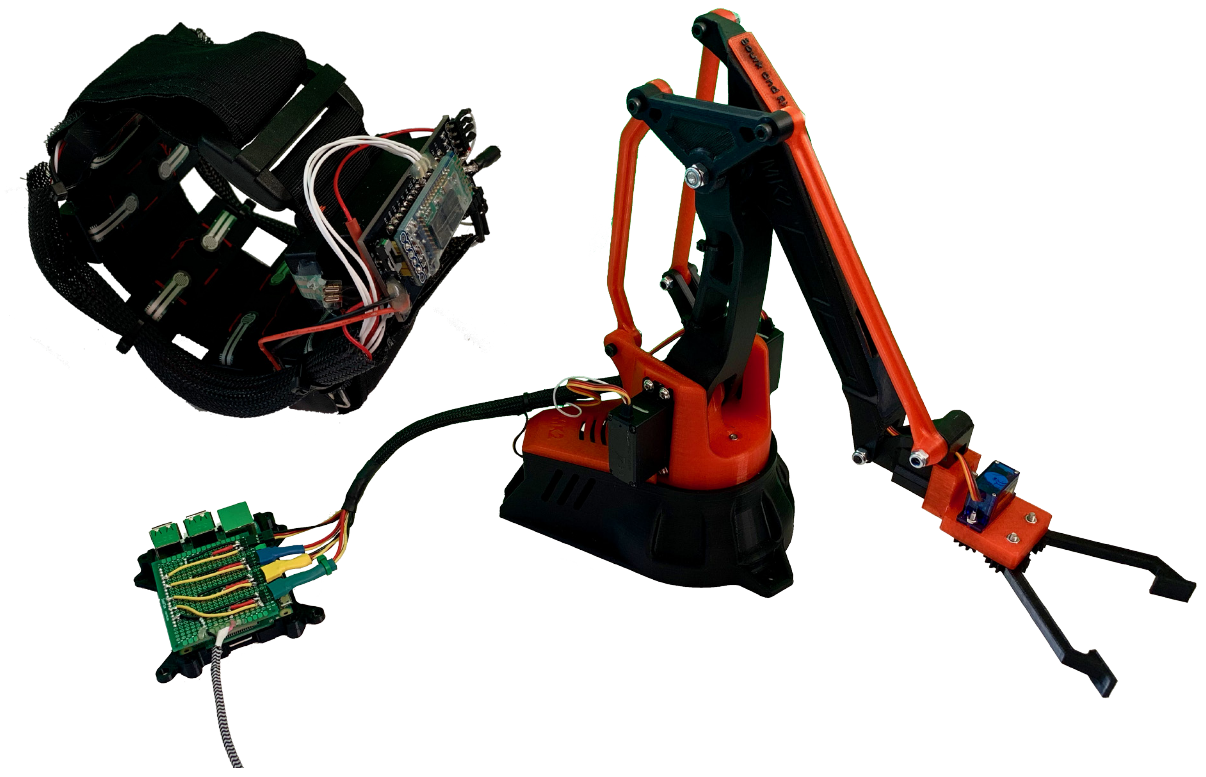
# Optimizing Force Myography for Real-Time Prosthetic Control Using Force Sensitive Resistors and an Inertial Measurement Unit



*Force Myography Sensor Band and Simulated Prosthetic Robotic Arm*

**Eric Wells and Mark Sherstan**

Contributions: 50 50

MEC E 653 - Final Report

Instructor: Dr. Albert Vette

April 10th, 2019

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## BACKGROUND AND RATIONALE

Upper limb amputation inevitably results in loss of both motor and sensory function causing an impact on an individual's economic, psychological, and social well-being. Prosthetics technology research attempts to mitigate these effects by restoring functionality to the lost limb. Recent research in the area focuses on electrically powered prostheses controlled by the muscle signals in the residual limb, termed myoelectric control [1]. These myoelectric devices utilize the existing neural pathways responsible for natural movement, offering an intuitive control interface. Although promising advancements have been made, studies report rejection rates for electrically powered prosthetics as high as 75% in children and 50% in adults [2]. Another study showed that the most common reason for abandonment was amputees being more functional in daily activities without the use of a prosthesis (98% of responses) [3].

Although there are many multifunctional hands available for prosthesis use [4], [5], commercial devices are unable to utilize them due to restricted control interfaces from the user. Recent research has focused on improving pattern recognition algorithms to decode the electromyographic signals (EMG) into multi degree of freedom (DOF) movements to enable more robust control for the user [1], [6]. Although results are promising in laboratory conditions, this technique has been unable to be transferred into commercial devices. This is due to current surface mounted EMG sensors having inherent unpredictable issues that affect the algorithm's performance such as cross-talk, electrical interference, muscle fatigue effects, perspiration effects, and skin impedance changing over the course of a day [7].

An alternative to this technology for use in prosthetics is force myography (FMG) [8]. This technology seeks to exploit the deformation resulting from residual muscle movement to obtain prosthesis control. This is typically done in a very similar manner to EMG, however instead of measuring electrical activity, an array of pressure sensors wrapped around the user's arm measures muscle deformation. The measured sensor values are used as inputs into a pattern recognition algorithm to predict the desired motions from the user. Many studies have shown success in offline tasks [8]–[15]. However, little work has been done to investigate pre-processing of the raw sensor signals before being fed into the pattern recognition algorithm. Additionally, all of the previous studies were done using offline classification metrics, leaving the question of whether the offline results translate to online performance.

## OBJECTIVES

The objective of the project is to optimize an FMG sensor and algorithm package that can be used to robustly control upper limb prosthetics in real time. To measure muscle deformation signals, eleven Force Sensitive Resistor (FSR) sensors will be placed around the forearm. Identifying the physical response in multiple locations on the arm will yield greater control of the prosthetic as more of the arm's muscles and tendons can be interpreted. An Inertial Measurement Unit (IMU) will be used to determine the arms orientation in space and how the arm is moving in terms of acceleration and angular velocity which is an unprecedented approach with FMG technology. When combined with the FSR signals the extra data from the IMU will allow the prosthetic control to not only rely on the muscle activity, but also the arm dynamics. Such data provides alternative control schemes and finer control due to a more accurate mapping of the arm. Digital filters will then be systematically investigated and implemented. Upon filtering and processing of the FSR and IMU signals, machine learning will be utilized to create a classification controller for a desktop mounted robotic arm, simulating a prosthetic, which will be ran in real time. Minimizing the sensor acquisition to robotic gesture execution time through hardware and rapid execution filters will be a significant focus for successfully achieving real time control as only offline studies have been performed.

## METHODS

### DATA ACQUISITION AND HARDWARE

To measure the force created by musculotendinous changes in the forearm and the orientation and movement of the arm a low profile, variable sized armband was designed and manufactured. The armband includes a custom designed printed circuit board (PCB) which interfaces with eleven FSR's, an IMU, microcontroller, and Bluetooth module. The armband wirelessly sends raw sensor values to a microprocessor where all computation takes place such as digital signal processing and real time control of a desktop mounted robotic arm. A high level overview of the signal flow can be seen in Figure 1.

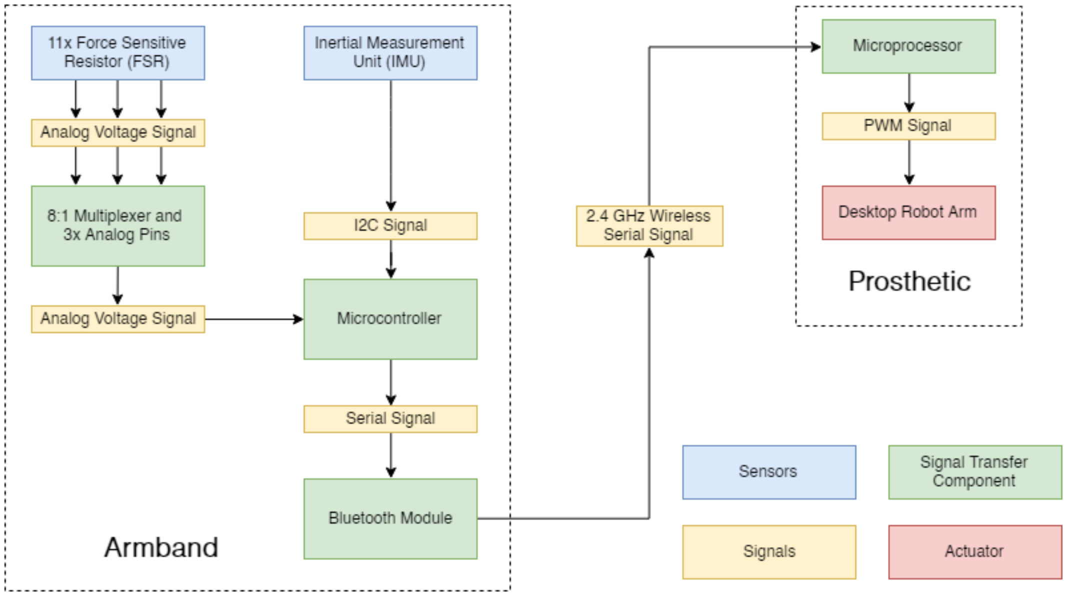


**Figure 1:** Signal Flow Chart

The IMU is a 6 DOF motion tracking device consisting of a three dimensional accelerometer and gyroscope. The accelerometer measures acceleration and the gyroscope angular velocity. Measurements are taken in the x (North), y (East), and z (Down) directions relative to a North-East-Down (NED) coordinate system. The right hand rule is followed for positive rotations with roll (ϕ) about the x axis, pitch (θ) about the y axis, and yaw (Ψ) about the z axis. Figure 2 displays the coordinate system on a users arm where the PCB with the IMU should be placed. The raw values from the IMU are passed to the microcontroller using an Inter-Integrated Circuit (I2C) with the combination of a serial data line (SDA) and serial clock line (SCL). To better interpret the data, the raw gyroscope values are subtracted from a calibration value to neglect any offset, and then divided by a sensitivity scale factor to yield a measurement in degrees per second (deg/s) (Appendix B.1 lines 210 to 244). The accelerometer is also divided by its respective sensitivity scale value to yield a measurement in g-force (g) (Appendix B.1 lines 230 to 244).

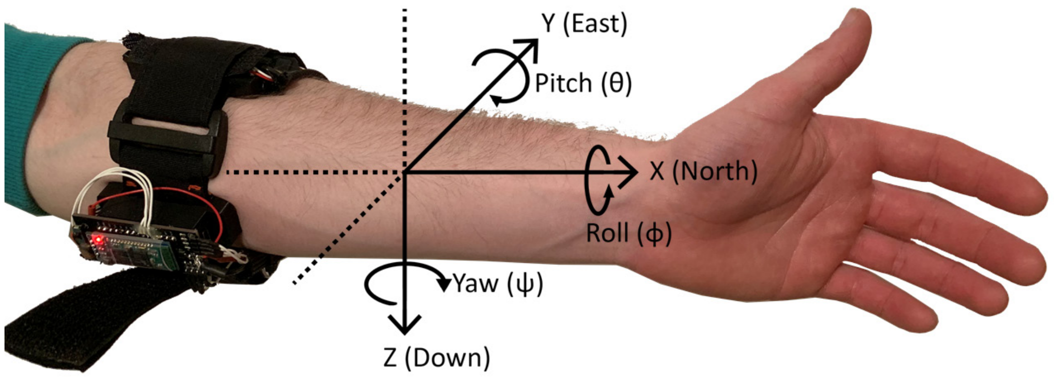


**Figure 2:** Armband Coordinate System with Slight Negative Roll

Eleven FSR pressure sensors are located in the armband, providing sufficient coverage of most forearms as in [9], [11]. The FSR's are equally spaced around the circumference of the forearm of the user and are responsible for measuring the pressure change resulting from muscle deformation. The FSR's are polymer thick film passive devices with an analog signal that decreases in resistance when a force is applied to the sensor pad. A multiplexer bridges eight FSR sensors to a single analog data line. The main purpose of the multiplexer is to increase the total analog input of the microcontroller and allow for sufficient coverage of the arm with sensors. The remaining three FSR's are read by three analog pins on a microcontroller. The FSR's ADC output is converted to force by calculating the resistance value of the sensor and linearly mapping it to the manufacturers conversion plot (Appendix B.1 lines 268 to 290).

Both the IMU and FSR raw signals are interpreted by a microcontroller. The raw values are then encoded for transmitting via a Bluetooth module which wirelessly transmits data at 2.4 GHz using a Bluetooth Serial Port Profile (SPP). A microprocessor equipped with Bluetooth technology decodes and processes the raw values so real time control can be achieved. It should be noted that the primary use of the microcontroller is to acquire data, and no calculations are performed on it. The signals are processed on a microprocessor due to the increased performance capabilities while microprocessors typically lack ease of interfacing with various analog sensors.

Once the signals are processed onboard the microprocessor they are passed to an onboard control system which maps the signal to a 4 DOF desktop mounted robotic arm. The robotic arm simulates a prosthetic and is controlled by servo motors receiving Pulse Width Modulation (PWM) signals generated by the control scheme. The robotic arm used is an open source, 3D printed arm termed EEZYbotARM MK2 [16] and is pictured on the title page. The remaining components including the manufacturer, manufacturer location, and manufacturer part number are tabulated in Table 1 and will be further discussed in the data acquisition section.

