BRADLEY
University

EMG Based Human Machine Interface
Final Report

By Aditya Patel and Jim Ramsay
Advised by Dr. Yufeng Lu and Dr. In Soo Ahn
Published May 3, 2018

Abstract

Surface Electromyography (EMG) is a non-invasive technique which records the electrical activity of muscles using electrodes placed directly on the skin. The use of EMG signals has been gaining prevalence in prosthetic control and gesture-control applications. This project aims to develop an EMG-based human machine interface system. A Myo Armband with eight electrode pairs is worn by a user to acquire and wirelessly transmit EMG data to a central controller. A pattern recognition algorithm is implemented on a central controller to recognize three different hand gesture commands. As a demonstration, we built a camera system equipped with servo motors. The recognized commands can remotely pan each camera and select one of multiple video feeds to display. Our study demonstrates that the EMG-based pattern recognition could be a viable human machine interface option for a broad range of applications in industrial, medical, and consumer markets.

Acknowledgements

We would like to thank our advisors, Drs. Yufeng Lu and In Soo Ahn. The time and effort they contributed in advising us was integral in the success of this project. We were the benefactors of their willingness to share their extensive knowledge in signal processing and neural network design. In addition to our advisors, we would also like to extend our gratitude to Bradley University and the ECE department faculty and staff for their dedication to providing the highest quality of learning experiences. Finally, we would be remiss to not express gratitude to our friends and family for all of their support. The support provided by our friends and family has enabled us to complete this project as well as our entire undergraduate degrees.

Table of Contents

List of Tables and Figures

1. Introduction

Electromyography (EMG) is a technique for monitoring electrical signals associated with movement of muscles. EMG signals can be obtained via an intramuscular needle, or by an electrode placed directly on the skin. Intramuscular EMG (iEMG) is more accurate than surface EMG (sEMG) but sEMG allows electrical signals to be measured without the need for intrusive or bulky measurement tools. Acquiring sEMG signals only requires electrodes to be placed directly above the target muscle. When placed on the forearm, sEMG electrodes detect muscle activity associated with the movement of a user's hand. Since this project is focused on the analysis of sEMG signals, when the term "EMG signal(s)" is used throughout this report, the reader should assume these signals were collected by using the sEMG method.

EMG signals can range from 0V to 10V, peak-to-peak. The difficult part of collecting and analyzing the raw EMG data is the wide range of frequencies it can produce. EMG signals can be anywhere between 10Hz to 500Hz, depending on the person and how active their muscles are. As shown in Figure 1, even when the same person holds the same gesture for a period of time, the amplitude and frequency of the EMG signal can still vary quite a bit.

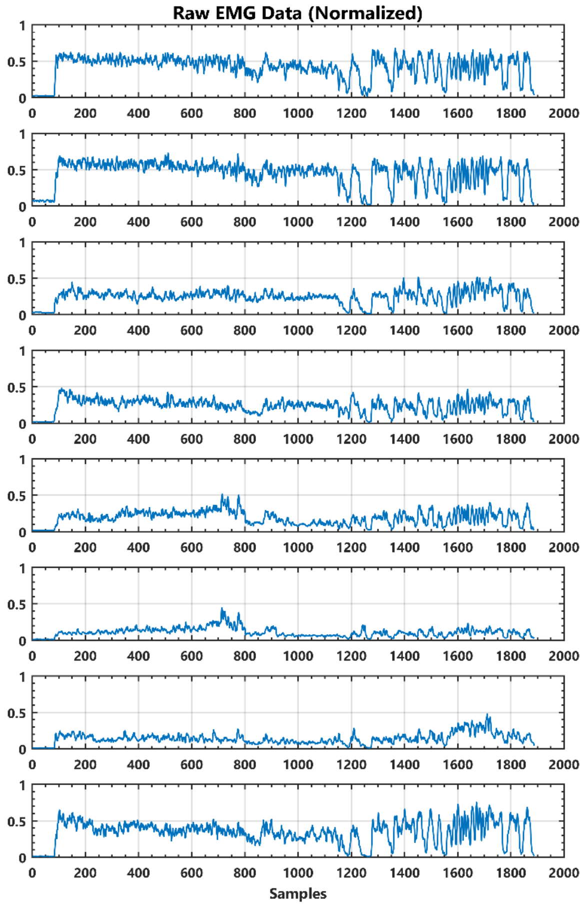


Figure 1 – Raw EMG data (normalized) from 8 sensors on one user, making one gesture

A. EMG Applications

Medical Diagnosis and Rehabilitation

Detection of EMG signals is becoming commonplace in the biomedical field. It is being used in medical research for diagnosis and rehabilitation [1]. In the most common case, an EMG test can be conducted to test for a variety of muscle and nerve related conditions and injuries [2]. Conditions that EMG testing helps diagnose include carpal tunnel syndrome, a pinched nerve, neuropathies, muscle diseases, muscular dystrophy, and Lou Gehrig's disease [3].

Prosthetic Control

In research, EMG signals are used to help recovering amputees control prosthetic limbs. Even if an amputee is missing a limb, their mind can still try to move the limb that is not there. In doing so, electrical impulses are sent to that region of the body as if the limb was still there. For example, an individual missing their forearm can have a prosthetic arm controlled by the EMG signals detected in their shoulder/upper arm [4].

There are great strides being made in EMG based prosthetics. For example, researchers at Japan's Hokkaido University developed an EMG prosthetic hand controller that uses real-time learning to detect up to ten forearm motions with 91.5% accuracy [5]. Additionally, research done at Abu Dhabi University aimed to develop a virtual reality simulation of an arm using EMG signals. They achieved an 84% success rate in simulating the correct movements made by amputees [6]

B. Pattern Recognition Algorithms

Pattern recognition is a subset of machine learning that can be broken into two main categories: supervised and unsupervised. In supervised learning, the algorithm is "trained" by giving the algorithm data that is already classified. This allows the program to have a baseline understanding of the pattern so that it knows what to look for in the future. In unsupervised learning, the algorithm is not given any classification information, and must draw inferences from data on its own [7]. "The most common unsupervised learning method is cluster analysis, which is used for exploratory data analysis to find hidden patterns or grouping in data. The clusters are modeled using a measure of similarity which is defined upon metrics such as Euclidean or probabilistic distance" [8].

A critical part of machine learning is an artificial neural network (ANN). ANN's are designed to mimic the human brain, where neurons and axons are represented by nodes and wires. Neural networks can be designed in countless different configurations. One form of neural network that is of interest to this project is a pattern recognition neural network (PNN). These algorithms are used to classify input data. The network is trained by associating training input data with known classifications. After the network is trained, new input data is entered and the output of the neural network is a classification for the input, based on the training stage results. The inputs for the network play a key role in the accuracy of the network. The network will get increasingly more accurate with more inputs, so long as there is a correlation to the classification. Some common inputs types are raw data, filtered data, averaged data, RMS data and other forms of data manipulation that help to relate each series of inputs to one classification.

2. Project Goals

The current market for gesture-based control of systems rely solely on the use of cameras to detect user movements. These systems require heavy processing and restrict the user to gesture only in the field of view of the cameras. To address these issues, this project created an EMG-based controlled system with the following goals.

A. Acquire EMG data from a user

The EMG data must be collected wirelessly so as to not restrict the user. The wireless communication needs to be reliable and quick to connect. Additionally, the data must be sampled at a rate high enough for real-time operation.

B. Detect different user hand gestures in real time

This system uses three different hand gestures to control it: a fist, wave inward, and wave outward. It has only been tested on the right hand, though it should be possible to use any hand. The system needs a calibration mode to allow for anybody to use it. The calibration should be quick and allow for fast and accurate gesture recognition. Users must receive feedback about the state of the system through the console.

C. Implement gesture detection to control a system

The system is comprised of two cameras, each attached to its own servo motor. The hand gestures allow the user to adjust the position of the motors, as well as the camera feed that is displayed on an external monitor. The motors rotate 180°, 90° in each direction from the initial position. The cameras operate at 30 frames per second and 720p resolution.

3. System Design

In this project, gestures are captured by a Myo Gesture Control Armband made by Thalmic Labs Inc. The armband houses eight electrodes for capturing EMG signals as well as an inertial measurement unit (IMU). Since this project is focused on creating a system interface with the use of EMG signals, the IMU data is ignored while collecting data from the armband. The gestures are used to control a camera system.

