

Control Systems Lab - Experiment No. 2:

Inverted pendulum

Group No.: 11

Members:

Sanjay Meena(22B3978)

Devtanu Barman(22B3904)

October 7, 2024

1 Aim

To design and implement control action for maintaining a pendulum in the upright position (even when subjected to external disturbances) through LQR technique in an Arduino Mega..

2 Objectives

- To restrict the pendulum arm vibration (α) within ± 3 degrees.
- To restrict the base angle oscillation (θ) within ± 30 degrees.

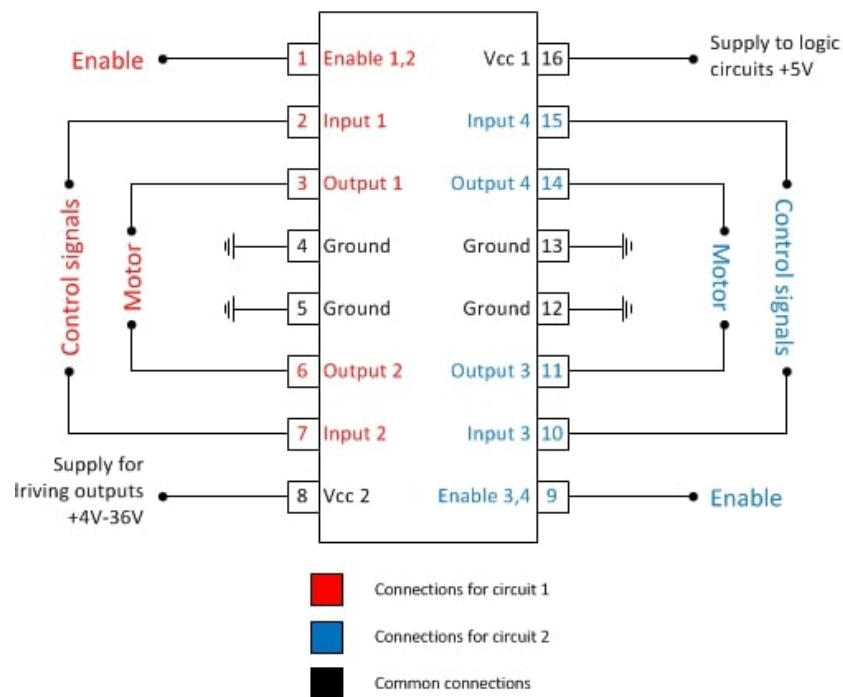
3 Prerequisites

- Arduino programming
- Matlab coding
- LQR technique
- State space modelling







4 Materials and Equipment Required

- Inverted pendulum setup
- Arduino Mega
- A-B cable
- Power supply
- L293D IC
- Jumper wires
- Single-stranded wires
- Breadboard
- Screwdriver
- Wire stripper

5 Pin Diagram of L293D



6 Pin Diagram of Encoder

Function	Encoder Pin Number	ARDUINO	Pin Number	AMT-036-1-036 Colors
+5 V	1	5V		BLACK
SCLK	2	52		BROWN
MOSI	3	51		RED
GND	4	GND		ORANGE
MISO	5	50		YELLOW
CHIP SELECT	6	2		GREEN

7 Procedure

7.1 State-Space Modeling

The inverted pendulum is a classic problem in control systems where the goal is to keep the pendulum balanced upright. This problem is typically modeled using a state-space approach. The state variables include:

θ : The angular position of the base of the pendulum.

α : The angular displacement of the pendulum arm from the vertical.

$\dot{\theta}$: The angular velocity of the base.

$\dot{\alpha}$: The angular velocity of the pendulum arm.

These four variables make up the system's state vector:

$$x = \begin{bmatrix} \theta \\ \alpha \\ \dot{\theta} \\ \dot{\alpha} \end{bmatrix}$$

The dynamics of the system are represented by matrices A , B , C , and D in the standard state-space form:

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

Where:

A : Represents the system dynamics.

B : Represents how the control input affects the system.

C and D : Relate the state variables to the system's output.

7.2 LQR Design

The Linear Quadratic Regulator (LQR) was used to find an optimal control strategy. The LQR aims to minimize a cost function that balances two competing objectives:

- Keeping the state variables $(\theta, \alpha, \dot{\theta}, \dot{\alpha})$ small, i.e., ensuring the pendulum stays upright and stable.
- Keeping the control input (force or torque applied to the system) as small as possible to avoid excessive energy consumption or control effort.

