

Cursor Control using EMG Muscle Sensor Electronic Systems in Action

Seif Salama – 202200639

Omar Reda – 202201648

April 2024

Table of Contents

I. Introduction

- i. Background and Motivation**
- ii. Project Overview**

II. Methods and Experimental Work

- i. EMG Signal Processing**
 - 1. Signal Acquisition and Amplification**
 - 2. Filtration**
 - 3. Rectification**
 - 4. Smoothing**
 - 5. ADC**
- ii. IMU Signal Processing**
- iii. Hardware Setup**
 - 1. Circuit Schematic**
 - 2. Power Source**
 - 3. Resistors and Capacitors**
 - 4. Instrumentation Amplifier**
 - 5. Operational Amplifier**
- iv. Software Preparation**
 - 1. EMG and IMU**
 - 2. Cursor and Speech to Text**

III. Results

- i. EMG**

1. Ideal Result (Simulation)

2. Practical Result (Implementation)

ii. IMU

iii. Cursor Control and Speech to Text

IV. Discussion

i. Result Interpretation

ii. Limitations

iii. Future Improvements

V. Conclusion

VI. Appendices

i. Appendix A

ii. Appendix B

iii. Appendix C

I. Introduction

i. Background and Motivation

In the intersection of electrical engineering and medicine lies biomedical engineering, a field where EMG muscle sensors emerge. Electromyography, being one of the most used sensors, tracks muscle movements by detecting electrical signals using electrodes. EMG sensors work by amplifying and filtering electrical signals from muscles, making them understandable by digital systems. By mixing EMG muscle data with info from IMUs or Flex Sensors, we can grasp not just movement but also muscle activity. This combination can be used in many applications, such as controlling a robotic arm remotely. Overall, this project shows how we can make our own EMG muscle sensors, then use other sensors to increase accuracy. The project explores the combination of an EMG muscle sensor with an IMU (Inertial Measurement Unit) to control a computer cursor and explore many functionalities without needing a keyboard and mouse. If successful, this project can make life easier for people with disabilities.

ii. Project Overview

This project seeks to revolutionize computer interaction by integrating an Electromyography (EMG) muscle sensor with an Inertial Measurement Unit (IMU) to control a computer cursor and enable various functionalities without the need for a traditional keyboard and mouse setup. The EMG sensor serves as the click mechanism for the mouse cursor, with mild contractions triggering a click and strong contractions activating the Windows speech-to-text functionality for typing. The IMU's gyroscope is utilized for cursor movement, eliminating the need for a keyboard and mouse setup. The potential impact of this project is significant, particularly for individuals with disabilities who may face challenges using conventional input devices. By providing an alternative, more accessible means of computer interaction, this project has the potential to enhance the quality of life for such individuals.

II. Methods and Experimental Work

i. EMG Signal Processing

1. Signal Acquisition and Amplification

Designing an Electromyography sensor from scratch is certainly a challenge. First, we need to collect the muscle signal. This is done through electrodes, which collect muscle signals. EMG muscle signals are usually between 10 and 30 mV. Their frequency is usually less than 500 Hz. After collection, the signals are amplified using an instrumentation amplifier. Two electrodes placed on the target muscle are connected to the inverting and non-inverting pins of the instrumentation amplifier. A third electrode is placed away from the muscle and is connected to the ground. The instrumentation amplifier removes some noise, as it uses the common mode method, which detects noise visible in only one of the sensors and rejects it.

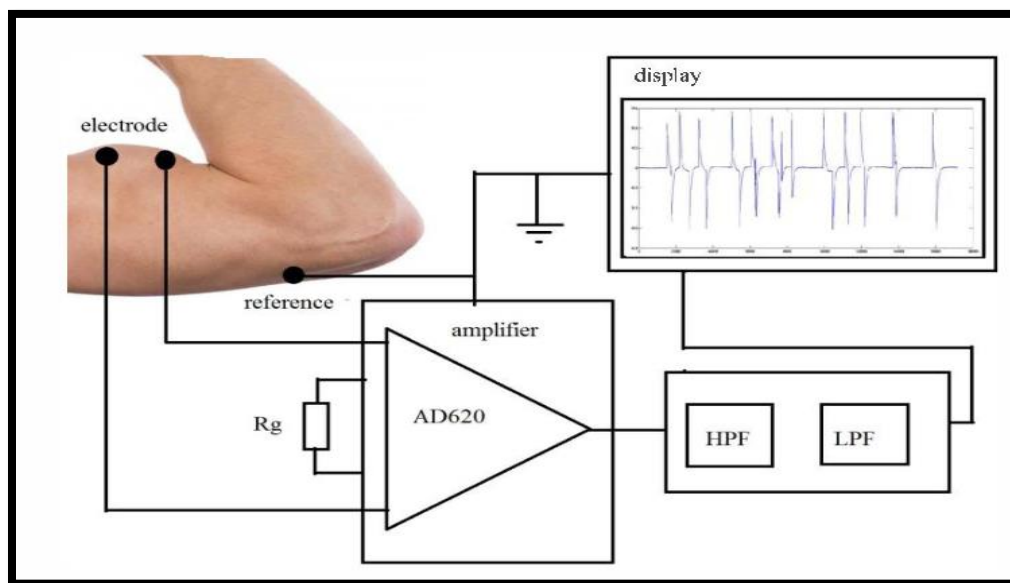


Figure 1: Electrode Placement and Amplification

2. Filtration

The next step is using a low pass filter. A low pass filter is essential as EMG muscle signals usually do not exceed 500 Hz. Consequently, we need to filter out higher frequencies using a low pass filter. In ideal circumstances, low pass filters are accurate. Unfortunately, in practice, low pass filters filter out some higher and lower frequencies than the one chosen. For this reason, low pass filters are rated first order, second order and third order. First order filters are the least accurate, and third order are the most accurate. For this application, first order filters are far from accurate, but second order filters are accurate enough, so considering other factors such as cost, we will be using second order filters.

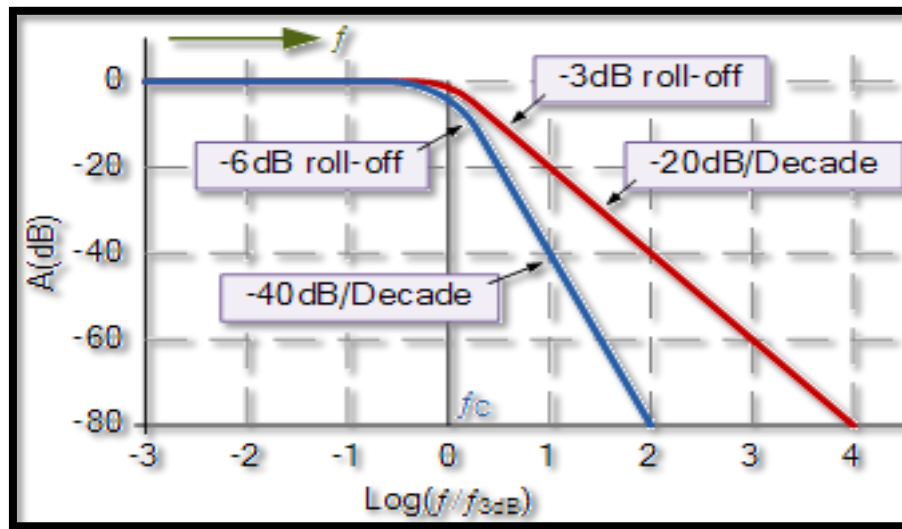


Figure 3: Difference in cut-off Frequency between first (red) and second (blue) order filters

3. Rectification

After filtration, we decided to do more signal processing by adding a full wave rectifier. The full wave rectifier would help smoothen the wave when the frequency is extremely lowered, helping us convert the signal into a digital signal and making it usable. Rectification would result in all values being above zero, which would make it easy to convert the signal into an RMS signal.

