Acrobot swing-up using Iterative LQR Model Predictive Control

Barbara Bazzana Tommaso Belvedere Christian D'Amico

Master of Science in Control Engineering
Sapienza, University of Rome

1 Introduction

In this project we analyze and implement an MPC based controller for the swing-up manoeuvre of an acrobot robot.

Using Model Predictive Control (MPC), our control architecture consists of an *Optimization Engine* which, given the current state of the system, computes an optimal state trajectory and control action (knowing the dynamics of the system), then the first control is applied to the robot and the next state is fed to the *Optimization Engine* for the next step.

Following our reference [1], the *Optimization Engine* is based on the *iterative Linear Quadratic Regulator* (iLQR) trajectory optimizer, which consists in an iterative algorithm for nonlinear least squares optimization.

We will define the swing-up problem as an optimization problem where the cost function is biased towards the final goal, with soft constraints on the control action and on the state trajectories. Being the dynamics and the cost function nonlinear, each optimization will be approximated as a least squares problem minimized iteratively at each step of the MPC.

Clearly, to be able to use the MPC as an online controller, the optimization phase must occur in a short time frame between each step. While the cold-started optimization may require several iterations to converge to a locally optimal trajectory, the following steps will make use of the previously computed trajectory and in many instances converge in very few iteration, allowing the online use of the whole MPC.

In the following sections we will analyze the theoretical aspects of iLQR, along with some improvements necessary for the functioning of the implementation. Then we will show the results we obtained on the acrobot robot along with a discussion of the problems we have faced.

2 iLQR

In the present work *Iterative Linear Quadratic Regulator* (iLQR) was chosen as a trajectory optimization engine. iLQR is a variation on usual *Differential Dynamical Programming* (DDP). While in DDP first and second derivatives of the dynamics are used, iLQR only uses the first derivatives. Because of this, it lacks quadratic convergence properties, but is computationally faster. In this section, a general Finite Horizon Control context will be discussed, then iLQR will be described in its base form, and finally, it will be modified to improve its performance and robustness.

2.1 Finite Horizon Optimal Control

Given discrete time dynamics of the form

$$x_{i+1} = f(x_i, u_i)$$

the total cost of a trajectory can be expressed as a sum of a running cost ℓ and final cost ℓ_f as

$$J_0(x, U_i) = \sum_{i=0}^{N-1} \ell(x_i, u_i) + \ell_f(x_N)$$

where x is the entire state evolution and $U = u_0, u_1, ..., u_{N-1}$ is the control sequence. Thus, a solution to the optimal control problem, is a control sequence which minimizes this cost

$$U^*(x) \equiv \arg\min_U J_0(x, U)$$

Performing *trajectory optimization*, an optimal $U^*(x)$ will be found for a particular x only.

2.2 iLQR Base Method

The main solution idea is exploiting the Dynamic Programming Principle to reduce the minimization of a single control which can be obtained by proceeding backwards in time. To achieve this, the cost-to-go J_i is defined as the partial sum of costs from i to N

$$J_i(x, U_i) = \sum_{j=i}^{N-1} \ell(x_j, u_j) + \ell_f(x_N)$$

then the *Value* will be

$$V(x, i) \equiv \min_{U_i} J_i(x, U_i)$$

By setting the *Value* at the final time instant as the final cost $V(x, N) \equiv \ell_f(x_N)$, the dynamic programming estimation of the value will take the form:

$$\min_u \left[\ell(x, u) + V(f(x, u), i + 1)\right]$$

This equation can be expanded as a function of perturbations as

$$Q(\delta x, \delta u) = \ell(x + \delta x, u + \delta u, i) - \ell(x, u, i) + V(f(x + \delta x, u + \delta u), i + 1) - V(f(x, u), i + 1)$$

which expanded to the second order takes the form

$$Q(\delta x, \delta u) \approx \frac{1}{2} \begin{pmatrix} 1 \\ \delta x \\ \delta u \end{pmatrix}^T \begin{pmatrix} 0 & Q_x^T & Q_u^T \\ Q_x & Q_{xx} & Q_{xu} \\ Q_u & Q_{ux} & Q_{uu} \end{pmatrix} \begin{pmatrix} 1 \\ \delta x \\ \delta u \end{pmatrix} \tag{1}$$

where the expansion coefficients are

$$Q_x = \ell_x + f_x^T V_x' \tag{2a}$$
$$Q_u = \ell_u + f_u^T V_x' \tag{2b}$$
$$Q_{xx} = \ell_{xx} + f_x^T V_{xx}' f_x + V_x' f_{xx} \tag{2c}$$
$$Q_{uu} = \ell_{uu} + f_u^T V_{xx}' f_u + V_x' f_{xx} \tag{2d}$$
$$Q_{ux} = \ell_{ux} + f_u^T V_{xx}' f_x + V_x' f_{ux} \tag{2e}$$

In equations (2c, 2d, 2e), the last terms are ignored in iLQR as they are second order terms of the dynamics. Note how, to lighten the notation, V' has been used as $V' \equiv V(i+1)$. Then, minimizing the quadratic approximation of the value function (1) with respect to the input δu, a closed form solution can be found:

$$\delta u^* = \arg\min_{\delta u} \; Q(\delta x, \delta u) = -Q_{uu}^{-1}(Q_u + Q_{ux}\delta x) \tag{3}$$

This solution can be interpreted as composed by two terms: an open loop gain term operating on the input $k = -Q_{uu}^{-1}Q_u$ and a feedback term which also takes into account the input $K = -Q_{uu}Q_{ux}$. Substituting the optimal policy (3) into (1), an estimate of the evolution of the Value can be obtained in a quadratic model as

$$\Delta V(i) = -\frac{1}{2}Q_u Q_{uu}^{-1} Q_u \tag{4a}$$

$$V_x(i) = Q_x - Q_{xu}Q_{uu}^{-1}Q_u \tag{4b}$$

$$V_{xx}(i) = Q_{xx} - Q_{xu}Q_{uu}^{-1}Q_{ux} \tag{4c}$$

The above equations[1] may be used iteratively from the end of the time horizon to the beginning together with the equations 2 to compute the control gains and the estimate of the value.

