EE 324: Experiment 2
Inverted Pendulum

Sravan K Suresh	22B3936
Swarup Dasharath Patil	22B3953
Amol Milind Pagare	22B3971

October 10, 2024

Contents

1 Objective

To design and implement a control action for maintaining a pendulum in the upright position, even when subjected to external disturbances, using the Linear Quadratic Regulator (LQR) technique on an Arduino Mega. The specific goals are:

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

2 Control Algorithm

blablablablablablablablablabla
The Arduino code we used is given below-

```
/* Include the SPI library for the arduino boards */
#include <SPI.h>

/* Serial rates for UART */
#define BAUDRATE        115200

/* SPI commands */
#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

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);

  // pinMode(11, OUTPUT);  // PWM output to motor
  pinMode(10, OUTPUT);  // Motor direction control 1
  pinMode(8, OUTPUT);
  // Arduino 4 --> IC EN (1)
  // Arduino 6 --> IC 2
  // Arduino 5 --> IC 7


  //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 encoderPosition0, encoderPosition1;
  //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;
  float theta, alpha;
  float start_pos_arm = (float)getPositionSPI(ENC_0, RES14)*360/16383;
//  float start_pos_arm=180;
  float error_pendulum_cur,error_arm_cur, error_pendulum_prev,error_arm_prev;
  float velocity_arm, velocity_pendulum, Vm_out, min_error_arm, max_error_arm, min_error_pend, max_error_pend;
  int fbsignal;
  float k[4] = {-7.8000 , 190.7521 ,  -6.8396 ,  26.3879};
//float k[4] = {-10.0000 , 190.7521 ,  -6.8396 ,  26.3879};


  //if you want to set the zero position before beggining uncomment the following function call
//  setZeroSPI(ENC_0);
//  setZeroSPI(ENC_1);
  encoderPosition1 = getPositionSPI(ENC_1, RES14);
  encoderPosition0 = getPositionSPI(ENC_0, RES14);
  theta = (float)encoderPosition0*360/16383;
  alpha = (float)encoderPosition1*360/16383;


  error_pendulum_prev = alpha - 180;
```

```
  error_arm_prev = theta - start_pos_arm;
  min_error_arm = error_arm_prev;
  max_error_arm = error_arm_prev;

  min_error_pend = error_pendulum_prev;
  max_error_pend = error_pendulum_prev;
  //once we enter this loop we will run forever
  while(1){

    encoderPosition0 = getPositionSPI(ENC_0, RES14);
    encoderPosition1 = getPositionSPI(ENC_1, RES14);

    theta = (float)encoderPosition0*360/16383;
    alpha = (float)encoderPosition1*360/16383;
    alpha += 10;

    error_pendulum_cur = alpha - 180;
    error_arm_cur = theta - start_pos_arm;
    velocity_pendulum = (error_pendulum_cur - error_pendulum_prev)/0.020;
    velocity_arm = (error_arm_cur - error_arm_prev)/0.020;