**Table 1:** Major Component Part List and Manufacturer Table

| General Part Name | Manufacturer | Manufacturer Location | Manufacturer Part Number |
|---|---|---|---|
| Inertial Measurement Unit (IMU) | TDK InvenSense | San Jose, USA | MPU-6050 |
| Force Sensitive Resistor (FSR) | Interlink Electronics | Westlake Village, USA | 30-49649 |
| 8:1 Multiplexer | Nexperia | Nijmegen, Netherlands | 74HC4051BQ,115 |
| Microcontroller | ELEGOO Inc. | Shenzhen, China | Arduino Nano V3.0 |
| Bluetooth Module | ITEAD Studio | Shenzhen, China | HC-05 |
| Microprocessor | Raspberry Pi Foundation | Cambridge, UK | Raspberry Pi 3B+ |

To ensure that the acquired signals are reliable and repeatable a custom PCB was designed, manufactured, and assembled. The PCB was designed as a shield to directly interface with the Arduino Nano V3.0 microcontroller. Creating the PCB allowed for permanent and reliable sensor connections with a small physical footprint preventing hindrance and discomfort to the user; another significant reason for abandonment of prosthetics [3]. The PCB was integrated into the armband and interfaced with the eleven FSR sensors while the IMU was directly mounted onto the PCB. When designing the PCB special consideration was taken for component selection, placement, and signal routing to minimize hardware noise and signal interference as per each manufacturers recommendation. With the implementation of the Bluetooth module the user does not need to be tethered to a computer or other fixed point further enhancing the user experience and reducing the risk of wires or connections failing from movement. A picture of the PCB and armband can be observed in Figure 3. A detailed schematic of the PCB and a list of all its components can be found in Appendix A along with references to datasheets for all components.

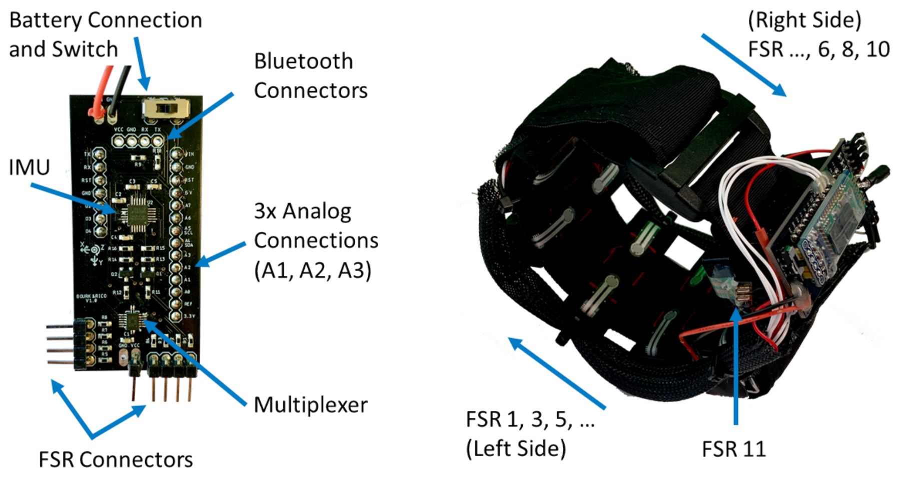

Battery Connection and Switch
Bluetooth Connectors
IMU
3x Analog Connections (A1, A2, A3)
Multiplexer
FSR Connectors
(Right Side) FSR ..., 6, 8, 10
FSR 1, 3, 5, ... (Left Side)
FSR 11

**Figure 3:** Custom PCB (Left) and Armband with FSR's (Right)

## EXPERIMENTAL PROCEDURE

The wristband was tightened onto the forearm of a test subject as seen in Figure 2. The subject then underwent a series of hand gestures as shown in Figure 4 where each position was held for approximately two seconds and 300 data points were obtained. Each gesture was executed once and over 30 different ~~trails~~ were conducted between two participants.

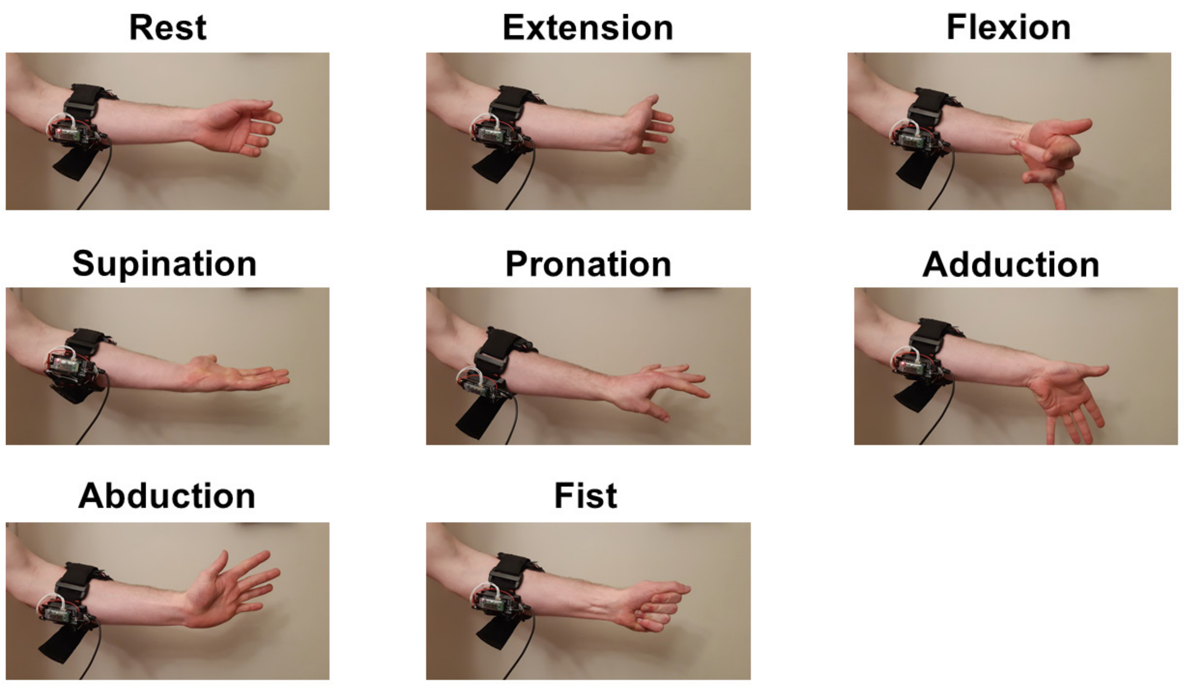


**Figure 4:** Eight Unique Gestures with Armband

## SIGNAL PROCESSING THEORY ON DATA ACQUISITION

### SAMPLE RATE

Human gestures can be classified as a random quantized signal, since the gesture (dependent variable) is happening continuously in time (independent variable), but has discrete values. However, digital sampling of these signals discritizes the time domain, shifting the signals into a random digital signal classification. A study done on hand motion frequency properties gave an upper bound on the frequency range of human gestures at 4.5 Hz [17]. Recent work in the literature undergoing similar implementations typically has a sample rate below 80 Hz ([9], [11]–[13],) and/or has a low pass filter with a cutoff frequency of less than 4 Hz ([8], [13], [14]). The Arduino Nano V3.0 is centered around the ATmega328P microcontroller which was programmed at a sampling frequency of 200 Hz, which exceeds that of the previous studies. The Arduino Nano V3.0 was selected due to the accessibility of the ATmega328P microcontroller pins, availability of a micro USB port for programming, and the proven track record and documentation of the microcontroller. The sampling frequency of 200 Hz yields a Nyquist frequency of 100 Hz, which also far exceeds the range of human gestures, allowing ample room for filter design. In order to maintain a constant sample rate, a time synchronization function was implemented in the embedded code which uses the internal crystal oscillator clock with a resolution of 4 microseconds. This sample and hold technique yields a sample frequency of 200 ± 0.16 Hz, with an error of ± 0.008%. It is ran once per sample and can be seen in detail Appendix B.2, lines 58 to 71.

Upon acquiring all 17 signals (11 FSR and 6 IMU), the data was cast into bytes for transmission by the HC-05 Bluetooth module. The HC-05 was selected as it can be configurable to be a master or slave, supports serial port profile (SPP) allowing for easy communication with microprocessors, and has a range up to 100 meters. As all raw signals are two byte integers a total of 34 bytes are packaged for transmission. To prevent reading incorrect or incomplete data a two byte header and two byte footer check is implemented (Appendix B.1 lines 135 to 159) where four unique hexadecimal numbers, two at the start and two at the end, of the transmission must be read before that package of data can be accepted and processed. Should the check fail the data package is discarded, however upon testing less than 0.1% of packets were lost. Although a fail slightly drops the real time sampling rate, it is still well above the Nyquist sampling rate and previous data points can be used without greatly affecting the real time control. This processes yields a bit rate of 60.8 kbps and a baud rate of 115200 was selected to ensure no bottlenecks affect the 200 Hz sampling rate. The signal processing and calculations are performed on a Raspberry Pi 3B+ as the microprocessor clock speed is nearly 90 times faster than the Arduino Nano V3.0 and the hardware allows for the implementation of various control techniques and digital signal processing not available in an Arduino environment.

## DIGITAL SIGNAL PROCESSING THEORY

The analog to digital conversion for the FSR signals occurs in the ATmega328P microcontroller onboard the Arduino Nano V3.0. The microcontroller uses a 10-bit analog to digital converter (ADC) giving a total of 1024 possible quantizations with a full scale range of 5 V. The MPU-6050 was selected due to its reliability, lost cost, and customizable full scale ranges. The IMU uses six internal ADC's to digitize the accelerometer and gyroscope values in the x, y, and z directions using the NED coordinate system defined earlier. The output of the ADC is a 16 bit signed integer (-32768 to 32768) with a 3.3 V full scale analog range.

The full scale range of the FSR is 0 to 2 kg which is set by the manufacturer. The full scale range was selected based on a ratio of cost to performance of the sensor where the Interlink 30-49649 provided the best results for the proposed application. The IMU has an adjustable full-scale range and is set at ± 2 g for the accelerometer and ± 250 deg/s for the gyroscope. The full scale range was experimentally determined by performing various common day tasks such as opening lids and picking and placing objects and observing the sensor response to maximize the use of all 16 bits and allow for more accurate measurements.

The quantization step *Q* and dynamic range *DR* are calculated using equations 1 and 2 respectively where *R* is the the full scale analog range and *N* is the number of bits. Table 2 tabulates a summary of the sensor attributes and the respective results.

$$Q = \frac{R}{2^N} \tag{1}$$

$$DR = 20\log_{10}(2^N) \tag{2}$$

**Table 2:** ADC and Digital Signal Processing Theory Summary

| Value | FSR | IMU |
|---|---|---|
| Analog to Digital Converter | ATmega328P | MPU-6050 |
| Full Scale Analog Range (R) | 5.0 V | 3.3 V |
| Number of Bits (N) | 10 | 16 |
| Quantization Step (Q) | 4.88 mV | 50.4 µV |
| Quantization Error ($Q_{error}$) | 2.44 mV<br>0.977 gram | 50.4 µV<br>$7.63 \cdot 10^{-3}$ deg/s<br>$6.10 \cdot 10^{-5}$ g |
| Dynamic Range (DR) | 60.2 dB | 96.3 dB |

As stated in the ATmega328P datasheet, the quantization error is half of the least significant bit (LSB), indicating that microprocessor has shifted the quantization levels to occur at the middle of the analog ranges. This creates a quantization error of half of the quantization step or 2.44 mV or 0.977 grams. The quantization error is approximately 0.5% of the average FSR values and the variability of the signals between gestures ranges on average at least 50 grams (2% of average) therefore the quantization error can be assumed negligible. Due to the small quantization error and low amounts of noise, a large signal to noise ratio was found further validating an acceptable quantization error. The quantization error can be reduced and SNR increased by changing to a 16 bit ADC which would increase the dynamic range from 60.2 dB to 96.3 dB, however with current results this has not been a concern.

The IMU does not shift the quantization step therefore, the quantization error and step are the same value. The IMU uses a 16 bit ADC and 3.3 V logic which provides a very small quantization error. When compared to the typical signals recorded during gestures the quantization error is well below 1% of the signal value for both the gyroscope and accelerometer and the quantization error can be assumed negligible. Although the IMU tends to be noisier than the FSR's, with 96.3 dB of dynamic range and a very small quantization error, a large SNR is achieved and the sensor and signals are reliable for the given application.

## FILTERS

### FSR FILTERING

As mentioned previously, the upper bound on human gesture frequency is given to be around 4.5 Hz. Any frequencies seen in the signals far above this value can thus be considered noise. Therefore, a low pass filter was designed to remove any high frequency noise that would affect the FSR signals. Figure 5 shows the time and spectral series of all eleven sensors while the user executed the eight unique hand gestures discussed previously. The frequency response shows almost no noise in the regions above 4.5 Hz. However, this is just one sample of the gestures executed, and does not represent the ensemble which may contain high frequency noise in a different environment or when expanding the experiment with different test participants. For this reason, a sinusoidal noise component at 50 Hz with a random amplitude between 0 and 15 grams was injected into the data, as observed in Figure 6. This will allow for better comparison between digital filters in a situation that may occur in the future.

