Taking account of underactuation in DDP

Assignement n° 2 of the course
Advanced Optimization-Based Robot Control

Students

Matteo Dalle Vedove
matteo.dallevedove@studenti.unitn.it
ID 232604

Alessandro Rizzardi
alessandro.rizzardi@studenti.unitn.it
ID 233221

Teachers

Andrea Del Prete
andrea.delprete@unitn.it

Gianluigi Grandesso
gianluigi.grandesso@unitn.it

Code implementation

The only thing that we are required to implement is a running cost that's used by the DDP (Differential Dynamic Programming) solver to avoid the actuation on the second joint. Having just a $\mathbb{R}^2$ vector of lagrangian coordinate q and wanting to limit the torque on the second joint, we design the selection matrix

$$S = \text{diag}(0,1) = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Given the control vector $u \in \mathbb{R}^2$, then it is clear that the matrix product Su (`u_2` in the code) provides us just the second actuation component (that we want to penalize). As explicitly required by the text, the cost due to the underactuation for each time-step is computed as

$$\frac{1}{2}\texttt{underact}\|u_2\|^2 \tag{1}$$

In practise the squared norm of u_2 can be computed as $u_2^T u_2$.

To use the DDP solver it's also necesary to compute the derivative of such cost with respect to the controls u; expanding $u_2^T u_2$ as $u^T S^T S u$, we observe that this equation is the canonical quadratic form $u^T A u$ (with $A = S^T S$ that due to the particular structure evaluates just to S) whose derivative is determined by the expression $u^T(A + A^T)$. This means that the overall derivative of (1) with respect to the inputs, considering the definition of our matrix S that's symmetrical, simplifies to

$$\frac{1}{2}\texttt{underact} \cdot u^T(2S) = \texttt{underact} \cdot u^T S$$

Lastly one more derivation of this running cost provides us the simple expression

$$\texttt{underact} \cdot S$$

```python
def cost_running(self, i, x, u):
    ''' Running cost at time step i for state x and control u '''
    S     = np.diag(np.array([0, 1]))
    u_2   = np.dot(S, u)
    cost  = 0.5*np.dot(x, np.dot(self.H_xx[i,:,:], x)) \  # already implemented
            + np.dot(self.h_x[i,:].T, x) + self.h_s[i] \  # already implemented
            + 0.5*self.lmbda*np.dot(u.T, u) \             # already implemented
            + 0.5 * self.underact * np.dot(u_2.T, u_2)
    return cost

def cost_running_u(self, i, x, u):
    ''' Gradient of the running cost w.r.t. u '''
    S     = np.diag(np.array([0, 1]))
    c_u   = self.lmbda * u \                              # already implemented
            + self.underact * np.dot(u.T, S)
    return c_u

def cost_running_uu(self, i, x, u):
    ''' Hessian of the running cost w.r.t. u '''
    S    = np.diag(np.array([0, 1]))
    c_uu = self.lmbda * np.eye(self.nu) \                 # already implemented
            + self.underact * S
    return c_uu
```

Question 1

At a first glance the solution obtained considering a selection matrix approach and a penalisation term on the second joint are equivalent. In the first case we simply use the matrix S to avoid, in both optimization and simulation, that the torque on the second joints affects the system's dynamics, while in the second case we weight the torque applied by the second joint that much ($1 \cdot 10^6$) that the solver is heavily penalised by actuating the second motor for the stabilisation.

Table 1: *minimisation results comparison between selection matrix and actuation penalty; in both cases it has been considered* `mu_factor = 10`.

SELECTION MATRIX	ACTUATION PENALTY	final cost	iterations at convergence
1	0	1481.179	33
0	1	1481.170	33

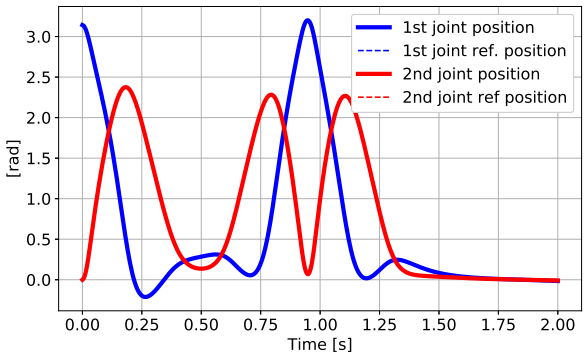
Taking a deeper look at the minimisation's result shown in table 1, also the final cost achieved by the two methods has not significant difference, with the actuation penalty slightly better with $6 \cdot 10^{-6}$ relative cost advantage. Also iterations at convergence are the same.