A final ingredient of the iLQR algorithm is a forward pass that constitutes the forward propagation of a new trajectory. This is achieved by

$$\hat{x}(1) = x(1) \tag{5a}$$

$$\hat{u}(i) = u(i) + k(i) + K(i)(\hat{x}(i) - x(i)) \tag{5b}$$

$$\hat{x}(i+1) = f(\hat{x}(i), \hat{u}(i)) \tag{5c}$$

Putting it all together, the base algorithm of iLQR is composed of the following steps:

1. **Derivatives:** given a nominal (x, u, i) sequence, compute the derivative of ℓ and f in (2).
2. **Backwards pass:** iterate Eqs. (2) and (4) from the end of the prediction horizon to the beginning $(i = N-1, ..., 1)$.
3. **Forward pass:** Propagate forward Eqs. (5).

This preliminary algorithm has several problems that will be discussed in the following section. In particular, the backwards pass will be improved with a regularization step, while a scaled version of equation 5b and associated search method will be introduced as far as the forward pass is concerned.

2.3 Improvements to the Trajectory Optimizer

Our reference [1] has introduced two major improvements to the basic algorithm. First of all, the authors suggest to introduce a regularization factor μ aimed to robustify the algorithm with respect to non positive-definite control-cost Hessian. As a general idea, steps should me made more conservative in such situations. Although a well-known standard regularization technique is already

[1] There was a mistake in the analog of 4b in [1]. As a consequence, the first and the second term on the right side were not compatible. [2] gave us the opportunity to test our ideas and verify their correctness.

available from the theory of the Gauss-Newton method, an improved regularization technique is proposed, which is more suitable for our control problem.

The second improvement consists in introducing a scaled (barring the feedback term which is not) version of the control equation from 5b to be explored as the scaling factor α changes in the range $(0, 1]$ in order to pick the biggest one that allowed the cost to decrease at a certain step. Only in case the cost function has actually decreased, the iteration is considered valid and the new trajectory is accepted as the current optimal one.

2.3.1 Improved Regularization

Being the iLQR, which is a variant of the classic DDP algorithm, the control analog of the Gauss-Newton method for nonlinear least-squares optimization, the steps taken by DDP are comparable to or better than a full Newton step for the entire control sequence. Thus, as in Newton's method, care must be taken when the Hessian is not positive-definite. The standard regularization is to add a diagonal term to the local control-cost Hessian

$$\widetilde{Q}_{uu} = Q_{uu} + \mu I_m$$

This modification makes the steps more conservative. However, it can cause the same control perturbation to have different effects at different times, depending on the control-transition matrix f_u. This is why our reference [1] introduced a scheme that only penalizes deviations from the states, rather than controls:

$$\widetilde{Q}_{uu} = l_{uu} + f_u^T (V'_{xx} + \mu I_n) f_u + V'_x \cdot f_{uu} \tag{6a}$$
$$\widetilde{Q}_{ux} = l_{ux} + f_u^T (V'_{xx} + \mu I_n) f_x + V'_x \cdot f_{ux} \tag{6b}$$
$$k = -\widetilde{Q}_{uu}^{-1} Q_u \tag{6c}$$
$$K = -\widetilde{Q}_{uu}^{-1} \widetilde{Q}_{ux} \tag{6d}$$

Robustness is thus improved since the feedback gains do not vanish as $\mu \to \infty$, but rather force the new trajectory closer to the old one. Moreover, the Value update from previous section is no longer valid as making the several cancellations of Q_{uu} and its inverse which lead to that update induces now an error. Thus an improved Value update is adopted

$$V_x(i) = Q_x + K^T \widetilde{Q}_{uu} k + K^T Q_u + \widetilde{Q}_{ux}^T k \tag{7a}$$
$$V_{xx}(i) = Q_{xx} + K^T \widetilde{Q}_{uu} K + K^T \widetilde{Q}_{ux} + \widetilde{Q}_{ux}^T K \tag{7b}$$

How the value of μ is chosen and updated is explained in section 2.3.3 as a few more concepts are needed. The idea though is that μ can be decreased to enjoy faster convergence only if the previous iteration was not affected by convergence issues, which basically means that the cost has decreased with respect to the previous iLQR iteration.

2.3.2 Improved Line Search

An important issue is that for a general nonlinear system the new trajectory may exit the quadratic model's region of validity, and the cost may not decrease. Given that we aim to reduce the cost

function, it is crucial to be able to discriminate between iterations that did provide a reduction of the cost and iterations that did not, which is kind of a failure from our perspective. If we were able to do this, we could decide whether to accept the new trajectory or not depending on a suitable termination condition that takes into account the reduction of the cost function. As a further improvement, a scaled version of the full control law from equation 5b could help in avoiding failures being somehow more conservative.

