EXERCISE 6
Linear-quadratic regulator and iLQR

Tue Herlau
tuhe@dtu.dk

11 March, 2022

Objective: In these exercises you will implement the LQR (linear quadratic regulator). We will use this to solve non-linear control tasks in two ways, the first is by simply expanding (linearizing) the system around a single point, the next by iteratively linearizing around a path (iLQR), and finally full iLQR which applies a line-search strategy. As this can be quite a mouthful, keep in mind that the main thing to take away from the exercise is the simple LQR method, and the idea that a non-linear system can be linarized so LQR can be applied to it. (41 lines of code)
Material: Obtain exercise material from our gitlab repository at
`https://gitlab.gbar.dtu.dk/02465material/02465students`

Contents

1 Discrete LQR (`dlqr.py`)

The discrete LQR updates are defined in algorithm 22. Our first task will be to implement LQR and apply it to a very small, linear system with stationary matrices A and B and

quadratic costs Q and R.

$$A = \begin{bmatrix} 1. & 1. \\ 0. & 1. \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

and

$$Q = \begin{bmatrix} \rho^{-1} & 0 \\ 0 & 0 \end{bmatrix}, \quad R = \begin{bmatrix} 1. \end{bmatrix}$$

This is known as a double integrator[1].

We will solve this problem using LQR. Note there is no terminal cost, no constant terms, and no linear terms. In other words, you do not have to implement the update rules for:

$$\boldsymbol{S}_{u,k}, \boldsymbol{l}_k, \boldsymbol{v}_k, v_k$$

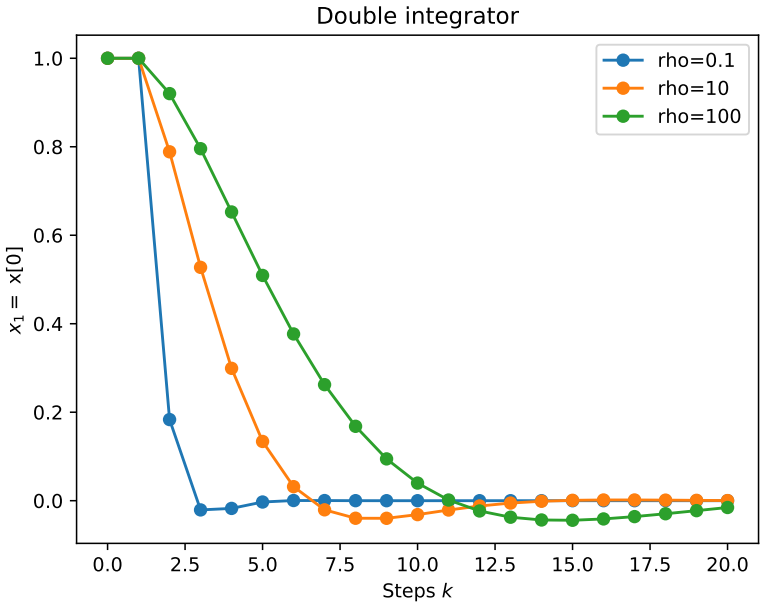
at this point, which will simplify the implementation somewhat.

Problem 1 *Implement basic LQR*

Implement the basic discrete LQR update rules for the double integrator. Pay close attention to the update rules; note that since we are progressing backwards in time check your implementation by looking at the last values of L, V, i.e. `L[N-1]`, etc.

[1]The example is taken from `http://cse.lab.imtlucca.it/~bemporad/teaching/ac/pdf/AC2-04-LQR-Kalman.pdf`, but see also `https://en.wikipedia.org/wiki/Double_integrator`.