A. Functions and Gestures

Table 1: Functions and their associated gestures

Function	Gesture
Toggle armband sleep / standby	Fist
System Control Activate	Palm in, wave out
Camera Control Activate	Palm in, wave in
Switch to Camera 1	Palm in, wave out
Switch to Camera 2	Palm in, wave in
Pan left	Palm in, wave in
Pan right	Palm in, wave out

The full flow of logic for our system is shown below.

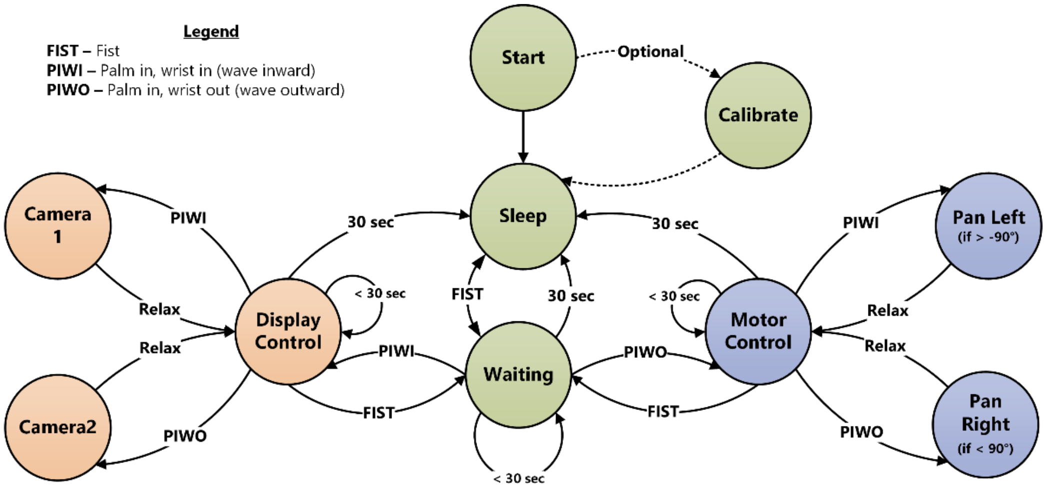


Figure 2: System flowchart

B. System Diagram

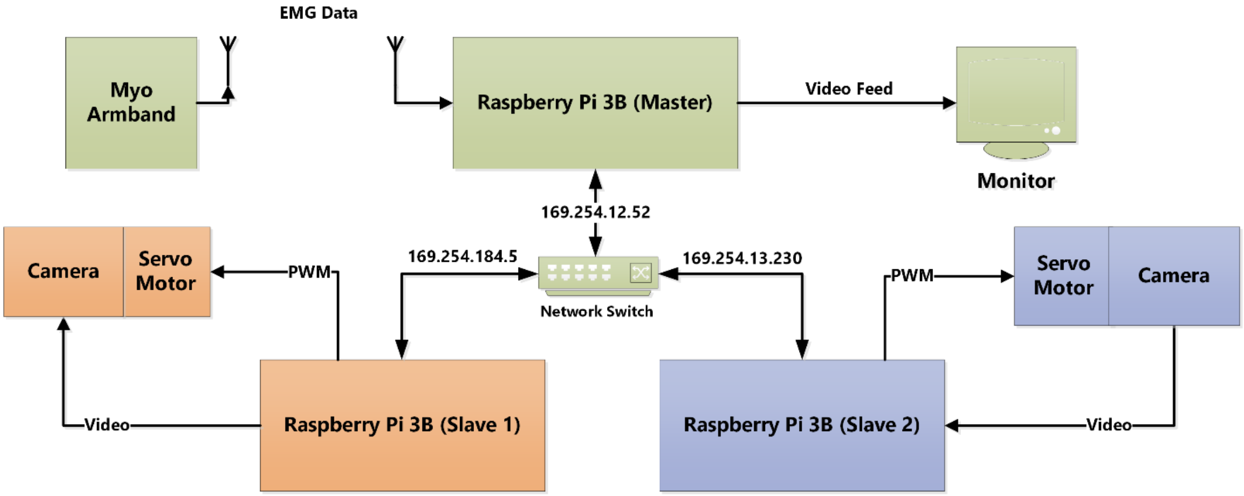


Figure 3: System Diagram

There are three Raspberry Pi 3B computers used in this project, one master and two slaves. The Myo armband sends EMG data via Bluetooth to the master Raspberry Pi. The master receives, processes and then communicates desired actions to one of the two slave Pi's. When the slave Pi gets a command, it executes a function to move the attached servo motor in a specific direction. The camera feeds stream on a webpage and, if the gesture calls for it, the master will switch the display to show the desired video stream. All master/slave communication is done on a local area network.

C. Myo Gesture Control Armband

The HMI device used for this project is an EMG armband, designed by Thalmic Labs. It is comprised of eight EMG sensors as well as a nine-axis IMU. Once connected to the armband via Bluetooth, the data is transmitted in real-time. The data is transmitted from the armband at 200 Hz and is in the form of an 8-bit unsigned integer. The raw EMG data is not the actual voltage that is sensed by the electrodes. Rather, it is a representation of muscle activation. The exact activation-to-voltage relationship is not made public by the developer of the armband.

D. Raspberry Pi

Master

The master Pi board is the heart of the EMG Security Monitoring System. It receives the armband signal via a Bluetooth USB dongle. This signal is then processed by algorithms that identify gestures made by the user. The master Pi also sends commands to the slaves when a gesture is made to move the motors. The master is responsible for keeping track of and making adjustments to the duty cycle of the control signal sent to the motors. The master also selects one of two different camera feeds to display on an external monitor.

Slave(s)

There are two Raspberry Pi 3B computers that act as slaves to the master Pi board. Each slave is equipped with an attached camera and servo motor. They process the video signals from their respective cameras and stream the video to a webpage. The Pi cameras connect directly to the Raspberry Pi 3b and have the ability to stream live video in 1080P. These Pi boards also run the scripts (when directed by the master Pi board) to generate a change in the servo motor's PWM duty cycle. This, in turn, controls the angle at which the camera is pointed.

E. Servo Motors

The system includes two servo motors (one per camera) that are used to pan the camera views from side to side. The motors are attached to a case which houses the Raspberry Pi and camera. The motors are powered by +5V, from the Pi GPIO pins. The Pi's also are equipped with a GPIO pin (12) that is designed to support PWM signal outputs—this is the pin used to transmit the PWM signal to the motor in this system. The camera angles are adjusted by increasing/decreasing the respective motors PWM duty cycle. The desired adjustment (per recognized gesture) is approximately 30 degrees. The master Pi board keeps track of the duty cycle and has built-in limitations of ± 90°.

F. Monitor

The monitor setup is initialized when the system is turned on. In this system, the display switches between camera feeds based on the gestures recognized by the master Raspberry Pi board. The display is in full screen and is changed by a Python script that toggles between browser tabs that each video feed is streaming to.

G. Software

Bluetooth Communication

Because the Myo armband does not come with first-party compatibility for Linux-based operating systems, we had to seek out open source software packages. The one that we found that worked best is called PyoConnect_v2.0, developed by *dzhu* and Fernando Cosentino [9][10]. This software package was designed to function in Linux just like the original software functions in Windows. The only part of this code that we used was the armband communication protocol. In this package is a file called "myo_raw.py" that executes the Bluetooth communication between the USB dongle and the armband. It is this file that we edited to function as our main code for gesture detection.

Video Feed

After exploring numerous different packages for the Raspberry Pi Camera module, we found one that perfectly aligned with our needs. Not only did it do everything we needed, it was much easier to install and configure than anything else that we had tried. The software is called the RPi-Cam-Web-Interface [11]. It allowed us to adjust the picture resolution, aspect ratio, framerate, overlayed text, and countless other items.

This software captures the video from the attached camera and streams the video feed to the Pi’s IP address, so that the URL looks like: 169.254.13.230/html/index.php. It works for local area networks as well as when connected to the internet. Because it is browser based, all that was needed to switch between video feeds was to have both open in their own tab, and to send the “ctrl+Tab” command to the Raspberry Pi.

