REPORT

Title: *Integration of EMG and IMU sensors to actuate a 1 DoF robotic arm in real time*

By*: Aditya Roy*

In coordination with
Vivek Chaudhary,
PhD scholar, Mechanical Engineering Department, IIT Delhi
Under the supervision of
Dr. Jitendra Prasad Khatait,
Professor, Mechanical Engineering Department, IIT Delhi

CONTENTS

INTRODUCTION

When it comes to helping robots understand what we want them to do — like moving an arm or picking something up — scientists often use a combination of muscle signals and motion sensors to make the interaction feel more natural and accurate.

One important way to do this is by using EMG sensors, which detect the tiny electrical signals your muscles make when they contract. These signals can be picked up from the surface of your skin, even without needles or surgery. So, if you flex your arm, EMG sensors can tell that your biceps are working. This is why EMG is a popular method for controlling things like prosthetic arms or robotic limbs — it lets machines "listen" to your muscle activity and understand your intent.

But there's a limit to what EMG can do. While it tells us that a muscle is active, it doesn't say how much you've actually moved — for example, how far you bent your elbow or how fast you did it. That's where IMU sensors come in. These sensors can track the actual motion of your arm by measuring movement, angle, and rotation. Think of it like this: EMG tells the robot you want to move, and the IMU shows how you're moving.

Georgi et al. (2015) and Zhang et al. (2011) were among the first to show that pairing inexpensive IMUs with surface EMG not only provides information about muscle activation but also captures movement patterns—like rotation or elevation—that EMG alone can't detect. This combination was found to greatly enhance gesture recognition, even more so than simply adding more EMG sensors

By using both sensors together, researchers can build smarter systems that not only detect your intention but also follow through with precise movement. For example, some advanced robotic arms and rehab devices use EMG to detect when you're trying to lift something, and then use IMU data to match your movement exactly — making the robot behave more naturally and safely.

Over time, this combination has become more powerful thanks to better signal processing. Now, robots can learn your motion patterns and respond in real-time, even when you're moving quickly or changing direction. This dual-sensor approach is already being used in

prosthetics, exoskeletons, and therapy robots to help people regain movement after injury or disability.

In short, using EMG and IMU sensors together is like giving robots both ears and eyes — one to hear your muscle signals and one to see your movements — so they can act just the way you want.

OBJECTIVE

This report proposes the formulation of a prototype of a 1 DOF robotic manipulator controlled by the instantaneous orientation of human forearm with the upper arm. The interaction between the actual elbow motion and the manipulator is facilitated by an IMU (Inertial Measurement Unit) sensor and a pair of EMG (Electromyography) sensors. The EMG sensors fetch the myoelectric signals generated during muscle contraction through the surface or skin. The Myoelectric interface is quite a reliable and accurate method to identify human motion and is considered an important tool in human robot interaction. IMU sensors, on the other hand, is primarily concerned with determining the extent of human motion, which cannot be comprehended from the EMG signal amplitude.

The surface EMG signal data has been obtained from three different subjects. The data showed conflicting results in terms of the peak amplitude at muscle activation. Thus, it can be inferred from the tests that activation signal amplitude cannot be used as a parameter to determine the extent of dynamic motion. This paper further illustrates the modelling of the manipulator and the procedures involved in processing the EMG and IMU data obtained from the subject.

Working with N20 motor and rotary encoder

- Due to its compact size and high torque generating capability, the N20 motor becomes highly appropriate for our application. The presence of inbuilt rotary encoder is a much bigger value addition for our needs.

- The first step is to determine the pulses per revolution generated by the encoder of the N20 motor. Different N20 motors available in the market offer different pulses per revolution (PPR). In order to validate it, we implemented a program that displays the encoder count and manually rotated the magnet attached at the end of the motor by 1 turn and observed the encoder count which turned out to be 28. The pulses per revolution of the output shaft can be obtained by:

$$PPR\ (output\ shaft) = PPR * Gear\ Ratio$$

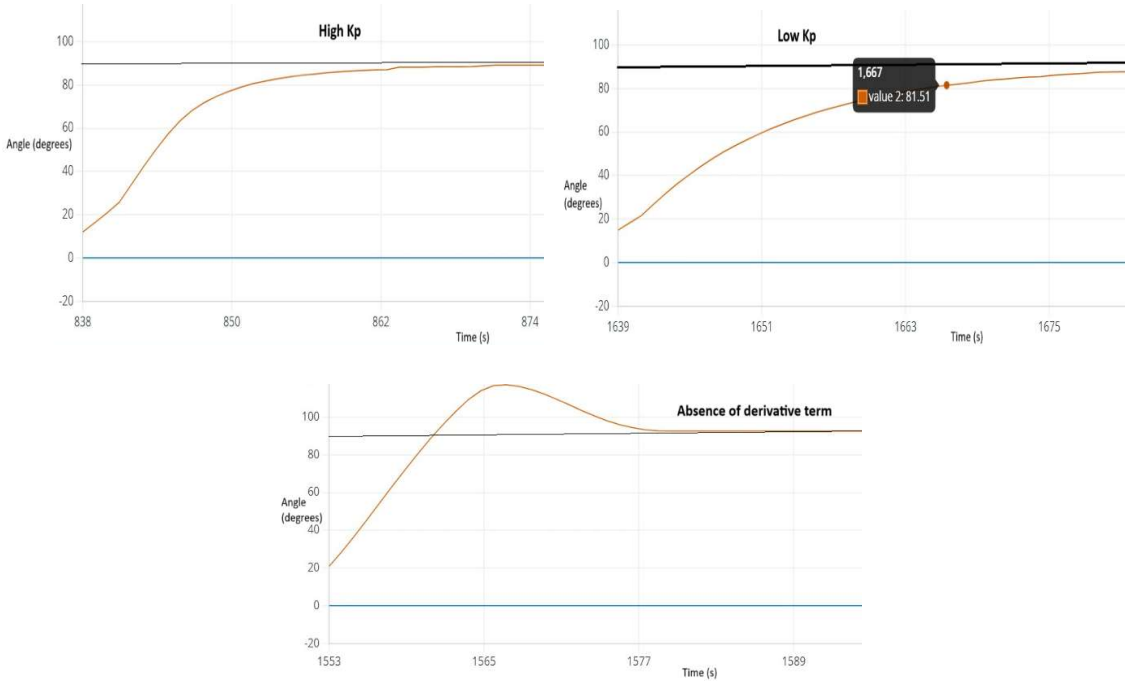
  where the PPR is 28 and the gear ratio is 100.