Info: The code should produce the following plot:



For debugging purpose check you get this exact input (and note the first place of divergence carefully)

```
L[19] is: [[-0. -0.]]
L[18] is: [[-0.0099 -0.0198]]
L[0] is: [[-0.0799 -0.4415]]
```

Problem 2 *Intuition checkup*

Look at the wikipedia article `https://en.wikipedia.org/wiki/Double_integrator`. Write out the cost function as a function of ρ. Why does the curves change? what is the controller trying to do with the states, and why do the curves depend on ρ the way they do? As a bonus, check the slides the example is taken from found here `http://cse.lab.imtlucca.it/~bemporad/teaching/ac/pdf/AC2-04-LQR-Kalman.pdf`. Our solutions is different, albeit very slightly. Why?

1.1 Implement full discrete LQR (`dlqr_check.py`)

So far so good, but LQR works with non-stationary matrices and this will important for iterative LQR in a moment. Therefore, it is time to implement the full LQR update (see algorithm 22). I have included a test for a simple, nonsense system which only operates over a few time steps. Make sure you get the same output, and otherwise notice the first step divergence happens.

Problem 3 *Implement full LQR*

Implement the full LQR algorithm, updating all terms. Carefully check the output below.

Info: For debugging purpose check you get this exact input (and note the first place of divergence carefully)

```
l[3]=[-9.5229  2.1249], l[2]=[-6.8482  4.8587], l[0]=[ 1.0649 -1.2575]
L[3]=[[ 0.9364  1.0472  0.5745]
 [-1.6524  0.9154  0.5854]]
L[2]=[[ 6.8717  0.2774  1.3766]
 [-7.6789 -0.6746 -0.9357]]
L[0]=[[-2.0035 -3.7411 -1.5262]
 [-0.135   3.8597  1.9371]]
V[0]=[[ 0.1544  5.2035  1.7752]
 [ 5.2035 -4.6053 -2.892 ]
 [ 1.7752 -2.892   0.2216]]
v[4]=[-0.7731  0.8494  0.7547], v[3]=[-4.2712  3.3003  3.9519], v[0]=[-1.7517
↪  0.9697  1.7765]
vc[4]=-0.4841, vc[3]=3.0585, vc[0]=1.8368
```

1.2 The Boing level flight example (`boing_lqr.py` and `lqr_agent.py`)

This problem will turn the basic LQR algorithm into an agent. One reason for doing this is simply convenience, however more importantly, it will allow us to automatically apply exponential integration which we implemented two weeks ago.

Recall once again the double-integrator system [Her21, eq. (14.16)]

$$\dot{\boldsymbol{x}}(t) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \boldsymbol{u}(t). \tag{1}$$

When we apply our discretization method described in section 11.2.6 this becomes a discrete update rule of the form:

$$\boldsymbol{x}_{k+1} = A\boldsymbol{x}_k + B\boldsymbol{u}_k = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{u}_k). \tag{2}$$

The matrices A, B in the above should be calculated using exponential discretization since the system is linear. What we then simply want is an agent which, when given a discrete model (of a stationary linear quadratic model), figures out what the A and B matrices are, apply LQR update rules to obtain control matrices L_k and l_k and then computes actions at time t by computing $k = \lfloor t/\Delta \rfloor$ and

$$\boldsymbol{u}_{k+1} = L_k \boldsymbol{x}_k + \boldsymbol{l}_k.$$

Note that in the linear case, if we have $\boldsymbol{f}_k$, we can recover A and B using the Jacobian:

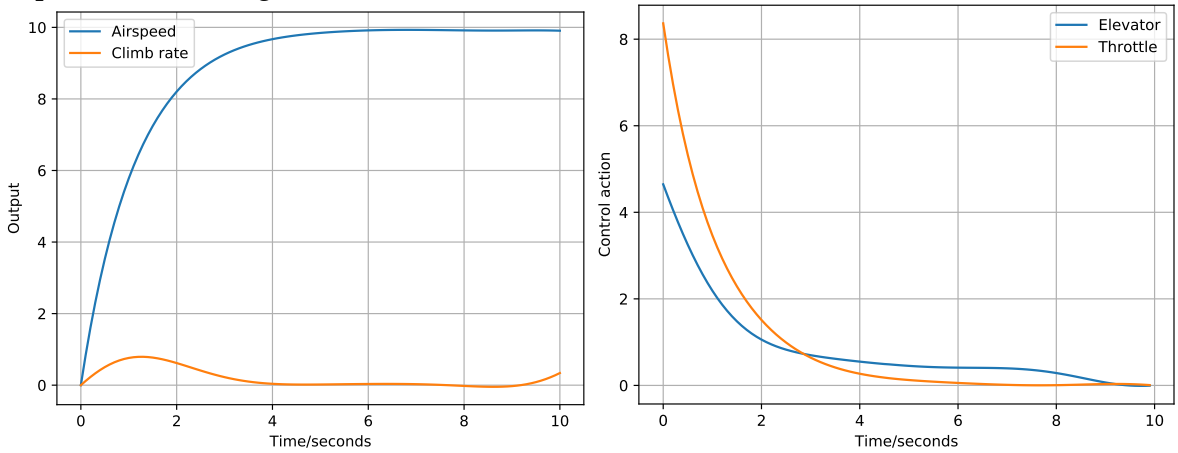
$$A = \boldsymbol{J}_{\boldsymbol{x}} \boldsymbol{f}_k(\boldsymbol{x}_0, \boldsymbol{u}_0), \quad B = \boldsymbol{J}_{\boldsymbol{u}} \boldsymbol{f}_k(\boldsymbol{x}_0, \boldsymbol{u}_0).$$

This is perhaps particularly And as it happens, the Jacobian is already implemented using sympy; we will for now only use it as a black-box method, and consider the implementation details in the next section.

Problem 4 *Boing example and the LQR Agent*

Implement a basic LQR agent and apply it to the Boing example. The implementation is fairly bare-bones, i.e. we do not take all potential terms in the cost function into account, but is sufficient for the Boing example and the double integrator. You should compute A and B using the build-in functionality for computing Jacobians, see hints in code.

Info: The examples which illustrates the difference between exponential integration and Euler integration in the notes are build using the code you just implemented and can be re-produced by changing which integration method the discrete model uses. This script uses exponential integration, and reproduce the Boing flight-change example. You should get:



2 Linearly approximating a system using the Jacobian (`linearization_agent.py`)

As discussed in the lectures, a simple idea is to linearly approximate the system and then solve the linear system using LQR. This corresponds to the first algorithmic idea for iterative LQR (with constant A, B matrices) discussed in the slides.

Since we are using the Jacobians extensively, let's briefly review how they are computed automatically using sympy (see `https://docs.sympy.org/1.5.1/tutorial/calculus.html`). The following code sets up a function `model.f_z(x, u)` which computes the derivatives with respect to all variables $\boldsymbol{z} = (\boldsymbol{x}, \boldsymbol{u})$:

```
# continuous_time_discretized_model.py
        u = symv("u", len(ud))
        x = symv('x', len(xd))
        """ x_next is a symbolic variable representing x_{k+1} = f_k(x_k, u_k) """
        x_next = self.f_discrete_sym(x, u, dt=dt)

        """ compute the symbolic derivate of x_next wrt. z = (x,u): d x_{k+1}/dz """
        dy_dz = sym.Matrix([[sym.diff(f, zi) for zi in list(x) + list(u)] for f in
        ↪  x_next])
        """ Define (numpy) functions giving next state and the derivatives """
        self.f_z = sym.lambdify((tuple(x), tuple(u)), dy_dz, modules=sympy_modules_)
```

