AI Enabled Control Engineering

Lecture 9: Optimal Control Experiments on the Rotary Inverted Pendulum

Professor Xun Huang

TA: Zhixiang Ju Haozhe Wang

Aeronautics and Astronautics
College of Engineering
Peking University

huangxun@pku.edu.cn
https://xunger99.github.io/xunger/

Recap: from Class 8 to Class 9

- In Class 8, we assembled the real rotary inverted pendulum circuit.
- We tested the potentiometer, encoder, DC motor, and data logging pipeline.
- We connected Bellman's optimality principle to HJB, LQR, MPC, and Q-learning.
- Today we use model-based optimal control to stabilize the real hardware.

Lecture 9

- Review the local linear model of the rotary inverted pendulum.
- Derive the discrete infinite-horizon LQR controller.
- Compute the LQR gain and deploy it to the real STM32 system.
- Introduce linear MPC as constrained finite-horizon optimal control.
- Run the MPC real-hardware experiment and compare it with LQR.

Learning objectives

By the end of this lab, students should be able to:

- explain why LQR and MPC are local stabilizing controllers for the upright region;
- compute a discrete-time LQR feedback gain from matrices A_d, B_d, Q, R;
- understand how low-pass filtered angle differences provide angular-velocity estimates;
- modify the firmware constants and run a real LQR balancing experiment;
- understand how MPC adds finite-horizon prediction and PWM constraints.

Experimental simplification

No swing-up in this class

In this experiment, we do not use DQN and we do not use an automatic swing-up stage.

- The controller is only valid near the upright equilibrium.
- Students should gently hold the pendulum close to the upright position.
- Then run the Python script and press Enter in the terminal to send the `GO` command.
- After `GO`, STM32 records the startup arm position as $\theta = 0$ and starts closed-loop control.

From continuous model to sampled-data model

The physical rotary inverted pendulum is a continuous-time system:

$$\dot{x}(t) = Ax(t) + Bu(t).$$

However, the STM32 controller does not update the motor command continuously. In our experiment:

$$T_s = 0.005 \text{ s}, \qquad f_s = 200 \text{ Hz}.$$

During one sampling interval, the PWM command is held constant:

$$u(t) = u_k, \qquad t \in [kT_s, (k+1)T_s).$$

Key point

The controller runs in discrete time. Therefore, the LQR and MPC gains used on STM32 should be computed from the discrete model, not directly from the continuous model.

Zero-order-hold discretization

For the continuous-time linear model

$$\dot{x}(t) = Ax(t) + Bu(t),$$

with zero-order hold input $u(t) = u_k$, the exact sampled model is

$$x_{k+1} = A_d x_k + B_d u_k.$$

The matrices are

$$A_d = e^{AT_s},$$

$$B_d = \int_0^{T_s} e^{A\tau} B \, d\tau.$$

In Python, students can compute them by

$$(A_d, B_d) = \texttt{cont2discrete}(A, B, T_s).$$

How do we get angular velocities?

The sensors directly measure only two angles:

$$\theta_k, \qquad \alpha_k.$$

In this class, use angle difference with low-pass filtering:

$$\dot{\theta}_k^{\text{raw}} = \frac{\theta_k - \theta_{k-1}}{T_s}, \qquad \dot{\alpha}_k^{\text{raw}} = \frac{\text{wrap}(\alpha_k - \alpha_{k-1})}{T_s}.$$

Then apply first-order low-pass filtering:

$$\dot{\theta}_k = (1-\lambda)\dot{\theta}_{k-1} + \lambda\dot{\theta}_k^{\text{raw}},$$

$$\dot{\alpha}_k = (1-\lambda)\dot{\alpha}_{k-1} + \lambda\dot{\alpha}_k^{\text{raw}}.$$

Why low-pass filtering is necessary

- Direct difference amplifies sensor noise.
- Potentiometer quantization and encoder count jumps may produce noisy velocity estimates.
- LQR and MPC both use the full state $x = [\theta, \dot{\theta}, \alpha, \dot{\alpha}]^\top$.
- Therefore, velocity estimates must be smooth enough for real-time feedback.

