EE324: Control Systems Lab

Experiment 2: Inverted Pendulum

Thursday batch - Table 13

Jatin Kumar (22B3922)
Aayush Kumar (22B0769)
Archisman Bhattacharjee (22B2405)

October 3, 2024

1 Objective

The aim of this experiment is 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.
This is to be done within the following constraints:

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

2 Control Algorithm

Using matrices A, B, C, D, Q, R (given), we determine a matrix K.
For the state $x = [x_1, x_2, x_3, x_4]^T$ where $x_1 = \theta,\ x_2 = \alpha,\ x_3 = \frac{\partial\theta}{\partial t},\ x_4 = \frac{\partial\alpha}{\partial t}$.
The linear state-space representation of the ROTPEN Inverted Pendulum is given by

$$\frac{\mathrm{d}}{\mathrm{d}t}x(t) = Ax(t) + Bu(x)$$

$$y(t) = Cx(t) + Du(x)$$

The matrices A, B, C and D were given to us in the datasheet of the pendulum.
The LQR Problem is to minimize the cost function

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

where Q is a 4×4 diagonal matrix and R is a constant. We need to find K in the state feedback control law $u = Kx$ such that the cost function J is minimized.
After tweaking, we arrived at the values of Q and R to be:

$$Q = \begin{bmatrix} 300 & 0 & 0 & 0 \\ 0 & 250 & 0 & 0 \\ 0 & 0 & 100 & 0 \\ 0 & 0 & 0 & 50 \end{bmatrix}$$

$$R = [1]$$

$$K = [-17.3205, 716.8129, -33.7449, 96.9336]$$

State-Space Matrix	Expression
A	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{rM_p^2 l_p^2 g}{J_p J_{eq} + M_p l_p^2 J_{eq} + J_p M_p r^2} & -\frac{K_t K_m (J_p + M_p l_p^2)}{(J_p J_{eq} + M_p l_p^2 J_{eq} + J_p M_p r^2) R_m} & 0 \\ 0 & \frac{M_p l_p g (J_{eq} + M_p r^2)}{J_p J_{eq} + M_p l_p^2 J_{eq} + J_p M_p r^2} & \frac{-M_p l_p K_t r K_m}{(J_p J_{eq} + M_p l_p^2 J_{eq} + J_p M_p r^2) R_m} & 0 \end{bmatrix}$
B	$\begin{bmatrix} 0 \\ 0 \\ \frac{K_t (J_p + M_p l_p^2)}{(J_p J_{eq} + M_p l_p^2 J_{eq} + J_p M_p r^2) R_m} \\ \frac{M_p l_p K_t r}{(J_p J_{eq} + M_p l_p^2 J_{eq} + J_p M_p r^2) R_m} \end{bmatrix}$
C	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
D	$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

Figure 1: State Space matrices

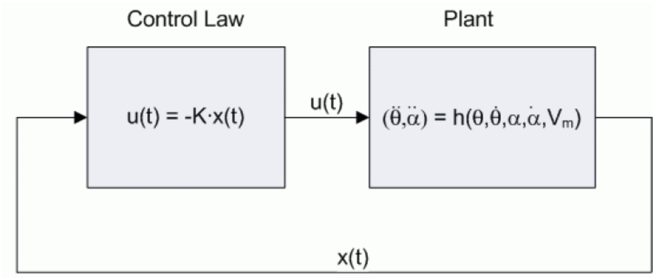

Figure 2: Closed loop control

3 Challenges Faced and their Solutions

- **Determining Values of Q and R:** There is no formula or algorithm to find the right values of Q and R for the system, We struggled with determining which state variables were causing problems **Resolution:** We started off with $diag[\frac{1}{\theta^2}, \frac{1}{\alpha^2}, 0, 0]$, and proceeded from there. Slight tweaking and intuition led us to the optimum values for Q and R matrix.

- **Extreme feedback initially**: Initial values of Q and R made the pendulum to act very violently, which made it very difficult to notice what was causing the problem
Resolution: Made small steps while changing and continuously plotted state variables to gauge what changes are taking place

- **Pendulum Specific Parameters:** We had to make sure we used the same pendulum each lab session **Resolution:** Used the same pendulum every lab and also ensured that the equilibrium points of alpha and theta were taken into account.

