State Space Speed Controller

Samarthya Bhatnagar and Soham Gadgil

ECE 4181 Fall 2018

System Requirements

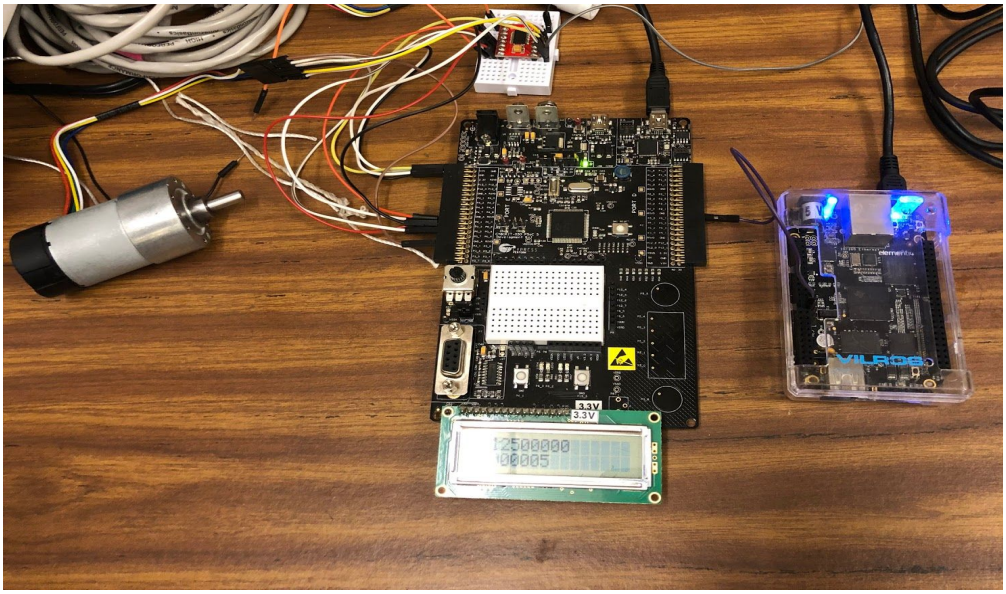
This project involves the development of a motor speed controller on the PSoC5 board. At the start of the program a reference speed is set for the motor to maintain. Then, if a load is placed on the motor or the reference speed is changed, the motor tries to adjust itself to account for the load or to get to the new reference speed. For example, if a load is added to the motor, it tries to provide a higher PWM value to bring it back to the reference speed.

System Specification

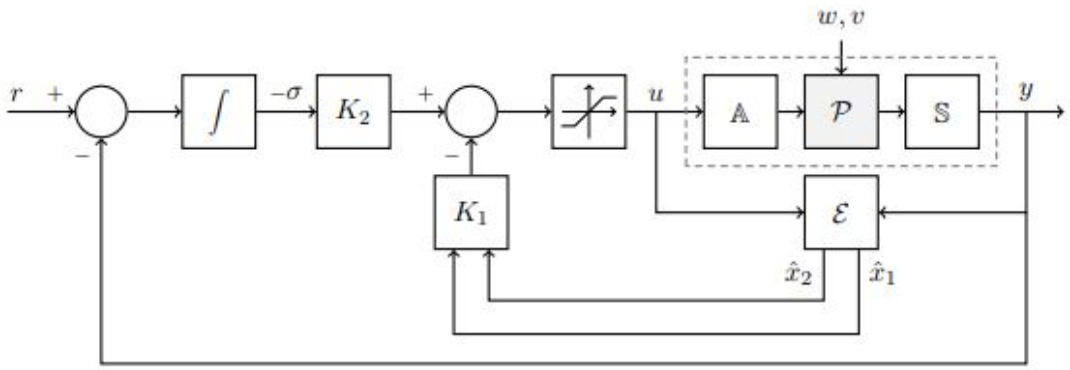
The following lists the components used in the project:
1) The project uses a PSOC5 LP035 board with a Cortex M3 CPU
2) The motor used is a Pololu 12V brushed DC motor with a 64 CPR encoder and a 30:1 gear ratio corresponding to 1920 counts per revolution.