Practical choice

The firmware uses a fixed low-pass coefficient. Students can compare different filtering strengths after the basic experiment works.

LQR problem setup

For the discrete linear system

$$x_{k+1} = A_d x_k + B_d u_k,$$

choose the infinite-horizon quadratic cost

$$J = \sum_{k=0}^{\infty} \left(x_k^\top Q x_k + u_k^\top R u_k \right).$$

Here

$$Q \succeq 0, \qquad R \succ 0.$$

Physical meaning

Q penalizes state deviation from the upright equilibrium. R penalizes large PWM commands.

Choosing Q and R for the RIP

The state order is

$$x = \begin{bmatrix} \theta & \dot{\theta} & \alpha & \dot{\alpha} \end{bmatrix}^\top.$$

A typical choice is

$$Q = \mathrm{diag}(1,\ 0.05, 80,\ 2), \qquad R = 0.001.$$

- q_α is large because the pendulum angle is the most critical state.
- $q_{\dot{\alpha}}$ suppresses pendulum angular velocity.
- q_θ limits long-term arm drift.
- R controls how aggressively the motor is used.

Discrete LQR value function

Assume the optimal value function has a quadratic form:

$$V(x_k) = x_k^\top P x_k, \qquad P = P^\top \succeq 0.$$

Bellman's optimality principle gives

$$V(x_k) = \min_{u_k} \left\{ x_k^\top Q x_k + u_k^\top R u_k + V(x_{k+1}) \right\}.$$

Using

$$x_{k+1} = A_d x_k + B_d u_k,$$

we obtain

$$x_k^\top P x_k = \min_{u_k} \left\{ x_k^\top Q x_k + u_k^\top R u_k + (A_d x_k + B_d u_k)^\top P (A_d x_k + B_d u_k) \right\}.$$

LQR feedback gain

The terms depending on u_k are quadratic:

$$u_k^\top (R + B_d^\top P B_d) u_k + 2 u_k^\top B_d^\top P A_d x_k.$$

Taking derivative with respect to u_k and setting it to zero gives

$$(R + B_d^\top P B_d) u_k + B_d^\top P A_d x_k = 0.$$

Thus

$$u_k^* = -K x_k,$$

where

$$K = (R + B_d^\top P B_d)^{-1} B_d^\top P A_d.$$

Discrete algebraic Riccati equation

The matrix P satisfies the discrete algebraic Riccati equation:

$$P = A_d^\top P A_d - A_d^\top P B_d (R + B_d^\top P B_d)^{-1} B_d^\top P A_d + Q.$$

After solving for P, compute

$$K = (R + B_d^\top P B_d)^{-1} B_d^\top P A_d.$$

For this lab

We use infinite-horizon discrete LQR. Therefore, we solve the algebraic Riccati equation and do not need a terminal condition $P(T) = H$.

Student task: compute the LQR gain

Students should find or load the model matrices A_d and B_d from the provided code or handout.

Then compute the gain in Python:

```
import numpy as np
from scipy.linalg import solve_discrete_are

Q = np.diag([1.0, 0.05, 80.0, 2.0])
R = np.array([[0.001]])

P = solve_discrete_are(Ad, Bd, Q, R)
K = np.linalg.inv(R + Bd.T @ P @ Bd) @ (Bd.T @ P @ Ad)
print(K)
```

Output

The result is a row vector $K = [k_\theta, k_{\dot{\theta}}, k_\alpha, k_{\dot{\alpha}}]$.