As the filter will be used during real time execution, any lag introduced due to filter implementation is to be kept to a minimum. Therefore, computation time and phase delay are of particular importance. Additional specifications included a pass and stop band frequency of 5 Hz and 15 Hz respectively. Having the pass band frequency slightly higher than the 4.5 Hz suggested allows some space for

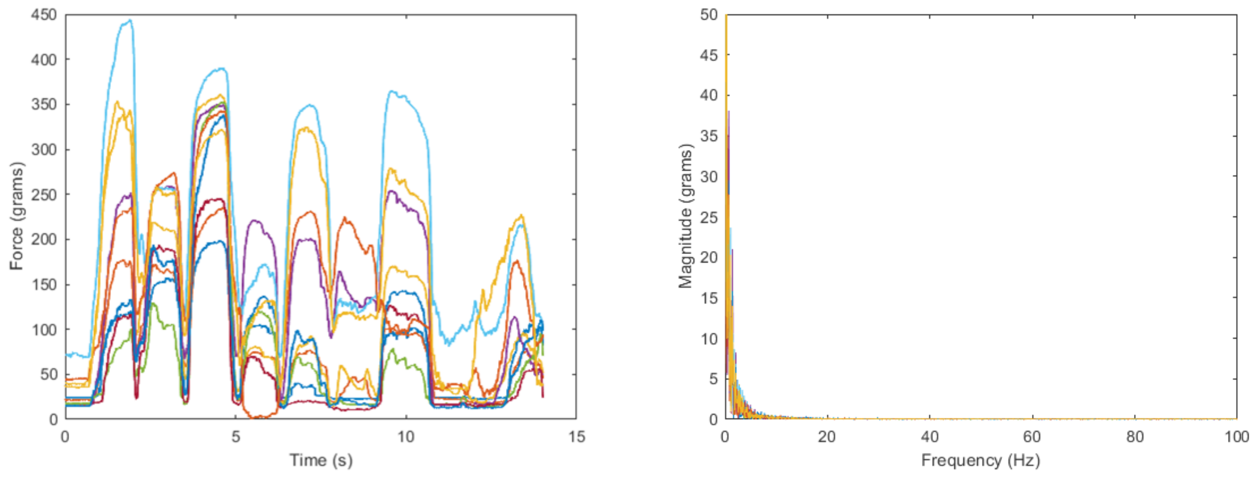


**Figure 5:** Raw FSR Time Series and Spectral Series Plots During the Eight Unique Gestures

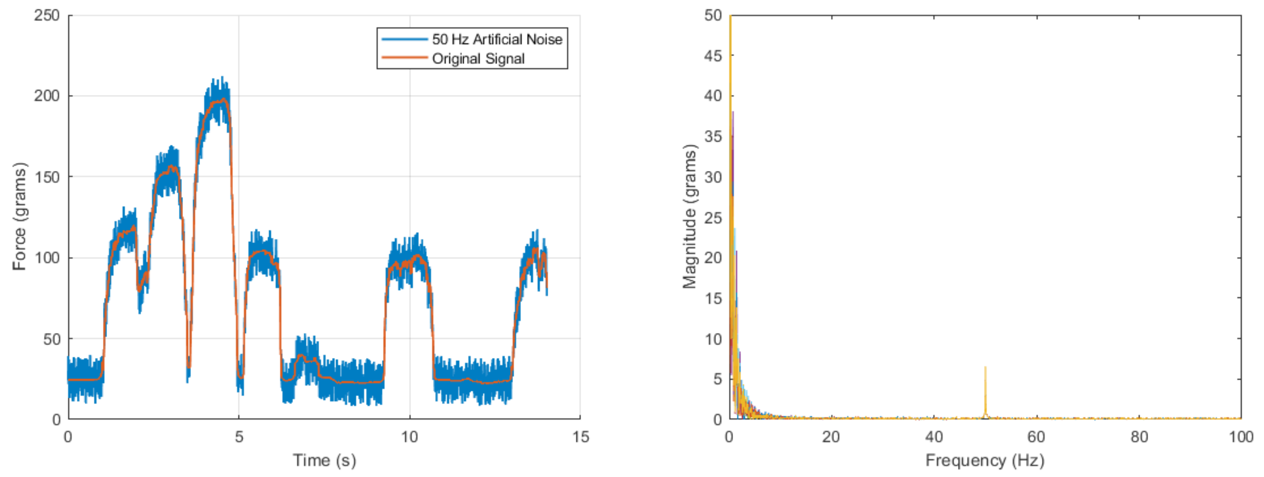


**Figure 6:** Raw FSR Time Series and Spectral Series Plots During the Eight Unique Gestures with added 50 Hz Noise

variation in different humans gesture speed. A transition width of 10 Hz corresponds to 10% of the Nyquist frequency, which will allow for some flexibility in filter design. Two finite impulse response (FIR) and two infinite impulse response (IIR) filters for a total of four digital low pass filter types were designed and compared for the given application; the Moving Average, Hanning Window, Butterworth, and Chebyshev Type II. For filters which did not allow for a specification of both pass and stop band frequency, a cutoff frequency at halfway between the two of 10 Hz was used. Additionally, the Butterworth and Chebyshev Type II filters were designed to have a stopband attenuation of 44 dB in order to more easily compare the result to the Hanning Window, which has a set stopband of 44 dB. The four filter shapes as well as phase delay responses can be seen in Figures 7 and 8. The red x's mark the location of the start and stop band frequencies for easy reference. All filter design was completed using the MATLAB signal processing toolbox where the details can be found in Appendix B.3.

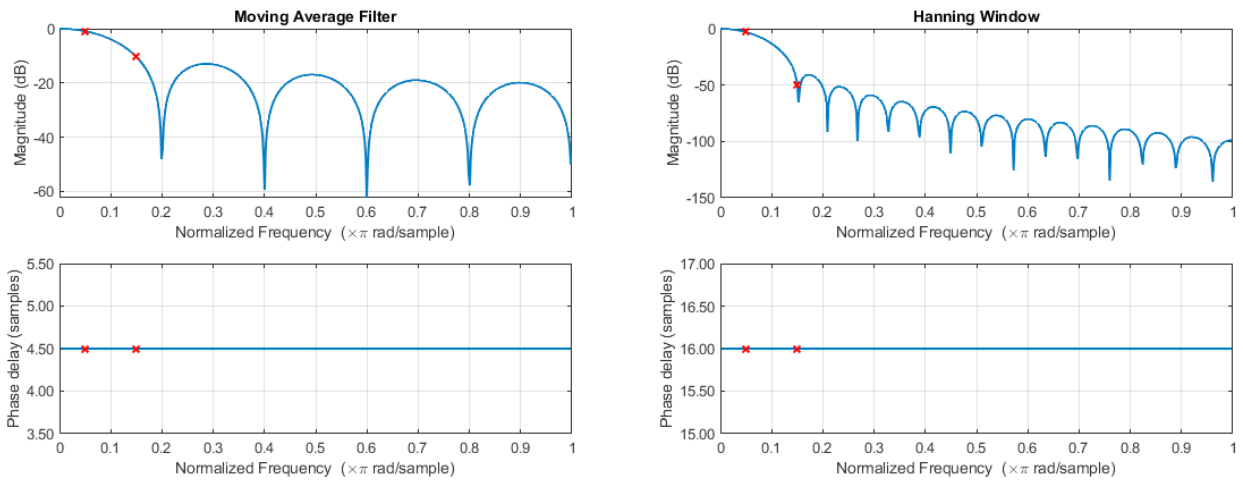


**Figure 7:** Filter Shapes and Phase Delays for FIR Designed Filters Moving Average (left) and Hanning Window (Right)

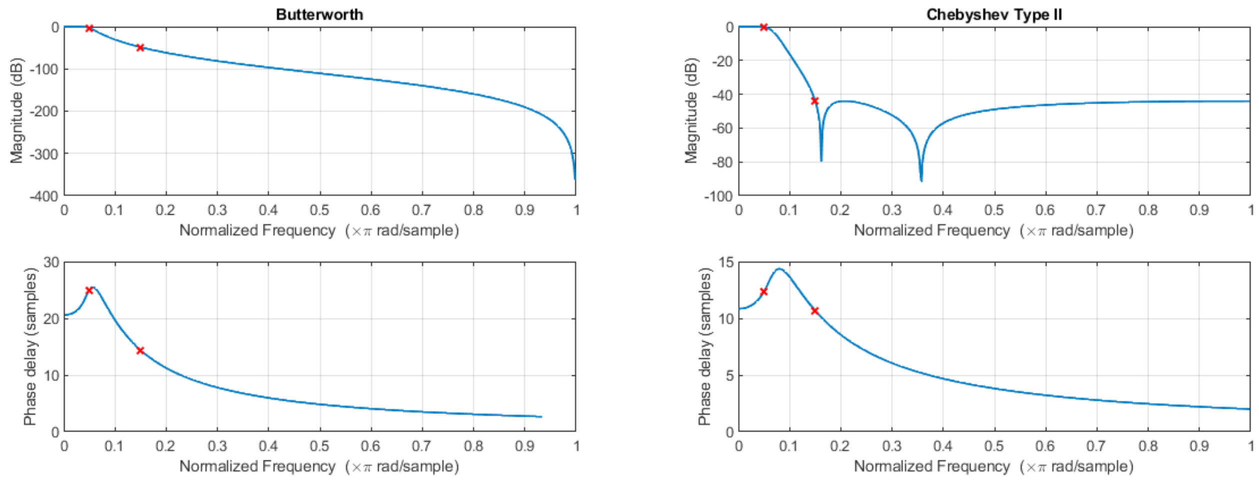


**Figure 8:** Filter Shapes and Phase Delays for IIR Designed Filters Butterworth (left) and Chebyshev Type II (Right)

## IMU FILTERING

To accurately establish the attitude of a users arm with respect to Figure 2 the accelerometer and gyroscope values are converted to an Euler angle that represents the arms orientation and is usable in a control scheme. It should be noted that yaw will be not considered due to sensor limitations and it was found to have no benefit in controlling the desktop mounted robotic arm. Neglecting detailed derivations [18] the roll and pitch are calculated using trigonometric relations between the output from the accelerometers different axis ($a_x$, $a_y$, and $a_z$) and is observed in Equations 3 and 4.

$$accelerometer_{roll} = \arctan 2\left(\frac{a_y}{a_z}\right)\frac{180}{\pi} \tag{3}$$

$$accelerometer_{pitch} = \arctan 2\left(\frac{-a_x}{\sqrt{a_y^2 + a_z^2}}\right)\frac{180}{\pi} \tag{4}$$

The gyroscope provides angular velocity, in deg/s and an angle can be found by integrating the sensor output based on the standardized sampling rate of 200 Hz. Each axis can be treated individually and adding the previous angle to the current angle, as in Equation 5, will provide roll and pitch.

$$\theta_i = \theta_{i-1} + \int_0^t \frac{d\theta_i}{dt}dt \tag{5}$$

Although roll and pitch is found two different ways, the results are less than optimal. The accelerometer provides accurate values when stationary, however when there is movement the signal becomes very noisey. Alternatively, the gyroscope provides great tracking during movement but the values drift when stationary. To achieve the desired results the sensors must be fused together in order to get the best attributes from both the sensors. This can be accomplished using a complementary filter.

A complementary filter is a first order low pass IIR filter sometimes called a leaky integrator or exponentially weighted moving average. It was selected over alternative IMU fusion and filtering algorithms such as the Kalman or Madgwick filter due to its ease of implementation, low computation time, and strong performance. It works using Equations 3 through 5 and applying a high pass and low pass coefficient for the gyroscope and accelerometer angle values respectively. The complementary filter can be observed in Equation 6.

$$angle_{current} = \alpha(angle_{previous} + gyroscope_{raw} \cdot dt) + (1-\alpha)(accelerometer_{angle}) \tag{6}$$

The coefficient α is related to the time constant τ and is defined in Equation 7. The time constant defines the boundary of trusting the gyroscope or accelerometer angle. For time periods greater than τ priority is placed on the accelerometer angle to mitigate gyroscope drift while values less than τ place the priority on the gyroscope angle and the noisy accelerometer data is filtered out.

$$\tau = \frac{\alpha dt}{1-\alpha} \leftrightarrow \alpha = \frac{\tau}{\tau + dt} \tag{7}$$

The implementation of the complementary filter not only fuses together the accelerometer and gyroscope data but also filters it to be usable. Although the filter is low pass in nature and additional filtering could be implemented, the results for the proposed application are more than acceptable. Additionally, many implementations in literature only require the use of a complementary filter to achieve their desired results to achieve robust control [19]–[21].

## RESULTS

### FSR FILTER DESIGN RESULTS

A summary of the results of the FSR filter design can be seen in Table 3. Additionally, a close up view of a single FSR with each filter applied separately can be seen in Figure 9.