The following procedure is followed to adjust the PWM values:
1) Set the reference speed
2) Obtain the ticks from the two encoders on the motor and calculate speed.
3) Use the controller module to obtain the error between the current motor speed and the reference speed
4) Adjust the PWM value to account for the error
5) Repeat the steps until the error is minimized.

The following is an image of the system setup:



The following is a model of the controller that we implemented:



Some things to note:
u - represents the controller input
y- represents the motor output
w,v - represents the noise from the motor and encoder readings
r - represents the reference signal to follow
sigma - represents the error from the current motor output
K1, K2 represent gain matrices that are applied on the input signal
epsilon - represents the estimator, which computes the estimated speed of the motor

The following is the model equations used in this controller scheme:

$$\begin{bmatrix}\dot{x}(t)\\ \dot{\sigma}(t)\\ \dot{\varepsilon}(t)\end{bmatrix} = \underbrace{\begin{bmatrix}A-BK_1 & -BK_2 & -BK_1\\ C & 0 & 0\\ 0 & 0 & A-LC\end{bmatrix}}_{A_a}\begin{bmatrix}x(t)\\ \sigma(t)\\ \varepsilon(t)\end{bmatrix} + \underbrace{\begin{bmatrix}0 & E & 0\\ -1 & 0 & 1\\ 0 & -E & L\end{bmatrix}}_{B_a}\begin{bmatrix}r(t)\\ w(t)\\ v(t)\end{bmatrix}$$

$$y(t) = \underbrace{\begin{bmatrix}C & 0 & 0\end{bmatrix}}_{C_a}\begin{bmatrix}x(t)\\ \sigma(t)\\ \varepsilon(t)\end{bmatrix} + \underbrace{\begin{bmatrix}0 & 0 & 1\end{bmatrix}}_{D_a}\begin{bmatrix}r(t)\\ w(t)\\ v(t)\end{bmatrix}.$$

This represents the overall system in state space form. The gains K1, K2, and L are configured so as to model the motor physics. Computing the following determinants will yield lambda values for the system:

$$\underbrace{\det\left(sI-(\mathcal{A}-\mathcal{B}K)\right)}_{\text{what we have}} = \underbrace{(s+\lambda_r)^{n+1}}_{\text{what we want}}$$

$$\underbrace{\det\left(sI-(A-LC)\right)}_{\text{what we have}} \quad = \quad \underbrace{(s+\lambda_e)^n}_{\text{what we want}}$$

Once the lambda values are calculated, we can calculate the gain matrices.

The controller equations are approximated by the following equations:

$$\begin{aligned}
\hat{x}_1[k+1] &= \hat{x}_1[k] + T\hat{x}_2[k] - TL_1\left(\hat{x}_1[k] - y[k]\right) \\
\hat{x}_2[k+1] &= \hat{x}_2[k] - T\alpha\hat{x}_2[k] + T\beta u[k] - TL_2\left(\hat{x}_1[k] - y[k]\right) \\
\sigma[k+1] &= \sigma[k] + T\left(y[k] - r[k]\right).
\end{aligned}$$

For our gain matrices, once the lambda values are obtained it is possible to compute K1, K2, and L:

$$K_{11} = \frac{1}{\beta}3\lambda_r^2, \quad K_{12} = \frac{1}{\beta}(3\lambda_r - \alpha), \quad K_2 = \frac{1}{\beta}\lambda_r^3.$$

$$L_1 = 2\lambda_e - \alpha, \quad L_2 = \lambda_e^2 - 2\alpha\lambda_e + \alpha^2.$$

where alpha and beta are pre-determined constants representing the motor physics.

Class and Object Diagrams

We make use of two classes:
(i) Controller Class - used to update the PWM values of the motor in accordance to the current speed and the reference speed
(ii) Encoder Class - used to obtain the encoder ticks and update the encoderCounter

Controller Class Diagram

Controller	
R	(reference signal that we want to follow)
X_hat	(Computed Model of motor physics, X)
Y_hat	(Computed output based on X_hat)
U	(Output of the Controller based on R and Y_hat, Y)
Y	(Output of the motor)
sigma	(Error in the motor speed)
speed	(Current motor speed)
updateController()	(compute X_hat, Y_Hat, U, sigma, speed)