More formally, given a backtracking line-search parameter $0 < \alpha \leq 1$, a scaled version of the full step can be computed as

$$\hat{u}(i) = u(i) + \alpha k(i) + K(i)(\hat{x}(i) - x(i)) \tag{8}$$

Thus, it is possible to decide whether to accept the improvements of a given iteration depending on the ratio between the actual and the expected reduction of cost

$$z = J(u_{1\ldots N-1}) - J(\hat{u}_{1\ldots N-1}))/\Delta J(\alpha) \tag{9}$$

where

$$\Delta J(\alpha) = \alpha \sum_{i=1}^{N-1} k(i)^T Q_u(i) + \frac{\alpha^2}{2} \sum_{i=1}^{N-1} k(i)^T \widetilde{Q}_{uu}(i) k(i)$$

In particular, in order for an iteration to be accepted, i.e. the new trajectory and the new cost are stored as current ones, we need the actual reduction to be at least k times the expected reduction, with $k > 0$. The most conservative choice is to set $k = 0$, in which case any positive reduction is considered valid. As a matter of fact, as we will discuss in section 3.2, we will also accept iterations where the cost is slightly increased.

2.3.3 Regularization Schedule

As we were mentioning above, now that we have introduced what accepting an iteration means and what are the conditions under which it occurs, the regularization schedule, i.e. the update law for μ, can be made explicit.

Actually, in case an iteration is accepted, which means that it has ended with no convergence issues, it is then possible to decrease the value of the regularization factor μ to enjoy fast convergence

$$\Delta_\mu = min\left(\frac{\Delta_\mu}{\Delta_0}, \frac{1}{\Delta_0}\right)$$

$$\mu = \begin{cases} \mu \cdot \Delta_\mu & \text{if } \Delta_\mu \Delta_0 > \mu_{min} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

On the contrary, in case an iteration is not accepted because of insufficient or even non-positive reduction ratio, μ is increased to enjoy more regularization, and thus improve robustness

$$\Delta_\mu = max(\Delta_\mu \Delta_0, \Delta_0)$$

$$\mu = max(\mu \cdot \Delta_\mu, \mu_{min}) \tag{11}$$

2.3.4 Improved algorithm sketch

After these improvements, a new sketch of the control algorithm is as follows:

1. **Derivatives:** given a nominal (x, u, i) sequence, compute the derivative of ℓ and f in (2). This step has not changed from the one in the previous algorithm sketch.

2. **Backwards pass:** iterate Eqs. (2), their improvements in (6) and the new value derivatives (7) from the end of the prediction horizon to the beginning ($i = N - 1, ..., 1$). Whenever an iteration is not accepted, increase μ according to section 2.3.3. If successful, decrease μ.

3. **Forward pass:** Propagate forward Eqs. (5) with $\alpha = 1$. If the integration diverges or actual reduction is insufficient, decrease alpha and restart the forward propagation.

3 Simulation

We have implemented the use of iLQR in an MPC fashion on the Acrobot as a simple example of an underactuated robot. In particular, we have been focusing on the swing up manoeuvre. The simulations have been performed using Matlab.

In the following, we first explain how the problem is formalized in our particular case study, then discuss how our program is organized, focusing in particular on the interaction between the iLQR trajectory optimizer and the MPC, and finally illustrate the results.

3.1 Settings

In order to interact with a Matlab program that is supposed to solve the deterministic finite-horizon optimal control problem with the iLQR, either in the context of MPC or not, we need to provide the discrete time dynamics of our system and the expressions of the total cost, split in running and final cost.

The Acrobot is a 2R robot which is by definition underactuated as only one input torque is available, in particular at the second joint, i.e. $\tau = \begin{pmatrix} 0 \\ \tau_2 \end{pmatrix}$. Thus, the Euler-Lagrange alternative model can be written as

$$M(q)\ddot{q} + c(q, \dot{q}) + e(q) = \begin{pmatrix} 0 \\ \tau_2 \end{pmatrix}. \tag{12}$$

Moreover, it can be shown that in our case

$$M(q) = \begin{pmatrix} a_1 + 2a_2c_2 & a_3 + a_2c_2 \\ a_3 + a_2c_2 & a_3 \end{pmatrix}, \quad c(q, \dot{q}) = \begin{pmatrix} -a_2s_2\dot{q}_2(2\dot{q}_1 + \dot{q}_2) \\ a_2s_2\dot{q}_1^2 \end{pmatrix}, \quad e(q) = \begin{pmatrix} a_4s_1 + a_5s_{12} \\ a_5s_{12}, \end{pmatrix},$$

where

$$a_1 = I_1 + m_1l_{c1}^2 + I_2 + m_2(l_1^2 + l_{c2}^2)$$

$$a_2 = m_2l_1l_{c2}$$

$$a_3 = I_2 + m_2l_{c2}^2$$

$$a_4 = g(m_1l_{c1} + m_2l_1)$$

$$a_5 = gm_2l_{c2}$$

Dynamic Parameter	Value
I_1	$0.7\ kg \cdot m^2$
I_2	$0.3\ kg \cdot m^2$
m_1	$1\ kg$
m_2	$0.5\ kg$
l_1	$1\ m$
l_2	$0.5\ m$
l_{c1}	$0.4\ m$
l_{c2}	$0.5\ m$

Table 1: Parameters used in the dynamics model.

For the sake of completeness, table 1 shows all useful values for defining the dynamics of the Acrobot.