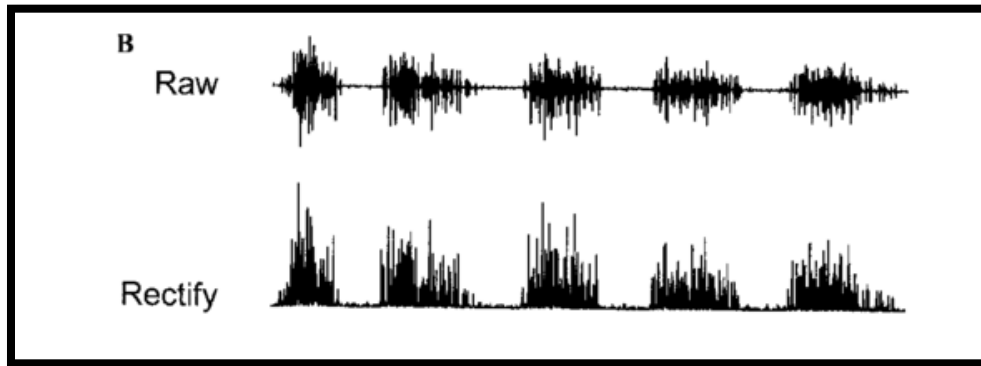


Figure 5: Difference between Raw and Rectified EMG Signal

4. Smoothing

An RMS signal would be perfect for an ADC (Analogue-to-Digital Converter). This can be achieved after full wave rectification by finally filtering the signal one last time using a low pass filter. This time, the filter would be used to achieve a very low frequency of 20 Hz. With such a low frequency, the EMG signal would be stable and wouldn't fluctuate or vulnerable to noise, as out of place signals are ignored. As shown in the following figure, RMS signals are accurate compared to the original amplified EMG signal.

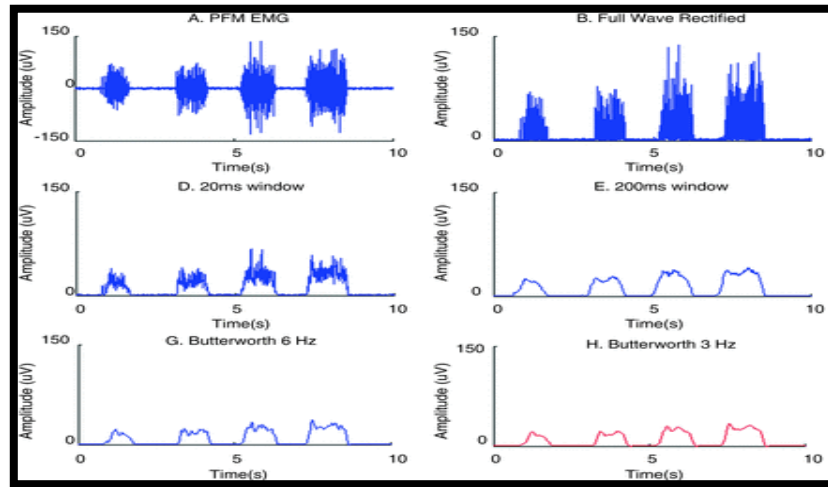


Figure 7: Filtered Signal vs RMS Signal

5. ADC

Finally, the RMS signal is delivered to the Arduino ADC pin. The signal is then programmed using the Arduino to do a specific function using embedded C. The Arduino R3 board is basically an ATMEGA328 microcontroller. We use the Arduino R3 for ease of use and uploading of code. It is also easy to connect external sensors to the Arduino board through its GPIO pins.

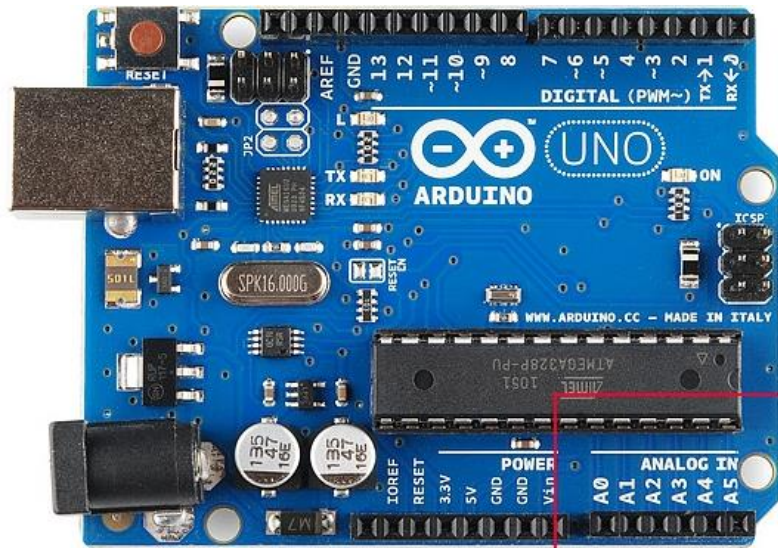


Figure 9: Arduino ADC Pins

ii. IMU Signal Processing

An Inertial Measurement Unit (IMU) sensor provides data on orientation, acceleration, and angular velocity. Integrating an IMU sensor alongside the EMG sensor enhances the understanding of movement by capturing information about the orientation and motion of the target muscle. This data complements the EMG signals by providing context for muscle activity, enabling more accurate assessments of movement and providing precise control. For our application, we will be using the IMU's gyroscope.

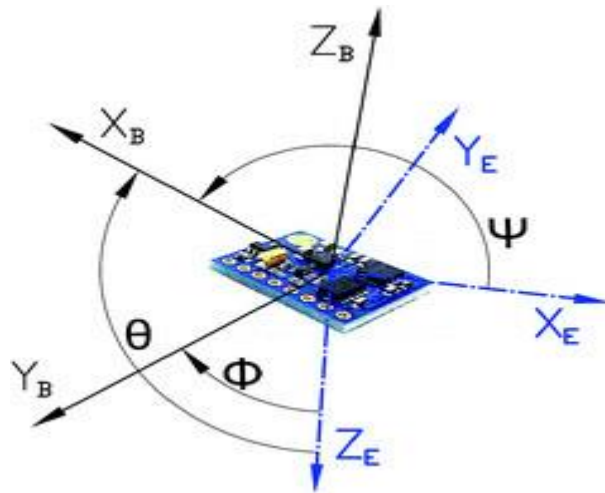
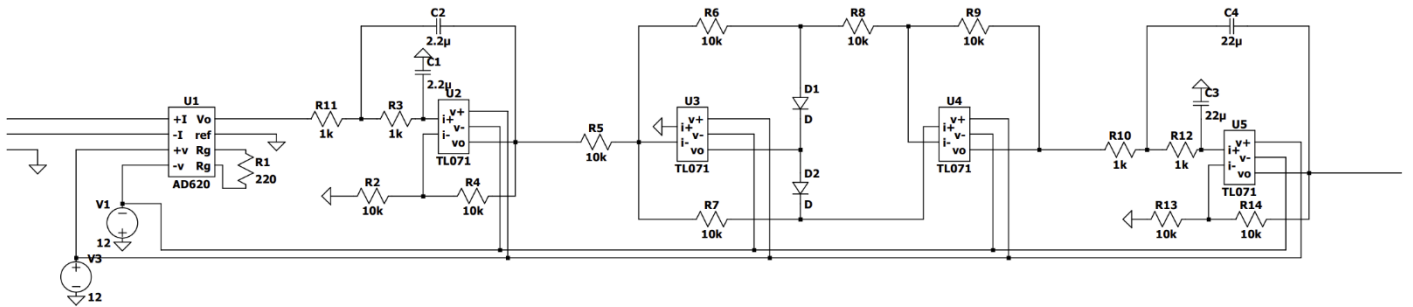


Figure 10: IMU Sensor

Using a certain formula, the IMU raw values would be converted into the angle of rotation of the imu. The imu would thus provide values that could easily be translated into a direction (left, right, forward and backward)

iii. Hardware Setup

1. Circuit Schematic



2. Power Source

The operational and instrumentational amplifiers are powered using two 9V batteries. One connected from the positive side (+9V) and one connected from the negative side (-9V). The other side of each battery is grounded. As for the imu, it is powered using the Arduino's 5V Power Supply.

