

EXERCISE 6

Linear-quadratic problems in control

Tue Herlau
tuhe@dtu.dk

8 March, 2024

Objective: In these exercises you will implement the LQR (linear quadratic regulator), and apply it to a variety of problems to archive optimal control for discrete linear-quadratic problems. (18 lines of code)

Exercise code: <https://lab.compute.dtu.dk/02465material/02465students.git>

Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex06.html

Contents

1	Exam question: DP and LQR	1
2	Discrete LQR (dlqr.py)	2
2.1	Implement full discrete LQR (dlqr_check.py)	3
2.2	Implement an LQR Agent (lqr_agent.py)	4
3	The Boeing level flight example (boeing_lqr.py)★	5
4	LQR and PID Control lqr_pid.py ★	7
4.1	Part 2: Set parameters in the PID controller	8

1 Exam question: DP and LQR



Consider the dynamical programming setting where we plan over a horizon $N > 0$. We consider a problem where:

- The terminal cost function is

$$g_N(x_N) = x_N^2$$

- For all $k = 0, \dots, N - 1$ the dynamics is $f_k(x, u, w) = ax + u - \lambda w$
- The noise disturbances are normally distributed with variance σ^2 :


$$P_W(w|x, u) = \mathcal{N}(w | \mu = 0, \sigma^2 = 1)$$

- The states and actions are real numbers $\mathcal{S}_k = \mathcal{A}_k(x_k) = \mathbb{R}$.
- The non-terminal costs are only affected by u , $g_k(x, u, w) = u^2$.

Thus, the relevant parameters of the problem are a and λ . We are concerned with optimal control.

- (a.) Although the problem has been formulated as being about dynamical programming, note that the structure of the problem is that of a 1-dimensional LQR model. In the case where $\lambda = 3$, derive the expected future cost $\mathbb{E}[g_N(x_N)|x_{N-1} = 0, u_{N-1} = 1]$ if we at time step $k = N - 1$ are in state $x_{N-1} = 0$ and take action $u = 1$
- (b.) Derive an analytical expression for the optimal policy $\mu_k(x_k)$ in time step $k = N - 1$

2 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

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

This is known as a double integrator¹.

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:

$$S_{u,k}, l_k, v_k, v_k$$

at this point, which will simplify the implementation somewhat.

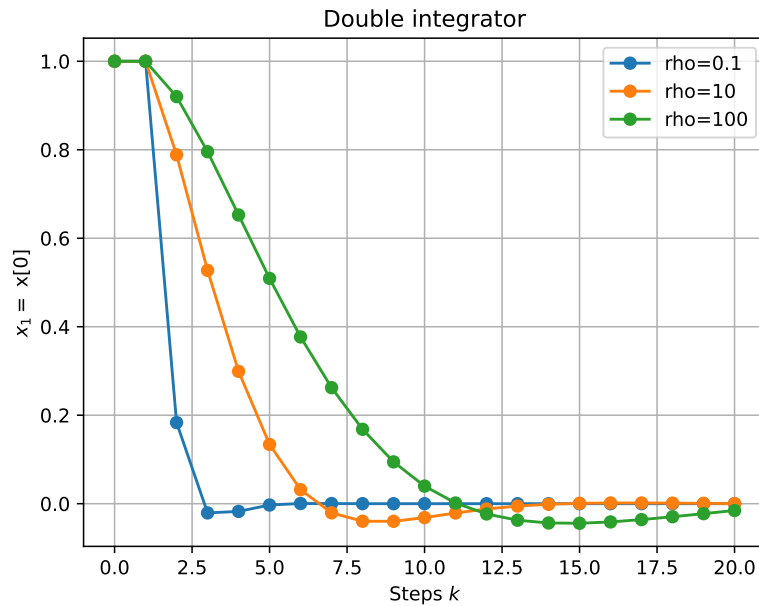
Problem 1 Implement basic LQR

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.