Finally, the discrete time dynamics has been obtained by inverting equation 12 to derive the expression of $\begin{pmatrix}\ddot{q}_1(k)\\ \ddot{q}_2(k)\end{pmatrix}$. Once $\ddot{q}(k)$ has been computed, by applying Euler integration method we obtain

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

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

As in any optimal control problem, we need to define the cost function that will be minimized by our controller. There are two main aspects which we need to consider at this stage. Of course, the cost function needs to be biased towards the goal that can be achieved by introducing a term $(x - x_{des})^T W (x - x_{des})$ with W suitable diagonal weighting matrix both in the final and in the running cost. At the same time though, the cost function can and must be used in such a way to introduce soft constraints on the control actions, that can be done by introducing two different weighting factors `stateWeight` and `InputWeight`, one for the state-dependent term of the running cost and the other for the input-dependent term of the running cost. The final cost only depends on the state so that only one scaling factor needs to be introduced `stateWeightFinal`, aimed to set the relative importance between the final and the running cost. Putting together the above considerations and the cost function suggested by our reference [1], we came down to the running cost of equation 13 and the final cost of equation 14.

$$l(x,u) = \texttt{stateWeight}\cdot(\sqrt{(x-x_{des})^T W (x-x_{des}) + \alpha^2} - \alpha) + \texttt{InputWeight}\cdot\alpha^2\cdot cosh(u/\alpha - 1) \quad (13)$$

$$l_F(x) = \texttt{stateWeightFinal}\cdot(\sqrt{(x-x_{des})^T W (x-x_{des}) + \alpha^2} - \alpha) \quad (14)$$

Once, the formulae for the final and the running cost have been established, obtaining a satisfactory trajectory is only a matter of parameter tuning. In particular, we have chosen $\alpha = 1$,$W = diag(1, 1, 0.1, 0.1)$ (that aims to weight more errors on the joint positions than errors in the velocities), `stateWeight` $= 0.01$, `inputWeight` $= 0.001$ and `stateWeightFinal` $= 100$.

Particular attention has been paid to using a suitable `wrap` function whenever needed, e.g. in the evaluation of $x - x_{des}$ that becomes $\begin{pmatrix} \texttt{wrap}(x(1) - x_{des}(1)) \\ \texttt{wrap}(x(2) - x_{des}(2)) \\ x_3 - x_{des}(3) \\ x_4 - x_{des}(4) \end{pmatrix}$.

Finally, the length of the time horizon `timeHorizon` needs to be tuned. Unfortunately, it is a problem-dependent quantity which must be found by trial-and-error, although we might have a guess given the length of the desired trajectory especially if we have already experimented the method on the same target robot. What we found in our case is `timeHorizon` $= 1000$. In particular, in case the time horizon is too short the *Optimization Engine* returns a trajectory that does not reach the goal.

3.2 MPC implementation

For every MPC iteration, as many iterations of iLQR are performed, i.e. forward and backward pass as illustrated in section 2.2, as needed to reach convergence, i.e. non-zero termination condition, provided that they are less than the maximum allowed number (we set it to 200). In particular, we have been using `dcost < tolFun` as termination condition, where `dcost` is the difference between the cost stored at the previous iteration and its current value and `tolFun` $= 10^{-4}$.

As discussed in subsection 2.3.2, in order for a forward pass to be accepted, the cost decrease must be positive and a minimum ratio between the cost decrease and the expected reduction can be imposed. In our experience however, particularly in the absence of noise and when the trajectory is approaching the target (so that variations in the cost become less and less relevant), the value of the new cost computed in the forward pass happened to be slightly more than the previous one, triggering the regularization schedule and loosing time in unnecessary iterations.

This said, it has proven very effective to also accept a forward iteration when the cost decrease is small in absolute value, thus allowing for small increases in the cost, keepeng in mind that in a MPC scheme we are going to optmize again the trajectory at the next step. In our case, we have achieved good results with the condition $\|\texttt{dcost}\| < 10^{-3}$. Note that the presence of noise helps in this matter since in general it will worsen the suboptimal trajectory and it will be more likely to have a decrease in cost even in the edge cases.

Once iLQR managed to converge, the first element of the optimized input trajectory (`uCurrent`) is fed to our actual Acrobot robot, which is represented by the function `dynamicsFunc`. In this way, the updated position and velocity of the robot (`xCurrent`) are computed. Finally, the prediction horizon is shifted forward as the alternative name of MPC, receding horizon control, suggests by setting the first element of `stateSuboptimal` to `xCurrent` and the first element of `inputSuboptimal` to `uCurrent`, while a new value, computed following suitable criteria, is added at the end of both arrays.

MPC terminates when either the error norm is less than `tolErrorNorm`, which we have set to 0.01, or the maximum allowed number of iterations has been reached. In particular, we set it to `timeHorizon` $+ 500$. Indeed, we believe that the goal will be reached within the `timeHorizon`, nevertheless we introduce some tolerance with respect to it.

An algorithmic description of the whole process is provided in the following, with a few more hints on how we actually implemented MPC in Matlab. It is important to notice that with the specific settings from section 3.1, we could set $\mu = 0$ at the beginning of every iLQR iteration without experiencing convergence issues. Moreover, we have monitored the variable μ and seen

that it never increases, which means that we have been enjoying the fastest possible convergence rate throughout the simulation.