3. Resistors and Capacitors

Resistors and Capacitors are a vital part of the circuit as they control aspects of the amplifiers used. For the instrumentation amplifier used, resistors control the gain of the voltage (V_{out}/V_{in}). As for the op amps, resistors control mainly the gain of the voltage while capacitors control the cut-off frequency of the low pass filter.

4. Instrumentation Amplifier

Instrumentation amplifier is used in the project to reject noise while amplifying the EMG muscle signal. It is important because EMG values are usually no higher than 30 mV. The model

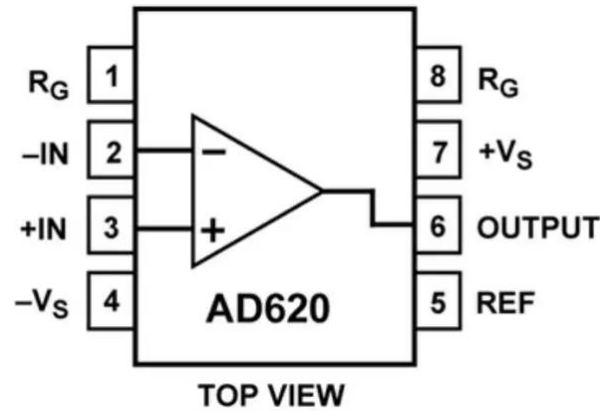


Figure 12: AD620 Pins

of instrumentation amplifier used in the AD620 amplifier, which is known for its low noise and good quality.

5. Operational Amplifier

As for the op-amp used, we opted for the TL071. The TL071 is a higher quality op-amp than the UA741, which is old. It is used four times in our current circuit. First, it is used to create a low pass filter. Then it is used twice to create a full wave rectifier. Finally, it is used to create an RMS signal using a low pass filter that is set to filter the wave to a very low frequency.

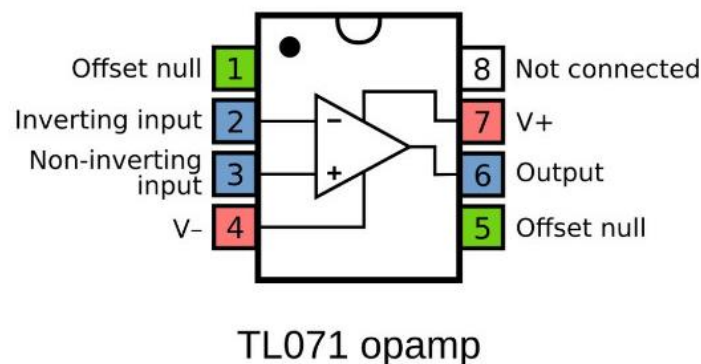


Figure 14: TL071 Pinout

iv. Software Preparation

1. EMG and IMU

For the EMG muscle sensor, we took the challenge of programming it in Embedded C. To do so, we had to make several files for data transfer protocols as well as the analogue-digital converter. As for the IMU, we used I2C protocol. To simplify the coding process, we used built-in Arduino functions for the IMU software and combined it with the Embedded C code for the EMG muscle sensor. For the EMG, a mild contraction prints a 1 to the serial monitor while a strong contraction prints a 2. To view the code used, refer to Appendix B.

2. Cursor and Speech to Text

Converting IMU values into mouse cursor movement was certainly a fun challenge. Using the python library “pyautogui”, we were able to program the computer cursor to move according to the gyroscope value provided. We were also able to program a mild contraction from the EMG (which returns 1) to do a mouse click, while a strong contraction (which returns 2) would open the speech to text module by clicking WIN+H hotkey on windows. To view the code, refer to Appendix C.

III. Results

i. EMG

1. Ideal Result (Simulation)

To make the simulation as accurate and practical as possible, two sources were used. First, an artificial EMG signal was created through sinusoidal waves with different frequencies and amplitudes. Then, we downloaded a raw EMG file from the internet to use in our simulation. The simulation results were ideal, thus, it expected that the practical results be different.

1.1 Artificial Signal

By Simulating several sinusoidal waves with different frequencies, we were able to create a noisy sinusoidal wave like an EMG signal. The following is the simulation result of the signal after filtering using a second order filter compared to after amplification.

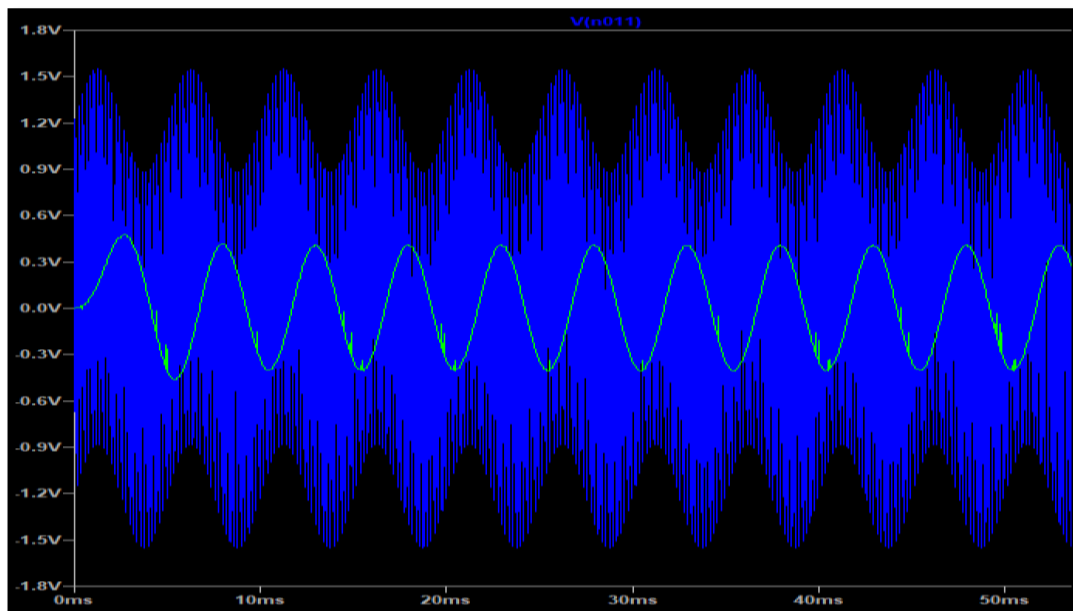


Figure 15: Sinusoidal Signal (Blue) compared to Filtered Signal (Green)

1.2 Pre-Recorded EMG Signal

To make the simulation as accurate as possible, we used an EMG raw signal file found on a GitHub repository. The simulation was done using one file, as using two files with different signals would not work with the common mode function. The following images show the EMG signal after file after amplification compared to after filtering and compared to after rectification and RMS smoothing.