  Thus, it can be inferred that one complete rotation of the output shaft is capable of producing 2800 encoder pulse counts, thus we can accurately position the motor for as much as 0.12857° (360°/2800), which is capable of producing very accurate results.

Implementing PID control

The application of PID control is crucial to position the motor to its desired angle as mapped by the IMU sensor. At first, we tried actuating the motor without using PID control which produced quite disappointing results. Operating at a fixed high RPM produced constant jitters in the motor pertaining to its time complexity during execution. Correspondingly, operating at low fixed RPM resulted in slow positioning of the motor to the desired angle. Thus it becomes necessary to implement an algorithm where the angular velocity is proportional to the required target angle. In order to achieve this PID control algorithm becomes necessary. The presence of proportional, integral and derivative terms offers fast response, minimal error, and good stability. Tuning a PID controller involves finding the right balance between responsiveness, stability, and steady-state error. The constants have been determined through trial and error. It can also be determined through Simulink by fetching the correct

specifications of the motor. We plotted some real time curves based on how absence of any of these components affects the performance.



From the above plots, it can be concluded that a high Kp (proportional gain constant) value results in faster attainment of target but it should be optimal to control a motor. Absence of derivative term may result in overshooting, thus its presence smoothens the process.

IMU signal processing

The IMU sensor that we utilised for this project is MPU9250 which is a 9-axis motion sensor combining accelerometer, gyroscope and magnetometer.

- We initially implemented only the accelerometer readings to compute the angle, the algorithm behind this angle computation involved relative direction of gravity to the sensor depending on which we computed the angle described by the gravity vector to the corresponding two orthogonal acceleration vectors.

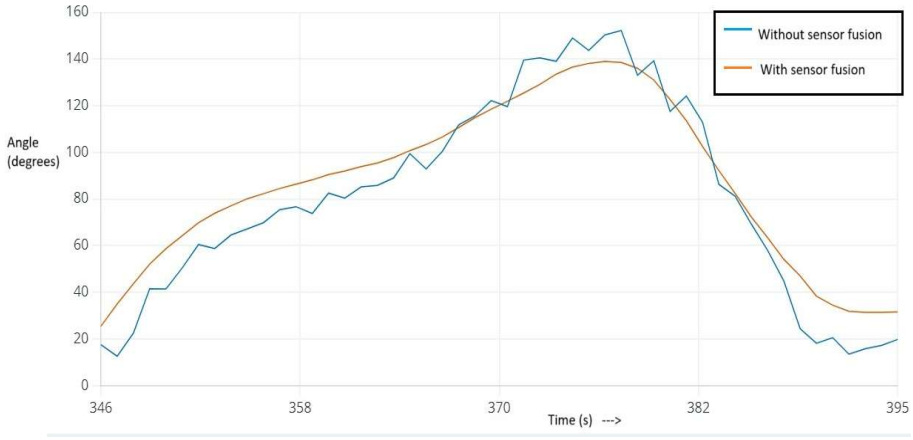
$$\text{Roll} = \arctan\left(\frac{a_y}{a_z}\right) \cdot \left(\frac{180}{\pi}\right)$$

- The angle computed ranges from -90 degrees to 0 degrees in the first quadrant followed by the 0 degrees to 90 degrees in the second quadrant. In order to obtain our desired range of 0 degrees to 180 degrees, we map the angle obtained to our desired one in the program itself.

- The implementation of only the accelerometer readings to compute the angle results in a drawback. The introduction of linear and rotational acceleration components to the sensor disrupts the direction of net acceleration and we observed inconsistent readings of the data. We verified this by manually rotating the sensor along a protractor attached setup and plotted the corresponding angle. In order to overcome this limitation, we implemented sensor fusion algorithm involving the gyroscope and accelerometer. This sensor fusion algorithm is known as complementary filter. In this we take a weighted average of both the accelerometer and gyroscope readings, thus the result becomes dependent on both of them. Therefore, any linear or rotational acceleration will have minimal effect as it is highly dependent on gyroscope reading or angular velocity of the sensor.

$$\theta_{\text{filtered}} = \alpha(\theta_{\text{prev}} + \omega_{\text{gyro}}\,\text{dt}) + (1-\alpha)\,\theta_{\text{acc}}$$

where, α -> constant of proportionality (= 0.98 in our case)
dt -> time gap between two consecutive sensor reading s
ω_{gyro} -> angular velocity computed by the gyroscope
θ_{acc} -> angle computed only by the accelerometer readings



EMG signal processing

In this project we utilised the MyoWare 2.0 muscle sensor by Sparkfun along with the Link Shield and Cable shield, 3 connector electrode pads, disposable surface electrodes. It is necessary to prepare the subject with application of isopropyl alcohol to the skin in order to obtain consistent results. IPA removes oils, dead skin, and moisture, reducing skin impedance. The sensor is capable of performing some preprocessing steps including amplification (by a gain of 200), band pass filtering (over a range of 20 Hz – 498 Hz), rectification and offset removal. However, in practice, these processing steps becomes insufficient considering the mechanical and electrical noise imparted by the surroundings. To further smoothen and obtain consistent data we have subjected the readings obtained to some more processing methods.

- The raw data obtained is in form of 10-bit ADC data, which needs to be converted to the corresponding analog voltage.

$$\frac{Resolution\ of\ the\ ADC}{System\ Voltage} = \frac{ADC\ Reading}{Analog\ Voltage\ Measured}$$

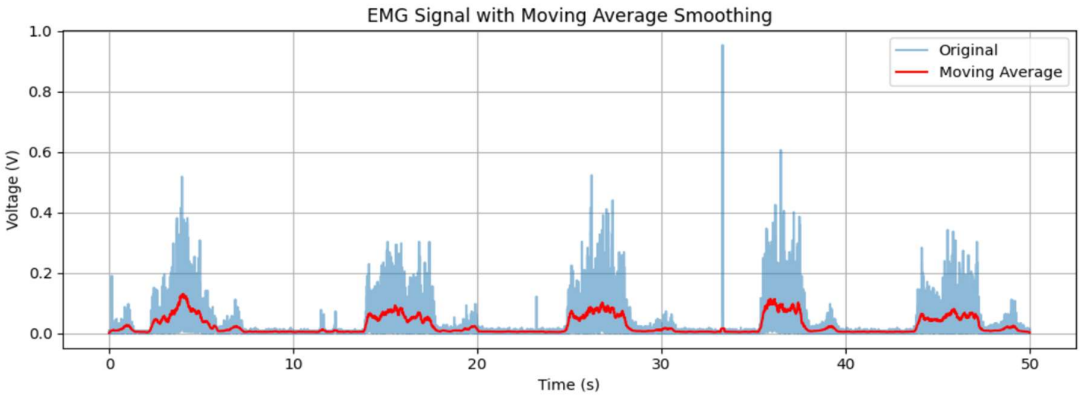
  In this case, Resolution of ADC = 1023, System Voltage = 5V.

