UNIVERSITY OF TRENTO

Department of Industrial Engineering
Master's degree in Mechatronics Engineering

TAKING ACCOUNT OF UNDERACTUATION IN DDP

Second assignment report

Mattia Pettene 239145
Gabriele Poggesi 233220

Course of Optimization-based robot control

Academic Year 2022 - 2023

Introduction

In this assignment we are asked to make practice using Differential Dynamic Programming (DDP) for generating reference trajectories and to compare two methods for considering underactuation in DDP, applied to the case of a double pendulum without motor on the second joint. This pendulum has to perform a swing-up maneuver.

Since our model is a fully actuated double pendulum, it is necessary to properly modify the code in order to take into account the underactuation. In order to do that, the are two different methods which could be applied: the "selection matrix" method and the "additional penalty" one. The first is based on the use of a diagonal selection matrix S, such that:

$$\begin{cases} S[i,i] = 1, & if\ i\ corresponds\ to\ an\ actuated\ joint \\ S[i,i] = 0, & if\ i\ corresponds\ to\ a\ not\ actuated\ joint \end{cases} \tag{1}$$

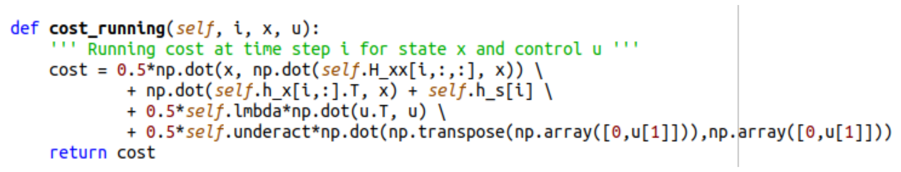
Multiplying by this matrix the control torque vector in the system dynamics equations, it is possible to force the torques associated with the joints which are not actuated, to be zero. The second method consists, instead, in adding a term to the running cost with a large weight to penalize the torque provided by a fictitious motor on the passive joint.

Question 1

In this first question, it is requested to implement the additional penalty method to make the double pendulum underactuated. To penalize the torque provided by the second motor, a large weight is added to the running cost in this way:

$$running\ cost\ +=\ \frac{\texttt{underact}}{2} \cdot ||u_i||^2 \tag{2}$$

where `underact` is the weight and it is equal to 10^6.



```
def cost_running(self, i, x, u):
    ''' Running cost at time step i for state x and control u '''
    cost = 0.5*np.dot(x, np.dot(self.H_xx[i,:,:], x)) \
            + np.dot(self.h_x[i,:].T, x) + self.h_s[i] \
            + 0.5*self.lmbda*np.dot(u.T, u) \
            + 0.5*self.underact*np.dot(np.transpose(np.array([0,u[1]])),np.array([0,u[1]]))
    return cost
```

Figure 1: Code implementation of the additional penalty method added to the running cost.

As it is possible to observe in Fig. 1, the term in eq. 2 is implemented in the last line before the *return*. In order to penalize a lot the second joint motor, it is considered only the term referred to this joint. Indeed, the square of $u[1]$ (control of the underactuated joint) is multiplied by `underact`, increasing the running cost. The idea is that, since controlling the second joint is very expensive in term of cost, the solver prefers to perform the swing-up maneuver acting only on the first motor; in this way, the whole system results underactuated. At the same time, since a new term that depends on u is added to the running cost, it is necessary to modify accordingly the gradient and the hessian with respect to u of it. The derivatives of the term shown in eq. 2 are computed and added.

After implementing the additional penalty method, it is tested to verify that it leads to the same solution found with the selection matrix method. The first simulation is carried out setting `SELECTION_MATRIX = 1`, `PUSH = 0`, `ACTUATION_PENALTY = 0`. The results are the following:

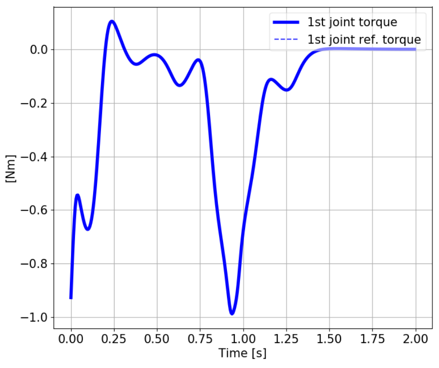
$$\text{iterations taken} = 33 \quad \text{cost} = 1481.1789 \quad \text{effort} = 6.5338$$

In order to test the second method, new flags are set in the configuration file. They are: `SELECTION_MATRIX = 0`, `ACTUATION_PENALTY = 1` and `PUSH = 0`.