How to put K into firmware

The control law is

$$u_k = -Kx_k.$$

For our firmware, insert the four gains into:

$$K_\theta, \quad K_{\dot\theta}, \quad K_\alpha, \quad K_{\dot\alpha}.$$

Then the firmware computes

$$u_{mraw} = -(K_\theta\theta + K_{\dot\theta}\dot\theta + K_\alpha\alpha + K_{\dot\alpha}\dot\alpha).$$

Finally, the PWM command is saturated:

$$u = \mathrm{sat}(u_{mraw}, -u_{\max}, u_{\max}).$$

LQR hardware experiment: preparation

1. Upload the LQR firmware to STM32.
2. Calibrate the potentiometer zero position: hold the pendulum upright, read the current raw value, and update `POT_ZERO` in the firmware.
3. Upload the firmware again after modifying `POT_ZERO`.
4. Close the Arduino Serial Monitor.
5. Connect motor power only after the wiring is checked.
6. Hold the pendulum gently near the upright position before pressing Enter in the Python terminal.
7. Keep fingers clear of the rotating arm.

LQR hardware experiment: run

1. Open a terminal in the LQR experiment folder.
2. Run the Python script:

```
python rip_lqr_run.py
```

3. Hold the pendulum close to upright.
4. Press Enter in the Python terminal to send `GO`.
5. The script records data and saves a CSV log and figures.

LQR hardware experiment: expected data

The Python script records:

$$t, \quad \theta, \quad \alpha, \quad \dot{\theta}_{\mathrm{LPF}}, \quad \dot{\alpha}_{\mathrm{LPF}}, \quad u.$$

Here, the angular velocities are not directly measured. They are computed from angle differences and then filtered:

$$\dot{\theta}_{\mathrm{raw}}(k) = \frac{\theta_k - \theta_{k-1}}{T_s}, \qquad \dot{\alpha}_{\mathrm{raw}}(k) = \frac{\alpha_k - \alpha_{k-1}}{T_s}.$$

$$\dot{x}_{\mathrm{LPF}}(k) = (1-\lambda)\dot{x}_{\mathrm{LPF}}(k-1) + \lambda \dot{x}_{\mathrm{raw}}(k).$$

The generated plots should include:

- angle response: $\theta(t)$ and $\alpha(t)$;
- filtered angular velocity response: $\dot{\theta}_{\mathrm{LPF}}(t)$ and $\dot{\alpha}_{\mathrm{LPF}}(t)$;
- PWM command $u(t)$.

Why introduce MPC after LQR?

LQR gives an explicit feedback law:

$$u = -Kx.$$

This is simple and fast, but standard LQR does not explicitly handle input constraints.

In real hardware, the PWM command must satisfy

$$-u_{\max} \leq u \leq u_{\max}.$$

MPC idea

MPC solves a finite-horizon optimal control problem at each sampling instant and includes PWM constraints directly inside the optimization problem.

MPC prediction model

Use the same discrete linear model:

$$x_{k+1} = A_d x_k + B_d u_k.$$

At current time k, predict future states:

$$x_{k+i+1|k} = A_d x_{k+i|k} + B_d u_{k+i|k}, \qquad i = 0, 1, \ldots, N-1.$$

The initial predicted state is

$$x_{k|k} = x_k.$$

Prediction horizon

In the provided embedded MPC code, the prediction horizon is small enough for STM32 real-time implementation at 200 Hz.

Stacked prediction form

Define the future input sequence

$$U = \begin{bmatrix} u_{k|k} & u_{k+1|k} & \cdots & u_{k+N-1|k} \end{bmatrix}^\top.$$

Stack future predicted states:

$$X = \begin{bmatrix} x_{k+1|k}^\top & x_{k+2|k}^\top & \cdots & x_{k+N|k}^\top \end{bmatrix}^\top.$$

Then

$$X = S_x x_k + S_u U.$$

Meaning

Once the current state x_k is known, all future predicted states are linear functions of the future input sequence U.

MPC cost function

The finite-horizon MPC cost is

$$J = \sum_{i=0}^{N-1} \left(x_{k+i|k}^\top Q x_{k+i|k} + u_{k+i|k}^\top R u_{k+i|k} \right) + x_{k+N|k}^\top P x_{k+N|k}.$$

Use the same stage weights as LQR:

$$Q = \mathrm{diag}(1, 0.05, 80, 2), \qquad R = 0.001.$$

Terminal weight

The terminal weight P can be chosen as the infinite-horizon LQR Riccati solution P_∞ to approximate the tail cost beyond the prediction window.