Programming Language

The programming language for this project was essentially chosen for us. The vast majority of documentation on programming the Raspberry Pi and the Myo Armband used Python, so this was the natural choice for this project. For us, python was a brand new language, with a steep learning curve. Once we got a grasp on the jargon and syntax, it became very easy to write our own code from scratch and not have to rely on finding reference code online.

The primary benefit of using Python (2.7.14) is that anything is possible with the language. The drawback of this though, is that the code can be very slow to execute, especially when combined with the slow clock speed and limited processing power of the Raspberry Pi 3B. Nonetheless, we were able to execute our entire gesture detection and control loop in less than five milliseconds. If the execution time got much longer than that, we found that the Bluetooth communication would hang up and the code would stall.

4. Technical Specifications

A. Myo Gesture Control Armband

- ***Physical***
    - Weight: 93g
    - Flexibility: Fits arms ranging between 7.5″ and 13″
    - Thickness: 0.45″
- ***Sensors***
    - Surface EMG electrode pairs (8 pairs)
    - 9-Axis IMU
        - 3-Axis gyroscope
        - 3-Axis accelerometer
        - 3-Axis magnetometer
    - Made of medical grade stainless steel
- ***Computer / Communication***
    - ARM Cortex M4 processor
    - Wireless Bluetooth 4.0 LE communication
    - Battery
        - Built-in Lithium Ion battery
        - Micro USB charge
        - 1 full day of usage
    - EMG Data
        - Sampling rate: 200 Hz
        - Unitless – muscle activation is represented as an 8-bit signed value
        - Time stamp is in milliseconds since epoch (01/01/1970)
    - Compatible Operating Systems (for the SDK)
        - Windows 7, 8, and 10
        - OSx 10.8 and up
        - Android 4.3 and up
    - Haptic feedback with short, medium and long vibration options

B. Raspberry Pi 3B

- ***Processor***
    - Broadcom BCM2387
    - 1.2 GHz Quad-Core ARM Cortex-A53
    - 802.11 b/g/n Wireless LAN
    - Bluetooth 4.1 (Classic and LE)
- ***GPU***
    - Dual Core VideoCore IV Multimedia Co-Processor
    - OpenVG and 1080p30 H.264 high-profile decoder
- ***Memory***
    - 1 GB LPDDR2
- ***Operating System***
    - Boots from Micro SD card
    - Runs Linux OS or Windows 10 IoT
- ***Dimensions***
    - 85 mm x 56 mm x 17 mm
- ***Power***
    - Micro USB socket 5v1, 2.5A
- ***Peripherals***
    - Ethernet
        - 10/100 BaseT socket
    - Video Out
        - HDMI (rev 1.3 & 1.4)
        - Composite RCA (PAL and NTSC)
    - GPIO
        - 40-Pin 2.54 mm expansion header 2x20 strip
        - 27-Pin GPIO
        - +3.3V, +5V and GND supply lines
    - Camera
        - 15-Pin MIPI Camera Serial Interface (CSI-2)

    - Display
        - Display Serial Interface 15-way flat flex cable connector with two data lanes and a clock lane

C. Raspberry Pi Camera Module v2

- ***Camera***
    - Sony IMX219 8-megapixel sensor
- ***Video***
    - 1080p30
    - 720p60
    - VGA90
- ***Photo***
    - 8 MP
- ***Compatibility***
    - Raspberry Pi 1, 2, 3 (all models)
    - Numerous open-source software libraries

5. Results

A. Raw Data Collection and Preliminary Results

While collecting preliminary data, our goal was to test the raw armband data to verify that we can see differences in the data when different motions are made. The armband was always placed onto the thickest part of the right forearm, with sensor 4 on top, and sensors 1 and 8 on the bottom. Two different motions were captured: palm in, wrist action out (wave out) and palm in, wrist action in (wave in).

The first thing we noticed, which can be seen in both Figure 4 and Figure 5, was that there is a distinct difference in the EMG data when the arm muscles are activated. To prove this, we took samples in 10-second intervals and performed the actions in sets of 1, 3 and 5 actions. We observed clear differences between when the user's arm was at rest and when the user was making a gesture.

The second important detail we noted was a noticeable difference between the EMG sensor data when we performed different actions. Figure 4 shows the EMG data when the wrist is moved outward and we can see that the most muscle activation is on sensors 3, 4, and 5. Some action is observed in 2 and 6, while a relatively low amount of action is seen in sensors 1, 7 and 8. Figure 5 shows the EMG data for when the wrist is moved inward. In this case, we see that the most activation occurs on sensors 1, 7, and 8. There is also some activation on sensors 2, 3 and 6, while almost no activation was observed on sensors 4 and 5.

From this point, we shifted our focus to filter and analyze the data and then implement pattern recognition algorithms. To validate the pattern recognition algorithms, we collected and tested data from multiple users, performing multiple actions/gestures.