- The presence of external noise may result in sudden unusually high voltage reading, the raw readings need to be first clipped to a legitimate magnitude before proceeding. This can be visually determined by looking at the plot of the raw readings. In our case it is 1 Volts.

- Although the signal obtained from the sensor is band pass filtered, to restrict the frequency to a shorter range in order to obtain signal with less noise, one can perform Fast Fourier Transform (FFT) to the data obtained and visually check the limits above which data loss might be negligible. After determining the limits, one can perform band pass filtering using Butterworth filter method. In our case, it was not necessary.

- It becomes essential to smoothen the data obtained by computing a feature of the clipped signal, commonly known as feature extraction. This feature is used for further

computation. The relevant features for smoothening might be RMS, moving average, variance, WAMP. This feature extracted is done over window size of n samples. The higher the value of n better will be the smoothening but may lead to loss of data leading to discrepancy. A window size of around 100 - 200 samples for a sampling frequency of 1000 Hz is considered a decent choice.



We can observe from the graph that moving average curve can provide reliable information to detect activation.

- Digital smoothing techniques like moving average (MA) create a smoother signal but don't strictly control frequency content. A low-pass filter (e.g., Butterworth) provides a more precise frequency-domain envelope extraction, offering better fidelity than a simple 100 ms moving average. The low pass filter (Butterworth) that we utilised is of order 1, implying it utilises the current and previous 1 reading to filter the signal.

$$y[n] = b_0 x[n] + b_1 x[n-1] - a_1 y[n-1]$$

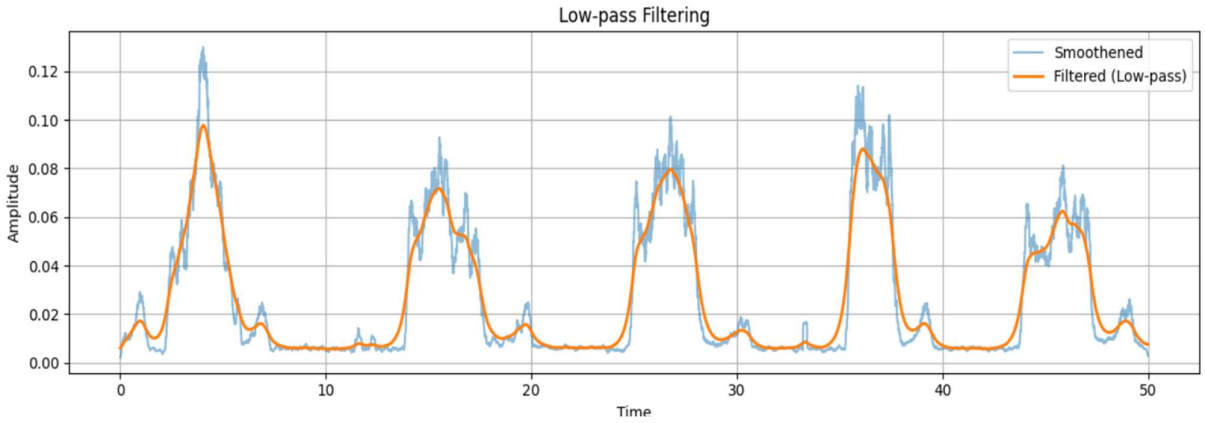
Where b0, b1, a1 → Butterworth coefficients

x[n] → nth input value

y[n] → nth filtered value

The filter coefficients are dependent on the sampling rate, cutoff frequency and order of filter. Thus, they remain constants and we have determined it by using the butter() method included in scipy.signal library in Python programming language.

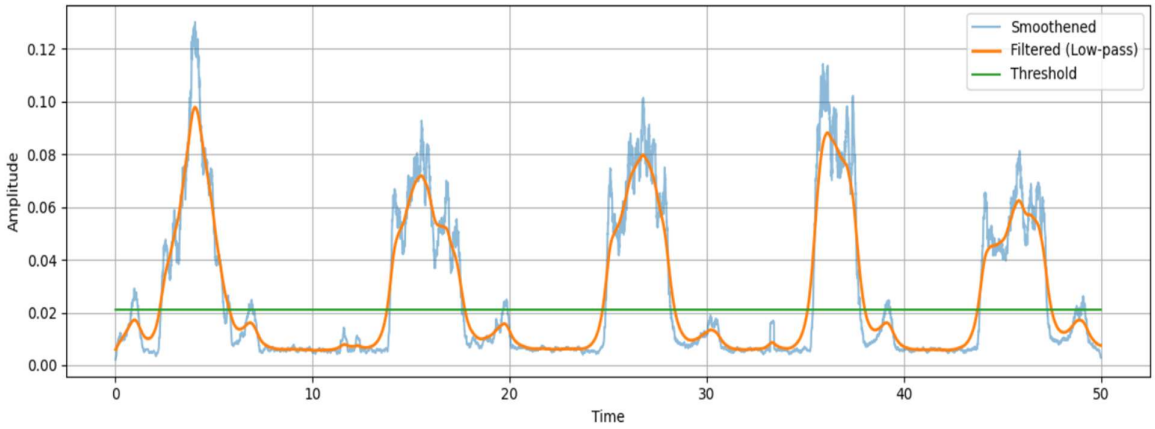
A higher order filter provides more accurate results but the time complexity involved is also high, therefore not recommended for real time processing.