From MPC cost to quadratic programming

Using

$$X = S_x x_k + S_u U,$$

the part of the cost depending on U can be written as

$$J_U(U) = \frac{1}{2} U^\top H U + f^\top U,$$

where

$$f = G x_k.$$

The matrices H and G depend only on

$$A_d, B_d, Q, R, P, N.$$

Embedded implementation

H and G are computed offline and stored in the firmware. Online computation only updates $f = G x_k$.

MPC input constraint

The motor PWM command is bounded:

$$-u_{\max} \le u_{k+i|k} \le u_{\max}, \qquad i = 0, 1, \ldots, N-1.$$

Therefore, the MPC problem is

$$\begin{aligned} \min_{U} \quad & \frac{1}{2} U^\top H U + (G x_k)^\top U, \\ \text{s.t.} \quad & -u_{\max} \le U_i \le u_{\max}. \end{aligned}$$

Main difference from LQR

LQR computes an unconstrained feedback and then the firmware may saturate it. MPC considers the PWM bound inside the optimization problem.

Projected-gradient solver on STM32

The gradient of the QP objective is

$$\nabla J(U) = HU + f.$$

A projected-gradient iteration is

$$U^{(j+1)} = \Pi_{[-u_{\max}, u_{\max}]} \left(U^{(j)} - \eta (HU^{(j)} + f) \right).$$

Here Π means element-wise saturation:

$$U_i \leftarrow \max(-u_{\max}, \min(u_{\max}, U_i)).$$

Real-time strategy

Use a fixed number of iterations and warm start from the previous control sequence.

MPC rolling optimization

At each control cycle:

1. read the current state x_k;
2. compute $f = Gx_k$;
3. warm start the control sequence U;
4. run projected-gradient iterations;
5. apply only the first input:

$$u_k = u^*_{k|k};$$

6. repeat at the next 5 ms control cycle.

MPC hardware experiment

1. Upload the MPC firmware to STM32.
2. Close the Arduino Serial Monitor.
3. Hold the pendulum near the upright position.
4. Run the Python script:

```
python rip_mpc_run.py
```

5. Press Enter in the terminal to send `GO`.
6. Record angle, angular velocity, PWM, and computation-time logs.

Comparing LQR and MPC in the lab

Item	LQR	MPC
Control law	explicit $u = -Kx$	QP-defined implicit feedback
Constraint handling	saturation after control	constraint inside optimization
Online computation	very small	projected-gradient iterations
Tuning parameters	Q, R	$Q, R, P, N, u_{\max}$
Hardware focus	fast local stabilization	constrained local stabilization

Student comparison task

Compare angle curves, PWM distributions, and whether the controller frequently hits the PWM bound.

Lab report requirements

Each group should record:

- the model matrices or file location used for computing K;
- at least three different LQR weight settings:

$$Q = \mathrm{diag}(q_\theta, q_{\dot\theta}, q_\alpha, q_{\dot\alpha}), \qquad R = r;$$

- the computed LQR gain K for each Q, R setting;
- one successful experimental log for each LQR setting;
- one successful MPC experimental log;
- comparison plots of $\theta, \alpha, \dot\theta_{\mathrm{LPF}}, \dot\alpha_{\mathrm{LPF}}$ and PWM for different LQR settings;
- a short comparison of LQR tuning effects and MPC performance.

Summary of Lecture 9

- LQR and MPC both rely on the local discrete model $x_{k+1} = A_d x_k + B_d u_k$.
- LQR solves a discrete Riccati equation and gives a fast explicit feedback law.
- MPC solves a finite-horizon constrained optimization problem at each sampling time.
- In the real experiment, students manually place the pendulum near upright and press Enter in Python to start control.
- Low-pass filtered angle differences provide the velocity estimates needed by both controllers.
- The key experimental comparison is between unconstrained feedback plus saturation and constrained rolling optimization.