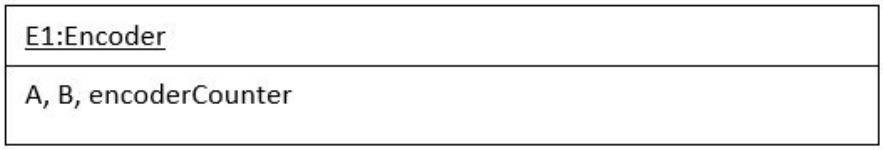
Controller Object Diagram

C1:Controller
R, X_hat, Y_hat, U, Y, sigma, speed

Encoder Class Diagram

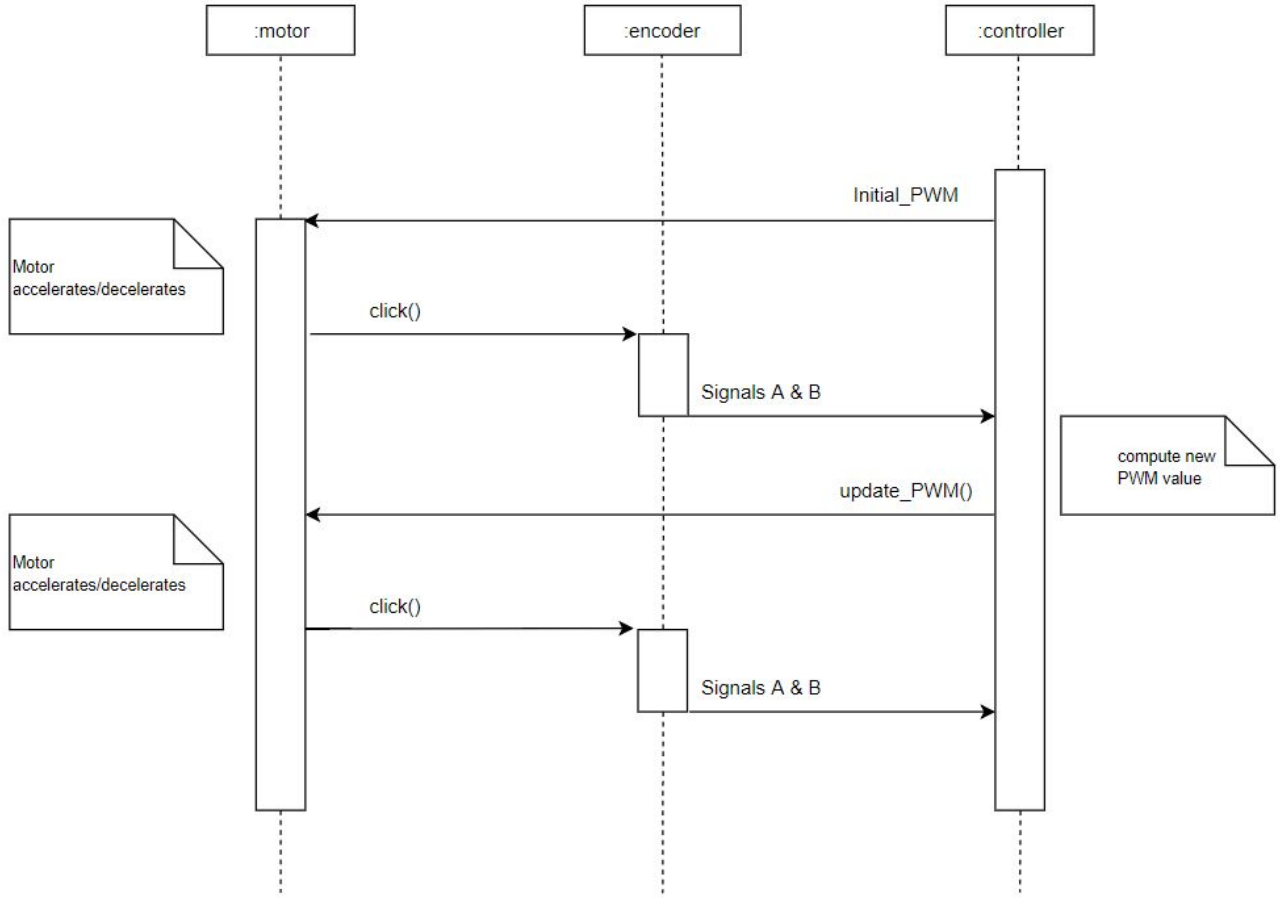
Encoder	
A	(Encoder 1 Output)
B	(Encoder 2 Output)
encoderCounter	(calculate number of encoder ticks)
pollEncoder	(Calculate encoder ticks using A, B and update encoderCounter)