**Table 3:** FSR Filter Design Results Summary

| Filter Type | Type | Stopband Attenuation (dB) | Computation Time (ms) | Order | Max Phase Delay in Passband (samples) |
|---|---|---|---|---|---|
| Moving Average | FFR | 10 | 0.0012 | 10 | 5 |
| Hanning Window | FFR | 44 | 0.0024 | 33 | 16 |
| Butterworth | IIR | 48 | 0.0012 | 5 | 25 |
| Chebyshev | IIR | 44 | 0.0012 | 4 | 12 |

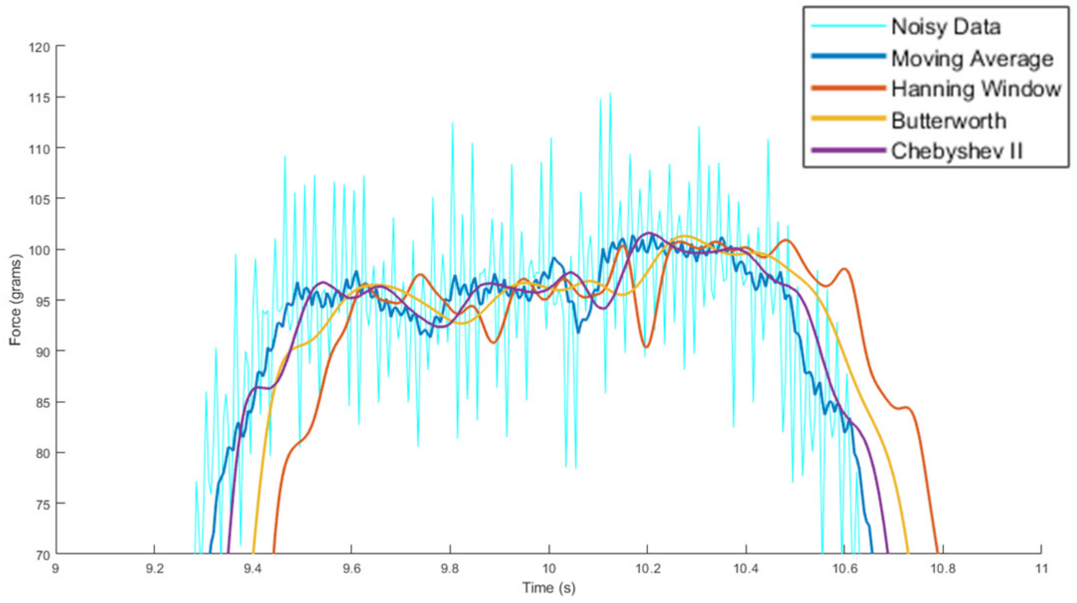


**Figure 9:** Close up View of Single FSR Signal with Four Designed Filters

The stop band attenuation of the moving average filter of 10 dB was significantly lower than the remaining three filters at 44 dB. This caused the noisy signal to still be visible in a close up view of the moving average signal, but not in the remaining three signals. This was expected since attenuations of 10 dB and 44 dB correspond to reducing the noise to 32% and 0.63% of the original value respectively. It should also be noted that the moving average filter cannot be tuned to fix this problem, as increasing the order will result in significant passband attenuation as well as a phase delay surpassing the other filtering methods. For this reason, the moving average filter was not considered for the given application. The major discernable difference in the remaining three filters was the amount of lag induced by the phase delay. Since this filter will be ran in real time, signal lag is an undesirable side effect and should be minimized. For this reason, the Chebyshev Type II filter was chosen as the best candidate from the group, since it had the smallest phase delay with a comparable stopband attenuation.

### IMU FILTER RESULTS

Through experimental tuning an α of 0.98 (τ = 0.245 s) provided optimal results for implementing a successful complementary filter. The results of a participant moving their arm in a down-up-down sequence in the x direction can be seen in Figure 10 where the resulting pitch is shown. As observed in the figure all signals start at the same value. While the accelerometer proves to be noisey especially during movement (2 s - 7 s) the gyroscope holds the resulting angle smooth and removes the noise and high frequency elements. At the end of the trail (7 s - 10 s) the arm is held stationary and the gyroscope begins to drift however the accelerometer holds the angle true therefore validating a successful sensor fusion and a stable value to used in control.

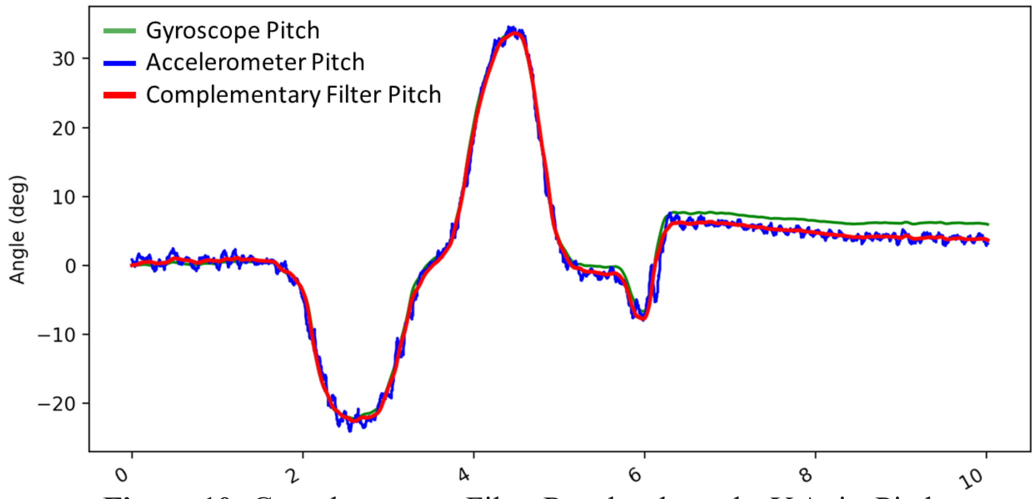


**Figure 10:** Complementary Filter Results about the Y Axis, Pitch

## DISCUSSION AND CONCLUSIONS

### IMPLEMENTATION

Using the experimental procedure defined, 300 data points for each of the eight gestures were filtered and labeled providing a total of 2400 data points of known gestures. Using Scikit Learn 0.20.3 and Python 3.7.2 the data was passed to a support vector machine (SVM) algorithm which generates a classifier by drawing hyperplanes about regions of interest (Appendix B.1, lines 17 to 87). With a trained classifier sensor data is acquired in real time, sent to the SVM, and the resulting gesture classification is mapped to the robotic arm in real time. The IMU data is not used in the classifier but is directly mapped from the user's arm attitude to the simulated prosthetic providing additional control options in tandem with the gestures.

To generate a robust classifier it is desirable to have all outputs be in a concentrated area that is separable from other clusters. To visually present the separability of the acquired data, a principal component analysis (PCA) was performed to compress the eleven dimension FSR data into three dimensions. To investigate the impact of the FSR lowpass filter designed, the artificial noise discussed earlier was injected into the training data. The PCA analysis done previously was reproduced with this new data and a visual comparison can be found in Figure 11. The clusters are now much more spread out, some even bleeding into others. This illustrates the importance of the designed Chebyshev Type II filter in FMG real time control as the presence of noise may introduce classification errors.

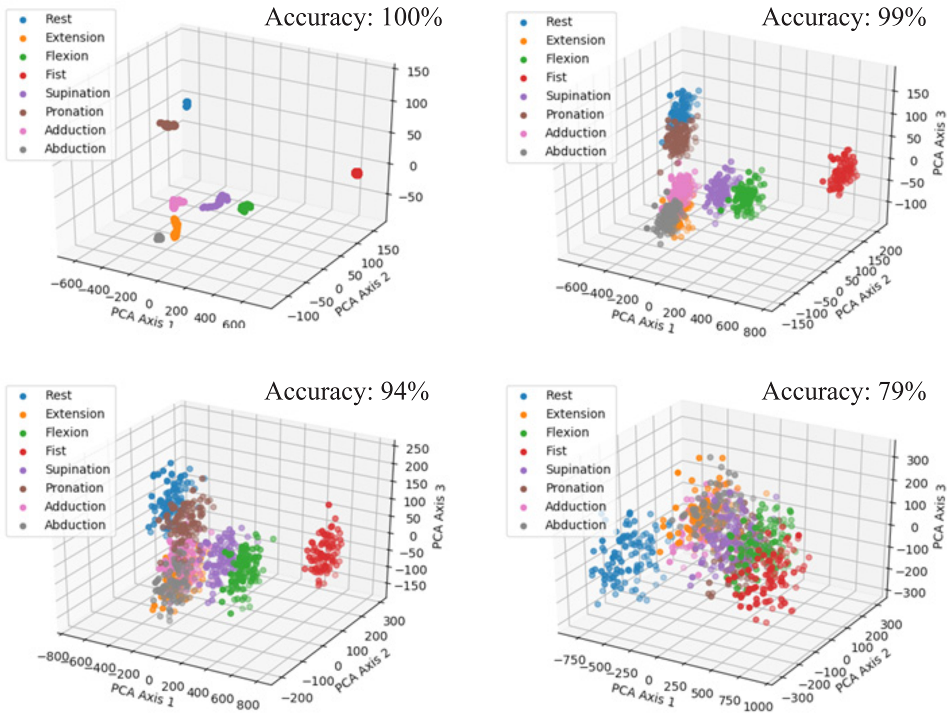


**Figure 11:** PCA Analysis of Eight Gestures Without Noise (Top-Left) and With 50 gram (Top-Right), 100 gram (Bottom-Left), and 200 gram (Bottom-Right) Noise Amplitude

## CONCLUSION

A FMG sensor and algorithm package was optimized to implement robust, real time control of a simulated prosthetic device. Two iterations of prototyping have been achieved, with the second design resulting in a wireless, low profile wristband containing 11 FSR sensors and an IMU capable of sampling reliably at 200 Hz. Moving Average, Hanning Window, Butterworth, and Chebyshev Type II low pass filters were investigated for unwanted high frequency noise reduction in the FSR sensors. The Chebyshev Type II was determine to be optimal due to its high attenuation, low order, and low phase delay properties. Sensor fusion was utilized to accurately determine wristband orientation while also compensating for high frequency noise and drifting values using a complementary filter. A SVM classification algorithm was used to successfully classify eight different hand gestures. The output of these hand gestures combined with the users arms orientation was mapped to a 4-DOF desktop robotic arm in real time, simulating the use of a prosthetic.

## FUTURE WORK

*Signal Processing:* It should be reiterated that the noise used to choose a low pass filter was added artificially with an arbitrarily chosen amplitude. The noise that the device may see in actual use will not necessarily be similar. The sensor signals should be monitored during common day situations to get an idea of the type of noise that will likely be seen by the end user of the device. The low pass filters used for the FSR sensors were analyzed using similar stop band attenuation to provide a fair comparison, however the amount of attenuation required may vary depending on the noise observed. In the future, the experimentally determined noise should be used to choose the parameters of the implemented low pass filter.

*Classifier Outputs:* Although the system is functional and working, there remains much to be optimized on the pattern recognition side of the device. The eight gestures were chosen arbitrarily, and many more could be investigated. Additionally having eight classes is not necessary for prosthetic use as commercial myoelectric devices typically only utilize one or at most two degrees of freedom. This would require only five classes to achieve control (e.g. rest, hand close, hand open, wrist-CW, wrist-CCW). A systematic experiment investigating multiple individuals in various gestures could be done to determine which gestures are the most separable, and therefore most robust for use in the classifier.

*Classifier Inputs:* Thus far, only the raw signals have been used as inputs into the classifier. Techniques such as normalization and mean removal have not been explored, but logically could result in a classifier that is more adaptable to varying wristband tightness. Many different feature extraction techniques have been investigated for EMG control, and have shown to increase the classification accuracy by up to 10% [7]. Using the IMU Euler angle output as an additional feature has also yet to be investigated. This could provide higher accuracy in mobile situations, as the wristbands inertia will compress the FSR sensors during acceleration.

*Training Data:* Every time the device it is taken on and off a new training session is required. The robustness of a classification model to wristband shift and varying tightness has not been investigated. Data could be gathered while varying wristband tightness and location. This dataset could be used to systematically investigate classifier response to varying physical wristband parameters. The device has largely been tested on only two individuals. Comparing the performance among many different subjects is essential to determining the effectiveness of this device. The device should also be tested on person's of amputation in order to confirm that performance on able bodied individuals transfers to the intended target.

## REFERENCES

[1] M. Asghari Oskoei and H. Hu, “Myoelectric control systems—A survey,” Biomedical Signal Processing and Control, vol. 2, no. 4, pp. 275–294, Oct. 2007.

[2] E. A. Biddiss and T. T. Chau, “Upper limb prosthesis use and abandonment: A survey of the last 25 years,” Prosthet Orthot Int, vol. 31, no. 3, pp. 236–257, Sep. 2007.

[3] E. Biddiss and T. Chau, “Upper-Limb Prosthetics: Critical Factors in Device Abandonment,” American Journal of Physical Medicine & Rehabilitation, vol. 86, no. 12, p. 977, Dec. 2007.