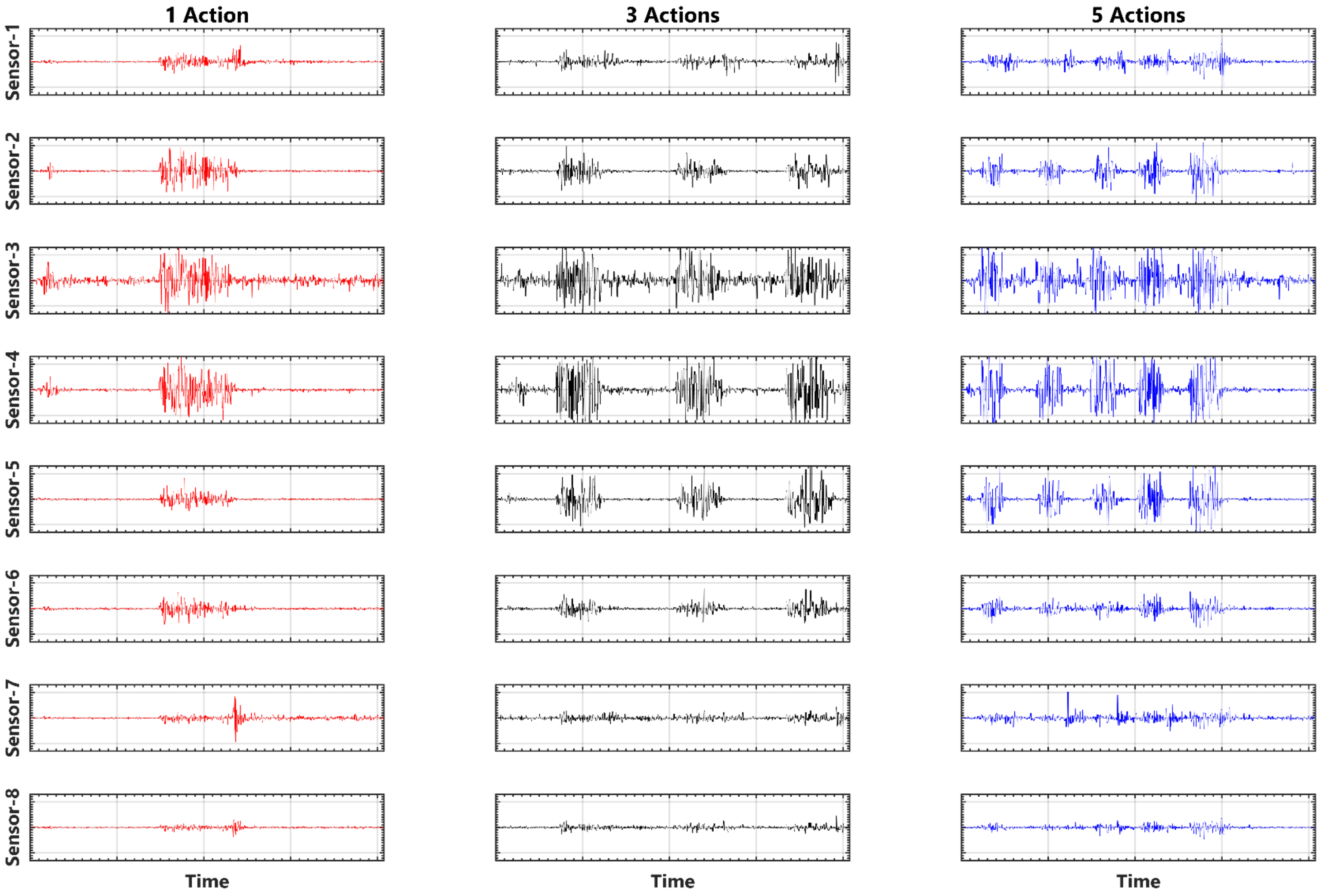


Figure 4: Raw EMG Data with Palm Facing In, Wrist Action Out

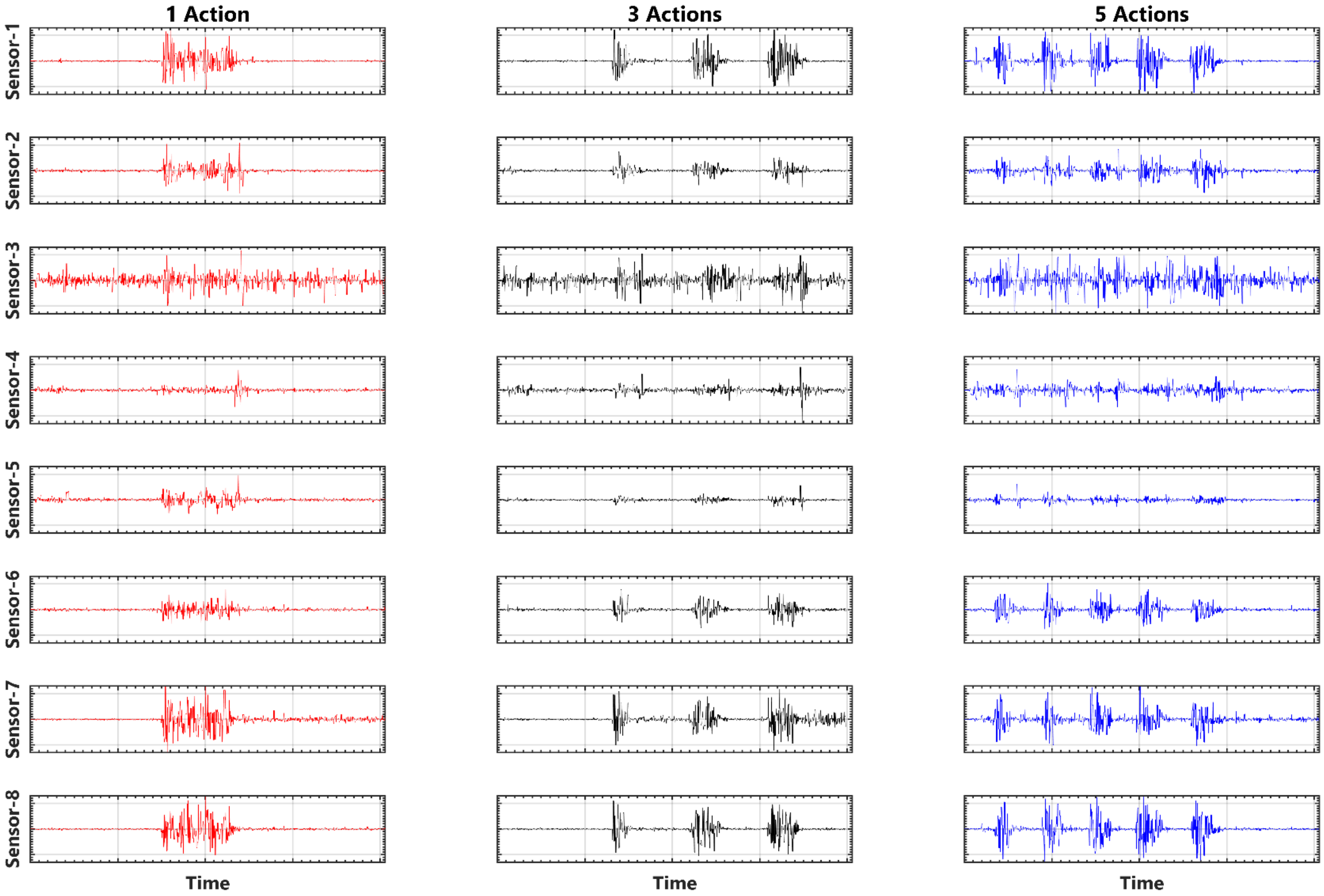


Figure 5: Raw EMG Data with Palm Facing In, Wrist Action In

B. Filtering

The first stage in preprocessing the data was applying a Kaiser filter. Multiple filter options were considered and built using MATLAB's filter design tool. The two best filters turned out to be a 50th order Hamming filter and a 248th order Kaiser filter, both pictured in Figure 6. After some experimenting, we decided that the Kaiser filter was our best option and that is what we went with.

Although the filtered data was better than just the raw signal, the data still needed to be processed further to make any sense of it. After the filter was applied, a 100-sample moving average was calculated to remove any residual noise and smooth out the data. This gave us the ability to clearly identify when the user was gesturing and which gesture was being performed.

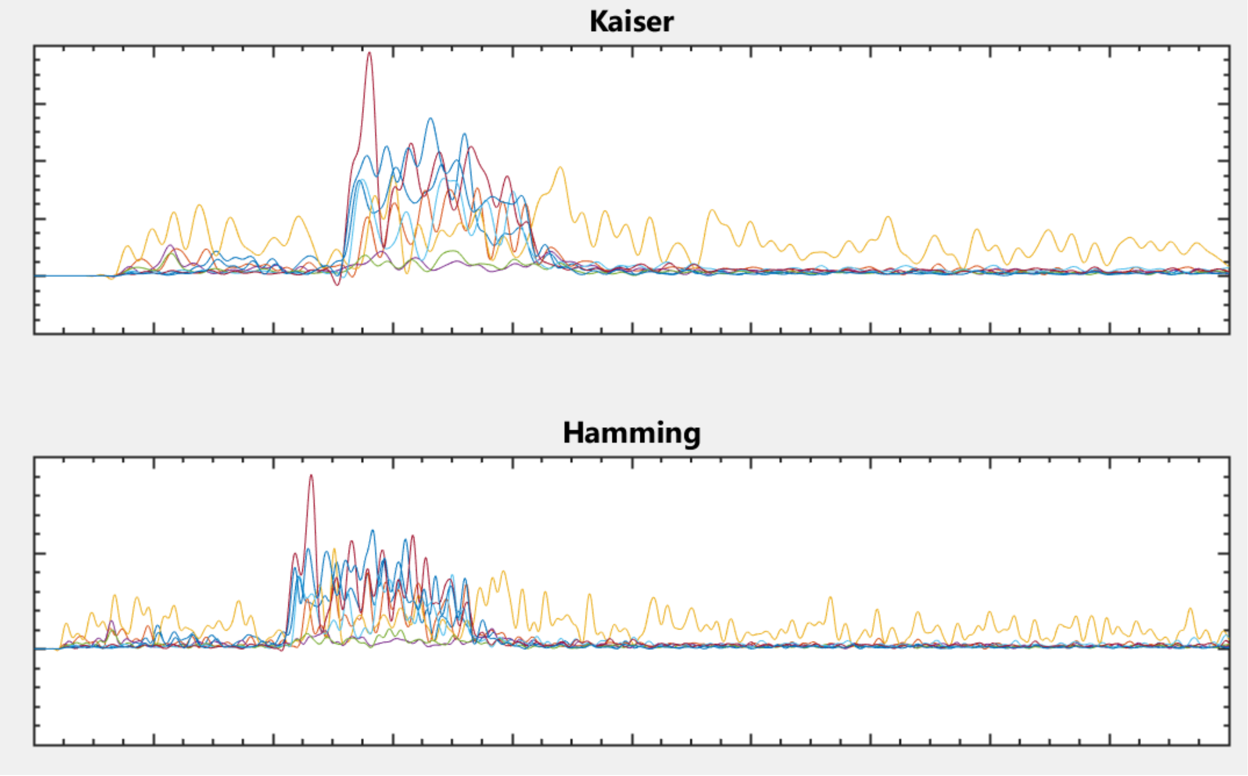


Figure 6: EMG sensor data (for all 8 sensors) when passed through each filter option

As shown in Figure 7, after each stage of preprocessing the data, the distinction between which sensors are active becomes clearer.