- The signal obtained after processing so far is quite informative enough to determine the point of activation of the muscles. Now the main task is to determine the threshold for activation of the muscle. The EMG signal magnitude varies hugely on different subjects. This provides a limitation to set the threshold at a given magnitude by the signals obtained from handful of subjects. For this case, different research papers have proposed different algorithms to determine the activation point. We have utilised the method proposed by DiFabio in 1987, which suggests setting activation when signal is greater than 3SD from baseline, or

$$\text{Threshold} = \mu + 3\sigma$$

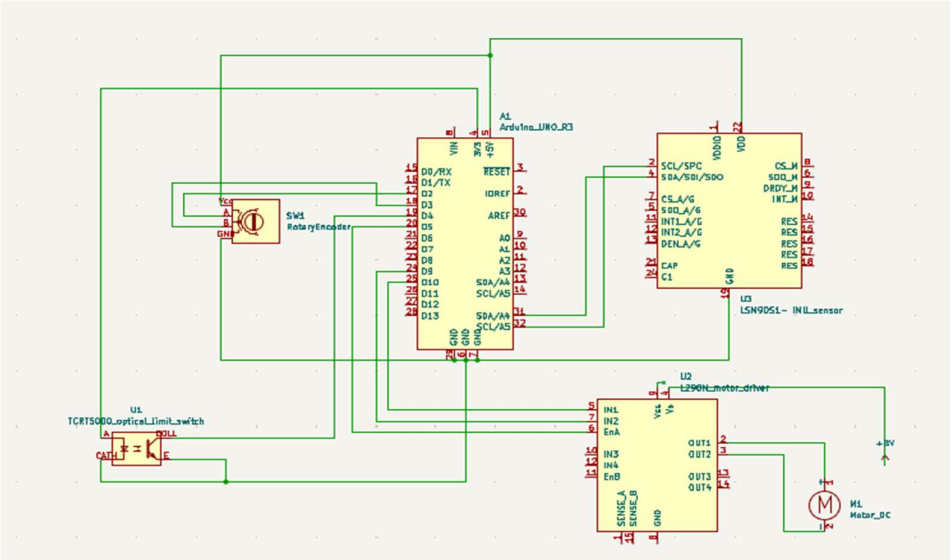
The results obtained using different parameters and on different subjects provided consistent results, and can be considered a decent method to determine muscle activation.



The above methods that we have used have been validated using the raw data obtained from a subject. We have saved the raw ADC data from the subject multiple times and implemented these methods on the obtained data. However, the real time processing does offer some anomalous results based on which we had to update the parameters involved.

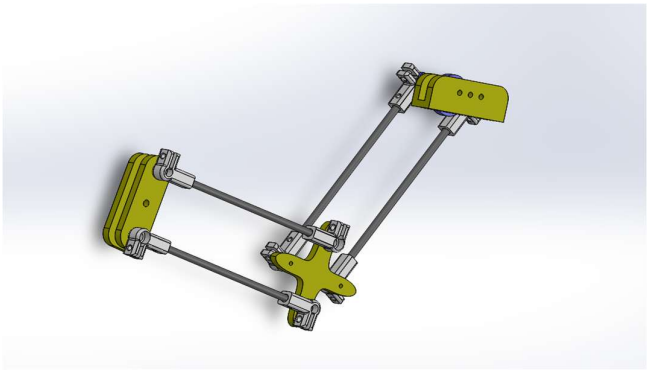
In context of saving the data for analysis, we made use of MATLAB's serialport object that is used to communicate with devices over a serial interface such as Arduino. This helped us saving the data as csv file and plot as png file. This data was further used to carry out the analysis.

Circuit Diagram



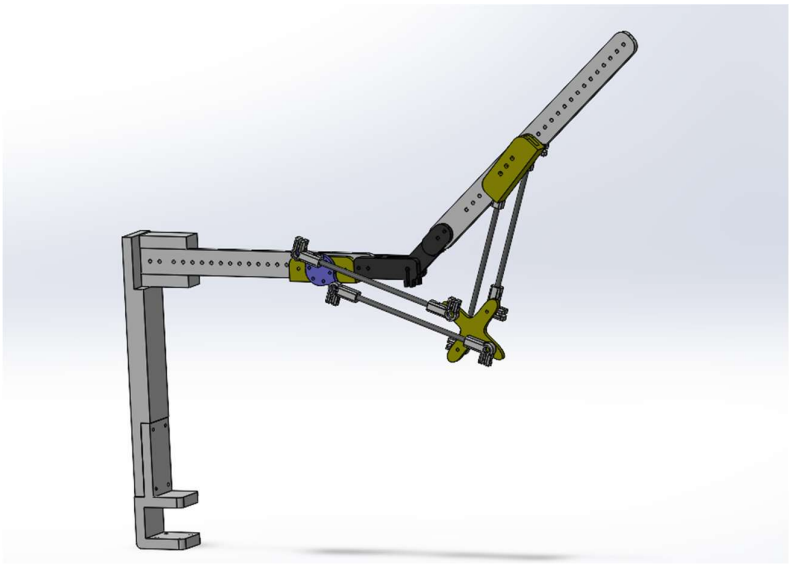
Hardware model

The model where our formulated objective is to implemented was designed in SolidWorks. The model prepared deploys a four-bar linkage and serves as a prototype in controlling the 1 DOF elbow joint.



The above figure refers to the link system that is externally attached to the upper arm to transmit the torque applied at its one end to the other end of the link system. Note that if we keep all the links of the same length, then the angle described by the arm (output) is equal to the angle described by the motor shaft. This correspondence was vital in constructing this link system.