Even there's not significant difference, this result is counter-intuitive, as we expect that the penalised system should have behaved at best as the system with selection matrix, as slightly non-zero torques at the second joint were allowed that would have weighted a lot in the running cost.

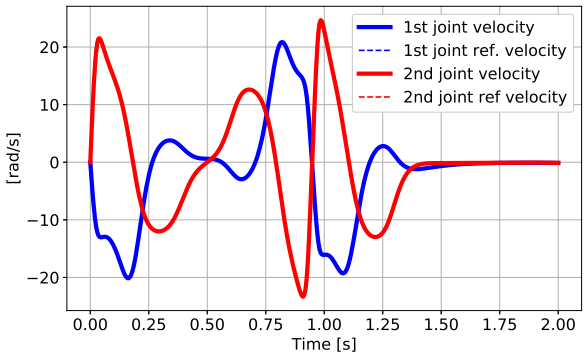
An explanation of this result can be found in light of what we discussed in class with

collocation method for solving optimal control problems. We think that in the very first iterations the solver uses some actuation on the second joint: this allows him to explore further solution space and somehow allows to find an even optimal control trajectory for the first joint; indeed DDP is a local minima solver. This is more a philosophical interpretation of the result as there's no notable difference from the two cases.

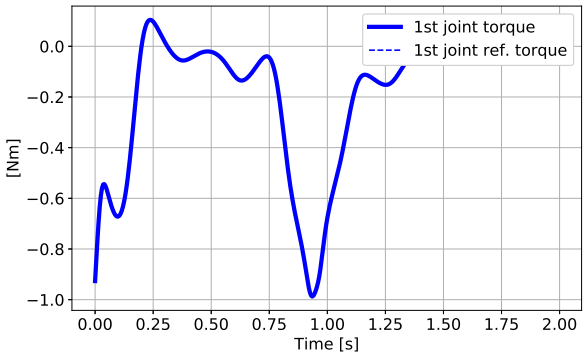
Fig. 1 shows the computed optimal trajectory computed by the DDP solver considering the selection matrix approach (that still, is not different - at a macroscopic level - from the results obtained with under-actuation whose results, for clarity, are reported at the end in fig. 4).



(a)



(b)



(c)

Figure 1: *optimal position (a), velocities (b) and joint torques (c) obtained using Differential Dynamic Programming; in the particular case, results are obtained considering the selection matrix approach.*

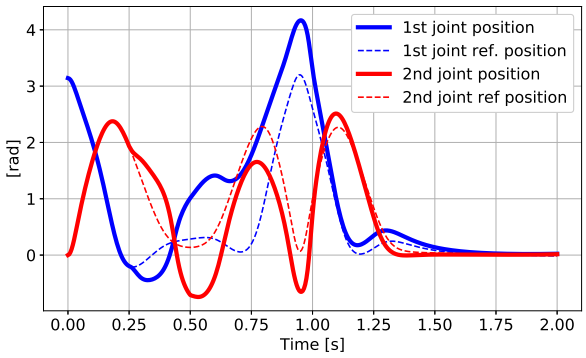
Question 2

The second task requires to set the pushing configuration that allow the simulation to add external pushes so we can obtained instantaneous increase of the second joint velocity. These pushes occur at times-samples $N/8$, $N/4$ and $N/2$ only during runtime simulation, not in DDP optimisation; fig. 2 shows the behviour of the system compared with the nominal computed trajectory of the DDP.