Then later, this functionality is used to compute the Jacobians in a more convenient format, namely the function `model.f`:

```
# continuous_time_discretized_model.py
    def f(self, x, u, k, compute_jacobian=False, compute_hessian=False):
        """
        By defult this functions returns f_k(x,u).

        If compute_jacobian=True it will return the derivatives as well:
        > f_k(x,u), df_k(x,u)/dx, df_k(x,u)/du.
        Code currently contains a stub for computing Hessians, but this is not
↪   implemented at the moment.
        """
        fx = np.asarray( self.f_discrete(x, u) )
        if compute_jacobian:
            J = self.f_z(x, u)
            if compute_hessian:
                raise Exception("Not implemented")
                f_xx, f_ux, f_uu = None,None,None  # Not implemented.
                return fx, J[:, :self.state_size], J[:, self.state_size:], f_xx,
                ↪  f_ux, f_uu
            else:
                return fx, J[:, :self.state_size], J[:, self.state_size:]
        else:
            return fx
```

Pay attention to how, if this function is called with `compute_jacobian=True`, it will return $\boldsymbol{f}_k(\boldsymbol{x}, \boldsymbol{u})$ and the Jacobians $\boldsymbol{J}_{\boldsymbol{x}}\boldsymbol{f}_k(\boldsymbol{x}, \boldsymbol{u})$ and $\boldsymbol{J}_{\boldsymbol{u}}\boldsymbol{f}_k(\boldsymbol{x}, \boldsymbol{u})$.

In this problem, you will implement [Her21, algorithm 23] and it will be applied to the cartpole task. The cartpole will be initialized slightly out of balance, similar to the PID cartpole example, and our job is to bring it in balance. We make the following assumptions/simplifications:

- We expand around the upright position $\bar{\boldsymbol{x}}$, and no action $\bar{\boldsymbol{u}} = 0$.

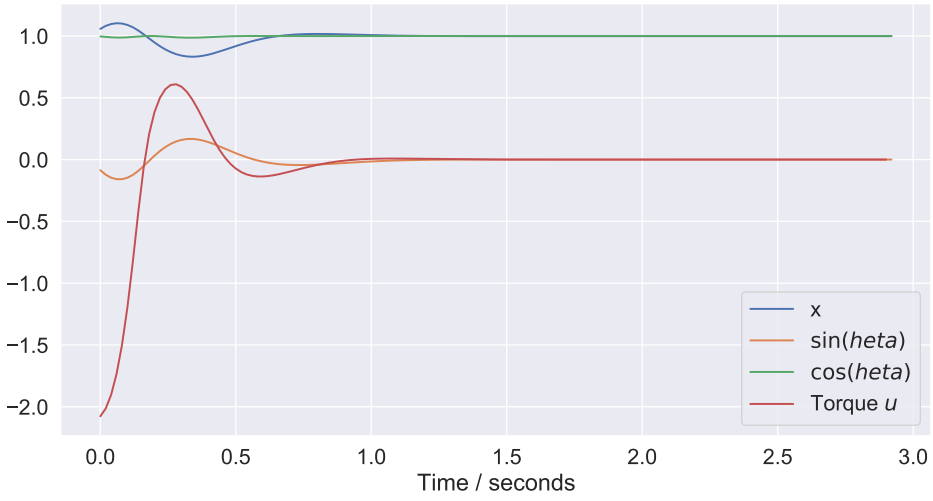
- We just fix the planning horizon to $N = 30$
- We only use the first control matrix/vector L_0, l_0 at all subsequent time steps

The later two points are reasonable since the problem is stationary.

Problem 5 *Implement linearization procedure*

Complete the missing code for the Cartpole system and check it can balance the cart. You can experiment with the no-control period to create more challenging problems. Since we use the agent/environment system, the LQR solution is simulated on a realistic system.

Info: The script should produce the following output, which shows a few of the coordinates of the simulated solution:



3 Iterative LQR (`ilqr.py`) ✸

The previous idea had the drawback that we linearized around a single point, and assumed that model was good enough to derive a controller for the entire trajectory. Obviously, once the trajectory begins to depart from that point, so will the controller, and the method will not be stable.