[4] J. T. Belter, J. L. Segil, A. M. Dollar, and R. F. Weir, “Mechanical design and performance specifications of anthropomorphic prosthetic hands: A review,” Journal of Rehabilitation Research & Development, vol. 50, no. 5, pp. 599–617, Sep. 2013.

[5] C. Cipriani, M. Controzzi, and M. C. Carrozza, “The SmartHand transradial prosthesis,” Journal of NeuroEngineering and Rehabilitation, vol. 8, no. 1, p. 29, May 2011.

[6] A. E. Schultz and T. A. Kuiken, “Neural Interfaces for Control of Upper Limb Prostheses: The State of the Art and Future Possibilities,” PM&R, vol. 3, no. 1, pp. 55–67, Jan. 2011.

[7] E. Scheme and K. Englehart, “Electromyogram pattern recognition for control of powered upper-limb prostheses: State of the art and challenges for clinical use,” Journal of Rehabilitation Research & Development, vol. 48, no. 6, pp. 643–659, Sep. 2011.

[8] M. Wininger, N.-H. Kim, and W. Craelius, “Pressure signature of forearm as predictor of grip force,” J Rehabil Res Dev, vol. 45, no. 6, pp. 883–892, 2008.

[9] A. Kadkhodayan, X. Jiang, and C. Menon, “Continuous Prediction of Finger Movements Using Force Myography,” J. Med. Biol. Eng., vol. 36, no. 4, pp. 594–604, Aug. 2016.

[10] H. K. Yap, A. Mao, J. C. . Goh, and C.-H. Yeow, “Design of a wearable FMG sensing system for user intent detection during hand rehabilitation with a soft robotic glove,” in 2016 6th IEEE International Conference on Biomedical Robotics and Biomechatronics (BioRob), Singapore, Singapore, 2016, pp. 781–786.

[11] E. Cho, R. Chen, L.-K. Merhi, Z. Xiao, B. Pousett, and C. Menon, “Force Myography to Control Robotic Upper Extremity Prostheses: A Feasibility Study,” Front Bioeng Biotechnol, vol. 4, Mar. 2016.

[12] A. Radmand, E. Scheme, and K. Englehart, “High-density force myography: A possible alternative for upper-limb prosthetic control,” J Rehabil Res Dev, vol. 53, no. 4, pp. 443–456, 2016.

[13] C. Castellini, R. Kõiva, C. Pasluosta, C. Viegas, and B. Eskofier, “Tactile Myography: An Off-Line Assessment of Able-Bodied Subjects and One Upper-Limb Amputee,” Technologies, vol. 6, no. 2, p. 38, Mar. 2018.

[14] Z. G. Xiao and C. Menon, “Towards the development of a wearable feedback system for monitoring the activities of the upper-extremities,” J Neuroeng Rehabil, vol. 11, p. 2, Jan. 2014.

[15] T. Stefanou, A. Turton, A. Lenz, and S. Dogramadzi, “Upper limb motion intent recognition using tactile sensing,” in 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vancouver, BC, 2017, pp. 6601–6608.

[16] Franciscone, C. (2018). EEZYbotARM MK2. [online] Eezyrobots. Available at: http://www.eezyrobots.it/eba_mk2.html [Accessed 20 Feb. 2019].

[17] Y. Xiong and F. Quek, “Hand Motion Gesture Frequency Properties and Multimodal Discourse Analysis,” International Journal of Computer Vision, vol. 69, no. 3, pp. 353–371, Sep. 2006.

[18] M. Pedley, “Tilt Sensing Using a Three-Axis Accelerometer - Rev 6”, Freescale Semiconductor, Inc. 2013.

[19] Gui, Pengfei & Tang, Liqiong & Mukhopadhyay, S.C.. (2015). MEMS based IMU for tilting measurement: Comparison of complementary and kalman filter based data fusion. 2004-2009. 10.1109/ICIEA.2015.7334442.

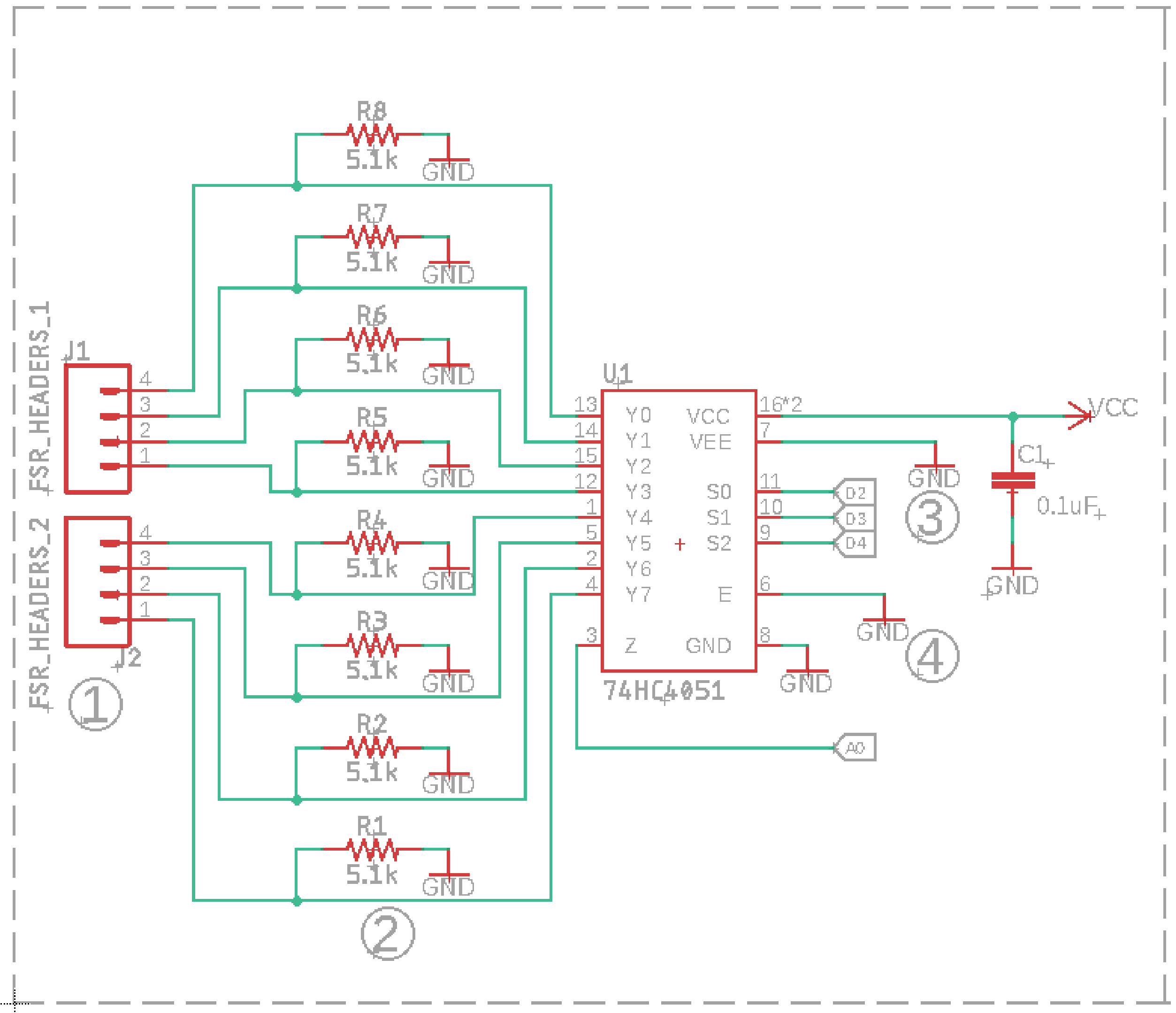
[20] Boyali, Ali & Hashimoto, N & Matsumoto, Osamu. (2015). A signal pattern recognition approach for mobile devices and it’s application to braking state classification on robotic mobility devices. Robotics and Autonomous Systems. 2073. 10.1016/j.robot.2015.04.008.

[21] S. Colton, The Balance Filter - A Simple Solution for Integrating Accelerometer and Gyroscope Measurements for a Balancing Platform. Rev.1: Submitted as a Chief Delphi white paper - June 25, 2007.

## APPENDIX A: PCB AND COMPONENTS

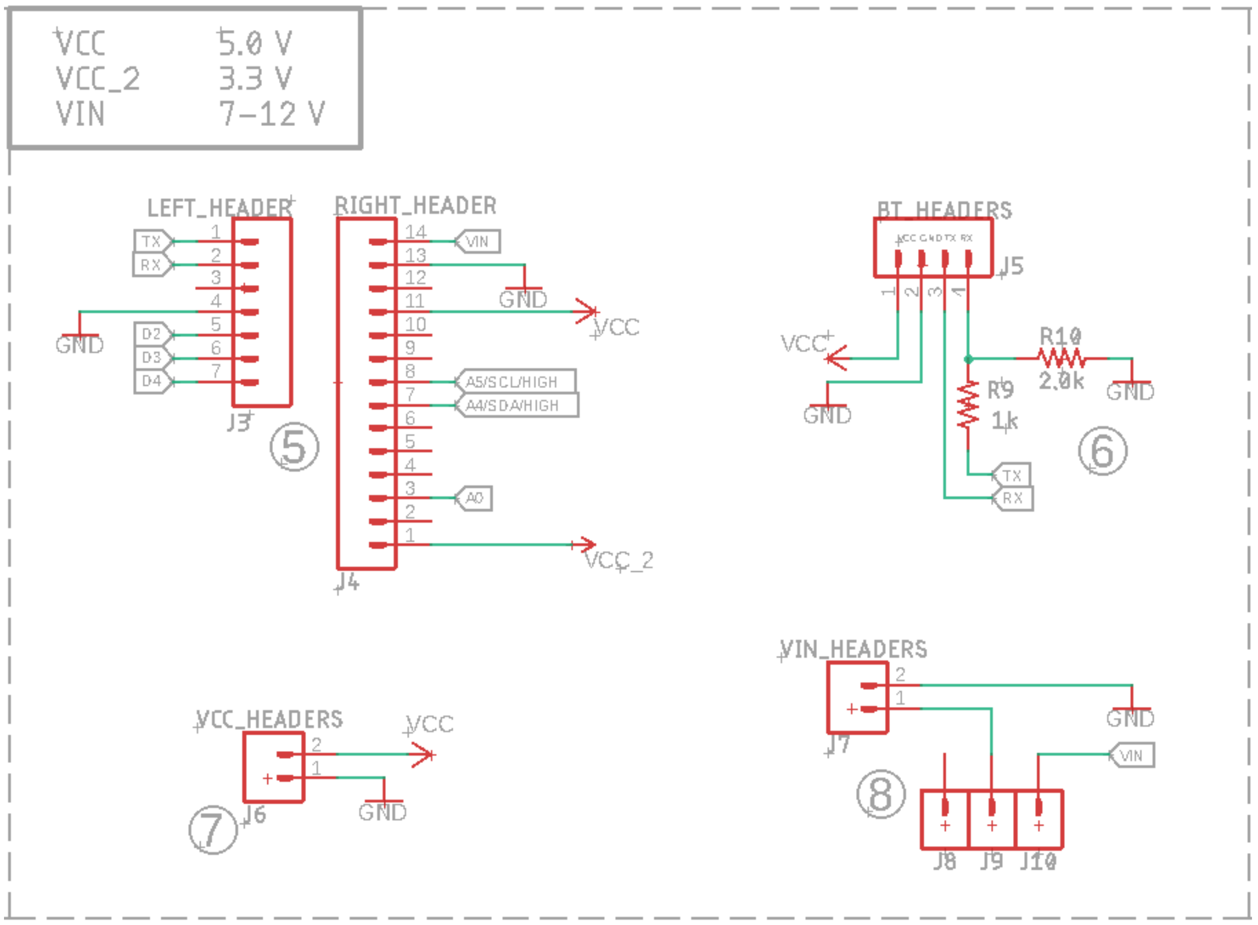
Section A.1 provides a schematic of the PCB with detailed design notes and Section A.2 displays the PCB board layout. Section A.3 tabulates all components used and a reference to the components data sheets. For complete design files refer to: https://github.com/Rico5678/GestureControlledRoboticArm_MecE653

