EXERCISE 7
Linearization and iterative LQR

Tue Herlau
tuhe@dtu.dk

15 March, 2024

Objective:
Today's exercises will continue our exploration of 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. (37 lines of code)
Exercise code: `https://lab.compute.dtu.dk/02465material/02465students.git`
Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex07.html

Contents

1 Exam question: Linearization of a simple problem **1**

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

3 Iterative LQR (`ilqr.py`) ✸ **3**
3.1 Basic iLQR (`ilqr_rendovouz_basic.py`) 3
3.2 Linesearch iLQR (`ilqr_rendovouz.py`) ✸✸ 4
3.3 Exploring basic and improved linesearch (`ilqr_pendulum.py`) ✸ . . . 5

4 An iLQR agent (`ilqr_cartpole_agent.py`, `ilqr_agent.py`) ✸ **6**

1 Exam question: Linearization of a simple problem

Consider a control problem where a control signal $u(t) \in \mathbb{R}$ is applied to control a system with state $x(t) \in \mathbb{R}$, and where the dynamics satisfy the following differential equation:

$$\dot{x} = f(x, u) = 4ux \tag{1}$$

The first two questions will assume the problem has been discretized using a time constant of $\Delta = 0.5$ to yield states $x_0, x_1, x_2, \cdots$ and control signals $u_0, u_1, u_2, \dots$.

(a.) Assume the problem is Euler discretized. Determine the discrete dynamics f_k used to compute $x_{k+1} = f_k(x_k, u_k)$.

(b.) Continuing the previous problem, suppose we wish to apply a LQR controller to control the system near a state $\bar{x}$. The system is therefore linearized around $\bar{x}$ and $\bar{u} = 1$ to give rise to the linearized dynamics $x_{k+1} = Ax_k + Bu_k + d$. Determine A, B and d in terms of $\bar{x}$.

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. Please look at the online documentation for more information on how to compute the Jacobians that are required for solving this problem.

You will implement [Her24, 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:

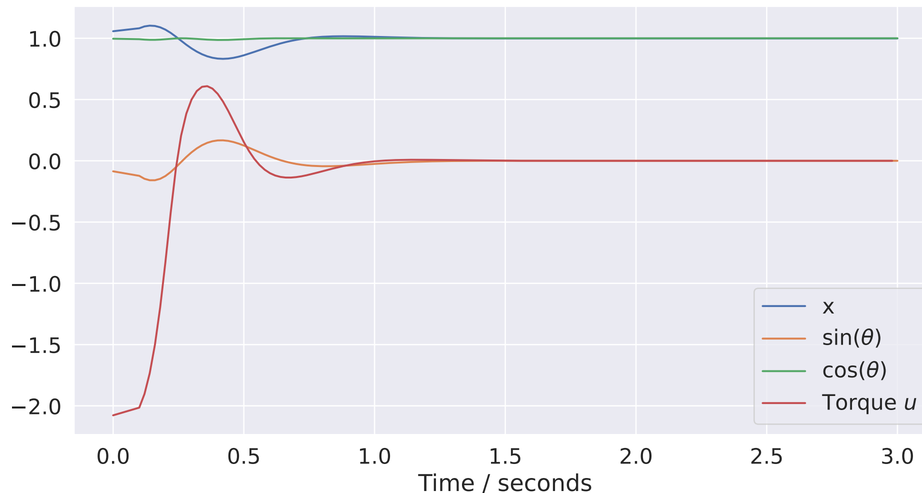
- We expand around the upright position $\bar{\boldsymbol{x}}$, and no action $\bar{\boldsymbol{u}} = 0$.
- We just fix the planning horizon to $N = 30$
- We only use the first control matrix/vector L_0, l_0 at all subsequent time steps

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

Problem 1 *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`) ✸

Note: With the previous exercise, you have everything required for the project. I think the iterative LQR exercise has some good points (and if you are a little stuck you can consult the online solution on gitlab), but if you prefer you can also choose to work on project 2.