Encoder Object Diagram



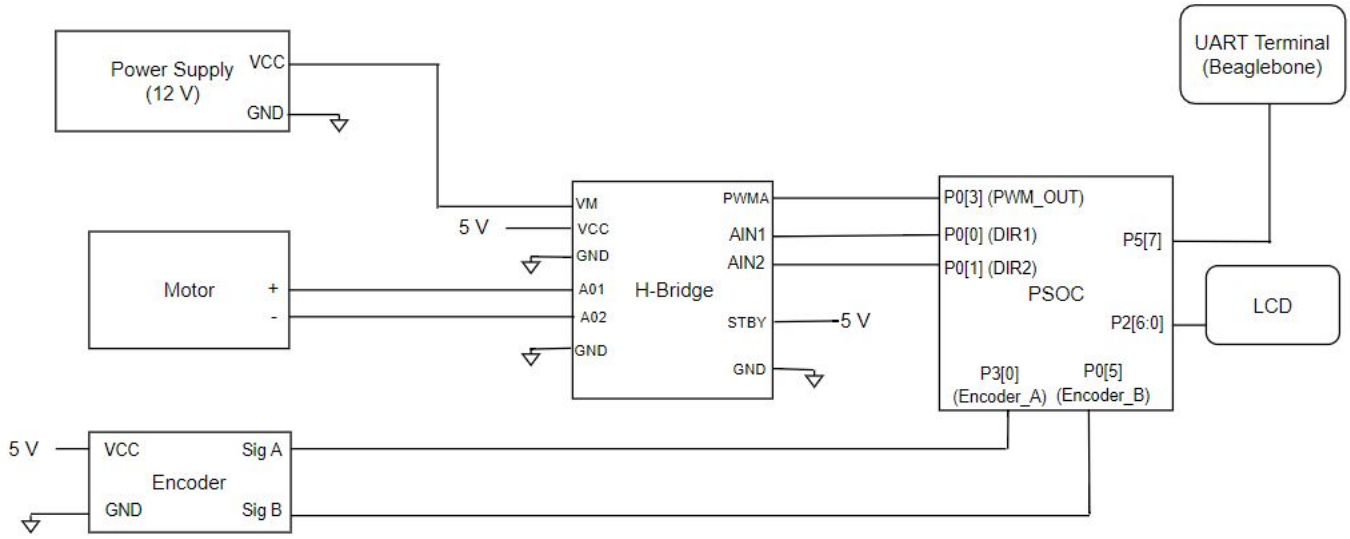
Sequence Diagram

Following is the sequence diagram for two clicks of the motor:

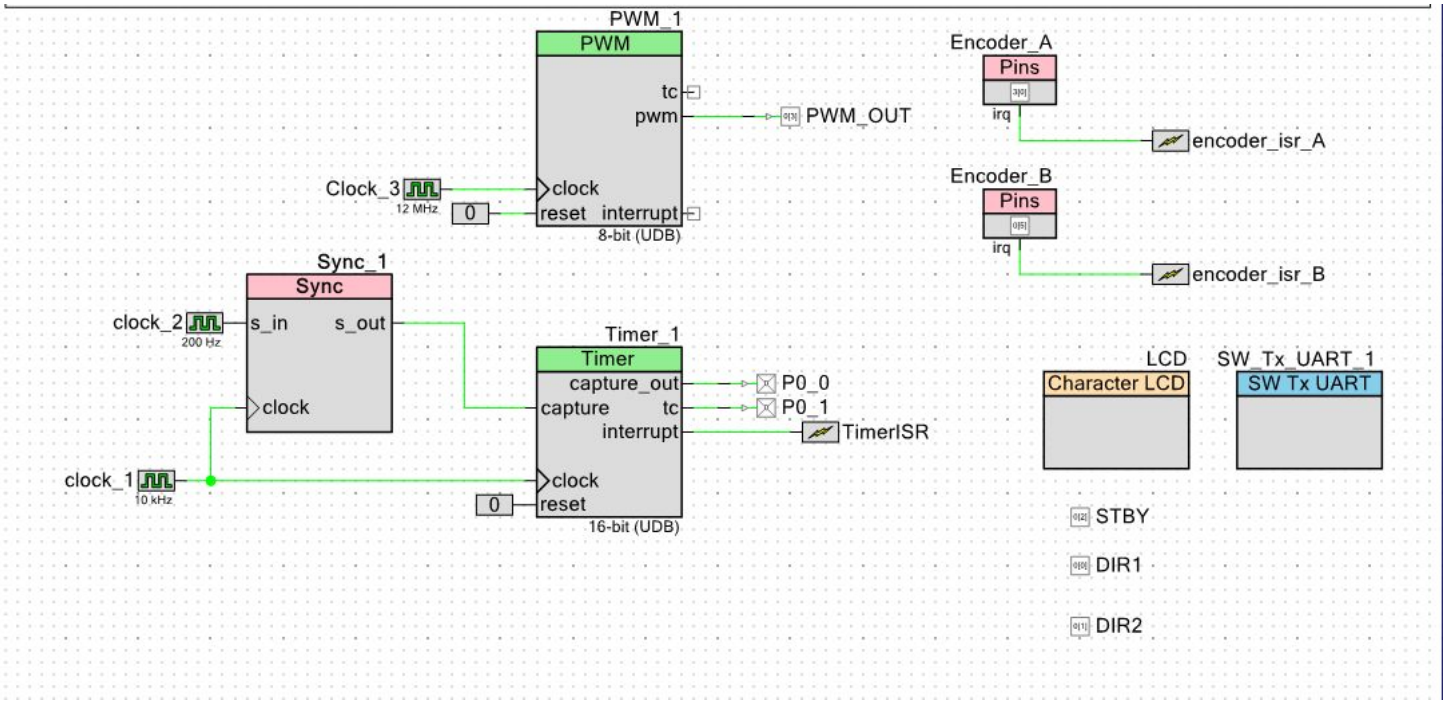


Schematics

Following is the circuit schematic diagram:



Following is the PSoC schematic:



The timer interrupt executes at the rate of 1 KHz (once per millisecond). This value was chosen after testing different rates since it worked well for real world simulations and controller update.

Code For All Functions

See Appendix A for Psoc Code, Appendix B for Python code for getting the data values, Appendix C for Matlab Code for plotting the graph.

Module Testing Procedure

The State Space Controller was tested by providing different reference trajectories and monitoring the response of the controller. Initially, a reference of 1 revolution/second was given to the controller, and then the reference changes to 4.5 revolutions/second after 4 seconds. This is done in the second stage of the updateController function, shown below, after the computation of the current speed and before the controller update computation.