Algorithmic description of the use of iLQR in an MPC fashion.

while errorNorm > tolErrorNorm **and** (numIterationMPC < maxNumIterationsMPC) **do**
  $terminationCondition \leftarrow 0$
  $numIterationILQR \leftarrow 1$
  $\mu \leftarrow 0$
  while (**not** terminationCondition) **and** (numIterationILQR < maxNumIterationsILQR) **do**
    stateOptimized, inputOptimized, terminationCondition ←
        trajectoryOptimizazion(stateSuboptimal, inputSuboptimal)
    if $\mu > \mu_{max}$ **then**
      break
    end if
    {`Reset trajectory for next iteration`}
    stateSuboptimal = stateOptimized
    inputSuboptimal = inputOptimized
    if terminationCondition = 1 **then**
      break
    end if
    numIterationILQR ← numIterationILQR+1
  end while
  {`Extract the first optimal input and apply it to the robot`}
  uCurrent ← inputOptimized(1)
  xCurrent ← dynamicsFunc(xCurrent, uCurrent)
  errorNorm ← norm(xCurrent(1:2)-xDesired(1:2))
  {`Add a new final state to the trajectory`}
  newFinalState ← dynamicsFunc(stateOptimized(:,end),inputOptimized(end))
  newFinalInput ← inputOptimized(end)
  stateSuboptimal ← [stateOptimized(:,2:end), newFinalState]
  stateSuboptimal(:,1) ← xCurrent
  inputSuboptimal ← [inputOptimized(2:end), newFinalInput]
  inputSuboptimal(1) ← uCurrent
  if numIterationMPC ≥ maxNumIterationsMPC **then**
    break
  end if
  numIterationMPC ← numIterationMPC+1
end while

3.3 Results

The described methods and parameters provide satisfactory results. The algorithm achieves convergence with a loss lower than the termination threshold in 1175 time steps (or 6 simulation seconds). The controller correctly understands the form that a swing-up motion should have, moving the

second link so as to increase the total amount of mechanical energy in the system in order to reach an orbit that takes it to the goal. A snapshot of the full trajectory can be observed in Figure (1).

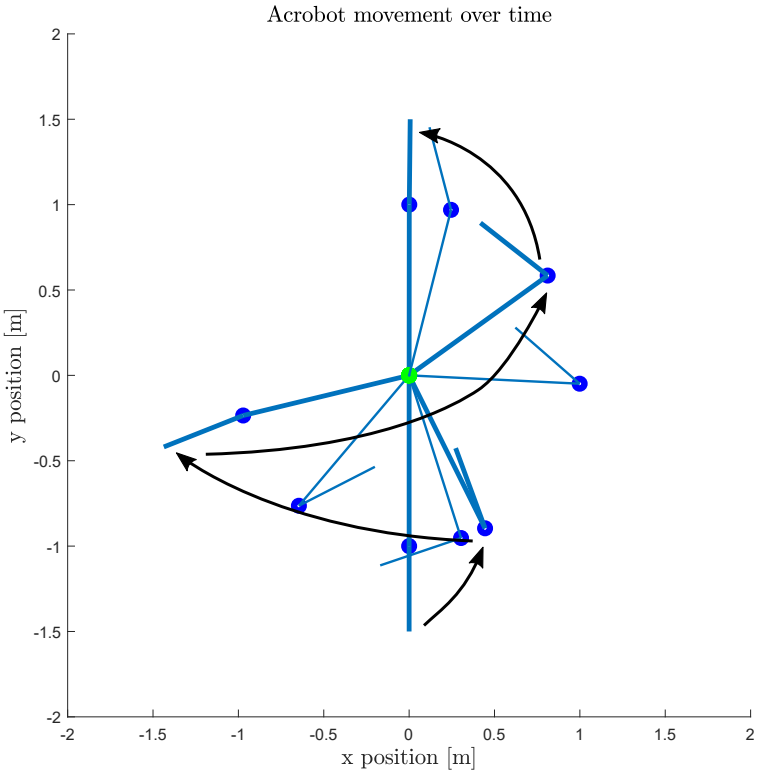

Figure 1: Trajectory of Acrobot when controlled by MPC

A more thorough representation of the behavior of Acrobot during execution is available in Figure (2) depicting its position and velocities over time. The input profile is also reported in Figure (3).

As predicted, convergence speed of the iLQR optimization procedure is quite slow at first (taking 55 iterations of the algorithm during the first MPC step) but quite fast during motion when warm started. In fact, after the first iteration almost all of the iLQR calls reach convergence in less than 3 iterations. Not only, when the MPC is close to the goal, performance further improves. Actually, it takes only one single iteration of iLQR for every MPC step. With the described parameters, faster convergence is observable after around 4.5 seconds of simulation time (900 simulation steps).

Interestingly, only one of the improvements on the basic form of iLQR described in Section (2.2) has been found to be critical for the success of the algorithm using the reported parameters. In fact, with the specific settings illustrated in section 3.1, the regularization parameter μ is always 0 during execution, in case we set it to be zero at the beginning of the iLQR iterations. This means that at no point a candidate solution is rejected. Actually, if a candidate solution were rejected, μ would increase. However, we never experience such a case.

Furthermore, in our simulations, we have been very conservative in that we consider any positive reduction as valid, no matter how big the ratio between expected and actual reduction is. Thus, saying that no candidate solution was rejected means that an even very small decrease in the cost function has been experienced at every step. Note how, having $\mu = 0$ through the whole simulation