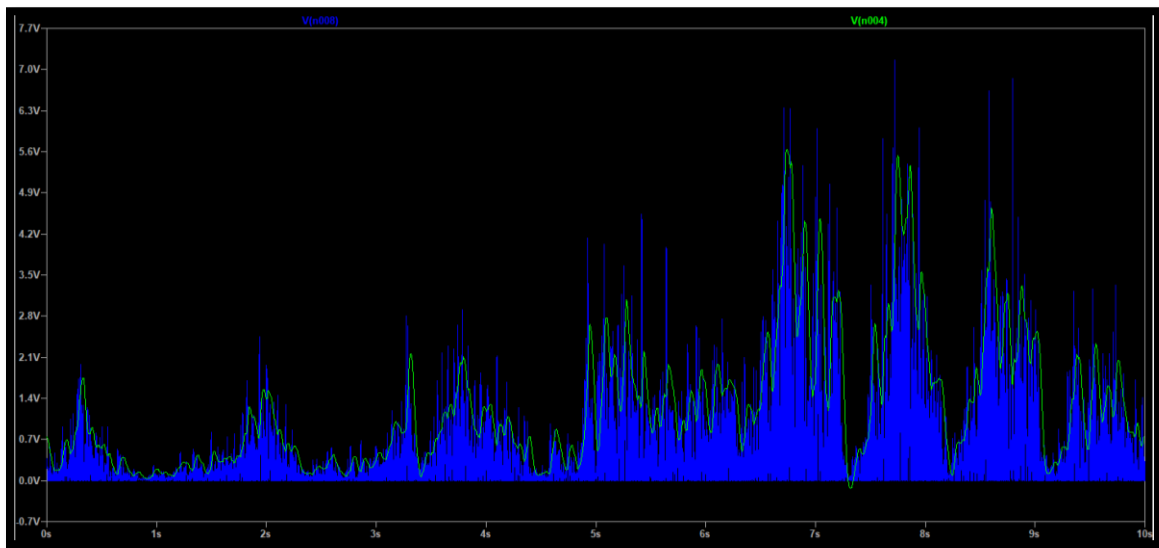


Figure 17: Rectified Signal

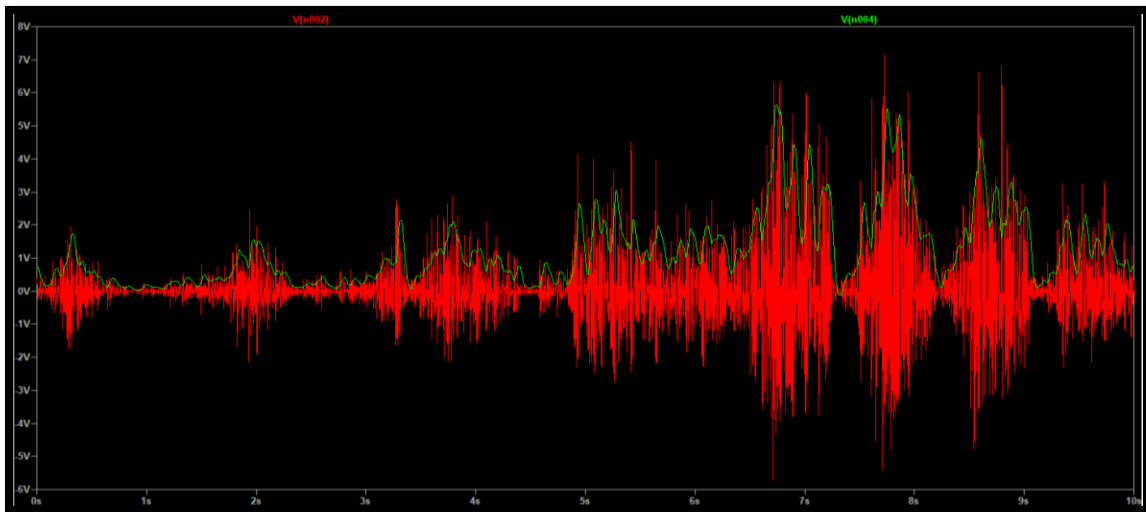


Figure 16: Amplified EMG Singal (Red) vs RMS Signal (Green)

2. Practical Result (Implementation)

The circuit worked when integrated. It successfully amplified and filtered the EMG signal. When it came to rectification and RMS value, the signal would successfully be converted to RMS. Motion Artifacts would affect the performance of the signal, especially the rectification and RMS stage. Regardless, the EMG signal was successfully integrated with the Arduino as an analog input. It also worked accordingly with the ADC code.

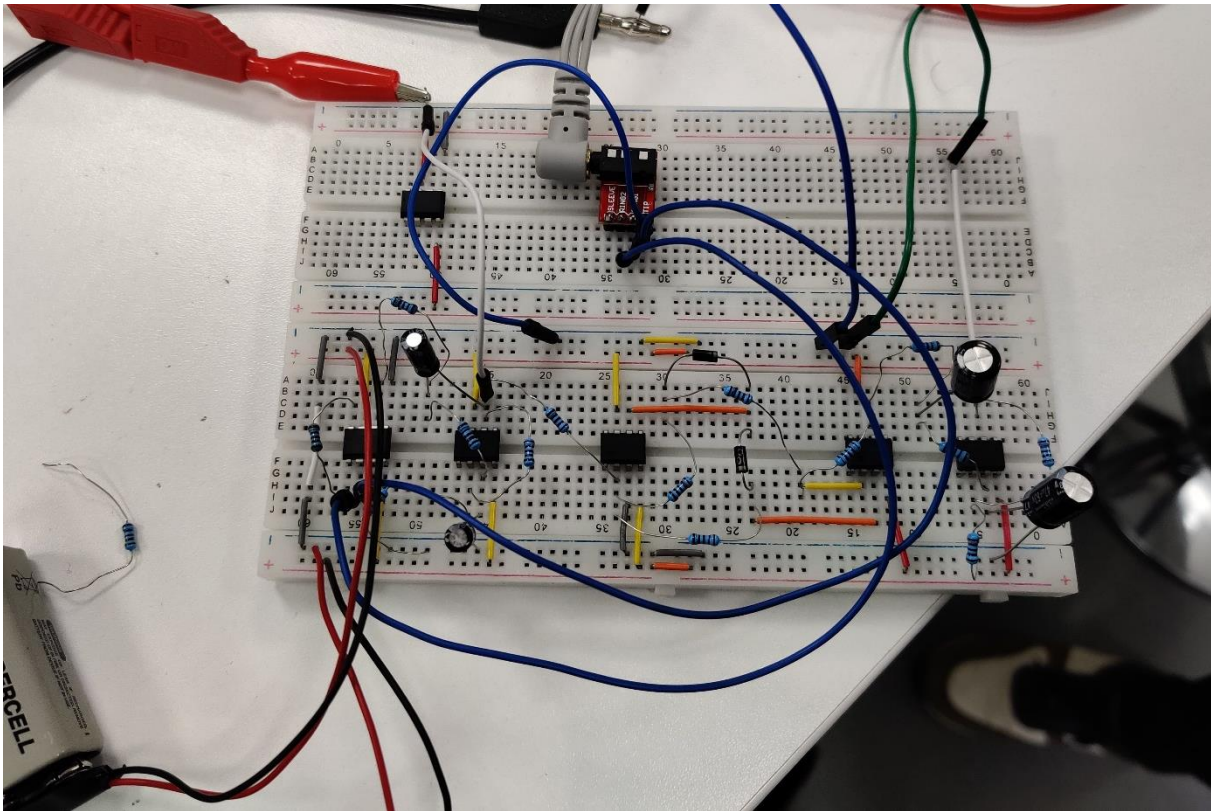


Figure 19: Implemented prototype

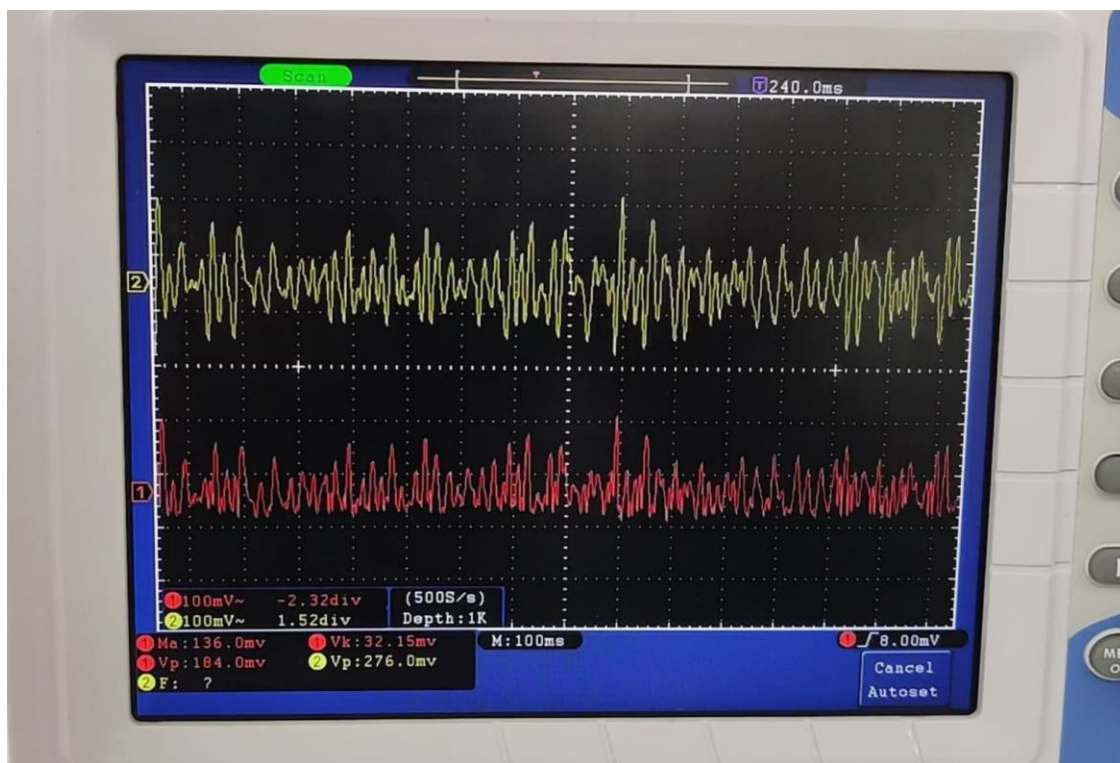


Figure 21: Filtered (Yellow) vs Rectified (Red)