    //LQR CODE
    Vm_out = (k[0]*error_arm_cur + k[1]*error_pendulum_cur + k[2]*velocity_arm
       + k[3]*velocity_pendulum)*3.1415926535/180;

    fbsignal =map(abs(Vm_out),0,12,0,255);
//   fbsignal =min((int) abs(Vm_out*390/max_vm)+190,255);
//     fbsignal = (int) abs(Vm_out*255/12);
    //Serial.println(fbsignal);
    if(Vm_out>0){
      analogWrite(8, constrain(fbsignal,0,255));  // Set motor direction
      analogWrite(10, 0);
    }
    else{
      analogWrite(10, constrain(fbsignal,0,255));  // Set motor direction
      analogWrite(8, 0);
    }

//     Serial.println(fbsignal);
//     Serial.println(error_arm_cur);
//     Serial.println(error_pendulum_cur);
  min_error_arm = min(min_error_arm,error_arm_cur);
  max_error_arm = max(max_error_arm,error_arm_cur);

  min_error_pend = min(min_error_pend,error_pendulum_cur);
  max_error_pend = max(max_error_pend,error_pendulum_cur);
  Serial.print("Error: ");
  Serial.print(error_pendulum_cur);

   Serial.print("| Theta: ");
   Serial.print(theta);
   Serial.print("| Alpha: ");
   Serial.println(alpha);
  //  Serial.print("| Encoder 0: ");
  //  Serial.print(encoderPosition0);
  //  Serial.print("| Encoder 1: ");
  //  Serial.println(encoderPosition1);
```

```
  // //  Serial.print("  Min Err Arm:  ");
  //  Serial.print(min_error_arm);
  //  Serial.print(" Max Err Arm: ");
  //  Serial.print(max_error_arm);
  //  Serial.print(" Min Err Pend: ");
  //  Serial.print(min_error_pend);
  //  Serial.print(" Max Err Pend: ");
  //  Serial.println(min_error_pend);

//     Serial.println(encoderPosition0);
//     Serial.println(encoderPosition1);
    error_pendulum_prev = error_pendulum_cur;
    error_arm_prev=error_arm_cur;
    delay(20);
  }
}

/*
 * 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 PID Control Code

3 Challenges Encountered

- Pendulum hi pendulum
- Pendulum hi pendulum

4 Results

XYZ controller parameters:-

- $K_i = 10^{-5}$
- $K_d = 10^{-2}$
- $K_p = 1$

Design Specifications:-

- **Rise time** = 351 ms
- **Settling time** = 543 ms
- **% overshoot** = 2.87 %

All of these parameters fall within our requirements from the controller.Thus, our designed controller is valid.

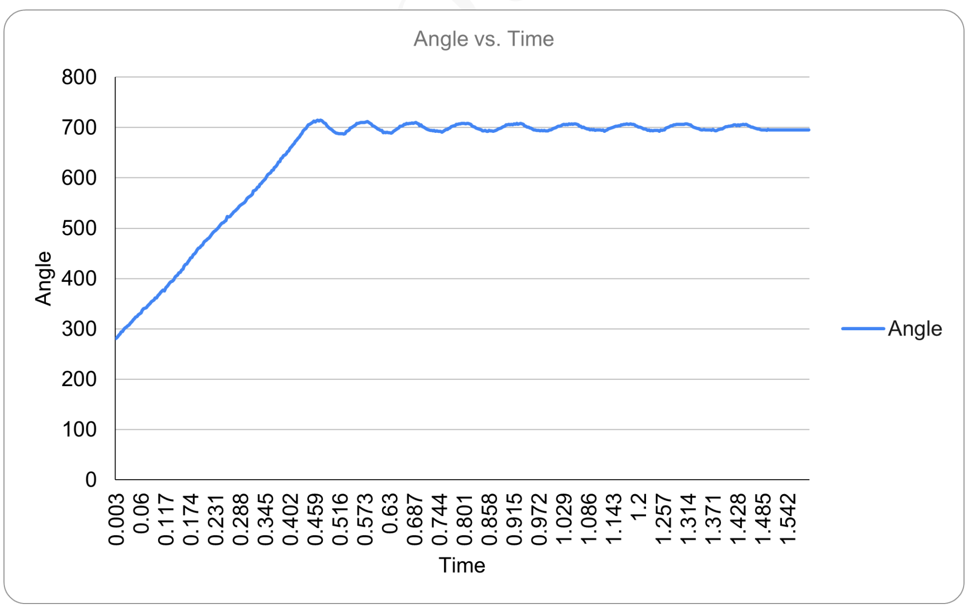

Figure 1: Response

5 Observations and Inferences

- The PID controller successfully achieved the desired performance criteria with a rise time within 0.5 seconds, settling time within 1 second, and overshoot under 10%.
- The final tuned PID parameters were Kp = 1, Kd = 10^{-2}, and Ki = 10^{-5}. These values provided a balance between responsiveness and stability, as observed in the final response plots.