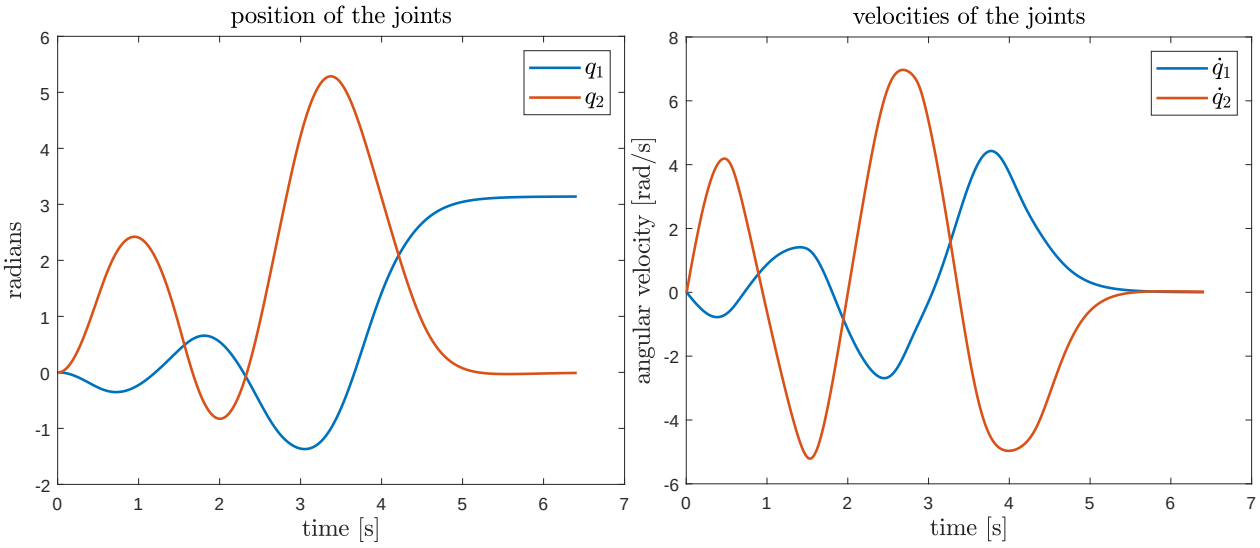

Figure 2: Evolution of position and velocities of the joints

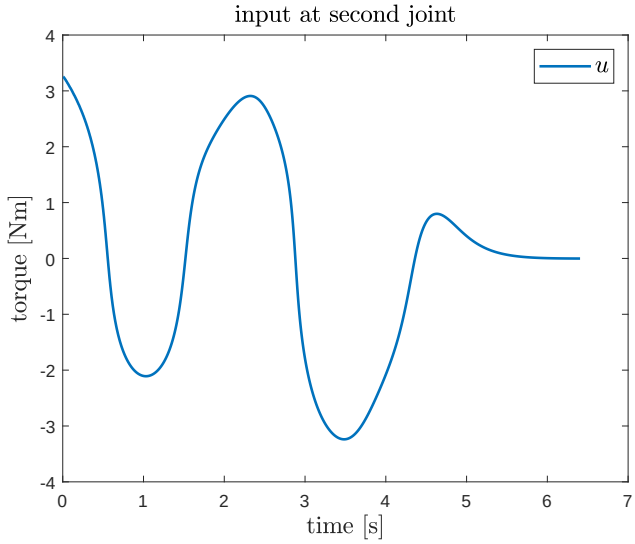

Figure 3: Input at the actuated joint

means that no convergence issues have been encountered, so that iLQR is allowed to converge at the fastest rate.

In the end, while there was not at all the necessity to increase μ, the other improvement from [1], i.e. the line search parameter α, is critical for the success of the algorithm. Actually, especially during the first MPC iterations, the iterations of iLQR end with $\alpha < 1$ which indicates that $\alpha = 1$ (the non-scaled version of the control input, i.e. 5b) does not provide a positive reduction of the cost function. Instead, once α is reduced, we do obtain a positive reduction and the iteration is in the end accepted. It is important to point out that, as expected, α is almost always equal to one as the `inputSuboptimal` and the `stateSuboptimal` get closer to the optimal ones, which happens after a very small number of MPC iterations indeed.

3.3.1 Choice of the initial guess

The previous results follow from the choice of setting the initial input and state trajectory identically to zero. To be precise, the initial state trajectory to be optimized is obtained by integration of the dynamical model with the chosen input trajectory, but since our starting point is an equilibrium an identically zero input results in no motion.

As mentioned, the first iLQR optimization, which is the most taxing and challenging for convergence, takes 55 iterations. If we were to choose an input closer to the optimal one, the number of iterations needed would decrease, as it is for the successive optimizations which are warm-started with the result of the previous one.

This said, we have generated a trajectory using a sinusoidal input of unitary amplitude and period equal to the time horizon, with interesting results. This choice makes no use of the information we have gathered from the previous simulations but is solely based on physical intuition of a primitive motion that the acrobot would do to kick-start the manoeuvre.

As we could expect, the first iLQR optimization takes 35 iterations to converge to the optimum, the overall MPC trajectory takes 1281 steps to stop with the same tolerance conditions and is reported in figure 4.

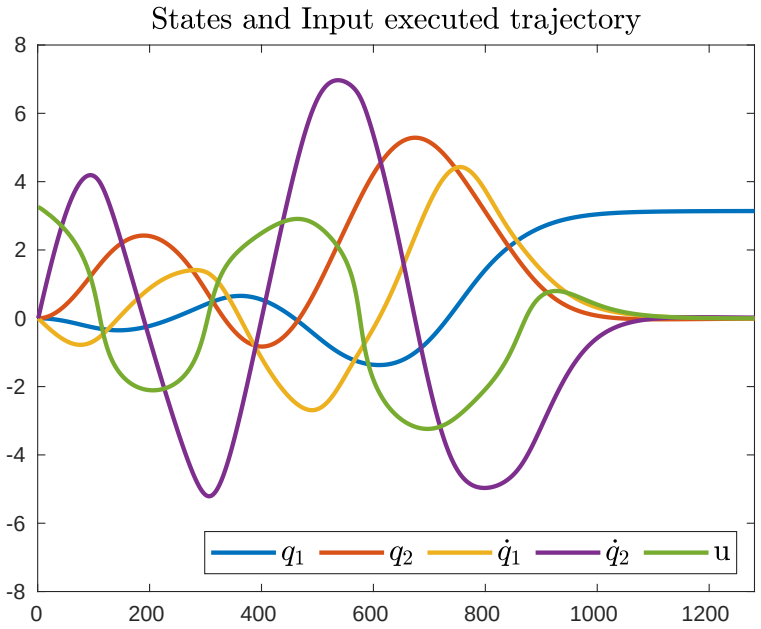

Figure 4: MPC trajectory with sinusoidal initial guess.