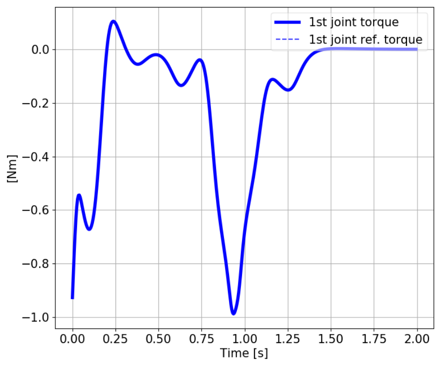
In this case the solution leads to:

$$\text{iterations taken} = 33 \quad \text{cost} = 1481.1789 \quad \text{effort} = 6.5338$$

It is immediate to notice that the solution found is exactly the same both in term of costs and in convergence rate.



(a)



(b)

Figure 2: First joint torque comparison using the two different methods to perform the underactuation.

In Figure 2 is shown the comparison of the first joint torque obtained using the selection matrix method (Fig. 2a) and the additional penalty one (Fig. 2b): it is trivial to observe that the two pictures are exactly the same, synonymous that in both cases only that motor joint is working and follows exactly the same function. This leads to the conclusion that the additional penalty method behaves very well and the solver prefer to deal with an underactuation rather then have a huge increase in term of cost.
To control how both joints rotate, their velocities are plotted in Fig. 3:

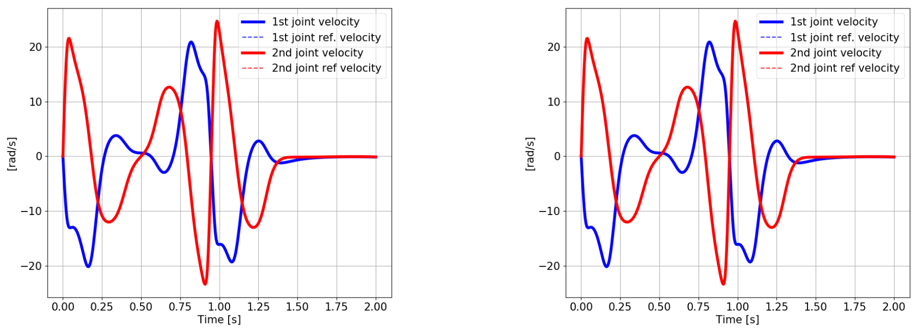


Figure 3: Joint velocities simulated with selection matrix method on the left, and with actuation penalty method on the right

Question 2

The second question consists in a simulation of the system with four external events that are going to affect the system behaviour. In particular, the flag `PUSH` in the configuration file introduces four instantaneous variations of the velocity in the second joint.

The main file contains the definition of this variations and it is possible to comprehend that they consist in an addition of 3 rad/s and they occur after N/8, N/4, N/2 and 3*N/2 simulation steps. The last velocity modification will not take place due to the fact that the simulation ends after N steps without reaching 1.5*N. In this simulation the total simulated time is two seconds so they will occur after 0.25, 0.5 and 1 second.

In order to see the effects of the disturbances, the graphs plotted after the simulation can be used.