- **Saturation in optimisation** We reached a level after which it was very difficult to decrease the error further by any changes made
Resolution: Made very small changes to the Q and R values till we thought that an optimal value had been achieved

- During the course of the project, we encountered several challenges. The most significant difficulty was determining appropriate values for the Q and R matrices in the Linear Quadratic Regulator (LQR) control design. Without an initial estimate for these matrices, stabilizing the inverted pendulum system was challenging.

- Moreover, tuning the LQR controller required multiple iterations, as each small adjustment to the Q and R values resulted in a noticeable change in system behavior. Without a guiding framework or benchmark, this trial-and-error process was time-consuming and difficult.

- We also faced challenges in setting up reliable communication between the controller and the hardware system. Implementing bi-directional communication with the Arduino to continuously update and fine-tune the control parameters based on real-time feedback was a critical step, but it required overcoming synchronization and data transfer issues.

- Additionally, practical limitations, such as sensor noise and hardware imperfections, complicated the process. These introduced unmodeled disturbances in the system, making it harder to achieve the theoretical performance metrics initially expected.

- Despite these challenges, we successfully utilized Bryson's Rule to establish an approximate neighborhood for the Q and R matrices, enabling us to make progress in controller tuning and ultimately stabilize the system.

4 Arduino Code

The following is the code we flashed onto the Arduino.

```
#include <SPI.h>

#define BAUDRATE        115200

#define AMT22_NOP       0x00
#define AMT22_RESET     0x60
#define AMT22_ZERO      0x70

/* Define special ascii characters */
#define NEWLINE         0x0A
#define TAB             0x09

/* We will use these define macros so we can write code once compatible with 12 or 14
    bit encoders */
#define RES12           12
#define RES14           14

/* SPI pins */
#define ENC_0            2
#define ENC_1            3
#define SPI_MOSI        51
#define SPI_MISO        50
#define SPI_SCLK        52

/* Motor Controller Pins */
#define CW 24
#define ACW 25
#define pwm_pin 9

  float theta =0;
  float alpha =0;
  float theta1 =0;
  float alpha1 =0;
  float t1 = 0;
  float t2 = 0;
  float theta_dot = 0;
  float alpha_dot = 0;
  float k1 = -17.3205;
  float k2 = 716.8129;
  float k3 = -33.7449;
  float k4 = 96.9336;
  float u=0;
void setup()
{
  //Set the modes for the SPI IO
  pinMode(SPI_SCLK, OUTPUT);
  pinMode(SPI_MOSI, OUTPUT);
  pinMode(SPI_MISO, INPUT);
  pinMode(ENC_0, OUTPUT);
  pinMode(ENC_1, OUTPUT);

  //Initialize the UART serial connection for debugging
  Serial.begin(BAUDRATE);

  //Get the CS line high which is the default inactive state
  digitalWrite(ENC_0, HIGH);
  digitalWrite(ENC_1, HIGH);

  //set the clockrate. Uno clock rate is 16Mhz, divider of 32 gives 500 kHz.
  //500 kHz is a good speed for our test environment
  //SPI.setClockDivider(SPI_CLOCK_DIV2);   // 8 MHz
  //SPI.setClockDivider(SPI_CLOCK_DIV4);   // 4 MHz
  //SPI.setClockDivider(SPI_CLOCK_DIV8);   // 2 MHz
  //SPI.setClockDivider(SPI_CLOCK_DIV16);  // 1 MHz
  SPI.setClockDivider(SPI_CLOCK_DIV32);    // 500 kHz
  //SPI.setClockDivider(SPI_CLOCK_DIV64);  // 250 kHz
  //SPI.setClockDivider(SPI_CLOCK_DIV128); // 125 kHz