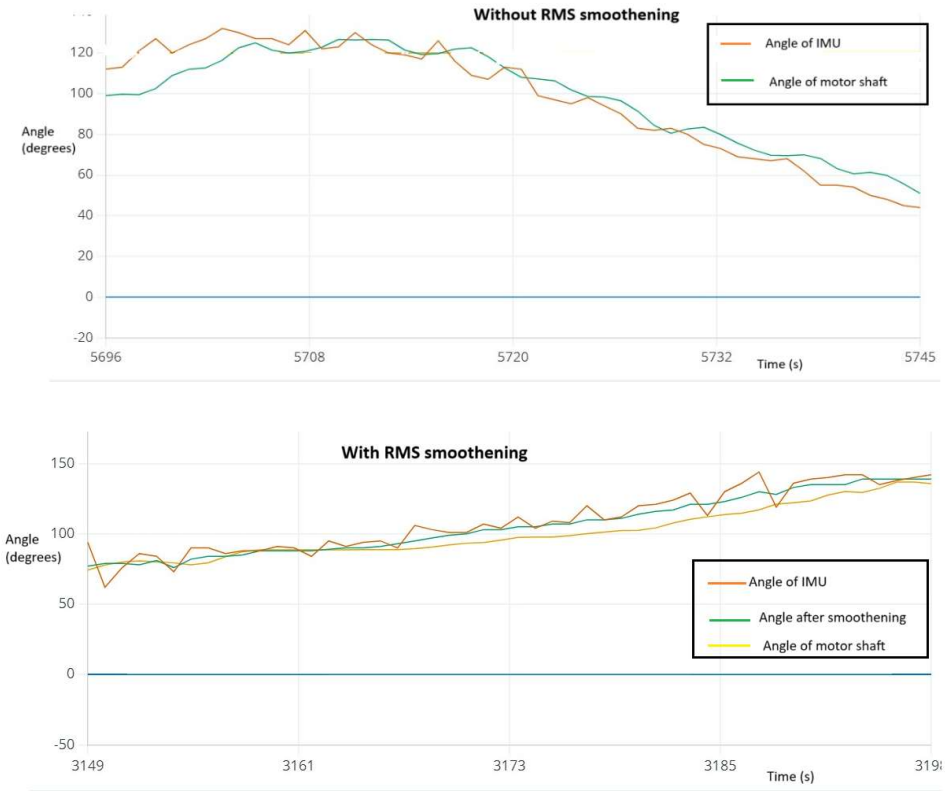
The assembled model is below.



Challenges faced during real time implementation

The above analysis from the data obtained from IMU sensor and EMG sensor did not turn out to be totally fruitful after assembling all components. Several unexpected anomalies occurred in the experimental setup which needed to be eliminated.

- First and foremost, homing of motor shaft was necessary to recalibrate the arm orientation at the start of the program execution. To achieve this, we made use of optical limit switches. We did not use stepper motor as it required external encoder.
- The angle obtained from the sensor after complementary filter when mapped to motor, still produced constant jitters in the motor. To overcome this, further smoothening of the IMU reading was required. We performed RMS smoothening of the sensor data over a window size of 10. The graphs below describe the effect of RMS smoothening on the output shaft angle.

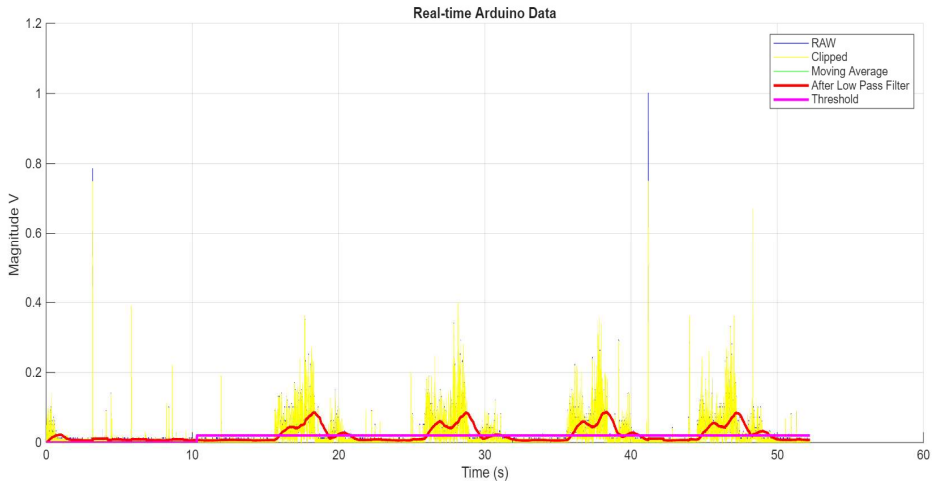


The RMS smoothened plot depicts that the motor shaft remains almost free from

jitters.

- Initially we were reading the EMG sensor reading simply using analogRead() and simply setting a delay of corresponding sampling rate, however from the saved data we observed that the variation in sampling varied hugely from time to time and in the process, there were severe data losses. In order to fix this issue, we used Interrupt from Hardware Timers using the library TimerOne.h and then we were able to obtain the reading at a fixed sampling rate of 1000Hz.
- Since our proposed algorithm of computing the threshold of muscle activation was based on calculations of the resting state, in real time implementation it becomes necessary for the subject to remain at a stationary state for a short interval (1-2 s), during which the baseline is obtained and the mean and standard deviation is calculated based on which the threshold is formed.
- Another big issue that we encountered is that we were supplying voltage to power the IMU sensor and the motor encoder from the 5V and 3.3V power supply pins of the Arduino UNO, this resulted in a power interference in the EMG signal obtained. To avoid this, we supplied all the voltages to the components through the external battery and kept the microcontroller as far as possible to the power source.
- Another issue that almost took a lot of time to resolve was that operating the motor alongside the EMG signal acquisition in real time imposed a huge delay and data loss in signal acquisition. The reason behind this was that we were using timer interrupts to read the EMG signals for which the statements executed should be short and take less time to execute, however, inclusion of motor control statements like digitalWrite() and analogWrite() takes significant time to process. In our revised logic we included the logic of running timer interrupts only during resting state, once muscle activation is reached, we disable the timer interrupt and proceed with motor control based on the input IMU readings.

The graph below depicts the real time EMG signal processing of a trial.



CONCLUSION

The integration of EMG (Electromyography) and IMU (Inertial Measurement Unit) sensing technologies offers a promising pathway to intuitive, responsive, and natural human–robot interaction. Through the implementation of a 1-DOF robotic manipulator controlled by elbow motion, the study successfully demonstrates how the intentional muscle activity, captured via EMG, and the physical motion, captured via IMU, can be integrated to actuate a robotic limb with precision. These find huge application in the field of rehabilitation robotics. The accurate measurement and replication of joint angles can help patients recover from stroke, orthopaedic injuries, or neuromuscular disorders perform assisted movement tasks. The system can be extended to multi-DOF rehabilitation exoskeletons or upper-limb assistive devices. Incorporating additional motors and sensors can expand the system to replicate full-arm movements, including wrist and shoulder, providing a more natural and complete motion model. Integrating machine learning models (e.g., SVM, LSTM, CNN) trained on fused EMG-IMU data can further improve gesture recognition and enable prediction of complex actions.