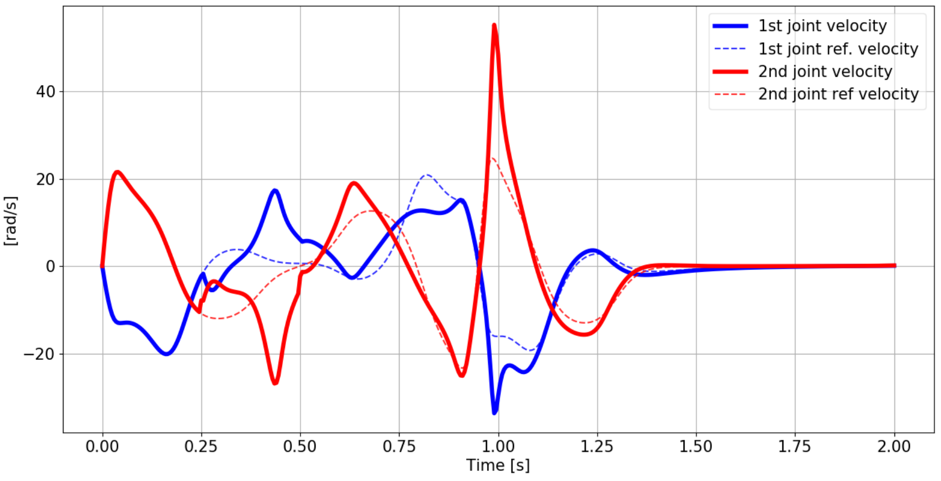


Figure 4: Joint reference velocities and the simulated ones

This first plot, Fig 4, shows the joints velocities in the whole simulated period. With respect to the previous question, Fig 3, this solution is very different because here it is evident how the robot moves away from its reference trajectory. Until 0.25 seconds, which corresponds to the N/8-th step, no external action occurs and the pendulum works as expected. Then, both the actual velocities deviate from their references and they will converge to the dotted reference function only at the end of the simulation.

The external actions act on the second joint velocities with an increment in the velocity. In all the instants they act, the red function moves upward the first and second times and slow down the falling trend at one second. The first link, blue function, is not directly affected by the external action but it is modified as a reaction coming from the other link.

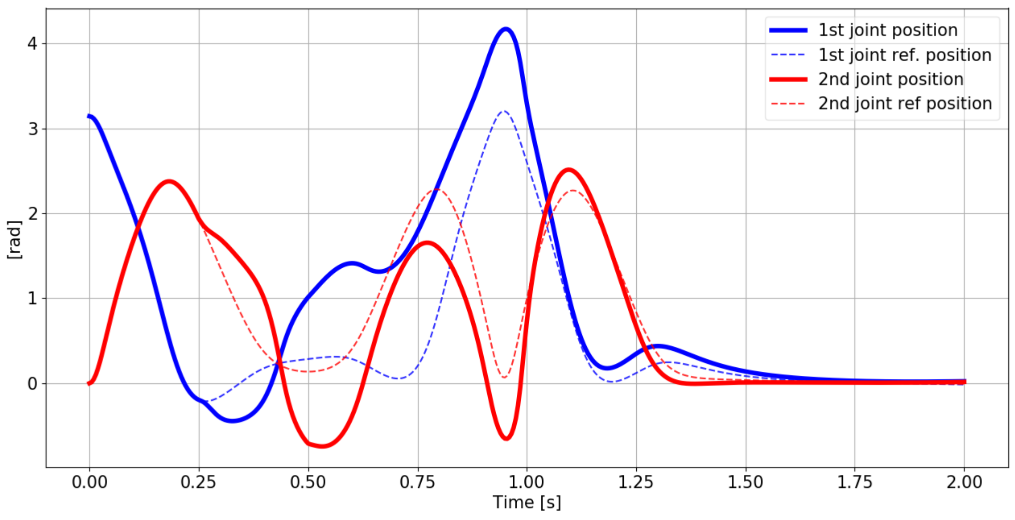


Figure 5: Joint reference angles and the simulated ones

Looking at Figure 5, it is possible to notice the simulated angles of the two joints. This result shows the same behaviour of the velocity plot with a perfect matching until the first push between the obtained ones and the references; a deviation from the desired one up to the end of the simulation with the effects of the two other external actions at 0.5 and 1 second. Differently from the velocity, the plots do not show local increments but changes in the slope. This is reasonable because the instantaneous velocity is the derivative of the position with respect to time and this definition can be observed in the plot looking at the slope of the functions.

The second joint angle is the one directly affected by the positive and instantaneous increment in velocity and that can be easily seen after 0.25 second where the function reduce its reduction because it receives the external $+3$ rad/s. This is not enough to rotate the link in the other direction because the velocity remains negative (with reference to Figure 4) despite the push. At 0.5 second the external action balances the little negative velocity that the second joint has just before that moment. Doing so, it stops the link rotation and the plot shows a little plateau in that moment. The third push occurs at

1 second when the joint is already rotating with a big positive velocity and the external action is barely visible. The blue function is not affected as the red one but it feels the actions because also its simulated behaviour deviates from the reference.

In order to study the tracking performance, a third plot can be examined. The torque of the first motor shows how the pendulum system deviates from the reference trajectory after the first external push where the motor try to counteract imposing big negative torque. Having only the motor in the first joint, the second link can not be controlled directly and the external action will destabilize the system. The motor try to stabilize the system and it reach this task in the last part of the simulation.