The cost function is typically defined as:

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

Where:

- Q : State cost matrix that penalizes deviations in the state variables.
- R : Control effort cost matrix that penalizes excessive control inputs.
- u : The control input that influences the system.

In this case, the Q and R matrices were carefully chosen to prioritize stability while minimizing control effort. A higher weight in Q on the angular displacement of the pendulum arm (α) and base position (θ) ensured that deviations from the upright position were minimized. The specific values used were:

$$Q = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 100 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$
$$R = 1$$

The gain matrix K is derived from the LQR algorithm. It represents the optimal feedback gains for each state variable $(\theta, \alpha, \dot{\theta}, \dot{\alpha})$. In this experiment, the gain matrix was calculated using MATLAB as:

$$K = \begin{bmatrix} -4.3497 \\ 73.6849 \\ -3.1068 \\ 9.502 \end{bmatrix}$$

This gain matrix was then hard-coded into the Arduino program for real-time control.

7.3 Controller Implementation

The implementation of the LQR controller in the Arduino system involved the following steps:

- **Real-time Sensor Input:** The angular positions $(\theta$ and $\alpha)$ were continuously measured using sensors. From these measurements, the angular velocities $(\dot{\theta}$ and $\dot{\alpha})$ were estimated by calculating the difference between consecutive measurements and dividing by the sampling time.
- **State Feedback Control:** Using the real-time state vector $[\theta, \alpha, \dot{\theta}, \dot{\alpha}]$, the control input was computed using the LQR gain matrix K . The control law is given by:

$$u = -Kx$$

Where:

- u : Control input (e.g., voltage to the motor controlling the base of the pendulum).
- K : LQR gain matrix.
- x : Current state vector $[\theta, \alpha, \dot{\theta}, \dot{\alpha}]$.

The control input u was applied to the system via the Arduino, driving the motor to adjust the position of the pendulum base, thus stabilizing the pendulum.

- **Sampling and Control Loop:** The system was sampled at regular intervals, and the control input was recalculated each time based on the new state measurements. A slight delay was introduced between sampling intervals to ensure that the pendulum had sufficient time to respond to the control input.

7.4 Performance Tuning

The Q and R matrices were fine-tuned through experimentation to ensure the system met the desired performance criteria. Initially, the system showed instability, which required adjusting the weighting matrices through trial and error. A balance had to be struck between:

- **Robustness:** The ability of the system to remain stable even when subjected to external disturbances (e.g., a push on the pendulum).
- **Responsiveness:** The speed at which the system corrected deviations from the upright position.

By increasing the cost in the Q matrix (particularly for θ), the system became more responsive, adhering more strictly to the constraints. However, this made the system less robust—if the pendulum was pushed too hard, it would fail to return to the upright position. A binary search was used to find the optimal Q and R matrices that balanced these factors.

8 Explanation of the Arduino Code

8.1 Code

The following Arduino code implements the LQR controller to stabilize the inverted pendulum system by controlling motor voltages based on real-time feedback from two encoders. Below is the code we implemented:

```
1  /* Include the SPI library for the Arduino boards */
2  #include <SPI.h>
3
4  /* Serial rates for UART */
5  #define BAUDRATE      115200
6
7  /* SPI commands */
8  #define AMT22_NOP      0x00
9  #define AMT22_RESET    0x60
10 #define AMT22_ZERO      0x70
11
12 /* Define special ASCII characters */
13 #define NEWLINE         0x0A
14 #define TAB              0x09
15
16 /* We will use these macros for 12 or 14 bit encoders */
17 #define RES12            12
18 #define RES14            14
```

```

19
20 /* SPI pins */
21 #define ENC_0          2
22 #define ENC_1          3
23 #define SPI_MOSI       51
24 #define SPI_MISO       50
25 #define SPI_SCLK       52
26
27 #define motorPin1      8
28 #define motorPin2      9
29
30 int alpha_raw;
31 int theta_raw;
32 float alpha_rad;
33 float theta_rad;
34
35 float theta_prev = 0;
36 float alpha_prev = 0;
37 float theta_dot;
38 float alpha_dot;
39
40 int t_old = 0;
41 int t_new = 0;
42 int dt = 0;
43
44 float K_theta = -4.242641;
45 float K_alpha = 72.910965;
46 float K_theta_dot = -2.655214;
47 float K_alpha_dot = 9.602371;
48
49 float control_signal;
50 int motor_voltage = 0;
51 float scaling_factor = 3;
52
53 void setup() {
54     pinMode(SPI_SCLK, OUTPUT);
55     pinMode(SPI_MOSI, OUTPUT);
56     pinMode(SPI_MISO, INPUT);
57     pinMode(ENC_0, OUTPUT);
58     pinMode(ENC_1, OUTPUT);
59
60     Serial.begin(BAUDRATE);
61     digitalWrite(ENC_0, HIGH);
62     digitalWrite(ENC_1, HIGH);
63     SPI.setClockDivider(SPI_CLOCK_DIV32); // 500 kHz
64     SPI.begin();
65 }
66
67 void loop() {