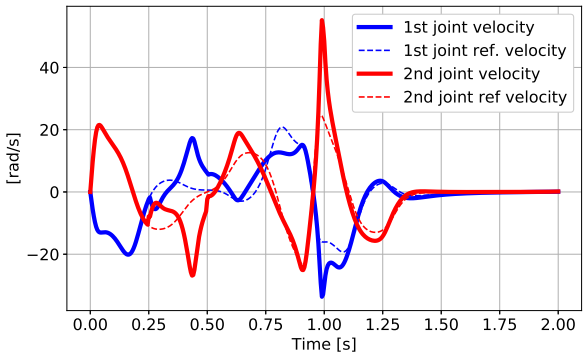
We can observe that at $N/8$ (time $t = 0.25s$) the torque's real trajectory moves away from the reference one, clearly identifying the first push: the torque in fact gets strongly distanced from it's reference motion; this heavily reflects also on the motion of the joints that get pushed apart from the references. This is probably due to the feedback gain computed for the time where the push happens that provides a strong deviation of the torques given an unbalance on the second joint velocity (where push is applied to). Obviously the push affects also position and velocities of the joints that get far away from their references. They still seems able to track the respective optimal trajectories with some offset that gradually reduces over time due to the feedback gain.

The second push happens at $N/4$ (time $t = 0.5s$) and is evidently shown in the figure with an instantaneous change of the torque(as for the previous case), while the third push, happening at $N/2$ (time $t = 1s$), does not provide a evident change in trajectories, probably because in that moment the velocity of the joints are already relatively high and the instantaneous change provided externally does not influence the joints behaviour that much. Joints coordinates seem not strongly affected by the second push; still they follow the trend of the optimal computed trajectory.

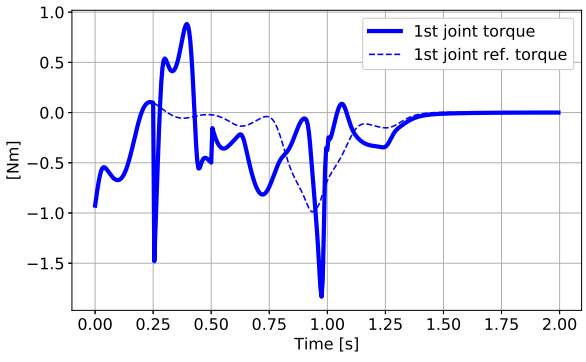
Toward the end of the simulation we have the convergence of the torque's trajectory as, while approaching the end, the DDP solver reduces the gain matrix; still, the system is able to converge to the desired configuration after $t = 1.75$, as reported in the plots. So we can state that the system is stable and able to converge resulting in a zero error in the trajectories when an external interference is added meaning



(a)



(b)



(c)

Figure 2: *position(a), velocity (b) and joint torques (c) time evolution for the optimised trajectory (dashed line) and the realisation with occurring pushes (solid line). The optimal trajectory has been obtained considering the selection matrix approach.*

Question 3

Setting the μ factor to zero provides the worse results simulation-wise where joint coordinates starts diverging from the nominal trajectories toward infinite values if we apply pushes to the system, making impossible the numerical solution of the problem (and even if the solver would have been able to catch up, results are very bad).

To somehow justify how this can happen, we should take a look at how the DDP algorithm works. As reported in the assignment's PDF, in order to compute the optimal trajectory w and feedback gains K, the algorithm is required to invert the matrix $Q_{uu} = \ell_{uu} + f_u^T V'_{xx} f_u$ for each time-stamp. To be able to perform this operation we must be sure that Q_{uu} is non-singular (or numerically *not-close* to such condition), however this cannot be stated a-priori; to limit this defective condition, we usually consider a regularised matrix

$$\overline{Q}_{uu} = Q_{uu} + \mu I$$

Intuitively, the matrix $\overline{Q}_{uu}$ is *bigger* then Q_{uu} as we expect an higher determinant. This means that the inverse of $\overline{Q}_{uu}$ should be somehow *smaller* then Q_{uu}, that in case of close-to-singular configuration would explode toward infinite values.

The regularisation factor μ comes handy for this reason, as it helps having a numerically invertible matrix, with the drawback of increasing the iterations at convergence; for this reason the value of μ is adaptively changed during the different iteration of the algorithm, in order to meet the best trade-off between convergence rate and numerical stability.

It's also intuitive that having a *big* regularisation factor μ, aside making $\overline{Q}_{uu}$ more easily invertible, heavily affects the expected behaviour of the solver, as we are no longer dealing with the linearization of the system's dynamic.