¹The 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)

```

1 L[19] is: [[-0. -0.]]
2 L[18] is: [[-0.0099 -0.0198]]
3 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?

2.1 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.

i

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

```

1  l[3]=[-9.5229  2.1249], l[2]=[-6.8482  4.8587], l[0]=[ 1.0649 -1.2575]
2  L[3]=[[ 0.9364  1.0472  0.5745]
3      [-1.6524  0.9154  0.5854]]
4  L[2]=[[ 6.8717  0.2774  1.3766]
5      [-7.6789 -0.6746 -0.9357]]
6  L[0]=[[ -2.0035 -3.7411 -1.5262]
7      [-0.135   3.8597  1.9371]]
8  V[0]=[[ 0.1544  5.2035  1.7752]
9      [ 5.2035 -4.6053 -2.892 ]
10     [ 1.7752 -2.892   0.2216]]
11 v[4]=[-0.7731  0.8494  0.7547], v[3]=[-4.2712  3.3003  3.9519], v[0]=[-1.7517
12     ↪ 0.9697  1.7765]
    vc[4]=-0.4841, vc[3]=3.0585, vc[0]=1.8368

```

2.2 Implement an LQR Agent (lqr_agent.py)

Now that we have implemented discrete LQR, an obvious next step is to implement an Agent that plan and takes actions using LQR. We will restrict ourselves to problems of the form:

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_k + \mathbf{d} \quad (1a)$$

$$\text{cost} = \sum_{k=1}^{N-1} \left[\frac{1}{2} \mathbf{x}_k^\top Q \mathbf{x}_k + \frac{1}{2} \mathbf{u}_k^\top R \mathbf{u}_k + \mathbf{q}^\top \mathbf{x}_k \right] \quad (1b)$$

Your task is to build an Agent that accepts A , B , \mathbf{d} and Q , R , \mathbf{q} as inputs, plug them into the discrete LQR planning algorithm to get the control matrices $(L_k, \mathbf{l}_k)_{k=0}^{N-1}$, and then compute the policy using:

$$\mathbf{u}_k = L_k \mathbf{x}_k + \mathbf{l}_k \quad (2)$$



Figure 1: The locomotive environment

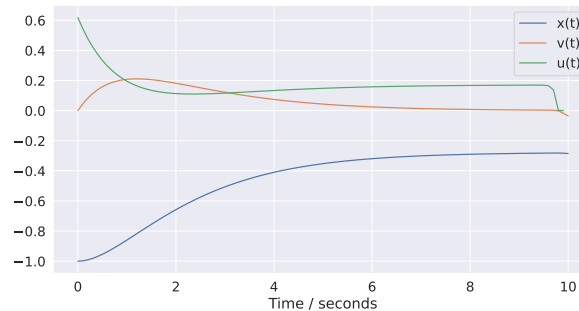
Problem 4 Implement the LQR agent

Complete the code for the LQR agent. The agent should call the LQR function you just implemented and compute the actions in the policy using eq. (2).

The agent is subsequently applied to the locomotive environment using a coarse guess on the system matrices A, B, \dots – we will return to where they come from in the next exercise.



Info: When done, the actions and states should be have as follows:



3 The Boeing level flight example (boeing_lqr.py)★

In this problem, we will apply discrete LQR, which we looked in the previous question in section 2.2, to the Boeing level flight example.

Recall once again that the dynamics in the Boeing flight example has the form:

$$\dot{\mathbf{x}}(t) = A^{(c)} \mathbf{x}(t) + B^{(c)} \mathbf{u}(t) + \mathbf{d}^{(c)} \quad (3)$$

$$\text{cost} = \int_{t_0}^{t_F} \left[\frac{1}{2} \mathbf{x}(t)^\top Q^{(c)} \mathbf{x}(t) + \frac{1}{2} \mathbf{u}(t)^\top R^{(c)} \mathbf{u}(t) + \mathbf{q}^{(c)\top} \mathbf{x}(t) \right] dt \quad (4)$$

When we apply our discretization method described in section 13.1.6 this becomes a discrete update rule of the form:

$$\mathbf{x}_{k+1} = A \mathbf{x}_k + B \mathbf{u}_k = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) \quad (5)$$

$$\text{cost} = \sum_{k=0}^{N-1} \left[\frac{1}{2} \mathbf{x}_k^\top Q \mathbf{x}_k + \frac{1}{2} \mathbf{u}_k^\top R \mathbf{u}_k + \mathbf{q}^\top \mathbf{x}_k \right] \quad (6)$$

The matrices A , B , d in the above should be calculated using exponential discretization since the system from the matrices $A^{(c)}$, $B^{(c)}$ and $d^{(c)}$ since the system is linear, and this will be our first task:

Problem 5 *Discretized the system using exponential discretization*

Complete the implementation of `compute_A_B_d` and `compute_Q_R_q`. These functions are given a control model and a discretization time and should use exponential discretization to compute the system matrices A , B , d and Q , R , and q for the discrete problem eq. (5).



Info: This example shows what the q -vector should look like. We can call the discretization function as:

```

1 # boeing_lqr.py
2 dt = env.dt
3 Q, R, q = compute_Q_R_q(model, dt)
4 print("Discretization time is", dt)
5 print("Original q-vector was:", model.get_cost().q)
6 print("Discretized version is:", q)

```

and we should get:

```

1 Discretization time is 0.1
2 Original q-vector was: [-100.  0.  0.  0.]
3 Discretized version is: [-10.  0.  0.  0.]

```

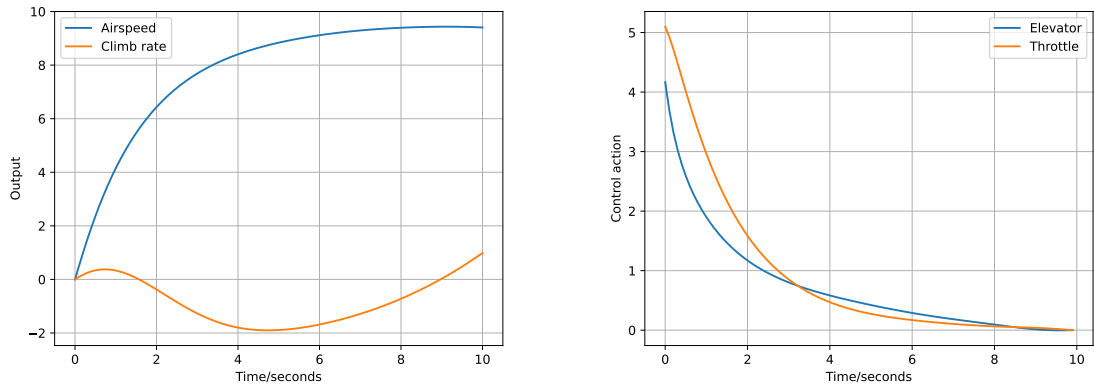
After you are done, you can call the `LQRAgent`, which we considered in section 2.2, to solve the optimal trajectory.

Problem 6 *Boing example and the LQR Agent*

Use the LQR agent and apply it to the Boing example. You need to use the two helper functions you defined earlier and call the LQR Agent as in the previous problem.



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 Boeing flight-change example. You should get:



4 LQR and PID Control `lqr_pid.py` ★

This example will explore how LQR can be used to set the K_p and K_d parameters in a PID controller. We will consider the Harmonic Oscillator described in [Her24, section 10.4.2] and show in fig. 2 Recall that the state consist of the position and velocity and we assume

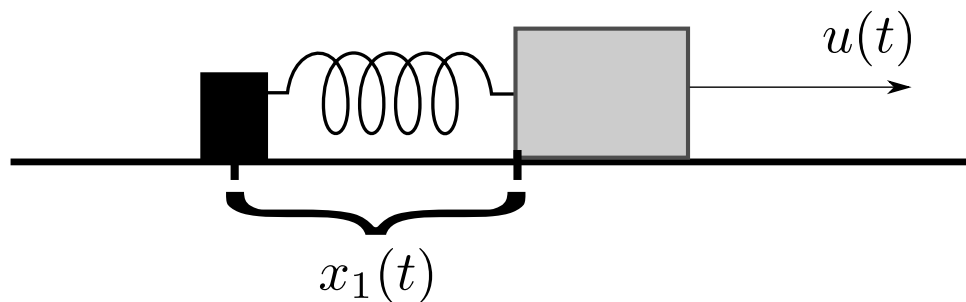


Figure 2: The harmonic oscillator. A frictionless ball is attached to a spring and can move back-and-forth. The ball is described by the position $x(t)$ and velocity $\dot{x}(t)$

the target is $x^* = 0$. The code will apply your `LQRAgent` to this problem after discretizing it appropriate and get the optimal control trajectory. Our job is to see if we can reproduce this trajectory using PID.

Recall that PID is a control law of the form ($e = x^* - x_k$)

$$u_k = K_p e + K_d \frac{e - e^{\text{prev}}}{\Delta}$$

The point is that the control law is the same for all time steps, which is different than LQR. Your first task is therefore to implement an `Agent` which behaves exactly the same

way as the `LQRAgent` considered earlier, but which always use the first control matrix, i.e.:

$$\mathbf{u}_k = L_0 \mathbf{x}_k + \mathbf{l}_0 \quad (7)$$

Problem 7 Implement the constant LQR Agent

Implement a variant of the `LQRAgent`, dubbed `ConstantLQRAgent`, which implement the control law in eq. (7). You only need to change the policy-function `def pi`.



Info: You only need to change the policy and can use that the LQR Agent already compute and store L_k and \mathbf{l}_k for all k . When done you should get this output:

```

1 # lqr_pid.py
2 x0, _ = env.reset()
3 print(f"Initial state is {x0=}")
4 print(f"Action at time step k=0 {constant_agent.pi(x0, k=0)=}")
5 print(f"Action at time step k=5 (should be the same) {constant_agent.pi(x0,
  ↪ k=0)=}")

```

and we should get:

```

1 Initial state is x0=array([1, 0])
2 Action at time step k=0 constant_agent.pi(x0, k=0)=array([-0.03309495])
3 Action at time step k=5 (should be the same) constant_agent.pi(x0,
  ↪ k=0)=array([-0.03309495])

```

4.1 Part 2: Set parameters in the PID controller

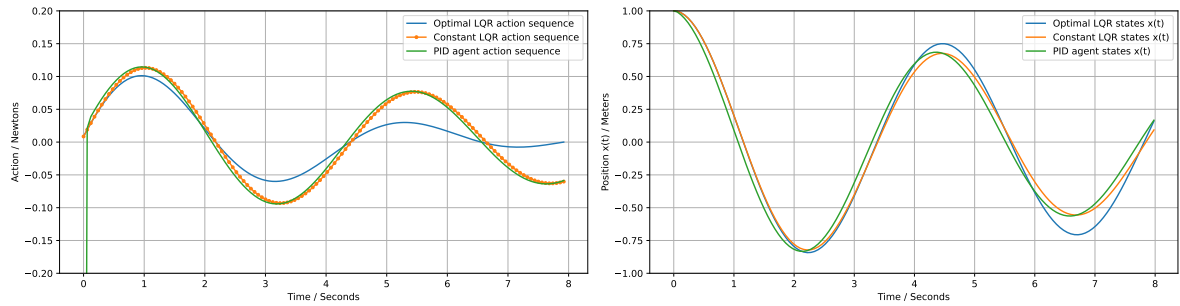
The way a PID controller select actions, and the way an infinite-horizon optimal LQR controller does so is in fact very similar if you just write them up as function $u_k = \dots$. In other words, the control matrices L_0, \mathbf{l}_0 from LQR control can be used to specify the constants K_P and K_D in a PID controller, so that they are *nearly* the same.

Problem 8 PID corresponding to optimal controller

Implement the function `get_Kp_Kd(L)`. Using the above observation, specify the parameters K_P, K_D in a PID controller as suggested by L_0 . The code will plot the PID action sequence. Can you account for the initial strange-looking action from the PID controller?

**Info:**

When done, the code will set up a PID controller using your parameters and run it. This will give you three action sequences and three state trajectories which are plotted below:



Although the PID controller computes an unusual first action, it very nearly tracks the constant LQR controller, which in turn is quite close to the optimal LQR controller.

References

[Her24] Tue Herlau. Sequential decision making. (Freely available online), 2024.