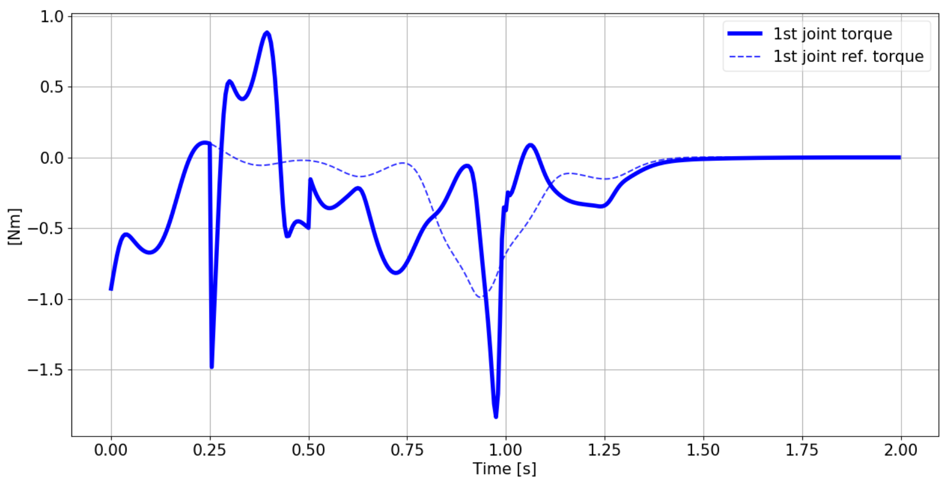


Figure 6: Joint reference torques and the simulated ones

The graph in Figure 6 has a direct relation with the blue function in Figure 4 that represents the velocity of the controlled joint. When the velocity is modified, the torque change accordingly and the external actions are very evident in Figure 6 because the torque experience the actions that went through the links.

Depending on the instant when they occur, the torque will be higher or lower. Only the first push creates a big effect on the motor because it is the first that destabilizes the system in just one action. The following two pushes have a smaller impact because the system is already trying to moving back to the desired trajectory with the motor that continuously adjusts its torque.

Question 3

In the last question it is asked to look at the algorithm. In particular, changing the value of `ddp_params['mu_factor']` changes the regularization part. In order to do so, this parameter is set to zero while all the other flags in the configuration file remain as in the previous question.

The Differential Dynamic Programming remains with a single hyperparameters for solving the problem, the α of the forward pass. Considering this limitation, the algorithm is not using the backward pass at its maximum capabilities because setting `ddp_params['mu_factor']` $= 0$ means keeping the regularization term μ constant. In this assignment is configured to be equal to 10.

The fact that μ is set constant will not bring to a surely bad situation because the resulting scenario depends also on the problem itself. For example, having large μ in some states may result in some problems for the solver because it faces a difficult function to be optimized.

If the problem has some states that do not need regularization, typical case of really simple problem, adding a μ will introduced useless quantities that slow down the process.

The general idea is to keep a reasonable initial value and, then, let the solver reduces this regularization parameter to a smaller value in order to speed up the algorithm, always maintaining the certainty of convergence.

Applying the changes to the regularization term μ, `mu_factor = 0` and μ `= 10`, there are many differences with respect to the previous simulations. First of all the simulation results are the following:

$$\text{cost} = \text{NaN} \quad \text{effort} = \text{NaN}$$

This fact leads to think that something goes wrong during the simulation.

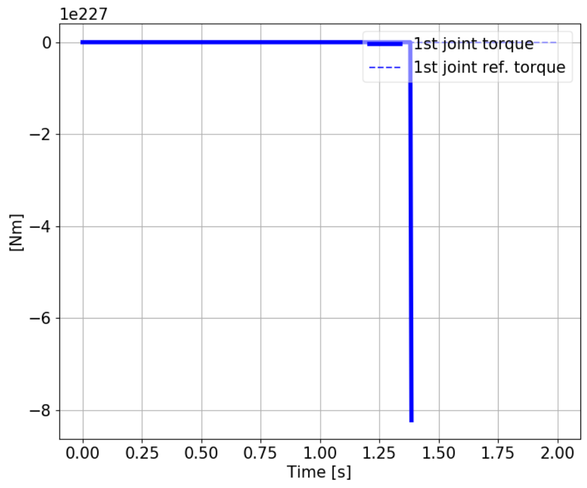


Figure 7: First joint torque representation in the case with μ constant equal to 10.