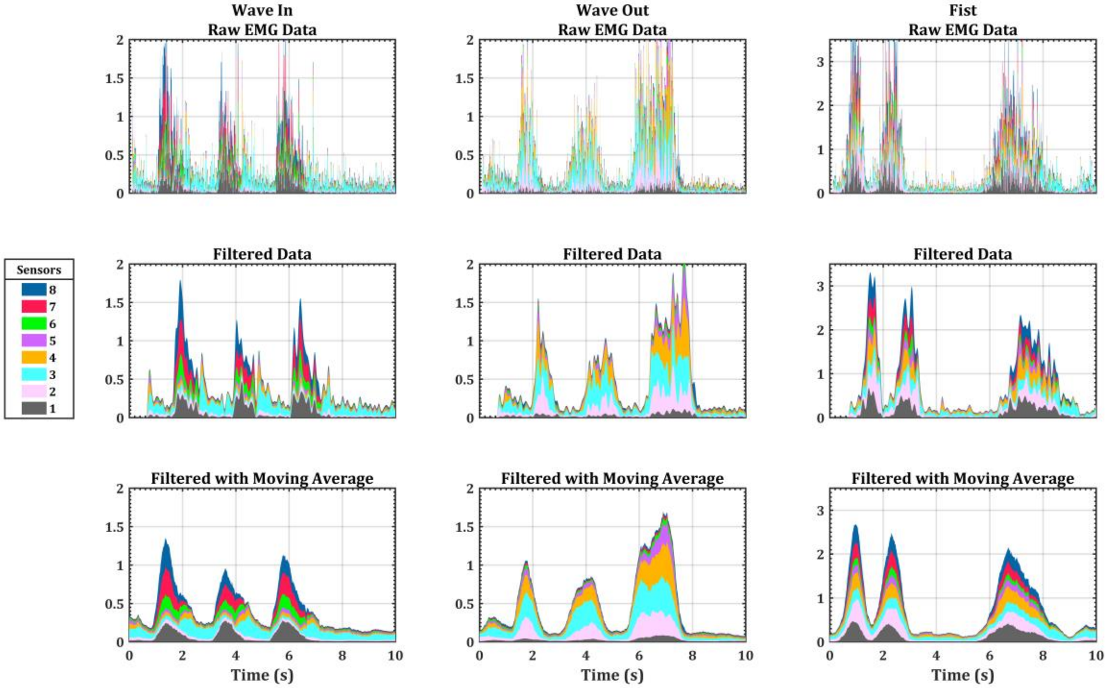


Figure 7: Three stages of preprocessing raw EMG data, for three different gestures

C. Gesture Detection Algorithm

During the process of setting up the Raspberry Pi boards, the motors and the cameras, we developed a somewhat simplistic, yet reliable method for gesture detection.

The first step in this process is get the averages of multiple groups of sensors. By grouping the sensors, we lower the chances of the algorithm being affected by inaccuracies or fluctuations in just one of the sensors. The sensor groups are shown in Table 2.

Table 2: Sensor groupings

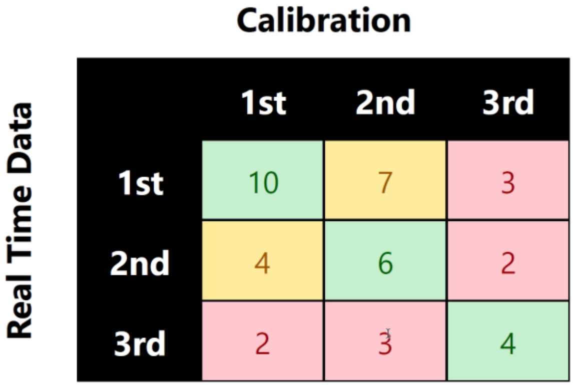
Group	Sensor Combination
1	1, 2, 3
2	2, 3, 4
3	3, 4, 5
4	4, 5, 6
5	5, 6, 7
6	6, 7, 8
7	7, 8, 1
8	8, 1, 2

The next step was to get some useful data from the groups. In both calibration and real-time data processing, the individual sensor values in each group are summed and divided by 3 to get a group average. The three groups with the highest averages are then selected to represent a gesture.

Calibration data is collected and the three highest group averages, for each gesture, are stored as a variable. When a user gestures with their arm, the confidence algorithm (described below) compares the groups of the real-time data to the groups of each calibrated gesture. Then, based on the relationship of matching groups, each calibrated gesture is assigned a confidence level. A threshold was implemented for the highest confidence level, which needed to be met to confirm that a perceived gesture was intentionally performed by the user. If the minimum threshold (10) is satisfied by at least one of the gesture comparisons, the algorithm has confirmation that the gesture with the highest confidence value is indeed the gesture that was made by the user.

To compare the real-time gesture data to the calibration data, each "match" of the top three groups are assigned a weight. For example, if the highest group average in one of the gestures calibration data matches the highest group of the real-time data, then the confidence in that action rises by 10. Then, if the second highest groups match, the confidence level for that action is increased, again, by 6. Finally, if the third groups also match, an additional 4 points are added to that gesture's total. In this case, the total points for that gesture would be 20 (10 + 6 + 4), which is the highest level of confidence possible. Lower confidence points are awarded for partial matches, which would be when the groups match but in a different order. See Table 3 for the confidence points assignments. See Table 4 for an example of how the totaling of points works.

<u>Table 3: Confidence values given for different pairings of top sensor groupings</u>



	1st	2nd	3rd
1st	10	7	3
2nd	4	6	2
3rd	2	3	4

Table 4: Example of confidence points awarded

Real Time Max Groups	5	6	2	Total Points
				20
PIWI Calibration	1	5	2	0+4+4=8
PIWO Calibration	5	6	2	10+6+4=20
FIST Calibration	1	4	8	0+0+0=0

In the example shown in Table 4, the confidence algorithm would have returned PIWO as the gesture detected because it had the highest confidence level (20), and the confidence level exceeded the threshold for gesture confirmation (9). The first, second and third highest values all matched, in the correct order, to the PIWO calibration data.

The values assigned for the confidence points were obtained experimentally through trial and error. We started out by using a plot (see Figure 8) to make a visual comparison between the highest group averages. As you can see, we started with the data on the plot but that format made it difficult to recognize patterns in the data. We then decided to remove the data and only keep the group average lines (see Figure 9), which significantly helped in identifying patterns in the top three group averages.
We then started to assign values by simply using values of one, two and three but quickly realized that was not going to work. We then realized that the correct order of the max averaged groups should increase the confidence, since it is another quality of the relationship between the real-time and calibrated data. We also lowered the point awarded for the matches that were out of order but did not want to completely disregard this relationship because the max group order does vary from time to time. We set it up so that if the max sensor groups matched, it was enough points to confirm a gesture. If not, then if the second and third highest matched, that would also tally enough confidence points that the gesture would also be confirmed. The final points we settled on were shown in Table 4.