For this last test with the DDP solver, we kept constant the regularisation factor $\mu = 10$; the optimal trajectories are achieved after 37 iterations of the solver for both selection matrix and under-actuation approaches, with a 12% performance worsening with respect to the adaptive regularisation term (see table 1). If we disregard the applied pushes, both methods present costs in computation and simulation of almost equal value 1481, that are the same achieved with adaptive regularisation.

Conversely, if we apply the push during simulation, the system diverges, as show in fig. 3. In light of our previous discussion, our interpretation of such bad result comprises two issues that at each time-step are concurring in worsening the result:

- the dynamic of the linearized system with respect to the nominal trajectory is unstable, that is, given a slight deviation from the desired configuration, while compensating it generates an even higher overshoot that amplifies over time;
- once the deviation from the reference motion becomes relevant, the linearization errors also start becoming dominant (as the feedback K is associated to a linearized dynamic that is holding no more).

Furthermore, the main issue in this simulation is that no bound for the control input is set, so the vicious circle that follows from the linearization error and unstable dynamic leads to the generation of infinitely high control torque that in practise cannot be achieved.

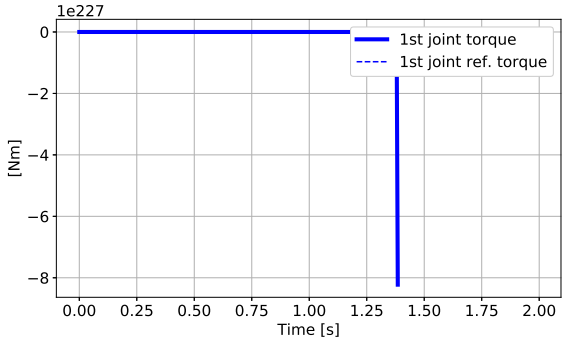
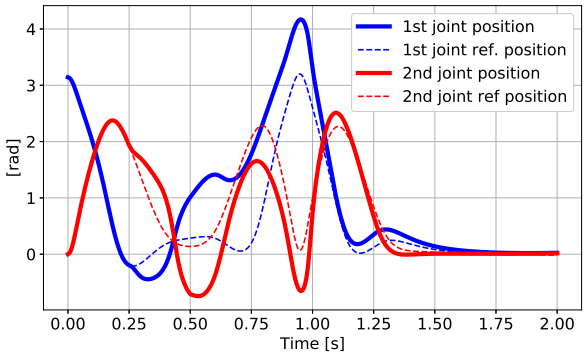
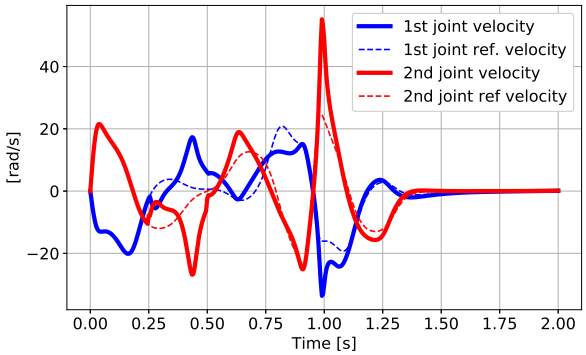


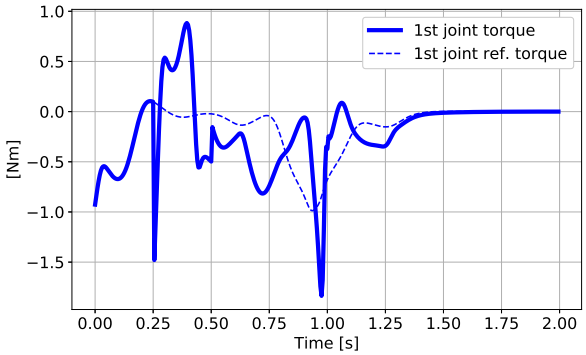
Figure 3: torque resulting from the simulation with applied pushes given a μ factor of 10; for the DDP configuration in this case it has been used the selection matrix.



(a)



(b)



(c)

Figure 4: optimal position (a), velocities (b) and joint torques (c) obtained using DDP and the under-actuation approach (with mu factor set to 10); the simulation is made considering also the applied pushes.