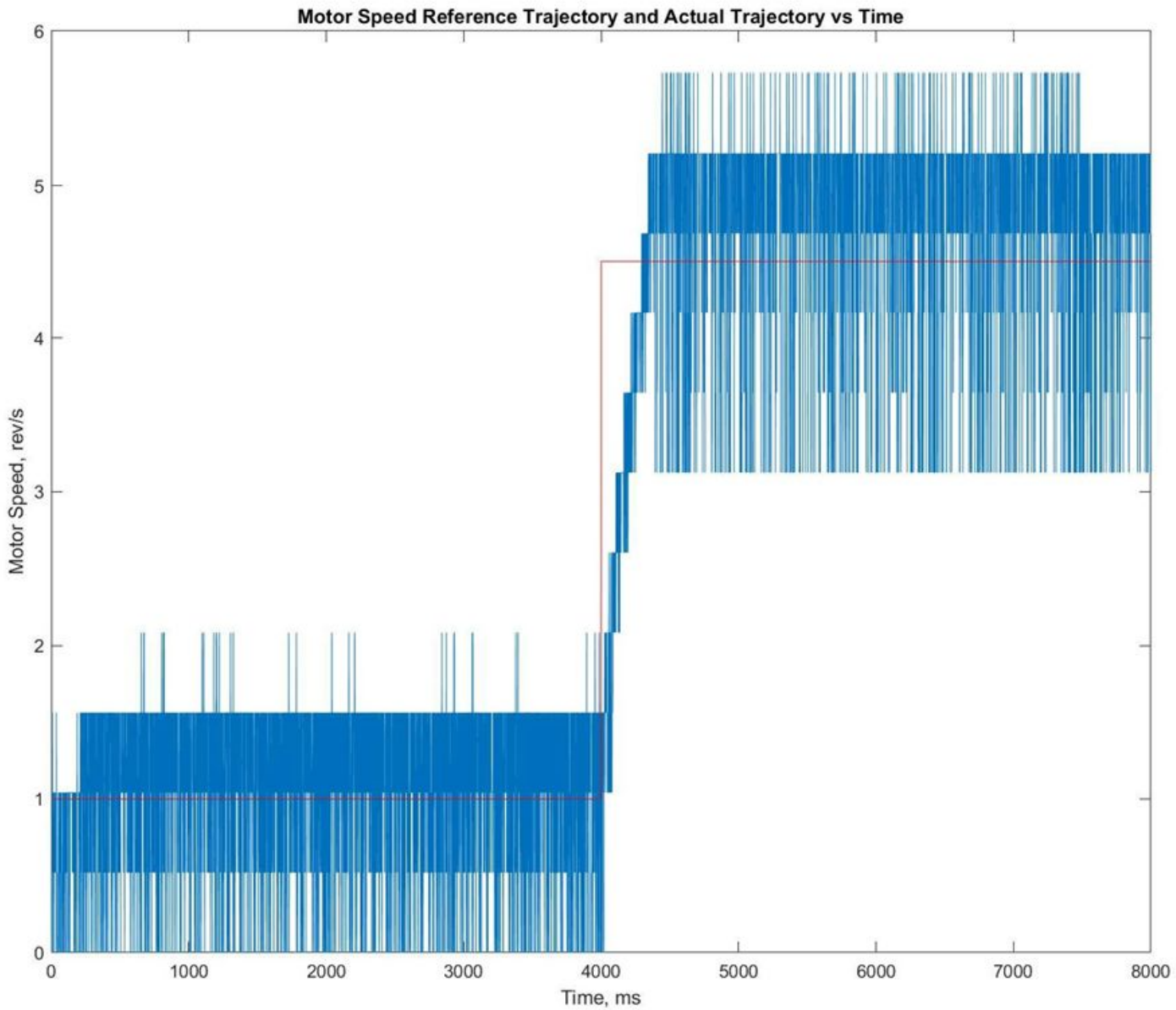
```
void updateController() {
    speed = encoderCounter * (1.0/1920.0) * 1000.0;
    encoderCounter = 0;
    r = 1.0;
    if ((pwmCounter > 4000)) {
        r = 4.5;
    }
    v_motor = -K11 * x1_hat - K12 * x2_hat - K2 * sigma;
```

The motor speed values are sampled at the same rate as the controller update, therefore we read the speed value and compute the controller output once every millisecond. The controller then stops once it reaches the max value of the array so as to avoid overflow. Then, the controller update function sets the sendDataFlag to high, and the scheduler calls the sendData functions as shown below.

```
speedLog[pwmCounter] = speed;
pwmCounter++;
if (pwmCounter > 8000) {
    pwmCounter = 8000;
    sendDataFlag = 1;
}
```

The sendData function transmits the speedLog using the UART pin. The data is sent to the Beaglebone Black, which has a python program running that caputres the data. The python program logs the data to a text file, which is then transferred to the host computer for processing. See Appendix B for python code.

The text file is fed into Matlab, where it can be processed and graphed. See appendix C for the Matlab script used to graph the data. The following is a graph of actual motor speed (Blue) and reference speed (Red) vs Time:


Motor Speed Reference Trajectory and Actual Trajectory vs Time
Motor Speed, rev/s
Time, ms
0
1
2
3
4
5
6
0
1000
2000
3000
4000
5000
6000
7000
8000

Results

As depicted in the graph, the controller follows the reference signal, but is oscillating frequently. There could be several reasons for this:

1. At 60 Mhz, the PSOC board just barely meets the requirements to fulfill this speed controller. Preferably, the controller would have a 200 Mhz System clock.
2. The encoder interrupt and timer interrupt have to be separate, but it is possible that they are interfering with each other. This was compensated by adjusting the encoder interrupt to have a higher CPU priority than the timer interrupt, but it is now possible that the encoder interrupt is interfering with the controller update. This was a compromise that had to be made, as we desired accurate speed values vs accurate controller update.
3. The PSOC base functions are very slow, and the LCD print functions slow down the execution of the main loop.
4. Last but not least, the controller constants have not been tuned specifically for this motor as we did not have enough time to develop a testing script. Therefore, we are using the "ideal values" for the controller physics, rather than testing and using the real world motor physics. This could defer from motor to motor, and it is desirable in the future to tune each motor individually.

Despite these reasons, the controller did respond to the change in reference signal, and attempted to adjust accordingly. The controller stayed with the new reference input, and oscillated back and forth around the reference value. This following video shows the motor adapting to the reference speed. The LCD shows the reference speed and the current speed of the motor. Please watch the video in full screen to clearly view the speeds. The link to the video is https://youtu.be/-zsoROHGO3s

Future Work

1. The motor needs to be tuned better to account for the real world motor physics of the specific motor that is used
2. Ideally, a more powerful MCU is used to calculate the controller update
3. Psoc must implement a better method of exporting data