ANNEXURE

In this section, we include the important code snippets that we have used in our project

- Program to display the angle computed after applying complementary filter on IMU reading.

```
#include <MPU9250_WE.h>
#include <Wire.h>
#define MPU9250_ADDR 0x68

MPU9250_WE myMPU9250 = MPU9250_WE(MPU9250_ADDR);

float accRoll;
float roll = 0.0;
float alpha = 0.98;          // Weight for gyro in complementary filter
unsigned long lastTime = 0;

void setup() {
  Serial.begin(115200);
  Wire.begin();
  if(!myMPU9250.init()){
    Serial.println("MPU9250 does not respond");
  }
  else{
    Serial.println("MPU9250 is connected");
  }
  Serial.println("Position you MPU9250 flat and don't move it -calibrating...");
  delay(1000);
  myMPU9250.autoOffsets();     //calibration
  Serial.println("Done!");
}
void loop() {

  unsigned long now = millis();
  float dt = (now - lastTime) / 1000.0;
  lastTime = now;

  xyzFloat acc = myMPU9250.getGValues();
  xyzFloat gyr = myMPU9250.getGyrValues();

  accRoll  = atan2(acc.y, acc.z) * 180.0 / PI;
  roll  = alpha * (roll + (gyr.x) * dt) + (1 - alpha) * accRoll;

  Serial.print(accRoll);
  Serial.print(",");
  Serial.print(roll);
  Serial.print(",");
  Serial.print(180);
  Serial.print(",");
  Serial.println(0);
  delay(50);
}
```

- Program to implement optical limit switch

```
const int limitSwitchPin = 7;  // Digital pin connected to limit switch
bool lastState = HIGH;         // Previous state of the switch (initially not
pressed)
void setup() {
  pinMode(limitSwitchPin, INPUT_PULLUP);  // Enable internal pull-up resistor
  Serial.begin(9600);
}

void loop() {
  lastState = digitalRead(limitSwitchPin);
  if(lastState == HIGH)
      Serial.println("HIGH");
  // Detect if switch has just been pressed
  else  Serial.println("LOW");
  delay(10);  // Optional debounce delay
}
```

- Program demonstrating the use of timer interrupts to read EMG signal at fixed sampling rate

```
#include <TimerOne.h>
#define SAMPLE_RATE 1000

const int sensorPin = A0;
volatile bool sampleFlag = false;  // Flag set by timer ISR
unsigned long timestamp = 0;       // For sampled time
bool logging = true;               // Enable/disable logging

void setup() {
  Serial.begin(250000);
  Timer1.initialize(1000000/SAMPLE_RATE);          // 1ms interval
  Timer1.attachInterrupt(timerISR);  // Lightweight ISR
}

void timerISR() {
  if (logging) {
    sampleFlag = true;               // Just set a flag
  }
}

void loop() {
  if (sampleFlag) {
    sampleFlag = false;             // Clear the flag immediately

   unsigned long t_us = micros();  // Capture timestamp as close to flag as
possible
    int sensorValue = analogRead(sensorPin);
    float voltage = sensorValue * (5.0 / 1023.0);

    Serial.print(t_us / 1000000.0, 4);  // Convert to seconds and print
    Serial.print(',');
    Serial.println(sensorValue);

  }
}
```

- Program that implements real time processing of the input EMG signal data

```
#include <TimerOne.h>

#define EMG_PIN A0
#define SAMPLE_RATE 1000        // Hz
#define CLIP_LIMIT 0.75         // volts
#define WINDOW_SIZE 200          // For moving average
#define LPF_B 0.00156           // Low-pass filter coefficient (0-1)
#define LPF_A -1.0
#define BASELINE_DURATION 2000  // First 2s = 2000 samples

volatile bool sampleFlag = false;  // Flag set by timer ISR

float clippedSignal = 0;
float filteredSignal = 0;
float lpfOutput = 0;

float baselineSum = 0;
float baselineSqSum = 0;
int baselineCount = 0;
float mean = 0, stddev = 0;

float window[WINDOW_SIZE];
int windowIndex = 0;
bool baselineDone = false;

float threshold = 0.0;
float threshold1 = 0.0;
static float prev = 0;

void setup() {
  Serial.begin(250000);
  Timer1.initialize(1000000 / SAMPLE_RATE);  // 1ms = 1000Hz
  Timer1.attachInterrupt(readEMG);
}

void readEMG() {
  sampleFlag = true;
}

void loop() {
  if(sampleFlag){
    sampleFlag = false ;

    unsigned long t_us = micros();

    int rawADC = analogRead(EMG_PIN);
    float voltage = (rawADC / 1023.0) * 5;  // Convert to volts

    Serial.print(t_us / 1000000.0, 4);  // Convert to seconds and print
    Serial.print(',');
    Serial.print(voltage,5);

    float clipped = clipAmplitude(voltage);
    float averaged = movingAverage(clipped);
    lpfOutput = lowPassFilter(averaged);

    Serial.print(',');
    Serial.print(clipped);
    Serial.print(',');
    Serial.print(averaged,5);
    Serial.print(',');
```

```
    Serial.print(lpfOutput,5);

    if (!baselineDone) {
      threshold1 = updateBaseline(lpfOutput);
    } else {
      threshold = threshold1;
      detectActivation(lpfOutput);
    }
  Serial.print(',');
  Serial.println(threshold,5);
  }

}

float clipAmplitude(float val) {
  if (val > CLIP_LIMIT) return CLIP_LIMIT;
  return val;
}

float movingAverage(float input) {
  window[windowIndex] = input;
  windowIndex = (windowIndex + 1) % WINDOW_SIZE;

  float sum = 0;
  for (int i = 0; i < WINDOW_SIZE; i++) sum += window[i];
  return sum / WINDOW_SIZE;
}

float lowPassFilter(float input) {
  float output = LPF_B * input + LPF_B * prev - LPF_A * prev;
  prev = input;
  return output;
}

float updateBaseline(float value) {
  baselineSum += value;
 baselineSqSum += value * value;
  baselineCount++;

  if (baselineCount >= BASELINE_DURATION) {
    mean = baselineSum / baselineCount;
    float variance = (baselineSqSum / baselineCount) - (mean * mean);
    stddev = sqrt(variance);
    baselineDone = true;
    return mean+3*stddev ;
  }
}

void detectActivation(float val) {
  float threshold = mean + 3 * stddev;
  if (val > threshold) {
    Serial.println("Muscle Activated");
  } else {
    Serial.println("Muscle Inactive");

  }
}
```