### A.1 - PCB SCHEMATIC



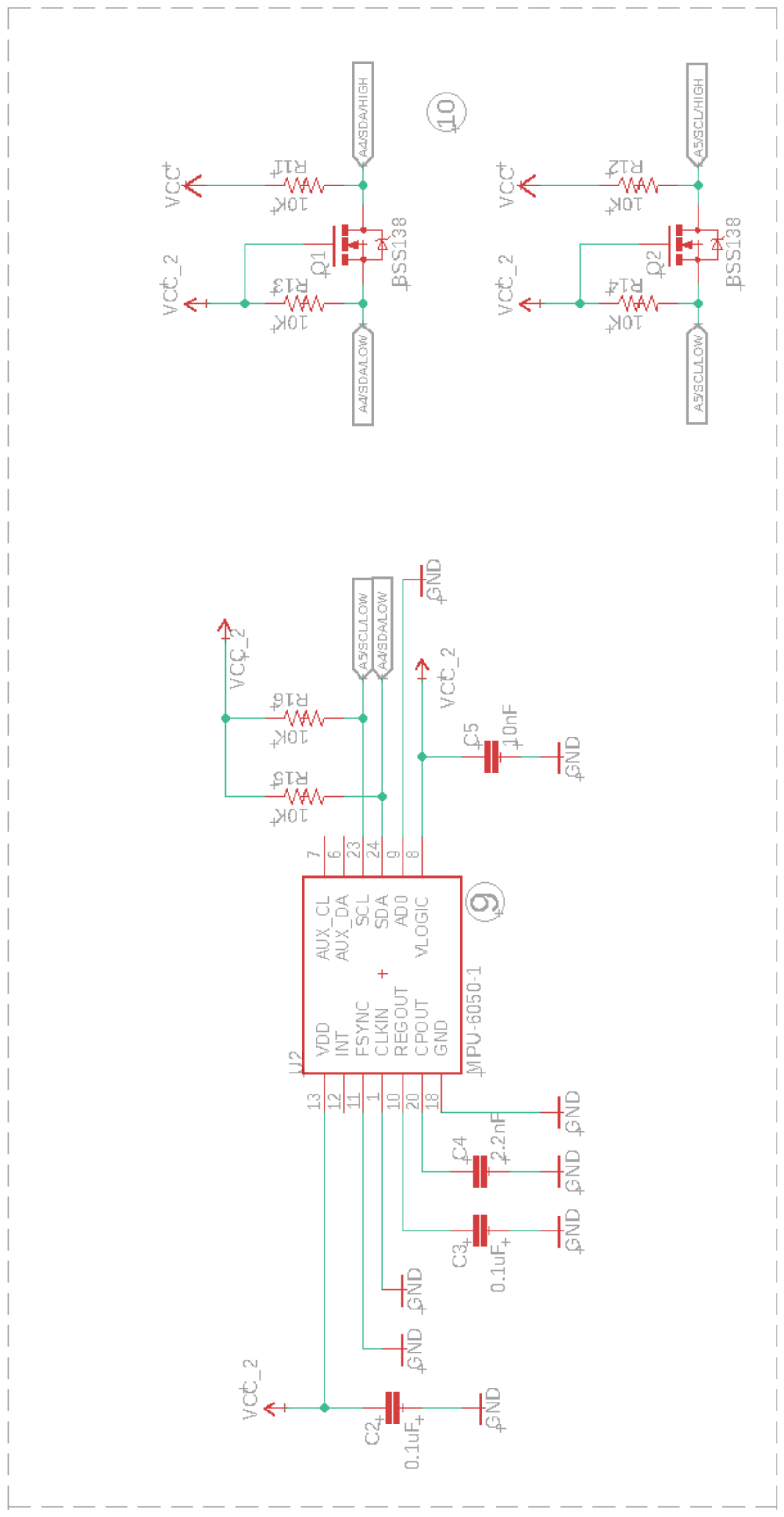
1. FSR – Force Sensative Resistor – Sparkfun SEN-09673. Attach one end to header and other to VCC.
2. FSR ranges between 1MΩ at low and 2.5kΩ with full pressure. Therefore, voltage divider range varries between 0–3.4 V.
3. VEE set to GND, no bipolar supply.
4. E set to GND to keep enable on.

**Figure A1:** 8 Channel Multiplexer and Voltage Dividers for FSR's



5. Shield for Arduino Nano V3 footprint.
6. Bluetooth module is HC-05. RX pin on module only supports logic 3.3 V hence the voltage divider. No voltage divider is required for the TX module to RX Arduino connection as high logic is satisfied.
7. VCC header to be used with the FSR's.
8. Two position SPDT switch used as SPST - Sparkfun COM-00102. To be soldered post manufacturing.

**Figure A2:** Pin and Header Assignment



9. As per MPU-6050 pinout:
INT – Interrupt digital output (totem pole or open-drain). Connection is optional.
FSYNC – Frame synchronization digital input. Connect to GND if unused.
CLKIN – Optional exter[n]al reference clock input. Connect to GND if unused.
AUX_DA – I2C master serial data, for connecting to external sensors. Connection is optional.
AUX_CL – I2C Master serial clock, for connecting to external sensors. Connection is optional.
AD0 – I2C Slave Address LSB (AD0). Set to GND for address of 0x68.
Otherwise connections follow the typical operating circuit defined on page 22 of 52 of Rev 3.4.

10. Conversion of 5 V logic to 3.3 V logic as required by MPU-6050.

**Figure A3:** 6-DOF Inertial Measurement Unit (IMU) and Logic Level Conversion

## A.2 – PCB BOARD LAYOUT

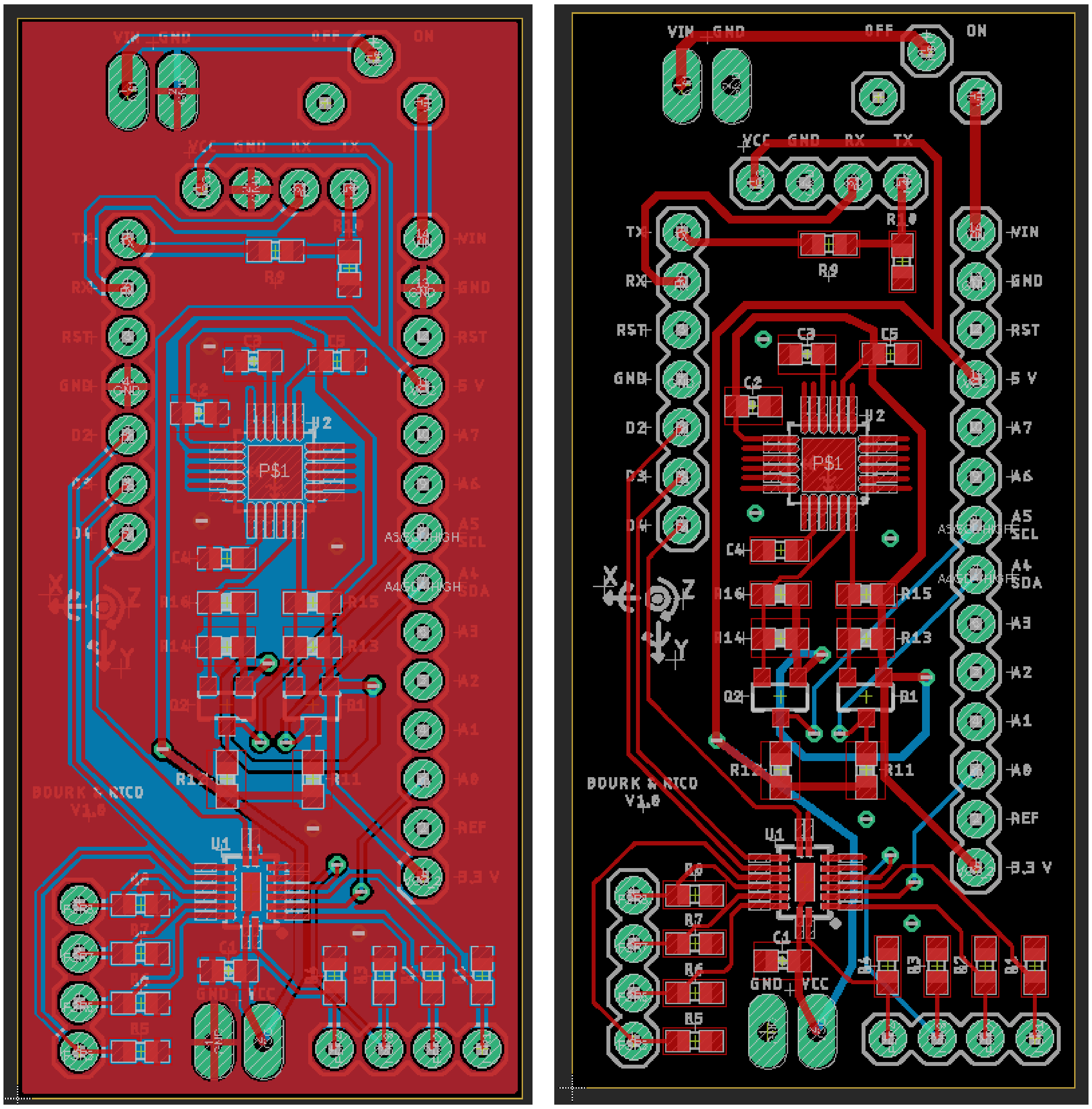


**Figure A4:** Top View of PCB Layout With Ground Planes (Left) and Without Ground Planes (Right)

## A.3 – BILL OF MATERIALS

**Table A1:** Bill of Materials for PCB and Data Sheet Hyperlinks

| Designator | Value | Manufacturer Part # | Rqd Qty | Cost / Unit | Total Cost | Data Sheet |
|---|---|---|---|---|---|---|
| R1-R8 | 5.1 kΩ | ERJ-3EKF5101V | 8 | $0.10 | $0.78 | Link |
| R9 | 1 kΩ | ERJ-3EKF1001V | 1 | $0.10 | $0.10 | Link |
| R10 | 2 kΩ | ERJ-3EKF2001V | 1 | $0.10 | $0.10 | Link |
| R11-R16 | 10 kΩ | ERJ-3EKF1002V | 6 | $0.10 | $0.59 | Link |
| C1-C3 | 0.1 uF | CL10B104KB8NNNC | 3 | $0.05 | $0.14 | Link |
| C4 | 2.2 nF | CL10B222JB8NNNC | 1 | $0.06 | $0.06 | Link |
| C5 | 10 nF | CL10B103KB8NNNC | 1 | $0.05 | $0.05 | Link |
| Q1,Q2 | - | BSS138 | 2 | $0.40 | $0.79 | Link |
| U1 | - | 74HC4051BQ,115 | 1 | $0.59 | $0.59 | Link |
| U2 | - | MPU-6050 | 1 | $11.01 | $11.01 | Link |
| | | | | **Total:** | **$14.21** | |

Additional data sheets for external components are referenced below:

- HC-05 Bluetooth Module - Link
- ATmega328P Microcontroller - Link
- FSR's - Link
- Raspberry Pi 3 Model B+ - Link

## APPENDIX B: CODE

The code used for reading the wireless signal and converting the raw signals to a physical interpretation on the microprocessor and then the robotic arm is in Section B.1. Section B.2 provides the main code used by the microcontroller for reading the raw sensor data and transmitting it wirelessly to the microprocessor. Section B.3 contains the code for designing the FSR filters in a MATLAB environment. For full documentation and the latest version of all code refer to: https://github.com/Rico5678/GestureControlledRoboticArm_MecE653

### B.1 – MICROPROCESSOR CODE (PYTHON)

```python
from sklearn.decomposition import PCA
from threading import Thread
from sklearn import svm
from mpl_toolkits import mplot3d
import RPi.GPIO as GPIO
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import serial
import time
import struct
import copy
import math
import pickle


class Model:
    def __init__(self, model=None):
        # Class / object / constructor setup
        self.model = model
        self.trainingxdata = None
        self.trainingydata = None

    def get_training_data(self, s, data, n, classnum, trainnum):
        # Get user to do the gestures and record the data
        xdata = []
        ydata = []
        counter = 0
        for i in range(0, trainnum):
            print("Training iteration: {0}".format(i))
            for k in range(0, classnum):
                input("Class number: {0}".format(k))
                for j in range(0, n):
                    data.getData(s)
                    a = [int(x) for x in data.force]
                    xdata.append(a)
                    ydata.append(k)

        self.trainingxdata = np.array(xdata)
        self.trainingydata = np.array(ydata)

    def plot_pca(self, show=True):
        # Squash down the dimensions of FSR data and show in 3 dimensions for groupings
        self.pca = PCA(n_components=3)
        self.pca.fit(self.trainingxdata)
        Xpca = self.pca.fit_transform(self.trainingxdata)
        ax = plt.axes(projection='3d')

        for i in range(0, int(self.trainingydata.max()) + 1):
            Xtemp = Xpca[self.trainingydata == i]
            ax.scatter3D(Xtemp[:, 0], Xtemp[:, 1], Xtemp[:, 2], cmap='Greens')

        if show is True:
            plt.figure(1)
            plt.show()

    def trainSVM(self):
        # Create a SVM classifier using sklearn
        self.model = svm.SVC(kernel='rbf', gamma='scale')
        self.model.fit(self.trainingxdata, self.trainingydata)

    def predict(self, data):
        # Identify the gesture and return the result
        prediction = self.model.predict([data])
```

```
            return prediction

    def score(self, xdata, ydata):
        # How accurate was the model?
        score = self.model.score(xdata, ydata)
        return score

    def savemodel(self, filename):
        # Save with pickles so that retraining is not required
        pickle.dump(self, open(filename, 'wb'))

    def data_split(self, p):
        # Arrange the data into something that is useful
        data = np.hstack((self.trainingxdata, np.transpose(np.array([self.trainingydata])
        datashuff = np.array(data)
        np.random.shuffle(datashuff)

        cutoff = int(p * data.shape[0])
        self.trainingxdata = datashuff[0:cutoff, 0:-1]
        self.trainingydata = datashuff[0:cutoff, -1]

        self.testingxdata = datashuff[cutoff::, 0:-1]
        self.testingydata = datashuff[cutoff::, -1]


class DAQ:
    def __init__(self, serialPort, serialBaud, dataNumBytes, numSignals):
        # Class / object / constructor setup
        self.port = serialPort
        self.baud = serialBaud
        self.dataNumBytes = dataNumBytes
        self.numSignals = numSignals
        self.rawData = bytearray(numSignals * dataNumBytes)
        self.dataType = None
        self.isRun = True
        self.isReceiving = False
        self.thread = None
        self.dataOut = []

        if dataNumBytes == 2:
            self.dataType = 'h'

        # Connect to serial port
        print('Trying to connect to: ' + str(serialPort) + ' at ' + str(serialBaud) + ' B
        try:
            self.serialConnection = serial.Serial(serialPort, serialBaud, timeout=4)
            print('Connected to ' + str(serialPort) + ' at ' + str(serialBaud) + ' BAUD.'
        except:
            print("Failed to connect with " + str(serialPort) + ' at ' + str(serialBaud)

    def readSerialStart(self):
        # Create a thread
        if self.thread == None:
            self.thread = Thread(target=self.backgroundThread)
            self.thread.start()

            # Block till we start receiving values
            while self.isReceiving != True:
                time.sleep(0.1)

    def backgroundThread(self):
        # Pause and clear buffer to start with a good connection
        time.sleep(2)
        self.serialConnection.reset_input_buffer()