This trajectory correctly resembles the fact that we are biased towards a positive initial input and while the performance is similar the trajectory is very different.

Interestingly, if instead of a sinusoidal input we choose a square wave with the same characteristics, we converge to the same trajectory (with 32 iterations for the first iLQR optimization), showing how the result is somewhat robust with respect to this class of initial guesses.

3.4 Addition of input noise

In order to test the robustness of the presented control strategy, we have then added input gaussian noise with zero mean aimed to simulate actuation uncertainty, obtaining quite satisfactory performances.

In particular, in the MPC framework, the actuation uncertainty affects the input of the iLQR only through the first sample of the state trajectory to be optimized. Indeed, while the input trajectory is just shifted in time before the next MPC iteration starts, the first sample of the state trajectory is overwritten with the actual state of the robot that is supposed to be measured, which in a simulation context means that its value is computed by applying the disturbed control input to the previous state. Figures 5 and 6 show the achieved results.

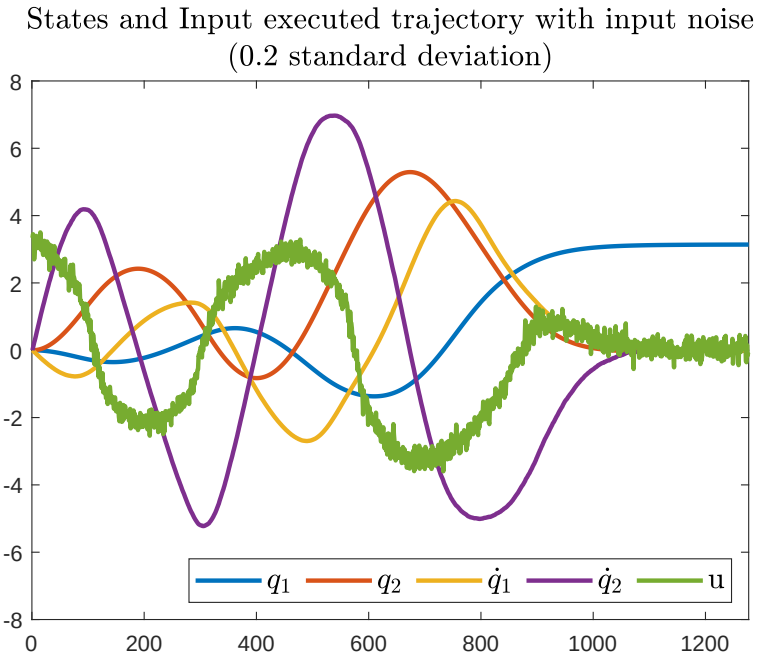

Figure 5: State and input trajectories with input noise with zero mean and 0.2 standard deviation.

As we were already pointing out in section 3.2, although having introduced the condition $\|\texttt{dcost}\| < 10^{-3}$ might still help in some cases, it is generally less crucial than in the noise free setting. Indeed, the presence of noise allows more iterations to be accepted as it is easier to get improvements during an iLQR iteration since the trajectory to be optimized is less similar to the optimal one.

3.5 Choice of cost function

The cost function that was used up to now is quite a peculiar one. Some more insight should be provided on the reasons of this choice.

The usual choice of for a cost function in most applications is that of a quadratic one. In this particular use case, that would have the form:

$$l(x,u) = \texttt{stateWeight} \cdot (x - x_{des})^T W (x - x_{des}) + \texttt{InputWeight} \cdot u^2 \tag{15}$$

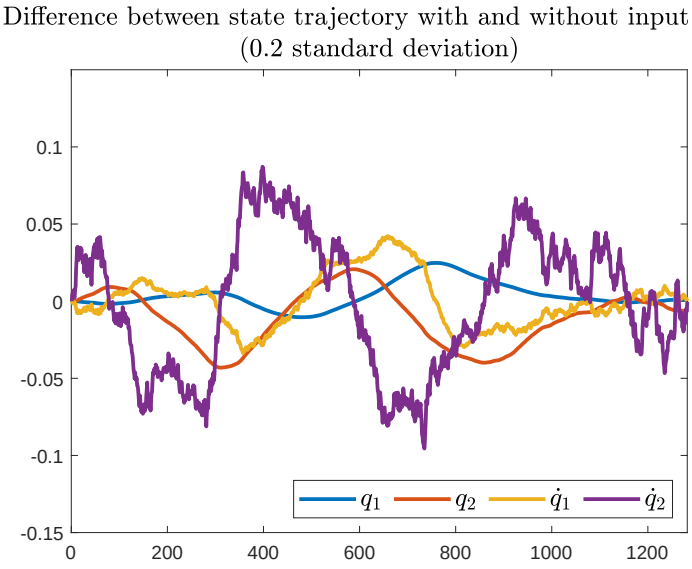

Figure 6: Difference between the state trajectories with and without input noise.

$$l_F(x) = \texttt{stateWeightFinal} \cdot (x - x_{des})^T W (x - x_{des}) \tag{16}$$

Focusing on the state part of these costs, their main disadvantage is the inherent difficulty in tuning parameters for a general task. When costs are *quadratic*, units of measurement between components of the state are not preserved. Thus, it's harder to find a suitable matrix W that encourages good behavior in many different situations. On the other hand, simply linear costs composed of absolute functions, have discontinuous derivatives around the goal, which make them subject to erratic behavior when first converging. Therefore, the choice of a smooth absolute function (such as the one shown in Eq: 13, 14) seems like an effective one. It has a continuous derivative near the minimum of the cost function, thus ensuring smooth convergence, while exhibiting asymptotically linear behavior elsewhere.