- Program to control motor shaft using IMU sensor only

```
#include <Encoder.h>
#include <MPU9250_WE.h>
#include <Wire.h>
```

```
#include <math.h>

#define MPU9250_ADDR 0x68
// Motor control pins (use with motor driver)
#define MOTOR_FWD 10
#define MOTOR_REV 9
#define PWM_EN    5  // PWM pin to control speed

// Encoder pins
#define ENCODER_A 2
#define ENCODER_B 3

float alpha = 0.98; // tuning factor
float roll =  0.0;

// Motor + Encoder specs
const int gearRatio = 100;
const int pulsesPerRev = 28;  // pulses per shaft revolution
const float degreesPerPulse = 360.0 / (gearRatio * pulsesPerRev);

MPU9250_WE myMPU9250 = MPU9250_WE(MPU9250_ADDR);
Encoder motorEncoder(ENCODER_A, ENCODER_B);

// PID variables
float Kp = 5;
float Ki = 2.5;
float Kd = 1;

float errorSum = 0;
float lastError = 0;

long lastTime = 0;
int targetAngle ;  // stores the IMU input
const int N = 10;
int idx = 0;
int angle_buffer[N] = {0};

void setup() {
  Serial.begin(9600);

  pinMode(MOTOR_FWD, OUTPUT);
  pinMode(MOTOR_REV, OUTPUT);
  pinMode(PWM_EN, OUTPUT);

  motorEncoder.write(0);
  lastTime = millis();

  Wire.begin();
  if(!myMPU9250.init()){
    Serial.println("MPU9250 does not respond");
  }
  else{
    Serial.println("MPU9250 is connected");
  }
  if(!myMPU9250.initMagnetometer()){
    Serial.println("Magnetometer does not respond");
  }
  else{
    Serial.println("Magnetometer is connected");
  }
  Serial.println("Position you MPU9250 flat and don't move it -calibrating...");
  myMPU9250.autoOffsets();
  Serial.println("Done!");
  delay(2000);
  lastTime = millis();
```

```
}

void loop() {
  // --- Compute shaft position in degrees ---

        long pulseCount = motorEncoder.read();
        float rawAngle = pulseCount * degreesPerPulse;
        float shaftAngle = rawAngle ;

        xyzFloat acc = myMPU9250.getGValues();
        xyzFloat gyr = myMPU9250.getGyrValues();

        unsigned long now = millis();
        float dt = (now - lastTime) / 1000.0;

        float accRoll = atan2(acc.y, acc.z) * (180.0 / PI);
        // Gyro + Accel
        roll = alpha * (roll + gyr.x * dt) + (1 - alpha) * accRoll;
        float roll_new = roll;

        angle_buffer[idx] = int(roll_new);
        targetAngle = int(computeRMS(angle_buffer));
        idx = (idx + 1) % N ;
        // --- Compute PID ---
        int error = targetAngle - shaftAngle;

        errorSum += ((error + lastError)/2) * dt;     //for integration term
        float dError = (error - lastError) / dt;  //

        float pidOutput = Kp * error + Ki * errorSum + Kd * dError;

        lastError = error;
        lastTime = now;

        // --- Apply motor control ---
        int pwm = constrain(abs(pidOutput), 0, 255);

        if (abs(error) < 0.5) {   //if rotor angle is close enough to the target
          analogWrite(PWM_EN, 0);
          digitalWrite(MOTOR_FWD, LOW);
          digitalWrite(MOTOR_REV, LOW);
        }
        else if (pidOutput > 0) {
      digitalWrite(MOTOR_FWD, HIGH);
          digitalWrite(MOTOR_REV, LOW);
          analogWrite(PWM_EN, pwm);
        } else {
          digitalWrite(MOTOR_FWD, LOW);
          digitalWrite(MOTOR_REV, HIGH);
          analogWrite(PWM_EN, pwm);
        }

        // --- Debug info ---
        Serial.print("0");
        Serial.print(",");      // Y-axis minimum
        Serial.print(roll_new);
        Serial.print(",");
        Serial.print(targetAngle);
        Serial.print(",");
        Serial.print(shaftAngle);
        Serial.print(",");
        Serial.println("180");  // Y-axis maximum
        delay(5);  // Small delay for stability
}
```

```
float computeRMS(int *buffer) {
  float sumSq = 0;
  for (int i = 0; i < N; i++) {
    sumSq += buffer[i] * buffer[i];
  }
  return sqrt(sumSq / N);
}
```