```

```python
        # Read until closed
        while (self.isRun):
            self.getSerialData()
            self.isReceiving = True

    def getSerialData(self):
        # Initialize data out
        tempData = []

        # Check for header bytes and then read bytearray if header satisfied
        if (struct.unpack('B', self.serialConnection.read())[0] is 0x9F) and (struct.unpack('B', self.serialConnection.read())[0] is 0x6E):
            self.rawData = self.serialConnection.read(self.numSignals * self.dataNumBytes

            # Copy raw data to new variable and set up the data out variable
            privateData = copy.deepcopy(self.rawData[:])

            # Loop through all the signals and decode the values to decimal
            for i in range(self.numSignals):
                data = privateData[(i*self.dataNumBytes):(self.dataNumBytes + i*self.dataNumBytes)]
                value, = struct.unpack(self.dataType, data)
                tempData.append(value)

        # Check if data is usable otherwise repeat (recursive function)
        if tempData:
            if (struct.unpack('B', self.serialConnection.read())[0] is 0xAE) and (struct.unpack('B', self.serialConnection.read())[0] is 0x72):
                self.dataOut = tempData
            else:
                return self.getSerialData()
        else:
            return self.getSerialData()

    def close(self):
        # Close the serial port connection
        self.isRun = False
        self.thread.join()
        self.serialConnection.close()

        print('Disconnected...')


class Sensors:
    def __init__(self, gyroScaleFactor, accScaleFactor, VCC, Resistor, tau):
        # Class / object / constructor setup
        self.gyroScaleFactor = gyroScaleFactor
        self.accScaleFactor = accScaleFactor
        self.VCC = VCC
        self.Resistor = Resistor
        self.tau = tau

        self.gx = None; self.gy = None; self.gz = None;
        self.ax = None; self.ay = None; self.az = None;

        self.gyroXcal = 0
        self.gyroYcal = 0
        self.gyroZcal = 0

        self.gyroRoll = 0
        self.gyroPitch = 0
        self.gyroYaw = 0

        self.roll = 0
        self.pitch = 0
        self.yaw = 0

        self.dtTimer = 0

        self.FSRvalues = []
        self.force = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

    def getRawSensorValues(self, s):
        # Get raw values from serial connection
        val = s.dataOut
        self.gx = val[0]
        self.gy = val[1]
        self.gz = val[2]
        self.ax = val[3]
        self.ay = val[4]
        self.az = val[5]
        self.FSRvalues = [val[6], val[7], val[8], val[9], val[10], val[11], val[12], val[

    def calibrateGyro(self, s, N):
        # Display message
        print("Calibrating gyro with " + str(N) + " points. Do not move!")

        # Take N readings for each coordinate and add to itself
        for ii in range(N):
            self.getRawSensorValues(s)
            self.gyroXcal += self.gx
            self.gyroYcal += self.gy
            self.gyroZcal += self.gz

        # Find average offset value
        self.gyroXcal /= N
        self.gyroYcal /= N
        self.gyroZcal /= N

        # Display message and restart timer for comp filter
        print("Calibration complete")
        self.dtTimer = time.time()

    def processIMUvalues(self):
        # Subtract the offset calibration values
        self.gx -= self.gyroXcal
        self.gy -= self.gyroYcal
        self.gz -= self.gyroZcal

        # Convert to instantaneous degrees per second
        self.gx /= self.gyroScaleFactor
        self.gy /= self.gyroScaleFactor
        self.gz /= self.gyroScaleFactor

        # Convert to g force
        self.ax /= self.accScaleFactor
        self.ay /= self.accScaleFactor
        self.az /= self.accScaleFactor

    def compFilter(self):
        # Get the processed values from IMU
        self.processIMUvalues()

        # Get delta time and record time for next call
        dt = time.time() - self.dtTimer
        self.dtTimer = time.time()

        # Acceleration vector angle
        accPitch = math.degrees(math.atan2(self.ay, self.az))
        accRoll = math.degrees(math.atan2(self.ax, self.az))

        # Gyro integration angle
        self.gyroRoll -= self.gy * dt
        self.gyroPitch += self.gx * dt
        self.gyroYaw += self.gz * dt
        self.yaw = self.gyroYaw

        # Comp filter
        self.roll = (self.tau)*(self.roll - self.gy*dt) + (1-self.tau)*(accRoll)
        self.pitch = (self.tau)*(self.pitch + self.gx*dt) + (1-self.tau)*(accPitch)

    def processFSRvalues(self):
        # Loop through all values
        for ii in range(len(self.FSRvalues)):
            # Analog value to voltage
            fsrV = self.FSRvalues[ii] * self.VCC / 1023.0

            # Use voltage and static resistor value to calculate FSR resistance
            try:
                fsrR = ((self.VCC - fsrV) * self.Resistor) / fsrV
            except:
                fsrR = 1e6

            # Guesstimate force based on slopes in figure 3 of FSR datasheet (conductance
            fsrG = 1.0 / fsrR

            # Break parabolic curve down into two linear slopes
            if (fsrR <= 600):
                self.force[ii] = (fsrG - 0.00075) / 0.00000032639
            else:
                self.force[ii] =  fsrG / 0.000000642857

        # Write the last entry of force to be the average value
        self.force[11] = np.mean(self.force)

    def getData(self,s):
        # Get data from serial
        self.getRawSensorValues(s)

        # Process IMU values
        self.compFilter()

        # Process FSR values
        self.processFSRvalues()

    def logData(self, s, T):
        # Set up timer, counter, and temp data storage
        startTime = time.perf_counter()
        count = 0
        tempData = []

        # Run for T seconds
        while time.perf_counter() < startTime + T:
            self.getData(s)
            tempData.append([time.perf_counter() - startTime] + self.force + [self.roll,
            count += 1

        # Close serial connection, write data, and display sampling rate
        print("Samping rate: ", count / (time.perf_counter() - startTime), " Hz")

        # Write data to CSV
        df = pd.DataFrame(tempData, columns=
['Time', 'FSR1', 'FSR2', 'FSR3', 'FSR4', 'FSR5', 'FSR6',
                                   'FSR7', 'FSR8', 'FSR9', 'FSR10', 'FSR11', 'Avg', 'Roll

        df.to_csv('data.csv', index=None, header=True)


class robotArm:
    def __init__(self):
        # Joint PWM ranges
        self.joint1Range = [500,2300]
        self.joint2Range = [1000,2000]
        self.joint3Range = [500,1200]
        self.joint4Range = [1100,1100]

    def startControl(self):
        # Run `pinout` to see the numbers
        GPIO.setmode(GPIO.BOARD)

        # Set up PWM pins on GPIO
        GPIO.setup(12, GPIO.OUT)
        GPIO.setup(13, GPIO.OUT)
        GPIO.setup(18, GPIO.OUT)
        GPIO.setup(19, GPIO.OUT)

        # Initialize all servos to center position
        self.joint1 = GPIO.PWM(12, 50)
        self.joint2 = GPIO.PWM(13, 50)
        self.joint3 = GPIO.PWM(18, 50)
        self.joint4 = GPIO.PWM(19, 50)

        # Write start position
        self.joint1.start(7.5)
        self.joint2.start(7.5)
        self.joint3.start(7.5)
        self.joint4.start(7.5)

    def updateState(self, state):
        # percentage / (20 ms * unit conversion)
        dutyCycleScale = 100 / (20*1000)

        # Check the different states and write a pule
        if state == 1:
            # Fist
            self.joint4.ChangeDutyCycle(self.joint4Range[1]*dutyCycleScale)
        elif state == 2:
            # Rest
            self.joint4.ChangeDutyCycle(self.joint4Range[0]*dutyCycleScale)
        elif state == 3:
            # Extension
            self.joint1.ChangeDutyCycle(self.joint1Range[0]*dutyCycleScale)
        elif state == 4:
            # Flexion
            self.joint1.ChangeDutyCycle(self.joint1Range[1]*dutyCycleScale)
        elif state == 5:
            # Forward
            self.joint2.ChangeDutyCycle(self.joint2Range[0]*dutyCycleScale)
            self.joint3.ChangeDutyCycle(self.joint3Range[1]*dutyCycleScale)
        elif state == 6:
            # Back
            self.joint2.ChangeDutyCycle(self.joint2Range[1]*dutyCycleScale)
            self.joint3.ChangeDutyCycle(self.joint3Range[0]*dutyCycleScale)
        else:
            print("No state found")

    def endControl(self):
        # Stop writing PWM signal to servos
        self.joint1.stop()
        self.joint2.stop()
```

```python
        self.joint3.stop()
        self.joint4.stop()

        # Clean up ports used
        GPIO.cleanup()


def main():
    # Set up serial connection
    portName = '/dev/rfcomm0'
    baudRate = 115200
    dataNumBytes = 2
    numSignals = 17

    s = DAQ(portName, baudRate, dataNumBytes, numSignals)
    s.readSerialStart()

    # Set up sensors
    numCalibrationPoints = 3000
    gyroScaleFactor = 131.0
    accScaleFactor = 16384.0
    VCC = 4.98
    Resistor = 5100.0
    tau = 0.98

    data = Sensors(gyroScaleFactor, accScaleFactor, VCC, Resistor, tau)
    data.calibrateGyro(s, numCalibrationPoints)

    # Set up robot arm
    bot = robotArm()
    bot.startControl()

    # set up, train, and save model
    model = Model()
    model.get_training_data(s, data, 2500, 8, 3)
    model.plot_pca()
    model.trainSVM()
    model.savemodel('8x3x2500-Pi')

    # Realtime control
    T = int(input("Enter how many seconds to run real time control: "))
    startTime = time.time()

    while(time.time() < (startTime + T)):
        data.getData(s)
        pred = model.predict(data.force)
        bot.updateclass(pred[0])
        print(pred)

    # Close all the connections and end program
    s.close()
    bot.endControl()
    print("Closed all")


if __name__ == '__main__':
    main()
```