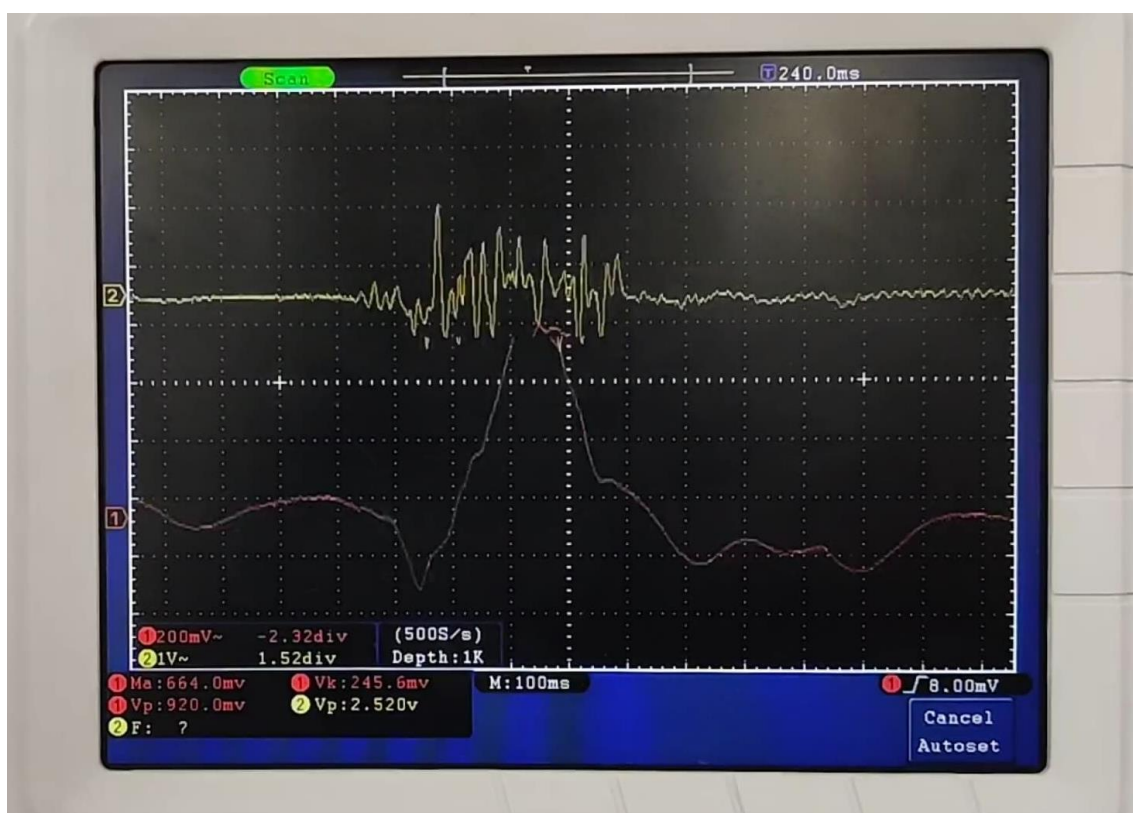


Figure 20: Filtered (Yellow) vs RMS (Red)

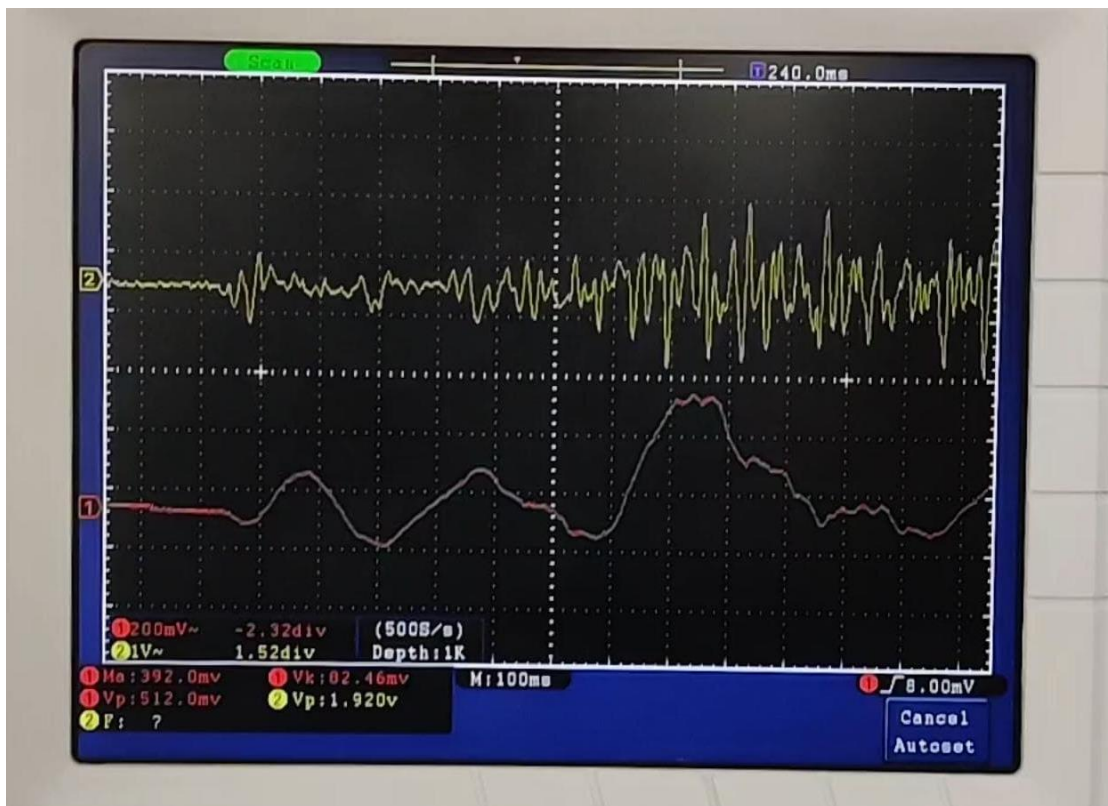
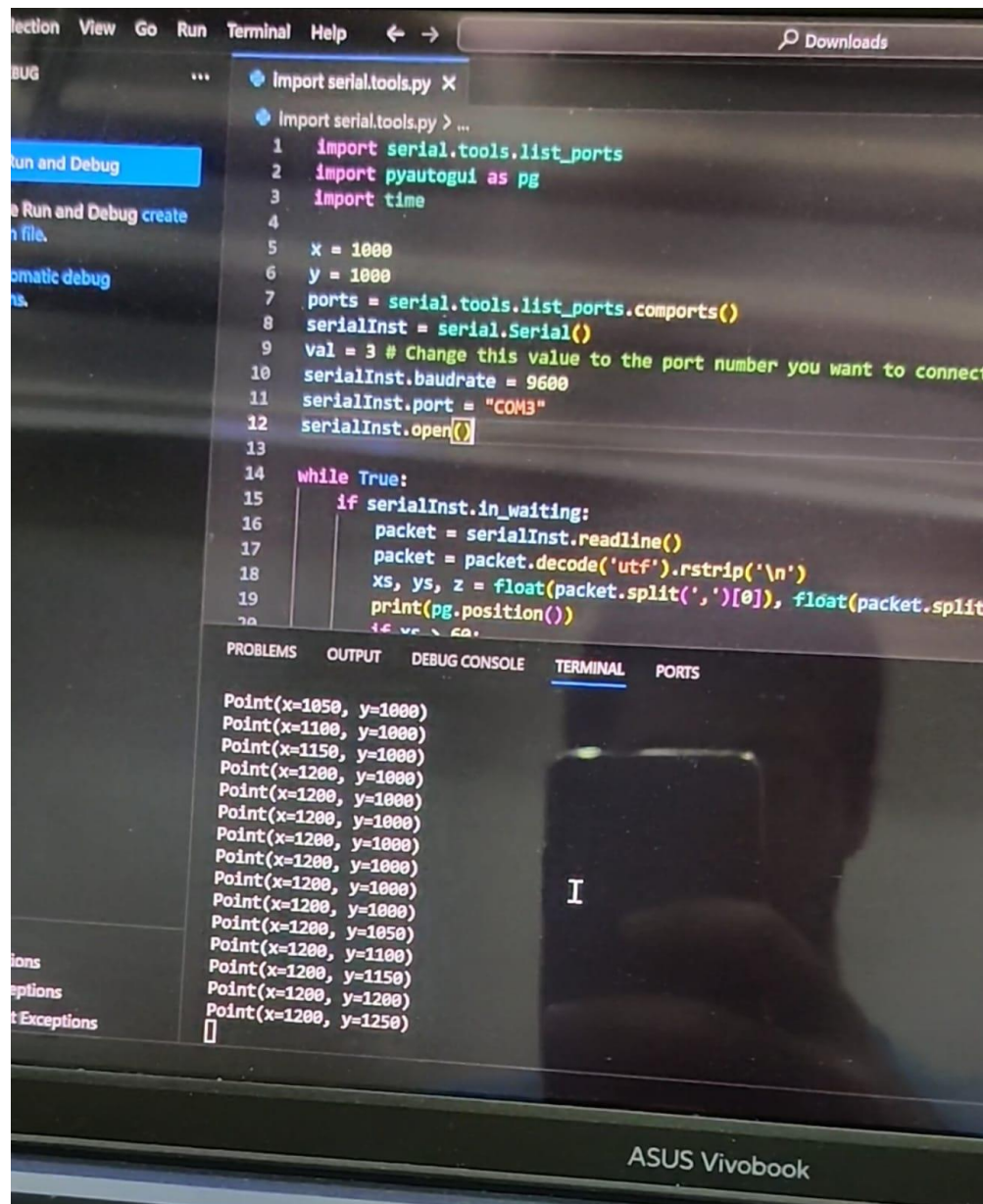


