Pendulum Controller: 1kHz Compensator / 20kHz Simulation

Analytical Derivation and Control of the Inverted Pendulum on a DC Motor

Subtitle: State-Space Linearization via Jacobian Method and LQR Implementation

Author: Diptopal Basu

Date: February 2026

Version: 1.0

Tags: [Control Theory, Robotics, Inverted Pendulum, LQR, Jacobian Linearization]---

Executive Summary & Configuration

This document provides a rigorous mathematical derivation of the Inverted Pendulum dynamics—specifically a pendulum mounted directly to a DC motor shaft—using **Lagrangian Mechanics**. The derivation accounts for both the mechanical inertia of the system and the electrical characteristics of the DC motor. It concludes with the linearization of the non-linear equations into a State-Space model ($Ax + Bu$) and provides the Python implementation for LQR gain calculation to achieve upright stability.

These parameters form the basis of the numerical matrices below.

Parameter	Description	Value
M_1	Pendulum Mass	**0.2 kg**
L_1	Pendulum Length	**0.3 m**
I_{tot}	Combined System Inertia	**0.00600575 kg·m²**
b_1	Mechanical Friction	**0.008 Nms/rad**
k_t / k_b	Torque / Back-EMF Constants	**0.12**
R	Terminal Resistance	**2.5 Ω**
g	Gravitational Acceleration	**9.81 m/s²**
$V_{deadzone}$	Minimum Motor Voltage	**0.4 V**

II. Step 1: Lagrangian Mechanics

Justification: We use the Lagrangian approach because it naturally handles the energy of rotating bodies without needing to decompose every individual force vector.

1. Kinetic Energy (T):

$$T = \frac{1}{2} I_{tot} \dot{\theta}^2$$

2. Potential Energy (V):

Using the convention where 0 is down and π is up:

$$V = M_1 g \frac{L_1}{2} \cos(\theta)$$

3. The Lagrangian ($L = T - V$):

$$L = \frac{1}{2} I_{tot} \dot{\theta}^2 - \frac{M_1 g L_1}{2} \cos(\theta)$$

4. The Equation of Motion:

Applying $\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = \tau_{motor} - \text{friction}$:

$$I_{tot} \ddot{\theta} - \frac{M_1 g L_1}{2} \sin(\theta) = \frac{k_t}{R}(V_{in} - k_b \dot{\theta}) - b_1 \dot{\theta}$$

III. Step 2: Jacobian Linearization

Justification: LQR requires a linear system ($\dot{x} = Ax + Bu$). Since $\sin(\theta)$ is non-linear, we use a Taylor series expansion (Jacobian) around the unstable equilibrium point $\theta = \pi$.

1. Define the Nonlinear State Functions:

Let $x_1 = \theta$ and $x_2 = \dot{\theta}$.

$\dot{x}_1 = f_1(x, u) = x_2$

$\dot{x}_2 = f_2(x, u) = \frac{M_1 g L_1}{2I_{tot}} \sin(x_1) - \frac{1}{I_{tot}}(b_1 + \frac{k_t k_b}{R})x_2 + \frac{k_t}{RI_{tot}} V_{in}$

2. Compute the Jacobians at $x_0 = [\pi, 0]^T$:

- **For Matrix A:**

$$A = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{M_1 g L_1}{2I_{tot}} \cos(\pi) & -\frac{1}{I_{tot}}(b_1 + \frac{k_t k_b}{R}) \end{bmatrix}$$

Since $\cos(\pi) = -1$ and we define the error as $\Delta\theta = \theta - \pi$, the sign flips to positive for the restoring force calculation.

- **For Matrix B:**

$$B = \begin{bmatrix} \frac{\partial f_1}{\partial u} \\ \frac{\partial f_2}{\partial u} \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{k_t}{RI_{tot}} \end{bmatrix}$$

3. Numerical Result:

$$A = \begin{bmatrix} 0 & 1 \\ 49.003 & -2.291 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 7.992 \end{bmatrix}$$

IV. Step 3: LQR and the Riccati Equation

Justification: LQR finds the "optimal" balance between system performance and energy usage. It solves the Algebraic Riccati Equation (ARE) to minimize total cost.

1. Penalty Selection (The Q Matrix):

We prioritize position error over velocity:

$$Q = \begin{bmatrix} 1010 & 0 \\ 0 & 15.7 \end{bmatrix}, \quad R = [0.1]$$

2. The Riccati Equation:

$$A^T P + PA - PBR^{-1}B^T P + Q = 0$$

Solving for P gives the energy surface of the system. The optimal gains are $K = R^{-1}B^T P$.

3. Theoretical vs. Manual Gains:

The Python solver yields $k_1 \approx -106$ and $k_2 \approx -13$. However, your project uses $k_1 = -220.0$ **and** $k_2 = -26.0$. This doubling of "stiffness" is a deliberate engineering choice to compensate for high hardware friction.

IV. Step 3: LQR Cost Weighting and Bryson's Rule

The LQR algorithm minimizes the cost function J, which balances the importance of state accuracy against the cost of control effort (voltage):

$$J = \int_0^\infty (x^T Q x + u^T R u) dt$$

1. Expanding the Q Matrix

For your 2-state pendulum system, the Q matrix is a 2×2 diagonal matrix. Each diagonal element represents the "penalty" for an error in that specific state:

$$Q = \begin{bmatrix} q_{11} & 0 \\ 0 & q_{22} \end{bmatrix}$$

- q_{11}: Penalty for **Angular Error** ($\theta - \pi$). A higher value makes the pendulum stiffer and more vertical.
- q_{22}: Penalty for **Angular Velocity** ($\dot{\theta}$). A higher value adds more electronic damping to prevent oscillation.