Appendix

Appendix A

```
#include <project.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int encoderCounter = 0;
int Counter = 0;
int pwmCounter = 0;
int pwmValue = 0;
int dir = 0;
int flag = 0;
int encoderFlag = 0;
int sendDataFlag = 0;
uint16 duty_cycle;
float32 r = 0.0;
float32 v_motor = 0;
float32 period = 0.001;
float32 sigma = 0;
float32 x1_hat = 0;
float32 x2_hat = 0;
float32 speed = 0;
float32 alpha = 127.0865;
float32 beta = 751.8797;
float32 K11 = 9.975;
float32 K12 = 0.035;
float32 K2 = 166.25;
float32 L11 = 272.91;
float32 L21 = 5316.4;
float32 speedLog[8000];
const int8_t encoder_table[] = {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};
uint8_t enc_val = 0;

/*******************************************************************************
*
* Define Interrupt service routines for timer and encoders
*******************************************************************************
*/
CY_ISR(InterruptHandler)
{
      Timer_1_STATUS;
    flag = 1;
}
```

```
void encoder_isr() {
    Encoder_A_ClearInterrupt();
    Encoder_B_ClearInterrupt();
    encoderFlag = 1;
}

/********************************************************************************
*
* Functions to convert float to string for printing on the LCD
********************************************************************************
*/
void reverse(char *str, int len)
{
    int i=0, j=len-1, temp;
    while (i<j)
    {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
        i++; j--;
    }
}
int intToStr(int x, char str[], int d)
{
    int i = 0;
    while (x)
    {
        str[i++] = (x%10) + '0';
        x = x/10;
    }

    // If number of digits required is more, then
    // add 0s at the beginning
    while (i < d)
        str[i++] = '0';

    reverse(str, i);
    str[i] = '\0';
    return i;
}
void ftoa(float n, char *res, int afterpoint)
{
    // Extract integer part
    int ipart = (int)n;

    // Extract floating part
    float fpart = n - (float)ipart;

    // convert integer part to string
    int i = intToStr(ipart, res, 0);

    // check for display option after point
    if (afterpoint != 0)
```

```
    {
        res[i] = '.';  // add dot

        // Get the value of fraction part upto given no.
        // of points after dot. The third parameter is needed
        // to handle cases like 233.007
        fpart = fpart * pow(10, afterpoint);

        intToStr((int)fpart, res + i + 1, afterpoint);
    }
}

/*****************************************************************************
*
* Scheduler Functions to update controllers, poll encoders, and send data
*****************************************************************************
*/

void updateController() {
    //Calculate current motor speed
    speed = encoderCounter * (1.0/1920.0) * 1000.0;
    encoderCounter = 0;
    r = 1.0;
    if ((pwmCounter > 4000)) {
        r = 4.5;
    }
    v_motor = -K11 * x1_hat - K12 * x2_hat - K2 * sigma;
    if (v_motor > 12) {
        v_motor = 12;
    }
    x1_hat = x1_hat + .001 * x2_hat - 0.001 * L11 * (x1_hat - speed);
    x2_hat = x2_hat - .001 * alpha * x2_hat + .001 * beta * v_motor - .001 *
L21 * (x1_hat - speed);
    sigma = sigma + .001 * (speed - r);
    duty_cycle = (uint16)(v_motor/12 * 255);
    PWM_1_WriteCompare(duty_cycle);
    speedLog[pwmCounter] = speed;
    pwmCounter++;
    //set sendDataFlag high for UART transfer
    if (pwmCounter > 8000) {
        pwmCounter = 8000;
        sendDataFlag = 1;
    }
}

void pollEncoder() {
    enc_val = enc_val << 2; // shift the previous state to the left
    uint8_t currState = (0b0010 & Encoder_A_Read()<<1) | (0b0001 &
Encoder_B_Read());
    enc_val = enc_val | currState; // or the current state into the 2 rightmost
bits
    encoderCounter += encoder_table[enc_val & 0b1111];    // preform the table
lookup and increment count accordingly
}
```

```
void sendData() {
    if (flag == 1) {
        while(1) {
            Timer_1_Stop();
            encoder_isr_A_Stop();
            encoder_isr_B_Stop();
            int i = 0;
            for (i = 0; i < 8000; i++) {
                char arr[20];
                ftoa(speedLog[i], arr, 8);
                SW_Tx_UART_1_PutString(arr);
                CyDelay(1);
                SW_Tx_UART_1_PutString("\n");
                CyDelay(1);
            }
            while(1){}
        }
    }
}

/*******************************************************************************
*
* Function Name: main
********************************************************************************
/

int main()
{
    PWM_1_Start();
    DIR1_Write(1u);
    DIR2_Write(0u);
    STBY_Write(1u);

    SW_Tx_UART_1_Start();

    encoder_isr_A_StartEx(encoder_isr);
    encoder_isr_B_StartEx(encoder_isr);

    LCD_Start();
    LCD_Position(0, 0);
    LCD_PrintString("Speed:");

    LCD_Position(1, 0);
    LCD_PrintString("Ref:");

    /* Enable the Interrupt component connected to Timer interrupt */
    TimerISR_StartEx(InterruptHandler);

      /* Start the Timer */
```

```
    Timer_1_Start();


    CyGlobalIntEnable;

    for(;;)
    {
        Counter++;
        if (Counter % 1000 == 0) {
            char arr[20];
            ftoa(speed, arr, 4);
            LCD_Position(0, 8);
            LCD_PrintString(arr);
            LCD_Position(1, 6);
            LCD_PrintInt32((int)r);
        }
        //update the controller on every timer interrupt (1 ms)
        if (flag == 1) {
            flag = 0;
            updateController();
        }
        //poll the encoders when either encoder A or encoder B ticks
        if (encoderFlag == 1) {
            encoderFlag = 0;
            pollEncoder();
        }
        if (sendDataFlag == 1) {
            sendDataFlag = 0;
            sendData();
        }
    }

}


/* [] END OF FILE */
```

Appendix B

```
import Adafruit_BBIO.UART as UART
import serial
import time

UART.setup("UART1")

ser = serial.Serial(port = "/dev/ttyO1", baudrate=9600)
ser.close()
ser.open()
value = str()
buffer = str()
f = open("output.txt", "w")
counter = 0
#if ser.isOpen():
#      print "Serial is open!"
#ser.write("Hello World!")
while(1):
      value = ser.read()
      buffer = buffer + value
      if (value == "\n"):
            print(counter)
            counter += 1
            f.write(buffer)
```

```
            buffer = str()


ser.close()


# Eventually, you'll want to clean up, but leave this commented for now,

# as it doesn't work yet

#UART.cleanup()
```

Appendix C

```
a = xlsread("data4.xlsx");
%filtering output
for i = 4200:8000
    if(a(i) < 3)
        a(i) = NaN;
    end
end
plot(1:8000,a);
hold on;
b = zeros(8000,1);
b(1:4000) = 1;
b(4001:end) = 4.5;
xlabel("Time, ms");
ylabel("Motor Speed, rev/s");
title("Motor Speed Reference Trajectory and Actual Trajectory vs Time");
plot(1:8000,b,"r");
```