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`)

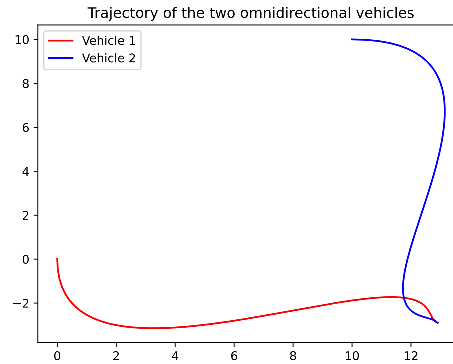
Iterative LQR attempts to overcome the aforementioned problem by first selecting an action sequence u_k, simulating a rollout x_k by applying u_k to the system, linearize the system around (u_k, x_k) to get matrices A_k, B_k, etc., and then creating an optimal control sequence for the linearised system by using discrete LQR, pseudo-code is given in [Her24, 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 2 *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=1.57068e+06, change in cost since last iteration    0
1> J=12294.9, change in cost since last iteration -1.55838e+06
2> J=12294.2, change in cost since last iteration -0.779608
3> J=12294.2, change in cost since last iteration -0.000169729
4> J=12294.2, change in cost since last iteration -5.24324e-08
5> J=12294.2, change in cost since last iteration -1.09139e-11
6> J=12294.2, change in cost since last iteration -5.45697e-12
7> J=12294.2, change in cost since last iteration    0
8> J=12294.2, change in cost since last iteration 1.27329e-11
9> J=12294.2, change in cost since last iteration    0
```

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 3 *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`)

