and the Pinocchio Development Team. Pinocchio: fast forward and inverse dynamics for poly-articulated systems. https://stack-of-tasks.github.io/pinocchio, 2015–2018.

[6] Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36, 2019.

[7] Yunfei Bai Erwin Coumans. *PyBullet Quickstart Guide*, 2016-2022.