```

```

68     delay(5);
69
70     alpha_raw = getPositionSPI(ENC_0, RES14);
71     alpha_rad = convertToRadians(alpha_raw);
72
73     theta_raw = getPositionSPI(ENC_1, RES14);
74     theta_rad = convertToRadians(theta_raw);
75
76     t_new = millis();
77     dt = t_new - t_old;
78
79     theta_dot = (theta_rad - theta_prev) * 1000 / dt;
80     alpha_dot = (alpha_rad - alpha_prev) * 1000 / dt;
81
82     control_signal = -(K_theta * theta_rad + K_alpha * (
83         alpha_rad - 177)
84     + K_theta_dot * theta_dot + K_alpha_dot * alpha_dot);
85
86     float duty_cycle = control_signal / 905.0;
87     if (duty_cycle >= 1) duty_cycle = 1;
88     else if (duty_cycle <= -1) duty_cycle = -1;
89
90     if (duty_cycle > 0) {
91         analogWrite(motorPin2, duty_cycle * 255);
92         analogWrite(motorPin1, 0);
93     } else {
94         analogWrite(motorPin1, -duty_cycle * 255);
95         analogWrite(motorPin2, 0);
96     }
97
98     theta_prev = theta_rad;
99     alpha_prev = alpha_rad;
100     t_old = t_new;
101
102     Serial.print("Theta: ");
103     Serial.print(theta_rad, DEC);
104     Serial.write(NEWLINE);
105     Serial.print(" Alpha: ");
106     Serial.print(alpha_rad, DEC);
107     Serial.write(NEWLINE);
108     Serial.print(" Control Signal: ");
109     Serial.println(duty_cycle * 255);
110 }
111
112 float convertToRadians(int raw_value) {
113     float angle_deg = (raw_value) * (360.0 / 16384);
114     return angle_deg;
115 }

```



```

116 uint16_t getPositionSPI(uint8_t encoder, uint8_t resolution)
117 {
118     uint16_t currentPosition = spiWriteRead(AMT22_NOP, encoder,
119         false) << 8;
120     delayMicroseconds(3);
121     currentPosition |= spiWriteRead(AMT22_NOP, encoder, true);
122     return (currentPosition &= 0x3FFF);
123 }
124
125 uint8_t spiWriteRead(uint8_t sendByte, uint8_t encoder,
126     uint8_t releaseLine) {
127     setCSLine(encoder, LOW);
128     delayMicroseconds(3);
129     uint8_t data = SPI.transfer(sendByte);
130     delayMicroseconds(3);
131     setCSLine(encoder, releaseLine);
132     return data;
133 }
134
135 void setCSLine(uint8_t encoder, uint8_t csLine) {
136     digitalWrite(encoder, csLine);
137 }

```

Listing 1: Arduino Code for LQR Controlled Inverted Pendulum

8.2 Explanation

8.2.1 Inclusion of SPI Library and Pin Definitions

The code begins by including the `SPI.h` library, which allows communication with the encoders.

```
#include <SPI.h>
```

Pins for the encoders and motor control are defined, including the SPI communication pins:

- `ENC_0`, `ENC_1`: Chip select lines for the two encoders.
- `motorPin1`, `motorPin2`: Pins that control the motor through an H-bridge driver.