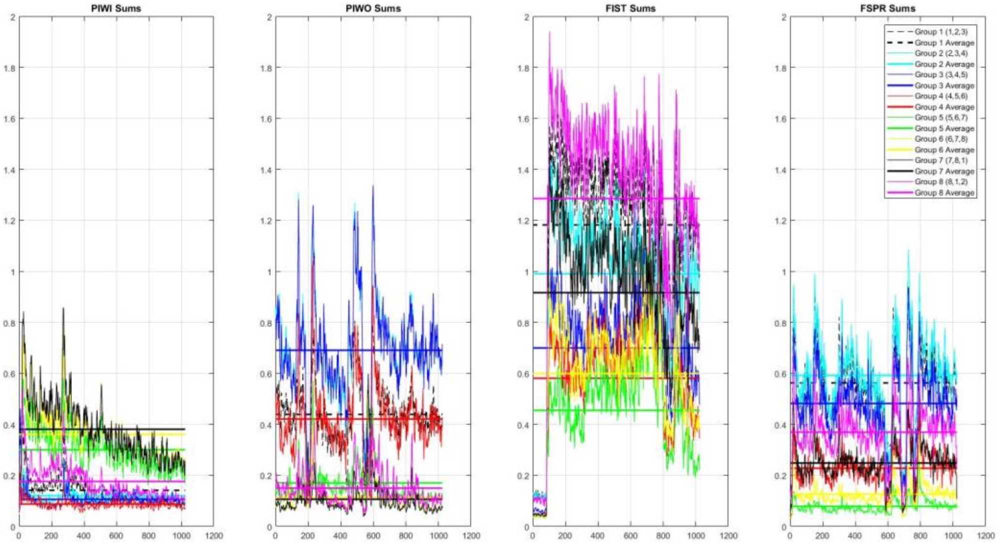


Figure 8: Sensor group sum data with the group average lines

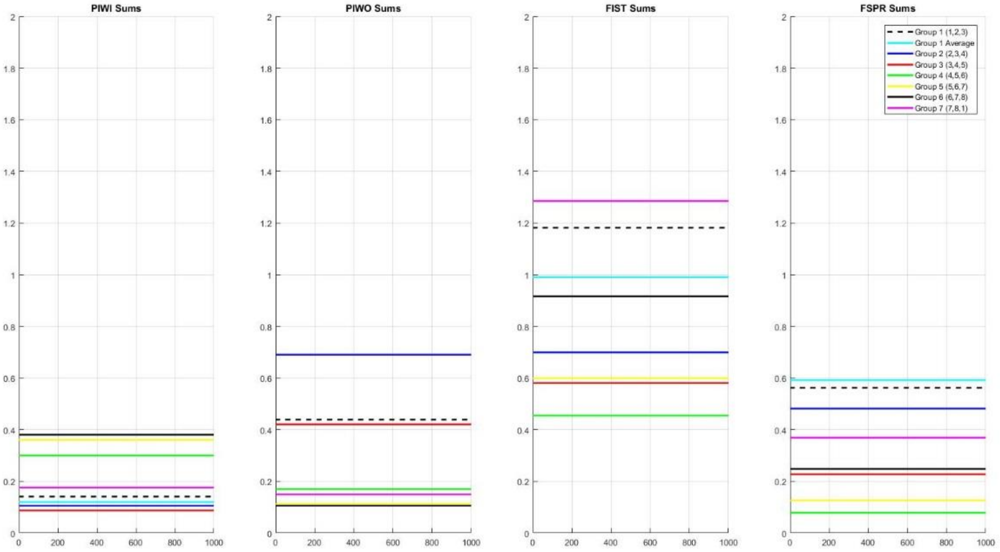


Figure 9: Line plots representing the group averages of different motions

6. Future Work

We highly suggest that research into using EMG data from the Myo armband be continued. For future projects, if our pattern recognition algorithms are not suitable, we would suggest implementing a neural network for gesture detection.

In this project, our efforts were not concentrated on implementing a neural network. We did, however, begin a small amount of research into the using a neural network. MATLAB has a built in neural network toolbox that we utilized. According to the MATLAB documentation on the *newpnn* function, a probabilistic neural network (PNN) is "a kind of radial basis network suitable for classification problems [12]."

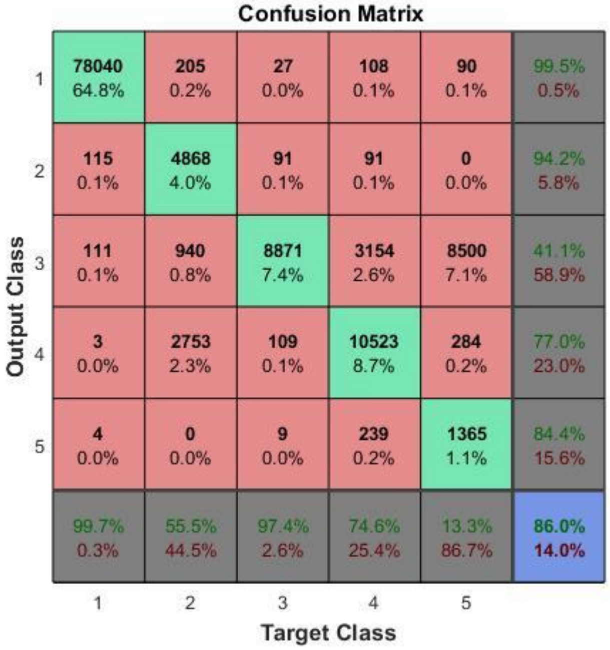
To start out with, we used raw data and used the sensor values and their sum (for each sample) as the training input and used a numerical representation of the gesture (1 through 5) for the target vector. We trained the neural network with one set of data and used data from a second person to validate the accuracy.

Using just the nine original inputs (eight raw sensor values and their sum) the PNN could only achieve an accuracy of 77.8%, which is shown in Figure 10. Since this accuracy would not be sufficient for use, we then used the preprocessed data (filtered and with the moving average), the average of the eight samples and their sum as the inputs. With more inputs to the training network, we were able to get a much higher accuracy of 86.0%. The confusion chart shown in Figure 11 represents the accuracy using these inputs.

It became clear that with more inputs, or at least more meaningful inputs, to the PNN, the accuracy would continue to increase. We did not, however, intend on the use of the PNN for this project due to the limitations of our system. Our research into the PNN was meant as a starting point, should this project be continued in the future. If a neural network is considered for a future project, more powerful microprocessors or digital signal processors (DSP) would be needed to replace the Raspberry Pi computers we used to process the data.

	1	2	3	4	5	
1	77517 64.3%	415 0.3%	87 0.1%	119 0.1%	212 0.2%	98.9% 1.1%
2	332 0.3%	4252 3.5%	3495 2.9%	6569 5.5%	1866 1.5%	25.7% 74.3%
3	30 0.0%	519 0.4%	5024 4.2%	667 0.6%	5162 4.3%	44.1% 55.9%
4	144 0.1%	1779 1.5%	371 0.3%	4949 4.1%	979 0.8%	60.2% 39.8%
5	250 0.2%	1801 1.5%	130 0.1%	1811 1.5%	2020 1.7%	33.6% 66.4%
	99.0% 1.0%	48.5% 51.5%	55.2% 44.8%	35.1% 64.9%	19.7% 80.3%	77.8% 22.2%

Figure 10: Confusion matrix using the raw EMG data as the PNN inputs



	1	2	3	4	5	
1	78040 64.8%	205 0.2%	27 0.0%	108 0.1%	90 0.1%	99.5% 0.5%
2	115 0.1%	4868 4.0%	91 0.1%	91 0.1%	0 0.0%	94.2% 5.8%
3	111 0.1%	940 0.8%	8871 7.4%	3154 2.6%	8500 7.1%	41.1% 58.9%
4	3 0.0%	2753 2.3%	109 0.1%	10523 8.7%	284 0.2%	77.0% 23.0%
5	4 0.0%	0 0.0%	9 0.0%	239 0.2%	1365 1.1%	84.4% 15.6%
	99.7% 0.3%	55.5% 44.5%	97.4% 2.6%	74.6% 25.4%	13.3% 86.7%	86.0% 14.0%

Figure 11: Confusion matrix using the preprocessed EMG data as the PNN inputs

7. Summary

Historically, the ability to control a system with hand gestures has been limited. Gesture control often required bulky equipment or relied on image processing to track user motion within the viewing range of a camera. This project addressed this limitation by developing a lightweight system that is controlled only by hand gestures detected via electromyography. Hand gestures are detected by analyzing the EMG signals produced by muscle activity in a user's arm. The EMG signals from the arm are captured by the Myo Gesture Control Armband. We developed an algorithm to process the EMG data to quickly and accurately recognize three unique hand gestures. It is these hand gestures that are used to control the camera system.

In addition to getting this system running, we explored advanced methods of pattern recognition, including neural networks and support vector machines. We achieved fairly high accuracy using a pattern recognition neural network, and are confident that given more time and effort, this method can be a viable form of gesture detection.

From the start of this project, our goal was to collect and analyze raw EMG data with the intention of putting it to use in a control system. Our goal has been accomplished by implementing an algorithm to detect gestures and control a camera system. Additionally, we have laid the groundwork for future projects in the EMG based HMI field by starting the research into more accurate methods of gesture detection.

8. References

[1] M. B. I. Reaz, M. S. Hussain, and F. Mohd-Yasin, “Techniques of EMG signal analysis: detection, processing, classification and applications,” Biological Procedures Online, vol. 8, no. 1, pp. 163–163, Oct. 2006.