2. Bryson's Rule for Initial Tuning

Bryson's Rule suggests that the weights should be the inverse of the square of the maximum acceptable error for that state:

$$q_{ii} = \frac{1}{\text{max acceptable value of } (x_i^2)}$$

$$R = \frac{1}{\text{max acceptable value of } (u^2)}$$

3. Numerical Transition: Ideal vs. Implementation

By comparing the "Ideal" weights (theoretical) to the weights required to reach your "Implementation" gains, we can see how Bryson's Rule was pushed to handle real-world friction.

Weight	Parameter	Ideal Selection	Implementation Selection	Physical Meaning
q_{11}	Position Penalty	**1010**	**4575**	We tolerate almost **zero** angular error.
q_{22}	Velocity Penalty	**15.7**	**64**	High damping to kill jitter.
R	Control Cost	**0.1**	**0.1**	Voltage is "cheap"; use as much as needed.

V. Step 4: Integral Justification ($k_i = -75.0$)

Justification: Why did you add k_i? Because LQR alone is mathematically "blind" to constant disturbances.

- **The 176° Settling Point:** Without k_i, gravity creates a constant torque. To counteract it, the LQR controller needs a non-zero error to produce voltage. This results in the pendulum settling slightly below the top (e.g., at 176°). This is the **steady-state error**.
- **The Integrator's Job:** k_i sums the error over time. As long as the pendulum is even 0.001 rad away from π, the sum grows. Eventually, it provides enough voltage to overcome the $0.4V$ **deadzone** and push the pendulum to exactly 180.000°.
- **Settling without Integrator:** Without the integrator, the system settles at the point where $k_1 \times \text{Error} = \text{Gravity Torque}$. With your mass, that offset is several degrees.

VI. Final Control Law

$$\text{Output Voltage}(V) = \underbrace{k_1(\theta - \pi) + k_2(\dot{\theta})}_{\text{LQR Stability}} + \underbrace{k_i \int (\theta - \pi) dt}_{\text{Precision Integral}} + \text{Deadzone}$$

Numerical Implementation (Python)

The LQR controller finds the optimal gain matrix K by minimizing a cost function that balances error (Q) and control effort (R). The core of this optimization is solving the **Algebraic Riccati Equation**:

$$A^T P + PA - PBR^{-1}B^T P + Q = 0$$

In the provided Python script, the line `solve_continuous_are(A, B, Q, R_lqr)` performs this calculation. It computes the unique positive-definite matrix P, which is then used to find the raw gains:

$$K_{raw} = R^{-1}B^TP$$

This script handles the mapping to the C-code implementation correctly. It takes the raw output K_{raw} and applies the negation logic ($k = -K$) so that the final **Pendulum gains** (k_1 for position and k_2 for velocity) result in the negative values used in the compensator. This ensures that a positive angular error produces a negative corrective voltage to restore the pendulum to the upright position.

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

def compute_lqr():
    # Parameters from Table
    M1, L1, I_tot = 0.2, 0.3, 0.00600575
    g, b1, kt, kb, R = 9.81, 0.008, 0.12, 0.12, 2.5

    # Jacobian Matrices
    A = np.array([[0, 1], [(M1*g*L1)/(2*I_tot), -(b1 + (kt*kb/R))/I_tot]])
    B = np.array([[0], [(kt/R)/I_tot]])

    # LQR Weights
    Q = np.diag([4575, 64])
    R_mat = np.array([[0.1]])

    # Solve Riccati
    P = solve_continuous_are(A, B, Q, R_mat)
    K = np.linalg.inv(R_mat) @ B.T @ P
    print(f"LQR Gains: k1={-K[0,0]:.4f}, k2={-K[0,1]:.4f}")

compute_lqr()
```

5.1 Final LQR Results

The table below summarizes the gains derived from the LQR optimization. The "Ideal" values represent standard mathematical weights, while the "Implementation" values reflect the high-stiffness tuning required to dominate hardware friction and stiction.

State Variable	Symbol	Ideal Gain	Implementation	Logic
Pendulum Position	k_1	≈ -106	-220.0	**Negative** (Restoring Torque)
Pendulum Velocity	k_2	≈ -13	-26.0	**Negative** (Damping Force)

5.2 Limitations and Future Work

The Impact of Noise and Drift

It is important to observe that in a purely theoretical environment—**without sensor noise or mechanical disturbances—the LQR balance is mathematically perfect.** Under ideal conditions, the pendulum would remain vertical indefinitely with near-zero control effort once it reaches the equilibrium point.

However, the physical implementation reveals two primary challenges:

1. **Sensor Noise and Jitter:** The raw encoder data contains high-frequency noise. When we derive velocity ($\dot{\theta}$) from these readings, the noise is amplified. This forces the motor to react to "phantom" movements, creating the micro-vibrations heard during operation.
2. **System Drift:** Despite a high k_1 gain, the system may experience a slow "drift" over time. This is caused by thermal changes in the motor resistance, slight misalignments in the center-of-gravity, or internal encoder scaling errors.

The Student Challenge

Because of these real-world constraints, even this high-stiffness derivation does not fully do justice to the complexity of a perfect, silent balance. **The student is encouraged to work on this further** by focusing on:

- **Noise Rejection:** Implementing a Kalman Filter to optimally estimate the state while ignoring sensor noise.
- **Drift Compensation:** Improving the z-pulse synchronization logic or adding an outer-loop to track and zero out long-term drift.
- **System Identification:** Refining the I_{tot} and b_1 parameters through empirical testing to closer match the mathematical model to the physical hardware.

Final Note: The implementation of $k_1 = -220.0$ and $k_i = -75.0$ provides a robust "stiff" balance that holds the pendulum upright despite noise, but a truly "perfect" balance requires the student to explore advanced filtering and non-linear modeling.