Figure 22: Filtered (Yellow) vs RMS (Red)

ii. IMU

After connecting the IMU and uploading the code to the Arduino, the output was clearly printed to the serial monitor. The values of the gyroscope angles were successfully displayed and ready to be used with the computer cursor program.



The image shows a screenshot of a code editor with a dark theme. The editor displays a Python script that imports necessary modules, sets up a serial connection to a COM3 port at 9600 baud, and enters a loop to read and process data from the IMU. The output of the script is visible in the terminal window at the bottom, showing a series of coordinate points.

```
Import serial.tools.py X
Import serial.tools.py > ...
1 import serial.tools.list_ports
2 import pyautogui as pg
3 import time
4
5 x = 1000
6 y = 1000
7 ports = serial.tools.list_ports.comports()
8 serialInst = serial.Serial()
9 val = 3 # Change this value to the port number you want to connect
10 serialInst.baudrate = 9600
11 serialInst.port = "COM3"
12 serialInst.open()
13
14 while True:
15     if serialInst.in_waiting:
16         packet = serialInst.readline()
17         packet = packet.decode('utf').rstrip('\n')
18         xs, ys, z = float(packet.split(',')[0]), float(packet.split(
19             if xs > 60:
```

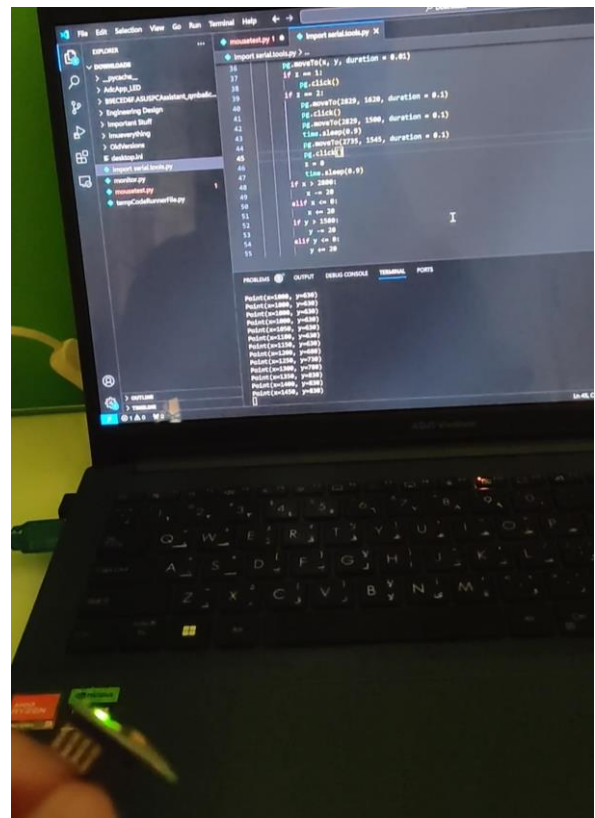
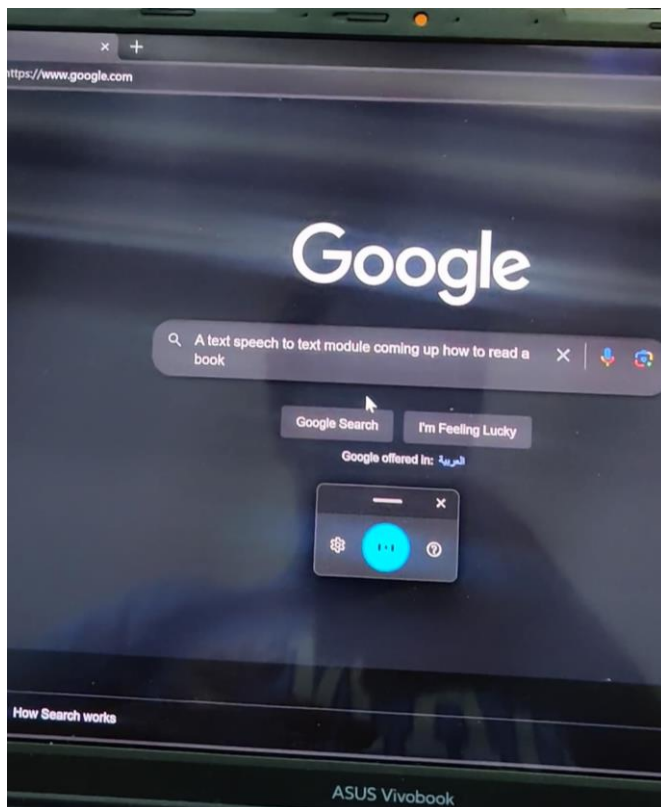
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Point(x=1050, y=1000)
Point(x=1100, y=1000)
Point(x=1150, y=1000)
Point(x=1200, y=1000)
Point(x=1200, y=1000)
Point(x=1200, y=1000)
Point(x=1200, y=1000)
Point(x=1200, y=1000)
Point(x=1200, y=1000)
Point(x=1200, y=1000)
Point(x=1200, y=1000)
Point(x=1200, y=1050)
Point(x=1200, y=1100)
Point(x=1200, y=1150)
Point(x=1200, y=1200)
Point(x=1200, y=1250)
[]
```

ASUS Vivobook

iii. Cursor Control and Speech to Text

Using the IMU and EMG outputs, we successfully integrated the Arduino with a Laptop to control the cursor using an IMU, and click using a mild EMG contraction, while also opening a speech to text module using a strong EMG contraction. The serial monitor output was successfully transferred to the python program and integrated. The cursor and speech to text also worked successfully.



IV. Discussion

i. Result Interpretation

The results of our project indicate a successful integration and functioning of both the Electromyography (EMG) muscle sensor and the gyroscope from the Inertial Measurement Unit (IMU). The EMG sensor accurately detected muscle contractions and translated them into corresponding cursor actions. Similarly, the gyroscope from the IMU reliably captured subtle hand movements and translated them into cursor movements on the computer screen. Eliminating the need for a mousepad. Overall, the successful integration of the EMG sensor and gyroscope not only achieved the project objectives but also showcased the potential of combining these technologies to enhance human-computer interaction.

ii. Limitations

Although the prototype has successfully worked, there are some limitations that exist. Firstly, the EMG sensor is sensitive to motion artifacts, which create an offset in the signal and makes it unusable. They also sometimes affect the rectification as a small portion of the signal gets downward during movement. Additionally, due to the usage of a breadboard, there is still a lot of noise in the EMG signal. Using a PCB would reduce this noise and improve stability. As for the IMU, it needs to be calibrated every time it is turned on, which means if it is turned on upside down it will not function correctly. This can be overcome by adding pre-set settings in the code instead of calibrating it each time.

iii. Future Improvements

Improvements can be made to enhance the prototype. Some of the improvements include the following:

- Implementing the circuit on a PCB instead of Breadboard to reduce noise and increase stability.
- Changing the Second Order Filter into a Third Order Filter to increase the accuracy of the low-pass filter.
- Adjusting the programming of the mouse cursor to be smoother and more like a mousepad.
- Adding more gesture control features by detecting patterns in EMG (ex. three quick contractions to open web browser).

V. Conclusion

In conclusion, our project successfully demonstrated the integration of an Electromyography (EMG) muscle sensor with a gyroscope from an Inertial Measurement Unit (IMU) to control a computer cursor and enable additional functionalities without the need for a keyboard and mouse setup. Throughout the project, the EMG sensor was able to convert a muscle contraction into a digital signal. The implementation of mild contractions for mouse clicks and strong contractions for activating speech-to-text functionality provided a seamless user experience. Positive results were shown when both sensors worked together to move a mouse cursor and click. The successful integration of EMG and IMU sensors opens up a wide range of possibilities for enhancing computer accessibility, particularly for individuals with disabilities. Future improvements suggested could enhance the prototype's usability and applicability. Finally, our project not only achieved its objectives but also highlighted the potential of combining EMG and IMU sensors to innovate human-computer interaction, ultimately contributing to a more inclusive and accessible digital environment.

VI. Appendices

i. Appendix A

[Video of Functioning Prototype](#)

ii. Appendix B

Main File