```

```
  //start SPI bus
  SPI.begin();
}

void loop()
{
  //create a 16 bit variable to hold the encoders position
  uint16_t encoderPosition;
  //let's also create a variable where we can count how many times we've tried to obtain the position in case there are errors
  uint8_t attempts;


  //if you want to set the zero position before beggining uncomment the following function call
//setZeroSPI(ENC_0);
//setZeroSPI(ENC_1);

  //once we enter this loop we will run forever
  while(1)
  {

    //set attemps counter at 0 so we can try again if we get bad position
    attempts = 0;

    //this function gets the encoder position and returns it as a uint16_t
    //send the function either res12 or res14 for your encoders resolution
    encoderPosition = getPositionSPI(ENC_0, RES14);

    //if the position returned was 0xFFFF we know that there was an error calculating the checksum
    //make 3 attempts for position. we will pre-increment attempts because we'll use the number later and want an accurate count
    while (encoderPosition == 0xFFFF && ++attempts < 3)
    {
      encoderPosition = getPositionSPI(ENC_0, RES14); //try again
    }

    if (encoderPosition == 0xFFFF) //position is bad, let the user know how many times we tried
    {
      Serial.print("Encoder 0 error. Attempts: ");
      Serial.print(attempts, DEC); //print out the number in decimal format. attempts - 1 is used since we post incremented the loop
      Serial.write(NEWLINE);
    }
    else //position was good, print to serial stream
    {
      if(float(encoderPosition)<8192){
        alpha = float(encoderPosition)*2*3.14/16384;
        }
      else{
        alpha = (float(encoderPosition)-16384)*2*3.14/16384;
        }
      Serial.print("Alpha  : ");
      Serial.print(alpha); //print the position in decimal format
      Serial.write(NEWLINE);

    }

    //////////again for second encoder////////////////////////////

    //set attemps counter at 0 so we can try again if we get bad position
    attempts = 0;

    //this function gets the encoder position and returns it as a uint16_t
    //send the function either res12 or res14 for your encoders resolution
    encoderPosition = getPositionSPI(ENC_1, RES14);

    //if the position returned was 0xFFFF we know that there was an error calculating the checksum
    //make 3 attempts for position. we will pre-increment attempts because we'll use the number later and want an accurate count
```

```
    while (encoderPosition == 0xFFFF && ++attempts < 3)
    {
      encoderPosition = getPositionSPI(ENC_1, RES14); //try again
    }

    if (encoderPosition == 0xFFFF) //position is bad, let the user know how many times we tried
    {
      Serial.print("Encoder 1 error. Attempts: ");
      Serial.print(attempts, DEC); //print out the number in decimal format. attempts - 1 is used since we post incremented the loop
      Serial.write(NEWLINE);
    }
    else //position was good, print to serial stream
    {
      if(float(encoderPosition)<8192){
        theta = float(encoderPosition)*2*3.14/16384;
        }
      else{
        theta = (float(encoderPosition)-16384)*2*3.14/16384;
        }
      Serial.print("Theta  : ");
      Serial.print(theta); //print the position in decimal format
      Serial.write(NEWLINE);

    }
    t1 = millis()/1000.0;
    theta_dot = (theta - theta1)/(t1-t2);
    alpha_dot = (alpha - alpha1)/(t1-t2);
    theta1 = theta;
    alpha1 = alpha;
    t2 = t1;
    u = -1*((k1*theta) + (k2*alpha) + (k3*theta_dot) + (k4*alpha_dot));

    u = constrain(u, -12, 12);
    u=u-0.27;
    u+= 0.27;
    Serial.print("Alpha_dot");
    Serial.println(alpha_dot);
    Serial.print("Theta_dot");
    Serial.println(theta_dot);
    Serial.print("u : ");
    Serial.println(u);

    if(u>0){
      digitalWrite(CW, HIGH);
      digitalWrite(ACW, LOW);
      analogWrite(pwm_pin, u*255/12);
      }
    else{
      digitalWrite(CW, LOW);
      digitalWrite(ACW, HIGH);
      analogWrite(pwm_pin, abs(u)*255/12);
      }

    //For the purpose of this demo we don't need the position returned that quickly so let's wait a half second between reads
    //delay() is in milliseconds
    delay(15);
  }
}

/*
 * This function gets the absolute position from the AMT22 encoder using the SPI bus. The AMT22 position includes 2 checkbits to use
 * for position verification. Both 12-bit and 14-bit encoders transfer position via two bytes, giving 16-bits regardless of resolution.
 * For 12-bit encoders the position is left-shifted two bits, leaving the right two bits as zeros. This gives the impression that the encoder
 * is actually sending 14-bits, when it is actually sending 12-bit values, where every number is multiplied by 4.
 * This function takes the pin number of the desired device as an input