3.1 Basic iLQR (`ilqr_rendovouz_basic.py`)

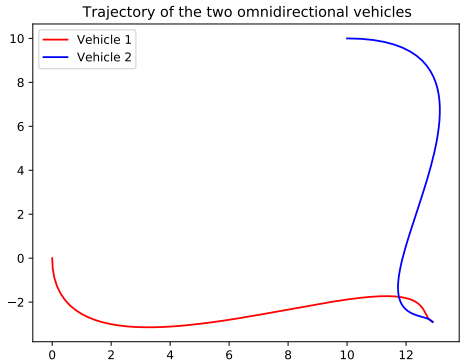
Iterative LQR attempts to overcome the aforementioned problem by first selecting an action sequence u_k, simulating a rollout x_k by applying u_k to the system, linearize the system around (u_k, x_k) to get matrices A_k, B_k, etc., and then creating an optimal control sequence for the linearised system by using discrete LQR, pseudo-code is given in [Her21, algorithm 24].

We will test the procedure on the rendevouz problem, where two vehicles has to fly towards and meet each other from given initial conditions.

Problem 6 *Implement linearizion procedure*

Implement the basic iLQR procedure defined as the function `ilqr_basic` in `ilqr.py`, i.e. make the script `ilqr_rendovouz_basic.py` run.

Info: Once completed, the paths of the vehicles should look as so:



We also compute the cost function along the paths. Your results may differ very slightly due to initialization, but should be similar to:

```
0> J=1.55665e+06, change in cost since last iteration    0
1> J=12295, change in cost since last iteration -1.54436e+06
2> J=12294.2, change in cost since last iteration -0.78954
3> J=12294.2, change in cost since last iteration -0.000176979
4> J=12294.2, change in cost since last iteration -5.54464e-08
5> J=12294.2, change in cost since last iteration -1.27329e-11
6> J=12294.2, change in cost since last iteration -3.63798e-12
7> J=12294.2, change in cost since last iteration -3.63798e-12
8> J=12294.2, change in cost since last iteration 1.27329e-11
9> J=12294.2, change in cost since last iteration 1.81899e-12
```

3.2 Linesearch iLQR (`ilqr_rendovouz.py`) ★★

Basic iLQR fails for most problems because the optimization problem is difficult and the linearizion procedure is only good in a small region around the expansion point. These problems can be somewhat overcome using linesearch, where we decrease the controller updates if they fail to find an improved solution (i.e., if the controller generates a higher cost, we search for policies with paths closer to the expansion point). This is done using a linesearch procedure exactly as discussed in [TET12], but with a single modification as documented in the source. We will test the procedure on the rendevouz environment where you should obtain identical outcomes.

Problem 7 *Implement linearizion procedure*

Implement the complete iLQR procedure in `ilqr.py` and ensure the script `ilqr_rendovouz.py` runs.

Info: As basic iLQR could solve the problem, we do not expect iLQR with linesearch will provide an improved solution; just ensure you get the same results as the basic scripts and you should be all set.

3.3 Exploring basic and improved linesearch (`ilqr_pendulum.py`)

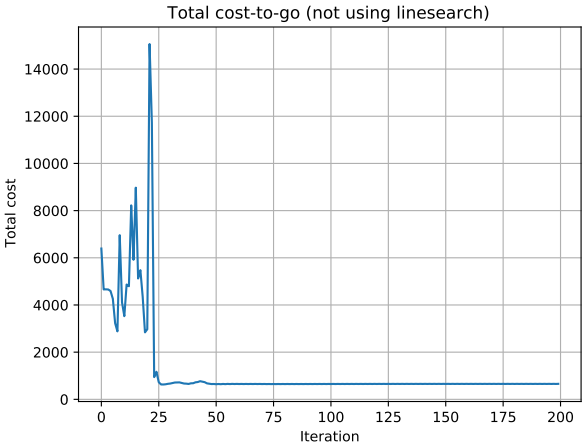
To test whether linesearch offers an improvement on the basic iLQR we will try the method on a more challenging problem, the inverted pendulum. The task is to swing up a pendulum which is originally hanging downwards. We will run the script in both settings and verify your implementation.