```
#include <util/delay.h>
#include "Adc_LED.h"
#include "Uart.h"
#include <stdlib.h>
#include "dio.h"
#include <Wire.h>

float elapsedTime, time, timePrev;
int gyro_error=0;
float Gyr_rawX, Gyr_rawY, Gyr_rawZ;
float Gyro_angle_x, Gyro_angle_y;
float Gyro_raw_error_x, Gyro_raw_error_y;
int acc_error=0;
float rad_to_deg = 180/3.141592654;
float Acc_rawX, Acc_rawY, Acc_rawZ;
float Acc_angle_x, Acc_angle_y;
float Acc_angle_error_x, Acc_angle_error_y;
float Total_angle_x, Total_angle_y;

void setupGyroAndAcc() {
    Wire.begin();
    Wire.beginTransmission(0x68);
    Wire.write(0x6B);
    Wire.write(0x00);
    Wire.endTransmission(true);
    Wire.beginTransmission(0x68);
    Wire.write(0x1B);
    Wire.write(0x10);
    Wire.endTransmission(true);
    Wire.beginTransmission(0x68);
    Wire.write(0x1C);
    Wire.write(0x10);
    Wire.endTransmission(true);
}
```

```

time = millis();

if(acc_error==0)
{
    for(int a=0; a<200; a++)
    {
        Wire.beginTransmission(0x68);
        Wire.write(0x3B);
        Wire.endTransmission(false);
        Wire.requestFrom(0x68,6,true);

        Acc_rawX=(Wire.read()<<8|Wire.read())/4096.0 ;
        Acc_rawY=(Wire.read()<<8|Wire.read())/4096.0 ;
        Acc_rawZ=(Wire.read()<<8|Wire.read())/4096.0 ;

        Acc_angle_error_x = Acc_angle_error_x +
((atan((Acc_rawY)/sqrt(pow((Acc_rawX),2) + pow((Acc_rawZ),2)))*rad_to_deg));
        Acc_angle_error_y = Acc_angle_error_y + ((atan(-
1*(Acc_rawX)/sqrt(pow((Acc_rawY),2) + pow((Acc_rawZ),2)))*rad_to_deg));

        if(a==199)
        {
            Acc_angle_error_x = Acc_angle_error_x/200;
            Acc_angle_error_y = Acc_angle_error_y/200;
            acc_error=1;
        }
    }
}

if(gyro_error==0)
{
    for(int i=0; i<200; i++)
    {
        Wire.beginTransmission(0x68);
        Wire.write(0x43);
        Wire.endTransmission(false);
        Wire.requestFrom(0x68,4,true);

        Gyr_rawX=Wire.read()<<8|Wire.read();
        Gyr_rawY=Wire.read()<<8|Wire.read();

        Gyro_raw_error_x = Gyro_raw_error_x + (Gyr_rawX/32.8);
        Gyro_raw_error_y = Gyro_raw_error_y + (Gyr_rawY/32.8);
        if(i==199)

```

```

    {
        Gyro_raw_error_x = Gyro_raw_error_x/200;
        Gyro_raw_error_y = Gyro_raw_error_y/200;
        gyro_error=1;
    }
}
}

void readAndPrintSensorData() {
    timePrev = time;
    time = millis();
    elapsedTime = (time - timePrev) / 1000;

    Wire.beginTransmission(0x68);
    Wire.write(0x43);
    Wire.endTransmission(false);
    Wire.requestFrom(0x68,4,true);
    Gyr_rawX=Wire.read()<<8|Wire.read();
    Gyr_rawY=Wire.read()<<8|Wire.read();
    Gyr_rawX = (Gyr_rawX/32.8) - Gyro_raw_error_x;
    Gyr_rawY = (Gyr_rawY/32.8) - Gyro_raw_error_y;
    Gyro_angle_x = Gyr_rawX*elapsedTime;
    Gyro_angle_y = Gyr_rawY*elapsedTime;

    Wire.beginTransmission(0x68);
    Wire.write(0x3B);
    Wire.endTransmission(false);
    Wire.requestFrom(0x68,6,true);
    Acc_rawX=(Wire.read()<<8|Wire.read())/4096.0 ;
    Acc_rawY=(Wire.read()<<8|Wire.read())/4096.0 ;
    Acc_rawZ=(Wire.read()<<8|Wire.read())/4096.0 ;

    Acc_angle_x = (atan((Acc_rawY)/sqrt(pow((Acc_rawX),2) +
pow((Acc_rawZ),2)))*rad_to_deg) - Acc_angle_error_x;
    Acc_angle_y = (atan(-1*(Acc_rawX)/sqrt(pow((Acc_rawY),2) +
pow((Acc_rawZ),2)))*rad_to_deg) - Acc_angle_error_y;
    Total_angle_x = 0.98 *(Total_angle_x + Gyro_angle_x) + 0.02*Acc_angle_x;
    Total_angle_y = 0.98 *(Total_angle_y + Gyro_angle_y) + 0.02*Acc_angle_y;

    Serial.print(Total_angle_x);
    Serial.print(",");
    Serial.print(Total_angle_y);
    Serial.print(",");
}

```