 * This funciton expects res12 or res14 to properly format position responses.
```

```
 * Error values are returned as 0xFFFF
 */
uint16_t getPositionSPI(uint8_t encoder, uint8_t resolution)
{
  uint16_t currentPosition;       //16-bit response from encoder
  bool binaryArray[16];           //after receiving the position we will populate this array and use it for calculating the checksum

  //get first byte which is the high byte, shift it 8 bits. don't release line for the first byte
  currentPosition = spiWriteRead(AMT22_NOP, encoder, false) << 8;

  //this is the time required between bytes as specified in the datasheet.
  //We will implement that time delay here, however the arduino is not the fastest device so the delay
  //is likely inherantly there already
  delayMicroseconds(3);

  //OR the low byte with the currentPosition variable. release line after second byte
  currentPosition |= spiWriteRead(AMT22_NOP, encoder, true);

  //run through the 16 bits of position and put each bit into a slot in the array so we can do the checksum calculation
  for(int i = 0; i < 16; i++) binaryArray[i] = (0x01) & (currentPosition >> (i));

  //using the equation on the datasheet we can calculate the checksums and then make sure they match what the encoder sent
  if ((binaryArray[15] == !(binaryArray[13] ^ binaryArray[11] ^ binaryArray[9] ^ binaryArray[7] ^ binaryArray[5] ^ binaryArray[3] ^ binaryArray[1]))
          && (binaryArray[14] == !(binaryArray[12] ^ binaryArray[10] ^ binaryArray[8] ^ binaryArray[6] ^ binaryArray[4] ^ binaryArray[2] ^ binaryArray[0])))
    {
      //we got back a good position, so just mask away the checkbits
      currentPosition &= 0x3FFF;
    }
  else
  {
    currentPosition = 0xFFFF; //bad position
  }

  //If the resolution is 12-bits, and wasn't 0xFFFF, then shift position, otherwise do nothing
  if ((resolution == RES12) && (currentPosition != 0xFFFF)) currentPosition = currentPosition >> 2;

  return currentPosition;
}

/*
 * This function does the SPI transfer. sendByte is the byte to transmit.
 * Use releaseLine to let the spiWriteRead function know if it should release
 * the chip select line after transfer.
 * This function takes the pin number of the desired device as an input
 * The received data is returned.
 */
uint8_t spiWriteRead(uint8_t sendByte, uint8_t encoder, uint8_t releaseLine)
{
  //holder for the received over SPI
  uint8_t data;

  //set cs low, cs may already be low but there's no issue calling it again except for extra time
  setCSLine(encoder ,LOW);

  //There is a minimum time requirement after CS goes low before data can be clocked out of the encoder.
  //We will implement that time delay here, however the arduino is not the fastest device so the delay
  //is likely inherantly there already
  delayMicroseconds(3);

  //send the command
  data = SPI.transfer(sendByte);
```

```
  delayMicroseconds(3); //There is also a minimum time after clocking that CS should remain asserted before we release it
  setCSLine(encoder, releaseLine); //if releaseLine is high set it high else it stays low

  return data;
}

/*
 * This function sets the state of the SPI line. It isn't necessary but makes the code more readable than having digitalWrite everywhere
 * This function takes the pin number of the desired device as an input
 */
void setCSLine (uint8_t encoder, uint8_t csLine)
{
  digitalWrite(encoder, csLine);
}

/*
 * The AMT22 bus allows for extended commands. The first byte is 0x00 like a normal position transfer, but the
 * second byte is the command.
 * This function takes the pin number of the desired device as an input
 */
void setZeroSPI(uint8_t encoder)
{
  spiWriteRead(AMT22_NOP, encoder, false);

  //this is the time required between bytes as specified in the datasheet.
  //We will implement that time delay here, however the arduino is not the fastest device so the delay
  //is likely inherantly there already
  delayMicroseconds(3);

  spiWriteRead(AMT22_ZERO, encoder, true);
  delay(250); //250 second delay to allow the encoder to reset
}

/*
 * The AMT22 bus allows for extended commands. The first byte is 0x00 like a normal position transfer, but the
 * second byte is the command.
 * This function takes the pin number of the desired device as an input
 */
void resetAMT22(uint8_t encoder)
{
  spiWriteRead(AMT22_NOP, encoder, false);

  //this is the time required between bytes as specified in the datasheet.
  //We will implement that time delay here, however the arduino is not the fastest device so the delay
  //is likely inherantly there already
  delayMicroseconds(3);

  spiWriteRead(AMT22_RESET, encoder, true);

  delay(250); //250 second delay to allow the encoder to start back up
}
```