[2] “Electromyography,” Medline Plus, 06-Nov-2017. [Online]. Available: https://medlineplus.gov/ency/article/003929.htm. [Accessed: 10-Nov-2017].

[3] J. H. Feinberg, “EMG Testing: A Patients Guide,” Hospital for Special Surgery, 21-Oct-2009. [Online]. Available: https://www.hss.edu/conditions_emg-testing-a-patient-guide.asp. [Accessed: 05-Nov-2017].

[4] S. Sudarsan and E. C. Sekaran, “Design and Development of EMG Controlled Prosthetics Limb,” Procedia Engineering, vol. 38, pp. 3547–3551, Sep. 2012.

[5] D. Nishikawa, Wenwei Yu, H. Yokoi and Y. Kakazu, "EMG prosthetic hand controller using real-time learning method," Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. 1999 IEEE International Conference on, Tokyo, 1999, pp. 153-158 vol.1.

[6] L. Fraiwan, M. Awwad, M. Mahdawi, and S. Jamous, “Real time virtual prosthetic hand controlled using EMG signals,” in Biomedical Engineering (MECBME), 2011 1st Middle East Conference on, 2011, pp. 225-227.

[7] C. Donalek, “Supervised and Unsupervised Learning,” Caltech Astronomy, Apr-2011. [Online]. Available: http://www.astro.caltech.edu/~george/aybi199/Donalek_Classif.pdf . [Accessed: 01-Nov-2017].

[8] “Unsupervised Learning,” MATLAB & Simulink. [Online]. Available: https://www.mathworks.com/discovery/unsupervised-learning.html. [Accessed: 2018].

[9] D. Zhu, “myo-raw,” Dec-2014. [Online]. Available: https://github.com/dzhu/myo-raw. [Accessed: May-2018].

[10] F. Cosentino, “PyoConnect_v2.0,” [Online]. Available: http://www.fernandocosentino.net/pyoconnect/ [Accessed: May-2018]

[11] “RPi-Cam-Web-Interface,” elinux.org. March 18, 2018. [Online]. Available: https://elinux.org/RPi-Cam-Web-Interface. [Accessed: May-2018].

[12] Mathworks. (2017). *Neural Network Toolbox: User's Guide* (r2017a). [Online]. Available: https://www.mathworks.com/help/nnet/ref/newpnn.html?searchHighlight=newpnn&s_tid=doc_srchtitle

Code Appendix

Please disregard any formatting abnormalities you see in the below code. The original code with clean formatting can be found on our github repository: https://github.com/abpatel2/2017-2018_EMG_senior_project.