## B.2 – MICROCONTROLLER CODE (C++)

```
//Include I2C library and declare variables
#include <Wire.h>

const int selectPins[3] = {2, 3, 4};
int FSR[11];
int temperature;
int acc_x, acc_y, acc_z;
int gyro_x, gyro_y, gyro_z;
unsigned long timer = 0;
long loopTimeMicroSec = 5000;


void setup() {
  // Start I2C and serial port
  Wire.begin();
  Serial.begin(115200);

  // Setup the registers of the MPU-6050
  setupMPU6050();

  // Set up the select pins as outputs for multiplexer
  for (int i=0; i<3; i++){
    pinMode(selectPins[i], OUTPUT);
    digitalWrite(selectPins[i], HIGH);
  }

  // Connect z on multiplexer to analog zero (A0)
  pinMode(A0, INPUT);

  // Reset the timer
  timer = micros();
}


void loop() {
  // Stabilize sampling rate
  timeSync(loopTimeMicroSec);

  // Read the raw data from MPU-6050
  readMPU6050();

  // Loop through all eight pins on multiplexer reading analog
  for (byte pin=0; pin<=7; pin++){
    selectMuxPin(pin);
    FSR[pin] = analogRead(A0);
  }

  FSR[8] = analogRead(A1);
  FSR[9] = analogRead(A2);
  FSR[10] = analogRead(A3);

  // Send raw values to Python
  writeBytes(&gyro_x, &gyro_y, &gyro_z, &acc_x, &acc_y, &acc_z,
    &FSR[0], &FSR[1], &FSR[2], &FSR[3], &FSR[4], &FSR[5], &FSR[6], &FSR[7], &FSR[8], &FSR
}


void timeSync(unsigned long deltaT){
  // Calculate required delay to run at 200 Hz
  unsigned long currTime = micros();
  long timeToDelay = deltaT - (currTime - timer);

  if (timeToDelay > 5000){
    delay(timeToDelay / 1000);
    delayMicroseconds(timeToDelay % 1000);
  } else if (timeToDelay > 0){
    delayMicroseconds(timeToDelay);
  } else {}

  timer = currTime + timeToDelay;
}


void selectMuxPin(byte pin){
  // Set the S0, S1, and S2 pins to yield Y0-Y7
  for (int i=0; i<3; i++){
    if (pin & (1<<i))
      digitalWrite(selectPins[i], HIGH);
    else
      digitalWrite(selectPins[i], LOW);
  }
}


void writeBytes(int* data1, int* data2, int* data3, int* data4, int* data5, int* data6,
  int* data7, int* data8, int* data9, int* data10, int* data11, int* data12, int* data13,
  int* data14, int* data15, int* data16, int* data17){

  // Cast to a byte pointer
  byte* byteData1 = (byte*)(data1);     byte* byteData2 = (byte*)(data2);
  byte* byteData3 = (byte*)(data3);     byte* byteData4 = (byte*)(data4);
  byte* byteData5 = (byte*)(data5);     byte* byteData6 = (byte*)(data6);
  byte* byteData7 = (byte*)(data7);     byte* byteData8 = (byte*)(data8);
  byte* byteData9 = (byte*)(data9);     byte* byteData10 = (byte*)(data10);
  byte* byteData11 = (byte*)(data11);   byte* byteData12 = (byte*)(data12);
  byte* byteData13 = (byte*)(data13);   byte* byteData14 = (byte*)(data14);
  byte* byteData15 = (byte*)(data15);   byte* byteData16 = (byte*)(data16);
  byte* byteData17 = (byte*)(data17);

  // Byte array with header for transmission
  byte buf[38] = {0x9F, 0x6E,
                  byteData1[0],  byteData1[1],     byteData2[0],  byteData2[1],
                  byteData3[0],  byteData3[1],     byteData4[0],  byteData4[1],
                  byteData5[0],  byteData5[1],     byteData6[0],  byteData6[1],
                  byteData7[0],  byteData7[1],     byteData8[0],  byteData8[1],
                  byteData9[0],  byteData9[1],     byteData10[0], byteData10[1],
                  byteData11[0], byteData11[1],    byteData12[0], byteData12[1],
                  byteData13[0], byteData13[1],    byteData14[0], byteData14[1],
                  byteData15[0], byteData15[1],    byteData16[0], byteData16[1],
                  byteData17[0], byteData17[1],    0xAE, 0x72};
  Serial.write(buf, 38);
}


void readMPU6050() {
  //Subroutine for reading the raw data
  Wire.beginTransmission(0x68);
  Wire.write(0x3B);
  Wire.endTransmission();
  Wire.requestFrom(0x68, 14);

  // Read data --> Temperature falls between acc and gyro registers
  while(Wire.available() < 14);
  acc_x = Wire.read() << 8 | Wire.read();
  acc_y = Wire.read() << 8 | Wire.read();
  acc_z = Wire.read() << 8 | Wire.read();
  temperature = Wire.read() <<8 | Wire.read();
  gyro_x = Wire.read()<<8 | Wire.read();
  gyro_y = Wire.read()<<8 | Wire.read();
  gyro_z = Wire.read()<<8 | Wire.read();
}


void setupMPU6050() {
  //Activate the MPU-6050
  Wire.beginTransmission(0x68);
  Wire.write(0x6B);
  Wire.write(0x00);
  Wire.endTransmission();

  // Configure the accelerometer
  // 2g --> 0x00, 4g --> 0x08, 8g --> 0x10, 16g --> 0x18
  Wire.beginTransmission(0x68);
  Wire.write(0x1C);
  Wire.write(0x00);
  Wire.endTransmission();

  // Configure the gyro
  // 250 deg/s --> 0x00, 500 deg/s --> 0x08, 1000 deg/s --> 0x10, 2000 deg/s --> 0x18
  Wire.beginTransmission(0x68);
  Wire.write(0x1B);
  Wire.write(0x00);
  Wire.endTransmission();
}
```

## B.3 – FILTER DESIGN CODE (MATLAB)

```
close all
% read data
filename = 'C:\Users\Rico5678\OneDrive\Documents\School\MecE 653\GestureControlledRobotic
delimiter = ',';
startRow = 2;
formatSpec = '%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%[^\n\r]';
fileID = fopen(filename,'r');
dataArray = textscan(fileID, formatSpec, 'Delimiter', delimiter, 'TextType', 'string', 'E
1, 'ReturnOnError', false, 'EndOfLine', '\r\n');
fclose(fileID);
data = [dataArray{1:end-1}];
clearvars filename delimiter startRow formatSpec fileID dataArray ans;

%% Raw values and FFT Plotting

% get data for filtering
fs = 200;
N = length(data);
t = (0:N-1)/fs;
T = N/fs;
data(:,2) = [];
FSRnum = 10;
FSR1 = data(:,1)';
noise = 15*rand(size(t)).*sin(2*pi*50*t);

%% Raw Plots
% plot all FSRs
fnum = 1;
figure(fnum)
set(0,'DefaultLineLineWidth',1.5)
plot(t, data(:,1:FSRnum))
xlabel('Time (s)')
ylabel('Force (grams)')
fnum = fnum+1;

% plot single FSR with and without noise
figure(fnum)
hold on
grid on
plot(t, FSR1 + noise)
plot(t, FSR1)
xlabel('Time (s)')
ylabel('Force (grams)')
legend("50 Hz Artificial Noise", "Original Signal")
fnum = fnum+1;

% plot FFT of all FSRS without noise
Y = fft(data);
Ymag = abs(Y);
Yscaled = Ymag(1:N/2+1,1:FSRnum)*2/N;
f = (0:1/T:fs/2);

figure(fnum)
grid on
plot(f, Yscaled)
xlabel('Frequency (Hz)')
ylabel('Magnitude (grams)')
xlim([0,100])
ylim([0,50])
fnum = fnum+1;

% plot FFT of all FSRS with noise
Y = fft(data + noise');
Ymag = abs(Y);
```

```
Yscaled = Ymag(1:N/2+1,1:FSRnum)*2/N;
f = (0:1/T:fs/2);

figure(fnum)
grid on
plot(f, Yscaled)
xlabel('Frequency (Hz)')
ylabel('Magnitude (grams)')
xlim([0,100])
ylim([0,50])
fnum = fnum+1;


%% Filter Design
fp = 5;
wp = fp/(fs/2);
fst = 15;
wst = fst/(fs/2);
%% Moving Average Filter

MAfilt_fc = (fp+fst);
psi = 2*pi*MAfilt_fc/fs;
MAn = ceil(pi/psi);
MAn = 30;
b = (1/MAn)*ones(1,MAn);
a = 1;
MAfiltFSR1 = filter(b,a,FSR1+noise);
tic

MAfiltdelay = 0;
tic
for i=1:100000
temp = filter(b,a,FSR1(1:MAn));
end
MAfiltdelay = toc/100000;
MAfiltphasedelay = phasedelay(b,a);

figure(fnum)
fnum = fnum+1;
subplot(2,1,1)
hold on
freqz(b,a)
[h, w] = freqz(b,a);
title("Moving Average Filter")
plot([wp, wst], interp1(w/pi, 20*log10(abs(h)), [wp,wst]), 'rx')
subplot(2,1,2)
hold off
phasedelay(b,a)
hold on
ytickformat('%.2f')
[phi,w] = phasedelay(b,a);
plot([wp, wst], interp1(w(2:end)/pi, phi(2:end), [wp,wst], 'linear','extrap'), 'rx')


%% Hanning Window
Hannwindowsize = ceil(3.32*fs/(fst-fp));
Hannwindow = hann((Hannwindowsize-1)/2, 'symmetric');
b = fir1(floor(Hannwindowsize/2-1), (fst-fp)/2/(fs/2), Hannwindow);
a = 1;
HannfiltFSR1 = filter(b,a,FSR1+noise);
tic

Hannfiltdelay = 0;
tic
for i=1:100000
```

```
temp = filter(b,a,FSR1(1:Hannwindowsize));
end
Hannfiltdelay = toc/100000;
Hannfiltphasedelay = phasedelay(b,a);

figure(fnum)
fnum = fnum+1;
subplot(2,1,1)
hold on
freqz(b,a)
[h, w] = freqz(b,a);
title("Hanning Window")
plot([wp,wst], interp1(w/pi, 20*log10(abs(h)), [wp,wst]), 'rx')
subplot(2,1,2)
ytickformat('%.2f')
hold off
phasedelay(b,a)
hold on
ytickformat('%.2f')
[phi,w] = phasedelay(b,a);
plot([wp,wst], interp1(w(2:end)/pi, phi(2:end), [wp,wst], 'linear','extrap'), 'rx')

%% Butterworth Filter

butterwp = 2*(fst-fp)/2/fs;
butterws = 2*fst/fs;
butterattenp = 1;
butterdp = 1 - 10^(-butterattenp/20);
butterattens = 44;
butterds = 10^(-butterattens/20);
butterN = ceil(log10(1/butterds^2-1)/(2*log10(butterws/butterwp)));

[b,a] = butter(butterN, butterwp);

butterfiltFSR1 = filter(b,a,FSR1+noise);

butterfiltdelay = 0;
tic
for i=1:100000
temp = filter(b,a,FSR1(1:butterN));
end
butterfiltdelay = toc/100000;
butterfiltphasedelay = phasedelay(b,a);

figure(fnum)
fnum = fnum+1;
subplot(2,1,1)
hold on
freqz(b,a)
[h, w] = freqz(b,a);
title("Butterworth")
plot([wp,wst], interp1(w/pi, 20*log10(abs(h)), [wp,wst]), 'rx')
subplot(2,1,2)
ytickformat('%.2f')
hold off
phasedelay(b,a)
hold on
[phi,w] = phasedelay(b,a);
plot([wp,wst], interp1(w(2:end)/pi, phi(2:end), [wp,wst], 'linear','extrap'), 'rx')

%% Chebyshev type 2 filter
cheby2wp = fp/(fs/2);
cheby2ws = fst/(fs/2);
cheby2attenp = 1;
cheby2dp = 1 - 10^(-cheby2attenp/20);
```

```
cheby2attens = 44;
cheby2ds = 10^(-cheby2attens/20);
epsilon = sqrt(1/(1-cheby2dp)^2-1);
d = sqrt(1/cheby2ds^2-1);
cheby2N = ceil(acosh(d/epsilon)/acosh(cheby2ws/cheby2wp));

[b,a] = cheby2(cheby2N,cheby2attens,cheby2ws);

cheby2filtFSR1 = filter(b,a,FSR1+noise);

cheby2filtFSRS = zeros(size(data(:,1:10)));
for i=1:10
cheby2filtFSRS(:,i) = filter(b,a,data(:,i)+noise');
end
cheby2filtdelay = 0;
tic
for i=1:100000
temp = filter(b,a,FSR1(1:cheby2N));
end
cheby2filtdelay = toc/100000;
cheby2phasedelay = phasedelay(b,a);

figure(fnum)
fnum = fnum+1;
subplot(2,1,1)
hold on
freqz(b,a)
[h, w] = freqz(b,a);
title("Chebyshev Type II")
plot([wp,wst], interp1(w/pi, 20*log10(abs(h)), [wp,wst]), 'rx')
subplot(2,1,2)
ytickformat('%.2f')
hold off
phasedelay(b,a)
hold on
[phi,w] = phasedelay(b,a);
plot([wp,wst], interp1(w(1:end)/pi, phi(1:end), [wp,wst], 'linear','extrap'), 'rx')

%% filter comparison
figure(fnum)
fnum = fnum+1;
hold on;
plot(t, FSR1)
plot(t, FSR1+noise)
plot(t,MAfiltFSR1)
plot(t,HannfiltFSR1)
plot(t,butterfiltFSR1)
plot(t,cheby2filtFSR1)
legend("Original Data", "Noisy Data", "Moving Average", "Hanning Window", "Butterworth",
xlabel('Time (s)')
ylabel('Force (grams)')


figure(fnum)
set(gca,'DefaultLineLineWidth',2)
fnum = fnum+1;
hold on;
plot(t, FSR1+noise, 'c-', 'LineWidth', 0.5)
plot(t,MAfiltFSR1)
plot(t,HannfiltFSR1)
plot(t,butterfiltFSR1)
plot(t,cheby2filtFSR1)
legend("Noisy Data", "Moving Average", "Hanning Window", "Butterworth", "Chebyshev II")
xlabel('Time (s)')
ylabel('Force (grams)')
```

```
xlim([9,11]);
ylim([70,120]);

figure(fnum)
fnum = fnum+1;
Y = fft(cheby2filtFSRS);
Ymag = abs(Y);
Yscaled = Ymag(1:N/2+1,1:FSRnum)*2/N;
f = (0:1/T:fs/2);

grid on
plot(f, Yscaled)
xlabel('Frequency (Hz)')
ylabel('Magnitude (grams)')
xlim([0,100])
ylim([0,50])
```