- Program to implement both IMU and EMG sensor to actuate the motor

```
#include <TimerOne.h>
#include <Encoder.h>
#include <MPU9250_WE.h>
#include <Wire.h>
#include <math.h>

#define MPU9250_ADDR 0x68

// Motor control pins (use with motor driver)
#define MOTOR_FWD 10
#define MOTOR_REV 9
#define PWM_EN    5  // PWM pin to control speed

// Encoder pins
#define ENCODER_A 2
#define ENCODER_B 3

#define EMG_PIN A0
#define SAMPLE_RATE 1000     // Hz
#define CLIP_LIMIT 0.75         // volts
#define WINDOW_SIZE 200          // For moving average
#define LPF_B 0.00156           // Low-pass filter coefficient (0-1)
#define LPF_A -1.0
#define BASELINE_DURATION 1000 // First 1000 samples

float alpha = 0.98; // tuning factor
float dt = 0.01;    // 10 ms loop

// Motor + Encoder specs
const int gearRatio = 100;
const int pulsesPerRev = 28;  // pulses per shaft revolution
const float degreesPerPulse = 360.0 / (gearRatio * pulsesPerRev);

MPU9250_WE myMPU9250 = MPU9250_WE(MPU9250_ADDR);
Encoder motorEncoder(ENCODER_A, ENCODER_B);

// PID variables
float Kp = 5;
float Ki = 2.5;
float Kd = 1;


float errorSum = 0;
float lastError = 0;
float roll = 0.0;

long lastTime = 0;
int targetAngle ;  // stores the IMU input
const int N = 10;
int idx = 0;

int angle_buffer[N] = {0};
volatile bool sampleFlag = false;  // Flag set by timer ISR
```

```
float clippedSignal = 0;
float filteredSignal = 0;
float lpfOutput = 0;

float baselineSum = 0;
float baselineSqSum = 0;
int baselineCount = 0;
float mean = 0, stddev = 0;

float window[WINDOW_SIZE];
int windowIndex = 0;
bool baselineDone = false;

float threshold = 0.0;
float threshold1 = 0.0;
static float prev = 0;

unsigned long lastMotorUpdate = 0;
const unsigned long motorInterval = 20; // ms

bool flag_detect = false;
bool imu_start = false;


void setup() {
  Serial.begin(250000);

  pinMode(MOTOR_FWD, OUTPUT);
  pinMode(MOTOR_REV, OUTPUT);
  pinMode(PWM_EN, OUTPUT);

  digitalWrite(MOTOR_FWD, LOW);
  digitalWrite(MOTOR_REV, LOW);
  analogWrite(PWM_EN, 0);

  motorEncoder.write(0);
  lastTime = millis();

  Wire.begin();
  if(!myMPU9250.init()){
    Serial.println("MPU9250 does not respond");
  }
  else{
    Serial.println("MPU9250 is connected");
  }
  if(!myMPU9250.initMagnetometer()){
    Serial.println("Magnetometer does not respond");
  }
  else{
    Serial.println("Magnetometer is connected");
  }
  Serial.println("Position you MPU9250 flat and don't move it - calibrating...");
  myMPU9250.autoOffsets();
  Serial.println("Done!");
  delay(1000);
  Timer1.initialize(1000000 / SAMPLE_RATE);  // 1ms = 1000Hz
  Timer1.attachInterrupt(readEMG);
}

void readEMG() {
  sampleFlag = true;
}

void loop() {
  if(sampleFlag & !imu_start){
```

```
    sampleFlag = false ;

    unsigned long t_us = micros();

    int rawADC = analogRead(EMG_PIN);
    float voltage = (rawADC / 1023.0) * 5;  // Convert to volts

    Serial.print(t_us / 1000000.0, 4);  // Convert to seconds and print
    Serial.print(',');
    Serial.print(voltage,5);

    float clipped = clipAmplitude(voltage);
    float averaged = movingAverage(clipped);
    lpfOutput = lowPassFilter(averaged);

    Serial.print(',');
    Serial.print(clipped);
    Serial.print(',');
    Serial.print(averaged,5);
    Serial.print(',');
    Serial.print(lpfOutput,5);

    if (!baselineDone) {
      threshold1 = updateBaseline(lpfOutput);
    } else if(flag_detect==false) {
      threshold = threshold1;
      flag_detect = detectActivation(lpfOutput);
    } else if(flag_detect==true) {
      Timer1.detachInterrupt();
      imu_start = true;
      lastTime = millis();
    }
    Serial.print(",");
    Serial.println(threshold);
  }
  else if(imu_start == true)
  {
      motor_control();
  }
}

float clipAmplitude(float val) {
  if (val > CLIP_LIMIT) return CLIP_LIMIT;
  //if (val < -CLIP_LIMIT) return -CLIP_LIMIT;
  return val;
}

float movingAverage(float input) {
  window[windowIndex] = input;
  windowIndex = (windowIndex + 1) % WINDOW_SIZE;

  float sum = 0;
  for (int i = 0; i < WINDOW_SIZE; i++) sum += window[i];
  return sum / WINDOW_SIZE;
}

float lowPassFilter(float input) {
  float output = LPF_B * input + LPF_B * prev - LPF_A * prev;
  prev = input;
  return output;
}

float updateBaseline(float value) {
  baselineSum += value;
  baselineSqSum += value * value;
```

```
  baselineCount++;

  if (baselineCount >= BASELINE_DURATION) {
    mean = baselineSum / baselineCount;
    float variance = (baselineSqSum / baselineCount) - (mean * mean);
    stddev = sqrt(variance);
    baselineDone = true;
    return mean+3*stddev ;
  }
}
bool detectActivation(float val) {
  float threshold = mean + 3*stddev;
  if (val > threshold) {
    return true;
  } else {
    return false;
  }
}

void motor_control(){

        long pulseCount = motorEncoder.read();
        float rawAngle = pulseCount * degreesPerPulse;
        float shaftAngle = rawAngle ;

        unsigned long now = millis();
        float dt = (now - lastTime) / 1000.0;  // seconds

        xyzFloat acc = myMPU9250.getGValues();
        xyzFloat gyr = myMPU9250.getGyrValues();

        float accRoll = atan2(acc.y, acc.z) * (180.0 / PI);

        // Gyro + Accel
        roll = alpha * (roll + gyr.x * dt) + (1 - alpha) * accRoll;
        float roll_new = roll;

        angle_buffer[idx] = int(roll_new);
        targetAngle = int(computeRMS(angle_buffer));
        idx = (idx + 1) % N ;
        // --- Compute PID ---
        int error = targetAngle - shaftAngle;

        errorSum += ((error + lastError)/2) * dt;     //for integration term
        float dError = (error - lastError) / dt;  //

        float pidOutput = Kp * error + Ki * errorSum + Kd * dError;

        lastError = error;
        lastTime = now;

                // --- Apply motor control ---
                int pwm = constrain(abs(pidOutput), 0, 255);

                if (abs(error) < 0.5) {   //if rotor angle is close enough to the target
                   analogWrite(PWM_EN, 0);
                  digitalWrite(MOTOR_FWD, LOW);
                  digitalWrite(MOTOR_REV, LOW);
                }
                else if (pidOutput > 0) {
                  digitalWrite(MOTOR_FWD, HIGH);
                  digitalWrite(MOTOR_REV, LOW);
                  analogWrite(PWM_EN, pwm);
                } else {
                  digitalWrite(MOTOR_FWD, LOW);
```

```
          digitalWrite(MOTOR_REV, HIGH);
          analogWrite(PWM_EN, pwm);
        }
        delay(10);  // Small delay for stability
}
float computeRMS(int *buffer) {
  float sumSq = 0;
  for (int i = 0; i < N; i++) {
    sumSq += buffer[i] * buffer[i];
  }
  return sqrt(sumSq / N);
}
```