8.2.2 Real-Time Sensing and State Estimation

The encoder values are read using the SPI protocol, and raw angular data is converted to radians. The angular velocities are calculated by differentiating consecutive measurements.

```

alpha_raw = getPositionSPI(ENC_0, RES14);
alpha_rad = convertToRadians(alpha_raw);

theta_raw = getPositionSPI(ENC_1, RES14);
theta_rad = convertToRadians(theta_raw);

```

The angular velocities $\dot{\theta}$ and $\dot{\alpha}$ are computed as:

$$\theta = \frac{\theta_{\text{new}} - \theta_{\text{prev}}}{\Delta t}$$

$$\alpha = \frac{\alpha_{\text{new}} - \alpha_{\text{prev}}}{\Delta t}$$

8.2.3 LQR Control Implementation

The control signal is calculated using the LQR control law $u = -Kx$, where K is the LQR gain matrix and x is the state vector containing θ , α , $\dot{\theta}$, and $\dot{\alpha}$.

```

control_signal = -(K_theta * theta_rad + K_alpha * (alpha_rad - 177)
+ K_theta_dot * theta_dot + K_alpha_dot * alpha_dot);

```

Here, the LQR gain values for each state variable are hard-coded. The value $\alpha_{\text{rad}} - 177$ is used to account for an angular offset.

8.2.4 Motor Control Using PWM

The control signal is converted to a duty cycle, which determines how much voltage is applied to the motor. The duty cycle is clamped to the range $[-1, 1]$.

```

float duty_cycle = control_signal / 905.0;
if (duty_cycle >= 1) { duty_cycle = 1; }
else if (duty_cycle <= -1) { duty_cycle = -1; }

```

The motor is driven based on the sign and magnitude of the duty cycle:

```

if (duty_cycle > 0) {
    analogWrite(motorPin2, duty_cycle * 255);
    analogWrite(motorPin1, 0);
} else {
    analogWrite(motorPin1, -duty_cycle * 255);
    analogWrite(motorPin2, 0);
}

```

8.2.5 SPI Communication for Encoders

The function `getPositionSPI()` reads the position data from the encoders using SPI. It verifies data integrity using checksums and extracts the 14-bit angular position:

```
uint16_t getPositionSPI(uint8_t encoder, uint8_t resolution)
{
    currentPosition &= 0x3FFF; // Mask to extract 14-bit position data
    return currentPosition;
}
```

8.2.6 System Feedback and Debugging

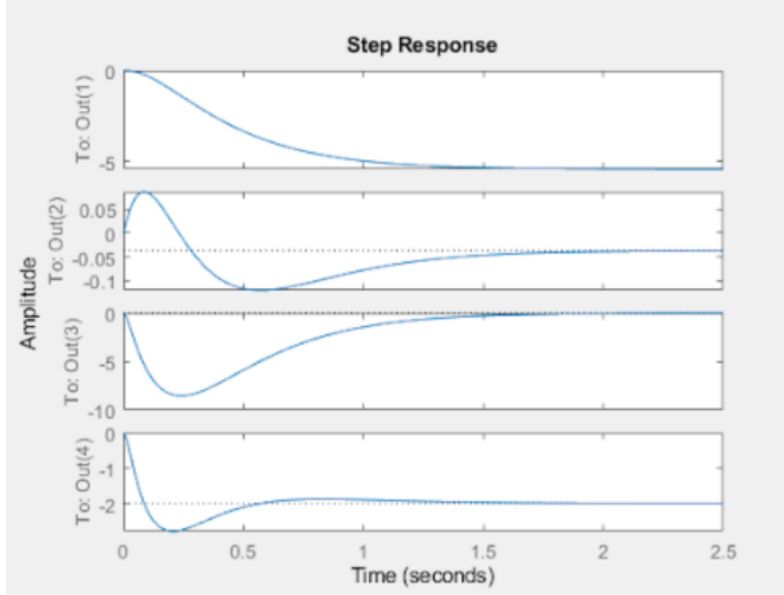
The program prints real-time data such as angular positions θ and α and the control signal for debugging purposes:

```
Serial.print("Theta: ");
Serial.print(theta_rad, DEC);
Serial.write(NEWLINE);
Serial.print(" Alpha: ");
Serial.print(alpha_rad, DEC);
Serial.write(NEWLINE);
Serial.print(" Control Signal: ");
Serial.println(duty_cycle * 255);
```

- Thus, this code demonstrates an LQR-controlled inverted pendulum system implemented on Arduino. It efficiently reads angular position data, computes state variables, and applies the appropriate motor voltage to maintain stability. The code integrates SPI communication for real-time sensor input and PWM for motor control, providing optimal performance for a highly dynamic and nonlinear system.