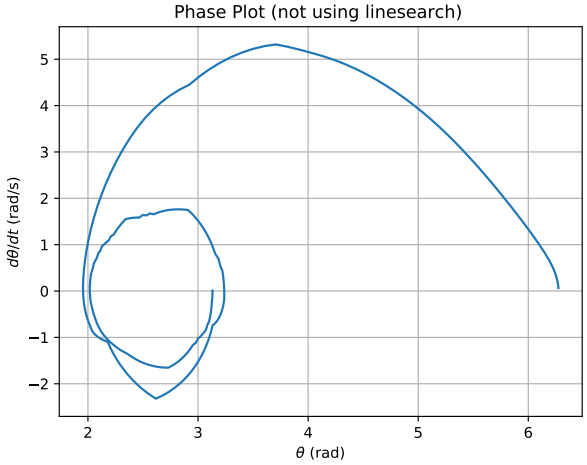
Problem 8 *Linesearch with the pendulum task*

Complete the script and test iLQR with linesearch on the inverted pendulum problem. Make sure it can swing up the pendulum. Set `render = True` in the code to see a small movie of what the controller does on the discretized environment. You can also experiment with the cartpole environment. Discuss what is shown in the plots.

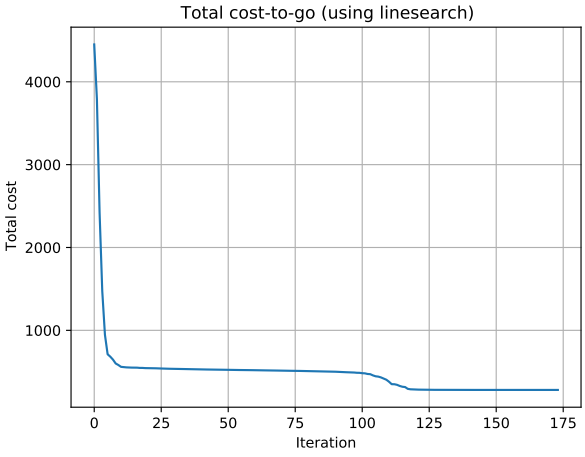
Discuss in the group whether this prove we can solve the pendulum-environment in this setup.

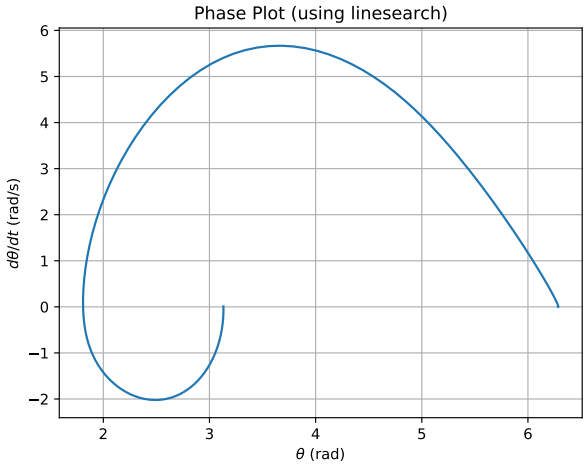
Info: The script runs both with and without linesearch, and in this case there should be a very noticeable difference. Without linesearch we get the following cost-to-go and phaseplot





This is obviously nonsense, and just shows the method has failed to converge to the up-right position. With linesearch things looks smoother:





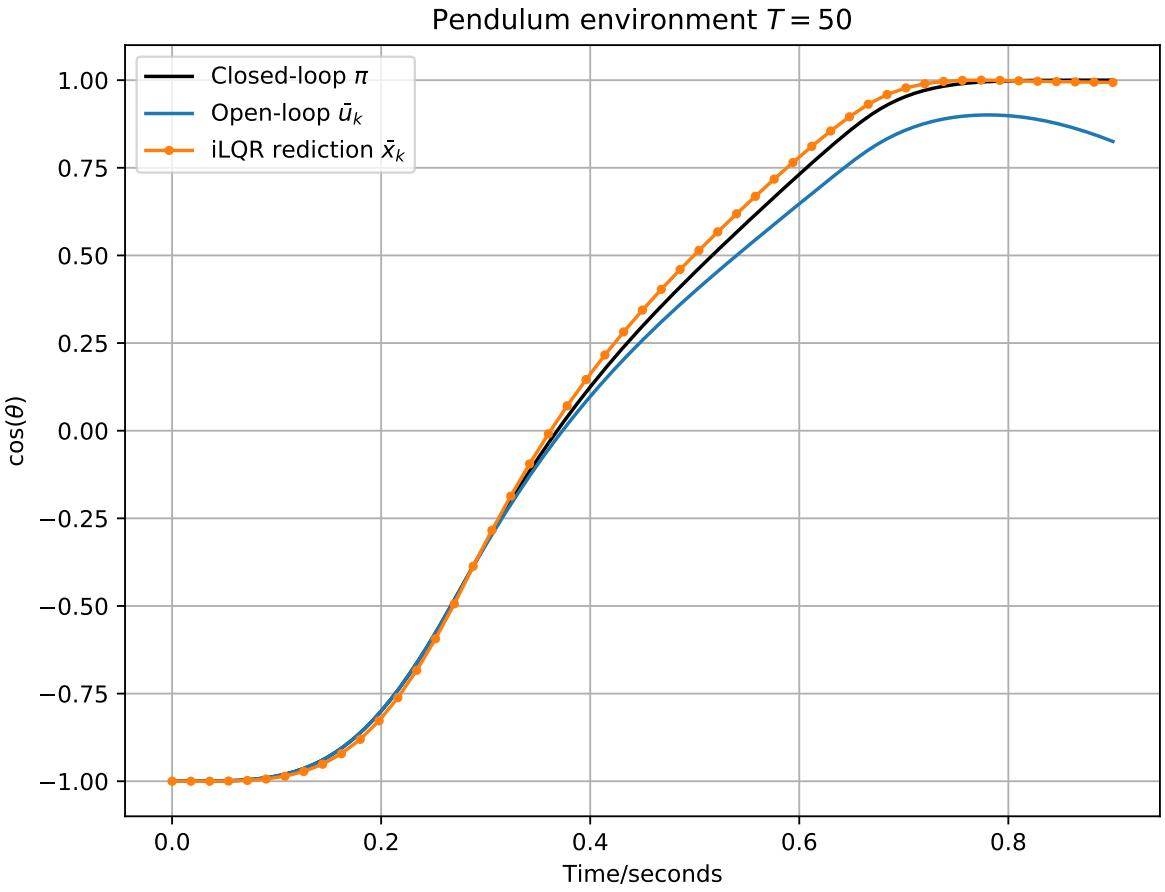
4 An iLQR agent (`ilqr_cartpole_agent.py`, `ilqr_agent.py`) ★

The big issue with the previous assignment is that we only visualized the iLQR predictions, and not the outcome of an actual simulation. We will fix this using an iLQR agent, which simply returns actions based on the output of the iLQR method. Note there are two methods for generating output actions: Either simply return $\bar{\boldsymbol{u}}_k$, or use [Her21, eq. (15.17)], and the script will implement both ideas and show a small animation. The computed trajectory is illustrated below:

Problem 9 *iLQR and the cartpole task*

Complete the implementation of the iLQR agent and check the outcome when applied to the cartpole problem. The method appears able to solve it in a stable manner.

Info: The outcome of the script is shown below, using both ways to compute the action.



References

[Her21] Tue Herlau. Sequential decision making. (See **02465_Notes.pdf**), 2021.

[TET12] 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. IEEE, 2012. (See **tassa2012.pdf**).