As it is possible to notice in Fig. 7, the first joint torque diverges to huge values, while, when it is possible to decrease μ, this does not happen. Investigating the reasons why keeping $\mu = 10$ this consequence is obtained, a lot of plots are carried out, to observe the behaviour of the simulator. What has been noticed is that X_sim and U_sim increase at each iteration step up to reach huge values and then NaN, since the solver can't deal with infinite numbers. Knowing that the cost depends only on X_sim and U_sim, it becomes NaN too. It is important to underline that this divergence occurs only if `PUSH = 1`: running the simulation keeping the same μ but setting `PUSH = 0`, the double pendulum converges to the optimal solution, following the reference trajectory.

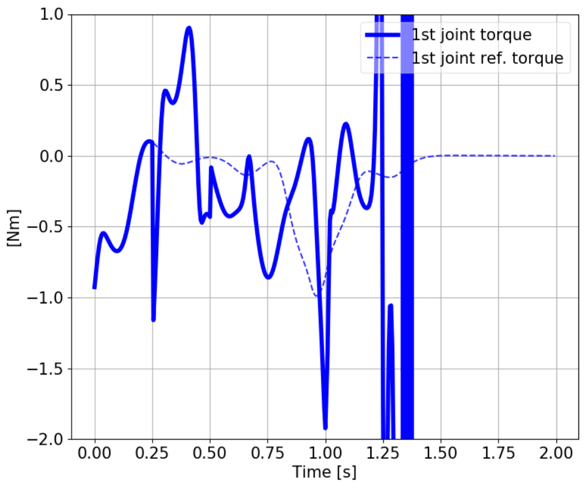


Figure 8: First joint torque representation in the case of μ constant equal to 10 and imposing bounds to observe the trend before the divergence.

The push effect is shown in the Fig. 8, where, looking only at a small range of the vertical axis, it is possible to study the behaviour of the joint torque until its divergence. A possible explanation of this phenomena is the following. From DDP theory is known that a great value of μ leads to a slower convergence therefore the number of iteration is increased. At the same time DDP algorithm works very well when it uses values that are near the reference and the push results in a great depart from that. Since the convergence is slow as it is reported, the algorithm is not able to bring back the pendulum to the right trajectory and the solution diverges a lot from the optimal one.

This justification is coherent with the DDP theoretical behaviour, but is not clear how this happen only with $\mu = 10$. Trying to find an explanation, several attempts are carried out: μ has been set equal to 9 or 11 that are values very similar to the problematic 10, and in both cases the convergence is reached.

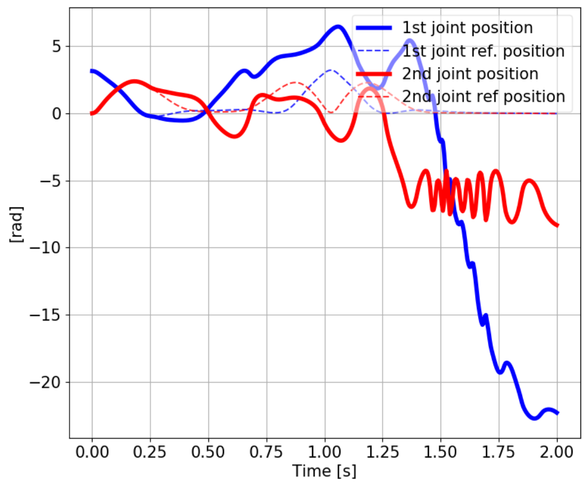


Figure 9: Joint positions setting $\mu = 100$.

For example, in Fig. 9 is reported the trend of the joint position in case of μ set constant and very high. This behaviour is coherent with the previously proposed idea: a high μ leads to a slow convergence (the number of iteration is very big) and it is possible to observe that as consequence of the push action, the solution is very different from the reference and the solver is no more able to converge, since the convergence rate is too much slow. The double pendulum doesn't perform the swing-up maneuver (because the solution diverges), high costs are reached, but they are always in a numerical form and NaN values are not presented.

Finally, after the several attempts and the reasoning reported, it isn't yet clear why setting $\mu = 10$ that bad consequence is reached. A possible explanation we have found is regarding numerical issues inside the simulator, that leads to reach huge values.