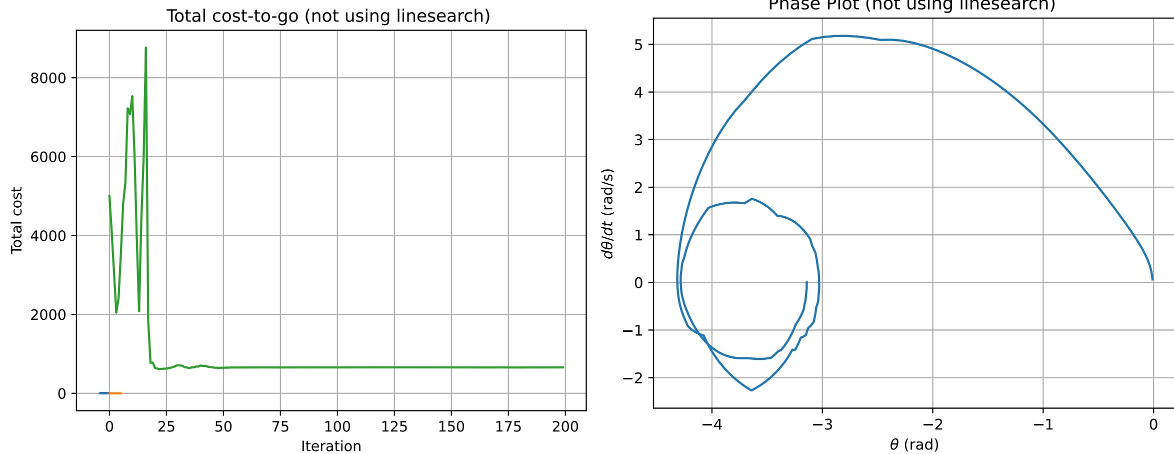
To test whether linesearch offers an improvement on the basic iLQR we will try the method on a more challenging problem, the inverted pendulum. The task is to swing up a pendulum which is originally hanging downwards. We will run the script in both settings and verify your implementation.

Problem 4 *Linesearch with the pendulum task*

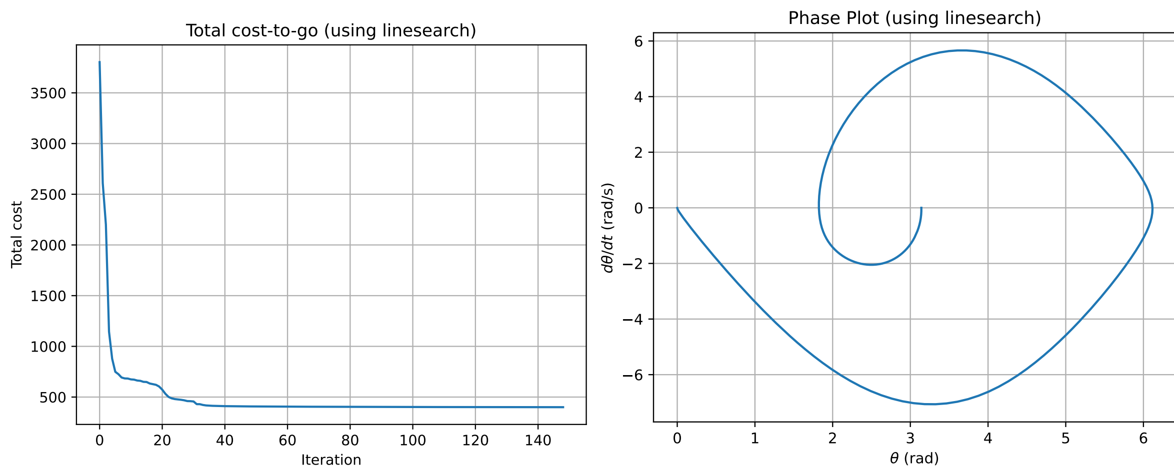
Complete the script and test iLQR with linesearch on the inverted pendulum problem. Make sure it can swing up the pendulum. Set `render = True` in the code to see a small movie of what the controller does on the discretized environment. You can also experiment with the cartpole environment. Discuss what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



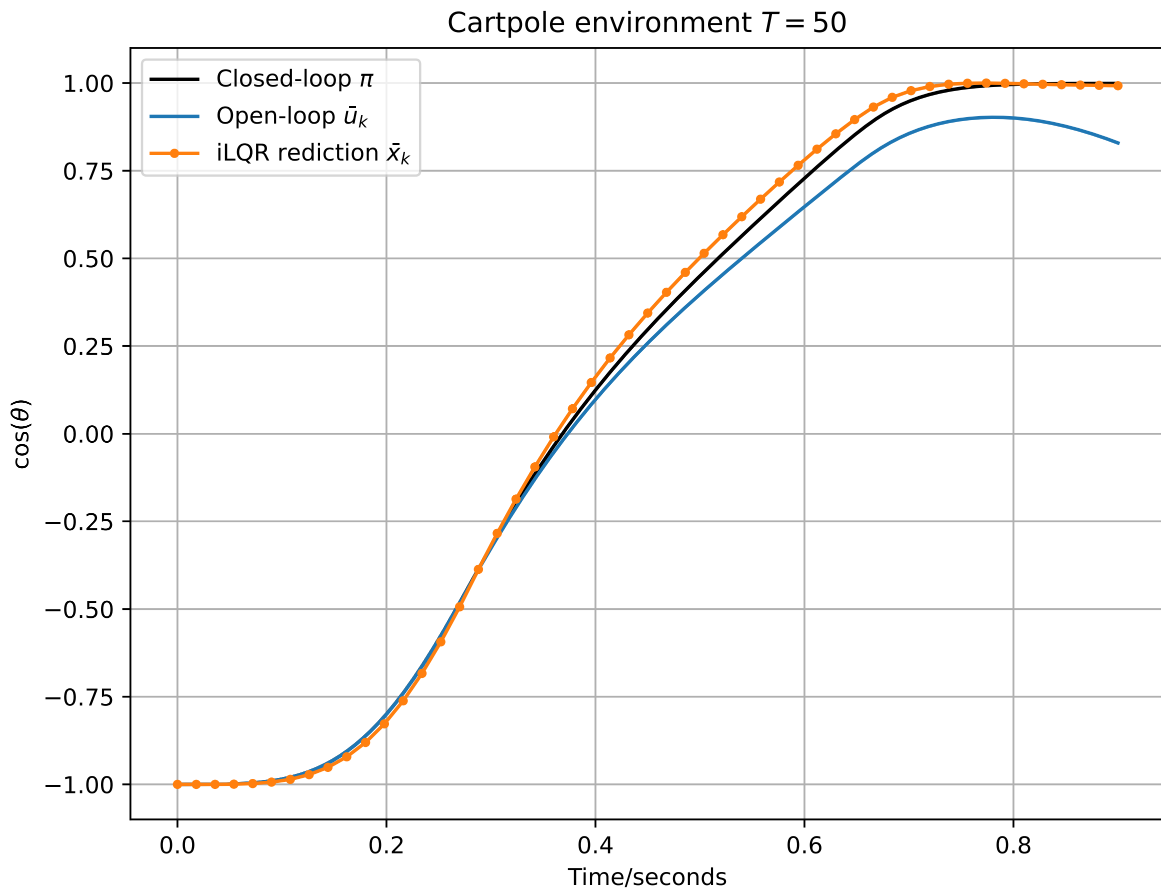
4 An iLQR agent (`ilqr_cartpole_agent.py`, `ilqr_agent.py`)

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 [Her24, eq. (17.17)], and the script will implement both ideas and show a small animation. The computed trajectory is illustrated below:

Problem 5 *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

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

[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**).