An added benefit of smooth absolute functions is encouraging cyclic asymptotic behavior when farther from the convergence point. Quadratic costs scale in a nonlinear way. Thus the controller has a stronger incentive than a linear controller to quickly reach states that are closer to the goal. Therefore, when using a quadratic cost, efficient cyclical behavior may be ignored in order to quickly reach regions that are closer to the goal. In the Acrobot control problem, no difference can be observed because of this consideration. The original authors imply that this is important in more complex systems (humanoid and snake robots).

Looking now at the input part of the cost function, it is often useful to have harder caps on the cost. Sometimes, when the input is constrained, MPC steps in the algorithm are wasted when the trajectory optimizer requires an input that is out of bounds. Quadratic functions do not grow fast enough for this to be systematically avoided. Thus, it is useful to use a function that is able to increase exponentially after a slow start around the no-input behavior. And that is exactly the behavior that can be observed from the chosen input cost in Eq: 13. Additionally, it is much easier to tune the parameter α in Eq 13 than to guess an appropriate weight `inputWeight` that ensures proper behavior in all cases. Thus in problems with no hard constraints on the input, a solution using quadratic costs may have higher input values and more aggressive trajectories.

All things considered, the underlying control mechanism works correctly even when using quadratic

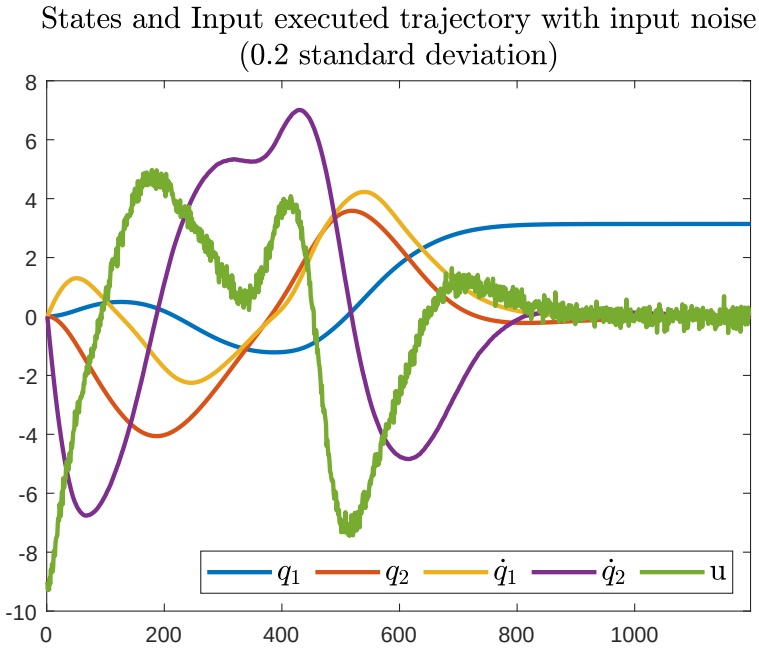

Figure 7: States and input trajectory with noisy input (0.2 standard deviation) when using quadratic cost functions

costs. In general, due to the just mentioned tuning issues for the input cost, solutions found with quadratic costs are much more aggressive and display higher input magnitude during motion. An example of this, is provided in Figure 7. This simulation was performed with the exact same parameters as the previous ones but using the costs in Eq: 15, 16. It can be noted how the top velocities and the highest inputs are significantly higher than they were in the previously described cases.

4 Conclusions

Throughout the development of the project, we have understood the theoretical background of the iLQR trajectory optimization method, both in its basic version, which is briefly recalled also in [1], and in its improved version again from [1]. Moreover, we have implemented it in an MPC fashion using Matlab.

First of all, our goal was to implement the iLQR optimization, leaving its employment in the context of MPC as a further step. In fact, once the iLQR optimizer is available, it can be easily emplaced in a very standard MPC algorithm, provided that prior to every call to iLQR a few parameters are reset. In this framework, we started with the implementation of the basic version of the algorithm. Unfortunately, the very first trials failed partly because we were not treating properly the cost function (in particular, we had the necessity to introduce two different functions, one for the running cost and one for the final cost), partly because we were not employing the improvements from [1]. As already highlighted in section 3.3, the introduction of the improved line-search turned out to be of crucial importance.

Once we managed to correctly deal with these two aspects, we did not encounter much difficulty in inserting the iLQR trajectory optimizer in the context of MPC. Finally, we could also experience

what happens in case we use different initializations for the suboptimal trajectories, which is also an interesting and important aspect in the analysis of this method.

In conclusion, we had the opportunity to study a very efficient and important optimization method and to work out a practical example in an MPC fashion, which allowed us to gain a much better understanding of it.

References

[1] Yuval Tassa, Tom Erez and Emanuel Todorov *Synthesis and Stabilization of Complex Behaviors through Online Trajectory Optimization* 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems

[2] Yuval Tassa, Nicolas Mansard and Emo Todorov *Control-Limited Differential Dynamic Programming* ICRA 2014