9 Results

The step responses for the 4 state variables (θ , α , $\dot{\theta}$, and $\dot{\alpha}$) are



where θ corresponds to step response for Out(1), α corresponds to Out(2), $\dot{\theta}$ corresponds to Out(3), and $\dot{\alpha}$ corresponds to Out(4).

10 Observations and Inferences

- The most dominant state was θ which corresponded to the horizontal sweep. This is directly inferred for the cost matrix Q and R , which are the inputs to the lqr method.
- We also noted that if we were to increase the cost of one particular(say for θ), while the system became faster in responding and adhered the constraints more rigidly, the system became less robust. So if for the lower cost system, even if we hit the pendulum hard, it was able to become stable, even if it had to break the constraints for horizontal and vertical deflection, but if the cost is made to high, if we hit it harder than a threshold, it lost its ability to come back to the initial configuration.

Therefore we had to a binary search to obtain a value which satisfied both of them.

- **θ and α Deviations:**
 - θ deviation:
 $\theta_{\max}(\text{left}) = -0.25 \text{ rad} = -14.26^\circ$

$$\theta_{\max}(\text{right}) = -0.46 \text{ rad} = 26^\circ$$

$$\text{Total Deviation} = 0.71 \text{ rad} = 40^\circ$$

– α deviation:

$$|\alpha| = 0.03 \text{ rad} = 1.8^\circ$$

11 Problems faced and solutions

- Getting a stable value. Even though the MATLAB routine showed that the system should die down to zero, in practice it never became stable. We changed Q and R matrices appropriately (mainly by hit and trial) to get the first stable solution. After this we were able to change the Q and R matrices systematically to get the stable inverted pendulum.
- We had to use a small delay in between two samplings. Extremely fast sampling rather deteriorated the performance, may be because reaction time of the inverted pendulum was much lesser than frequency of Arduino.

12 Why LQR is Chosen Over PID

LQR is chosen over PID control for several reasons, These reasons include:

- **Optimal Control:** LQR provides an optimal control solution by minimizing a cost function that balances state regulation and control effort. It systematically tunes the trade-off between system performance and energy consumption, which is difficult to achieve with a PID controller that relies on manually adjusting gains.
- **State-Space Representation:** LQR works efficiently in state-space models where multiple state variables (e.g., angular position and velocity) must be controlled simultaneously. In contrast, PID control is primarily effective for Single-Input Single-Output (SISO) systems and may not perform well in Multi-Input Multi-Output (MIMO) systems like the inverted pendulum.
- **Handling Coupled Dynamics:** In systems with coupled dynamics, such as the inverted pendulum where the base angle and pendulum arm angle interact, LQR can manage these interactions through the use of state feedback. PID control, on the other hand,

handles each state independently, which can lead to suboptimal performance or instability in coupled systems.

- **Robustness to Disturbances:** LQR inherently provides better robustness to external disturbances. By penalizing state deviations through the cost matrix Q , LQR ensures the system remains stable even under external disturbances. PID control can struggle to recover from large disturbances, often requiring additional tuning to maintain stability.
- **Tuning Simplicity:** In PID control, the gains K_p , K_i , and K_d need to be manually tuned, which can be time-consuming and require trial and error. LQR, on the other hand, computes an optimal feedback gain matrix K using mathematical algorithms based on the system's model, which simplifies the tuning process.
- **Performance for Nonlinear Systems:** For systems with nonlinear dynamics (like the inverted pendulum), LQR outperforms PID by taking into account the entire system's state and predicting the future trajectory, thus providing smoother and more accurate control. PID control, by contrast, can struggle with nonlinearities due to its reliance on past and current errors without predictive capabilities.

For these reasons, LQR is the preferred control method in systems like the inverted pendulum, where optimality, coupled dynamics, and robustness are essential considerations.

13 Conclusion

The application of LQR to this inverted pendulum control problem successfully demonstrated how optimal control techniques can stabilize inherently unstable systems. By carefully choosing the Q and R matrices, the LQR controller was able to maintain the pendulum in an upright position, even under disturbances, while minimizing control effort. The experiment highlighted the importance of balancing robustness and responsiveness in dynamic system control.

THANK YOU!