Listing 1: Arduino Code for PID Control

5 Conclusion and Inference

5.1 Performance Metrics

- The LQR controller successfully achieved the desired performance criteria, with the pendulum arm vibration (α) maintained within $\pm 3°$ and the base angle oscillation (θ) within $\pm 30°$.
- The final tuned LQR parameters were:
  - $K_\alpha = -17.3205$
  - $K_\theta = 716.8129$
  - $K_{\dot{\alpha}} = -33.7449$
  - $K_{\dot{\theta}} = 96.9336$

  These values provided a balance between responsiveness and stability, as observed in the final response plots.

5.2 Bi-Directional Communication with Arduino

- The implementation of bi-directional communication helped us to track the error term and update the penalty terms in the Q matrix, which generated modified proportionality constants and stabilized the inverted pendulum.
- By performing this iteratively, we were able to successfully stabilize the Inverted Pendulum.

6 Results

The following graphs show the step-response of the LQR-stabilized mode:

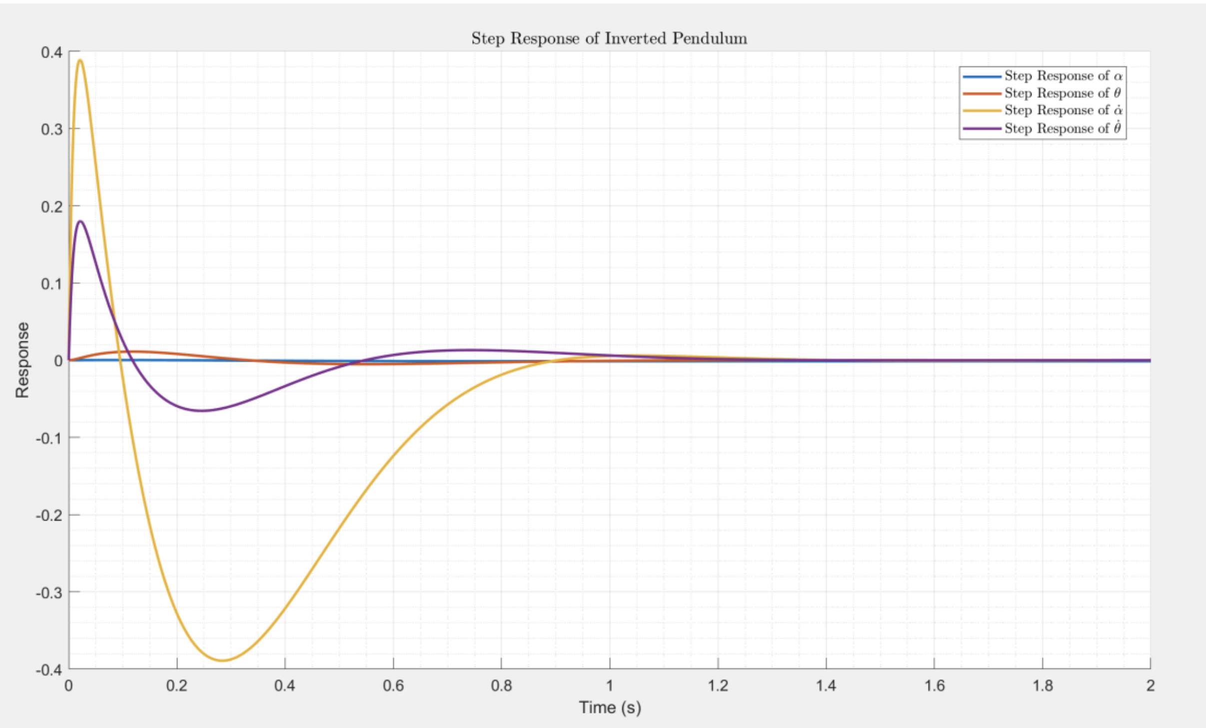

Figure 3: Step Response