```
void readAndPrintADC() {
    unsigned short adc_reading;
    adc_reading = Adc_ReadChannel(0);

    if(adc_reading > 300 && adc_reading < 700) {
        Serial.print("1");
    }
    else if (adc_reading > 700) {
        Serial.print("2");
    }
    else {
        Serial.print("0");
    }
    Serial.println();
}

void setup() {
    Adc_Init();
    Uart_Init();
    Serial.begin(9600);
    setupGyroAndAcc();
}

void loop() {
    readAndPrintSensorData();
    readAndPrintADC();
    _delay_ms(100);
}
```

ADC File

```
#if !defined(__AVR_ATmega328P__)
#include <avr/iom328p.h>
#endif
#include "Adc_LED.h"

void Adc_Init(void)
{
    // AREF = AVcc
    ADMUX = (1<<REFS0);

    // ADC Enable and prescaler of 128
    // 16000000/128 = 125000
    ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
}

unsigned short Adc_ReadChannel(unsigned char ch)
{
    // select the corresponding channel 0~7
    // ANDing with '7' will always keep the value
    // of 'ch' between 0 and 7
    ch &= 0b00000111; // AND operation with 7
    ADMUX = (ADMUX & 0xF8)|ch; // clears the bottom 3 bits before ORing

    // start single conversion
    // write '1' to ADSC
    ADCSRA |= (1<<ADSC);

    // wait for conversion to complete
    // ADSC becomes '0' again
    // till then, run loop continuously
    while(ADCSRA & (1<<ADSC));

    return (ADC);
}
```

UART File

```
#include "Uart.h"
#if !defined(__AVR_ATmega328P__)
#include <avr/iom328p.h>
#endif

#define _BV(bit) (1 << (bit))
#define bit_is_set(sfr, bit) (_SFR_BYTE(sfr) & _BV(bit))
#define bit_is_clear(sfr, bit) (!( _SFR_BYTE(sfr) & _BV(bit)))
#define loop_until_bit_is_set(sfr, bit) do { } while (bit_is_clear(sfr, bit))
#define loop_until_bit_is_clear(sfr, bit) do { } while (bit_is_set(sfr, bit))

void Uart_Init(void) {

    Uart_SetBaudRate(9600);

    /* Enable USART transmitter/receiver */
    UCSR0B = (1 << TXEN0) | (1 << RXEN0);

    /* 8 data bits, 1 stop bit */
    UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);
}

void Uart_SetBaudRate(unsigned short BaudRate)
{
    unsigned short UBBR = ( (F_CPU / 16) / BaudRate ) - 1;
    UBRR0L = (UBBR & 0xFF);
    UBRR0H = (( UBBR >> 8 ) & 0xFF);
}

void Uart_SendChar(unsigned char DataByte)
{
    // Wait until Write buffer is empty
    while ( ! (UCSR0A & ( 1 << UDRE0)) );
    UDR0 = DataByte;
}

unsigned char Uart_ReadData(void) {
    // Wait until data is received
    while ( ! (UCSR0A & ( 1 << RXC0)) );
    return UDR0;
}
```

```

void Uart_SendString(char DataString[], unsigned char Size){
    int i;
    for (i=0; DataString[i]; i++)
    {
        Uart_SendChar(DataString[i]);
    }
}

```

DIO File

```

void DIO_init(){
    //set pins PD2 - PD7 as output
    DIO_SetPinDirection('d', 2, 'o');
    DIO_SetPinDirection('d', 3, 'o');
    DIO_SetPinDirection('d', 4, 'o');
    DIO_SetPinDirection('d', 5, 'o');
    DIO_SetPinDirection('d', 6, 'o');
    DIO_SetPinDirection('d', 7, 'o');
    //set pins PB0 - PB2 as output
    DIO_SetPinDirection('b', 0, 'o');
    DIO_SetPinDirection('b', 1, 'o');
}

// set Pin to INPUT or OUTPUT
void DIO_SetPinDirection(unsigned char port, int pin, unsigned char direction){
    //
    if (port == 'b' & pin <= 7 & pin >= 0){ // pin must be from 0-7
        if(direction == 'o'){
            DDRB = DDRB|(1<<pin); //bit set to 1 for output
        }else if(direction == 'i'){
            DDRB = DDRB&!(1<<pin); // bit cleared to 0 for input
        }
    }
    if (port == 'c' & pin <= 7 & pin >= 0){ // pin must be from 0-7

```

```

    if(direction == 'o'){
        DDRC = DDRC|(1<<pin); // bit set to 1 for output
    }else if(direction == 'i'){
        DDRC = DDRC&!(1<<pin); // bit cleared to 0 for input
    }
}
}
if (port == 'd' & pin <= 7 & pin >= 0){ // pin must be from 0-7
    if(direction == 'o'){
        DDRD = DDRD|(1<<pin); //bit set to 1 for output
    }else if(direction == 'i'){
        DDRD = DDRD&!(1<<pin); //bit cleared to 0 for input
    }
}
}

// if pin is output it will set it to either HIGH or LOW
// if pin is input it will activate internal pull up resistor
void DIO_SetPinState(unsigned char port, int pin, unsigned char state){
    if (port == 'b' & pin <= 7 & pin >= 0){ // pin must be from 0-7
        if(state == 'h'){
            PORTB = PORTB|(1<<pin); //pin is set to 1 for high
        }else if(state == 'l'){
            PORTB &= ~(1<<pin); //pin is cleared to 0 for low
        }
    }
    if (port == 'd' & pin <= 7 & pin >= 0){ // pin must be from 0-7
        if(state == 'h'){
            PORTD = PORTD|(1<<pin); //pin is set to 1 for high
        }else if(state == 'l'){
            PORTD &= ~(1<<pin); //pin is cleared to 0 for low
        }
    }
    if (port == 'c' & pin <= 7 & pin >= 0){ // pin must be from 0-7
        if(state == 'h'){
            PORTC = PORTC|(1<<pin); //pin is set to 1 for high
        }else if(state == 'l'){
            PORTC &= ~(1<<pin); //pin is cleared to 0 for low
        }
    }
}
}

```


ADC Header

```
#ifndef __ADC__
#define __ADC__

void Adc_Init(void);

unsigned short Adc_ReadChannel(unsigned char channel);

#endif
```

UART Header

```
#ifndef __UART__
#define __UART__

void Uart_Init(void);

void Uart_SetBaudRate(unsigned short BaudRate);

void Uart_SendChar(unsigned char DataByte);

unsigned char Uart_ReadData();

void Uart_SendString(char DataString[], unsigned char Size);

#endif
```

DIO Header

```
void DIO_init();
void DIO_SetPinDirection(unsigned char port, int pin, unsigned char direction);
void DIO_SetPinState(unsigned char port, int pin, unsigned char state);
```

iii. Appendix C

```
import serial.tools.list_ports
import pyautogui as pg
import time

x = 1000
y = 1000
ports = serial.tools.list_ports.comports()
serialInst = serial.Serial()
val = 3 # Change this value to the port number you want to connect to
serialInst.baudrate = 9600
serialInst.port = "COM6"
serialInst.open()

while True:
    if serialInst.in_waiting:
        packet = serialInst.readline()
        packet = packet.decode('utf').rstrip('\n')
        xs, ys, z = float(packet.split(',')[0]), float(packet.split(',')[1]),
float(packet.split(',')[2])
        print(pg.position())
        if xs > 60:
            x -= 100
        elif xs > 30:
            x -= 50
        elif xs < -60:
            x += 100
        elif xs < -30:
            x += 50
        if ys > 60:
            y += 100
        elif ys > 30:
            y += 50
        elif ys < -60:
            y -= 100
        elif ys < -30:
            y -= 50
        pg.moveTo(x, y, duration = 0.01)
        if z == 1:
            pg.click()

        if z == 2:
            pg.hotkey('win', 'h')
```

```
if x > 2800:
    x -= 20
elif x <= 0:
    x += 20
if y > 1580:
    y -= 20
elif y <= 0:
    y += 20
```