A. main.py

```
'''
AUTHOR: Aditya Patel and Jim Ramsay
DATE CREATED: 2018-03-01
LAST MODIFIED: 2018-04-12
PLATFORM: Raspberry Pi 3B, Raspbian Stretch Released 2017-11-29
PROJECT: EMG Human Machine Interface
ORGANIZATION: Bradley University, School of Electrical and Computer Engineering
FILENAME: main.py
DESCRIPTION:
        Main script that:
                - initializes/executes bluetooth protocol (not written by Aditya/Jim -- see note below)
                - starts reading emg data.
                - detects gestures
                - commands two slave raspberry pi's to rotate servo motors
                - switches between displaying video feed from each of the slaves

        Gestures (right hand only, have not tested on left hand):
                rest -- do nothing, arm relaxed
                fist -- tight fist
                piwo -- palm in, wrist out (wave outward)
                piwi -- palm in, wrist in (wave inward)

        Master:
                emgPi_3 -- pi@169.254.12.52 password is "ee00"

        Slaves:
                ssh commands recognize the defined names for the slaves using ssh_keys. Using the defined
                names and saved keys bypasses password requirements.

                emgPi_1 -- pi@169.254.184.5  password is "ee00"
                emgPi_2 -- pi@169.254.13.230 password is "ee00"

NOTE:
        Original by dzhu
                https://github.com/dzhu/myo-raw

        Edited by Fernando Cosentino
                http://www.fernandocosentino.net/pyoconnect

        Edited further by Aditya Patel and Jim Ramsay
                There are a lot of global variables used to function like constants. This is likely not good practice
                but had to be done to meet deadlines.

                The majority of the code that we wrote is at the bottom of the script, after all of the Bluetooth and armband related code.
'''

from __future__ import print_function
import enum
import re
import struct
import sys
import threading
import time
import string
```

```python
import serial
from serial.tools.list_ports import comports
from common import *

''' Additional Imports '''
import os
import numpy as np
import csv
import datetime
from ringBuffer import ringBuffer
import displayControl as display
from calibrate import Calibrate
from guppy import hpy

'''
       GLOBAL VARIABLES
       note: a lot of these are meant to function like a "DEFINE" in C. They are never written to.
'''

''' ARRAYS '''
global emg_data
emg_data = []

global duty
duty = [50, 50]                                                                                                                                          # initial duty cycle for each motor

''' INTEGERS '''
global GETTINGCALDATA; global CALIBRATING; global SLEEP; global WAITING; global DISPLAYCONTROL; global MOTORCONTROL
GETTINGCALDATA = 0
CALIBRATING = 1
SLEEP = 2
WAITING = 3
DISPLAYCONTROL = 4
MOTORCONTROL = 5

global REST; global FIST; global PIWI; global PIWO
REST = 0
FIST = 1
PIWI = 2
PIWO = 3

global calMode
calMode = REST

global EMGPI_1; global EMGPI_2
EMGPI_1 = 0
EMGPI_2 = 1


global fistCalData;   global piwiCalData;   global piwoCalData;
fistCalData = []
piwiCalData = []
piwoCalData = []

global curPi
curPi = 0

t0 = time.time()
global t_endWaiting

gestureString = ["fist", "piwi", "piwo", ""]
```

```
modeString = ["", "", "SLEEP", "WAITING","DISPLAY CONTROL","MOTOR CONTROL"]

def multichr(ords):
	if sys.version_info[0] >= 3:
		return bytes(ords)
	else:
		return ''.join(map(chr, ords))

def multiord(b):
	if sys.version_info[0] >= 3:
		return list(b)
	else:
		return map(ord, b)

class Arm(enum.Enum):
	UNKNOWN = 0
	RIGHT = 1
	LEFT = 2

class XDirection(enum.Enum):
	UNKNOWN = 0
	X_TOWARD_WRIST = 1
	X_TOWARD_ELBOW = 2

class Pose(enum.Enum):
	RESTT = 0
	FIST = 1
	WAVE_IN = 2
	WAVE_OUT = 3
	FINGERS_SPREAD = 4
	THUMB_TO_PINKY = 5
	UNKNOWN = 255

class Packet(object):
	def __init__(self, ords):
		self.typ = ords[0]
		self.cls = ords[2]
		self.cmd = ords[3]
		self.payload = multichr(ords[4:])

	def __repr__(self):
		return 'Packet(%02X, %02X, %02X, [%s])' % \
			(self.typ, self.cls, self.cmd,
			 ' '.join('%02X' % b for b in multiord(self.payload)))

class BT(object):
	'''Implements the non-Myo-specific details of the Bluetooth protocol.'''
	def __init__(self, tty):
		self.ser = serial.Serial(port=tty, baudrate=9600, dsrdtr=1)
		self.buf = []
		self.lock = threading.Lock()
		self.handlers = []

	## internal data-handling methods
	def recv_packet(self, timeout=None):
		t0 = time.time()
		self.ser.timeout = None
		while timeout is None or time.time() < t0 + timeout:
			if timeout is not None: self.ser.timeout = t0 + timeout - time.time()
			c = self.ser.read()
			if not c: return None
```

```

            ret = self.proc_byte(ord(c))
            if ret:
                if ret.typ == 0x80:
                    self.handle_event(ret)
                return ret

    def recv_packets(self, timeout=.5):
        res = []
        t0 = time.time()
        while time.time() < t0 + timeout:
            p = self.recv_packet(t0 + timeout - time.time())
            if not p: return res
            res.append(p)
        return res

    def proc_byte(self, c):
        if not self.buf:
            if c in [0x00, 0x80, 0x08, 0x88]:
                self.buf.append(c)
            return None
        elif len(self.buf) == 1:
            self.buf.append(c)
            self.packet_len = 4 + (self.buf[0] & 0x07) + self.buf[1]
            return None
        else:
            self.buf.append(c)

        if self.packet_len and len(self.buf) == self.packet_len:
            p = Packet(self.buf)
            self.buf = []
            return p
        return None

    def handle_event(self, p):
        for h in self.handlers:
            h(p)

    def add_handler(self, h):
        self.handlers.append(h)

    def remove_handler(self, h):
        try: self.handlers.remove(h)
        except ValueError: pass

    def wait_event(self, cls, cmd):
        res = [None]
        def h(p):
            if p.cls == cls and p.cmd == cmd:
                res[0] = p
        self.add_handler(h)
        while res[0] is None:
            self.recv_packet()
        self.remove_handler(h)
        return res[0]

    ## specific BLE commands
    def connect(self, addr):
        return self.send_command(6, 3, pack('6sBHHHH', multichr(addr), 0, 6, 6, 64, 0))

    def get_connections(self):
```

```
        return self.send_command(0, 6)

    def discover(self):
        return self.send_command(6, 2, b'\x01')

    def end_scan(self):
        return self.send_command(6, 4)

    def disconnect(self, h):
        return self.send_command(3, 0, pack('B', h))

    def read_attr(self, con, attr):
        self.send_command(4, 4, pack('BH', con, attr))
        return self.wait_event(4, 5)

    def write_attr(self, con, attr, val):
        self.send_command(4, 5, pack('BHB', con, attr, len(val)) + val)
        return self.wait_event(4, 1)

    def send_command(self, cls, cmd, payload=b'', wait_resp=True):
        s = pack('4B', 0, len(payload), cls, cmd) + payload
        self.ser.write(s)

        while True:
            p = self.recv_packet()

            ## no timeout, so p won't be None
            if p.typ == 0: return p

            ## not a response: must be an event
            self.handle_event(p)

class MyoRaw(object):
    '''Implements the Myo-specific communication protocol.'''

    def __init__(self, tty=None):
        if tty is None:
            tty = self.detect_tty()
        if tty is None:
            raise ValueError('Myo dongle not found!')

        self.bt = BT(tty)
        self.conn = None
        self.emg_handlers = []
        self.imu_handlers = []
        self.arm_handlers = []
        self.pose_handlers = []

    def detect_tty(self):
        for p in comports():
            if re.search(r'PID=2458:0*1', p[2]):
                print('using device:', p[0])
                return p[0]

        return None

    def run(self, timeout=None):
        self.bt.recv_packet(timeout)

    def connect(self):
        ## stop everything from before
```

```
                self.bt.end_scan()
                self.bt.disconnect(0)
                self.bt.disconnect(1)
                self.bt.disconnect(2)


                ## start scanning
                print('scanning for bluetooth devices...')
                self.bt.discover()
                while True:
                        p = self.bt.recv_packet()
                        print('scan response:', p)

                        if p.payload.endswith(b'\x06\x42\x48\x12\x4A\x7F\x2C\x48\x47\xB9\xDE\x04\xA9\x01\x00\x06\xD5'):
                                addr = list(multiord(p.payload[2:8]))
                                break
                self.bt.end_scan()

                ## connect and wait for status event
                conn_pkt = self.bt.connect(addr)
                self.conn = multiord(conn_pkt.payload)[-1]
                self.bt.wait_event(3, 0)

                ## get firmware version
                fw = self.read_attr(0x17)
                _, _, _, _, v0, v1, v2, v3 = unpack('BHBBHHHH', fw.payload)
                print('firmware version: %d.%d.%d.%d' % (v0, v1, v2, v3))

                self.old = (v0 == 0)

                if self.old: # if the firmware is 0.x.xxxx.x
                        ## don't know what these do; Myo Connect sends them, though we get data
                        ## fine without them
                        self.write_attr(0x19, b'\x01\x02\x00\x00')
                        self.write_attr(0x2f, b'\x01\x00')
                        self.write_attr(0x2c, b'\x01\x00')
                        self.write_attr(0x32, b'\x01\x00')
                        self.write_attr(0x35, b'\x01\x00')

                        ## enable EMG data
                        self.write_attr(0x28, b'\x01\x00')
                        ## enable IMU data
                        self.write_attr(0x1d, b'\x01\x00')

                        ## Sampling rate of the underlying EMG sensor, capped to 1000. If it's
                        ## less than 1000, emg_hz is correct. If it is greater, the actual
                        ## framerate starts dropping inversely. Also, if this is much less than
                        ## 1000, EMG data becomes slower to respond to changes. In conclusion,
                        ## 1000 is probably a good value.
                        C = 1000
                        emg_hz = 50
                        ## strength of low-pass filtering of EMG data
                        emg_smooth = 100

                        imu_hz = 50

                        ## send sensor parameters, or we don't get any data
                        self.write_attr(0x19, pack('BBBBHBBBBB', 2, 9, 2, 1, C, emg_smooth, C // emg_hz, imu_hz, 0, 0))

                else: #normal operation
                        name = self.read_attr(0x03)
```

```
            print('device name: %s' % name.payload)

            ## enable IMU data
            self.write_attr(0x1d, b'\x01\x00')
            ## enable vibrations
            self.write_attr(0x24, b'\x02\x00')
            # Failed attempt to disable vibrations:
            # self.write_attr(0x24, b'\x00\x00')

            # self.write_attr(0x19, b'\x01\x03\x00\x01\x01')
            self.start_raw()

        ## add data handlers
        def handle_data(p):
            if (p.cls, p.cmd) != (4, 5): return
            c, attr, typ = unpack('BHB', p.payload[:4]) # unpack unsigned char, unsigned short, unsigned char
            pay = p.payload[5:]
            if attr == 0x27:
                vals = unpack('8HB', pay)    # unpack 8 unsigned shorts, and one unsigned char https://docs.python.org/2/library/struct.html
                                                                   ## not entirely sure what the last byte is, but it's a bitmask that
                                                                   ## seems to indicate which sensors think they're being moved around or
                                                                   ## something
                emg = vals[:8]
                moving = vals[8]
                self.on_emg(emg, moving)
            elif attr == 0x1c:
                vals = unpack('10h', pay)
                quat = vals[:4]
                acc = vals[4:7]
                gyro = vals[7:10]
                self.on_imu(quat, acc, gyro)
            elif attr == 0x23:
                typ, val, xdir, _,_,_ = unpack('6B', pay)

                if typ == 1: # on arm
                    self.on_arm(Arm(val), XDirection(xdir))
                    print("on arm")
                elif typ == 2: # removed from arm
                    self.on_arm(Arm.UNKNOWN, XDirection.UNKNOWN)
                    print("NOT on arm")
                elif typ == 3: # pose
                    self.on_pose(Pose(val))
            else:
                print('data with unknown attr: %02X %s' % (attr, p))

        self.bt.add_handler(handle_data)

    def write_attr(self, attr, val):
        if self.conn is not None:
            self.bt.write_attr(self.conn, attr, val)

    def read_attr(self, attr):
        if self.conn is not None:
            return self.bt.read_attr(self.conn, attr)
        return None

    def disconnect(self):
        if self.conn is not None:
            self.bt.disconnect(self.conn)

    def start_raw(self):
```

```
        '''Sending this sequence for v1.0 firmware seems to enable both raw data and
        pose notifications.
        '''

        self.write_attr(0x28, b'\x01\x00')
        #self.write_attr(0x19, b'\x01\x03\x01\x01\x00')
        self.write_attr(0x19, b'\x01\x03\x01\x01\x01')

    def mc_start_collection(self):
        '''Myo Connect sends this sequence (or a reordering) when starting data
        collection for v1.0 firmware; this enables raw data but disables arm and
        pose notifications.
        '''

        self.write_attr(0x28, b'\x01\x00')
        self.write_attr(0x1d, b'\x01\x00')
        self.write_attr(0x24, b'\x02\x00')
        self.write_attr(0x19, b'\x01\x03\x01\x01\x01')
        self.write_attr(0x28, b'\x01\x00')
        self.write_attr(0x1d, b'\x01\x00')
        self.write_attr(0x19, b'\x09\x01\x01\x00\x00')
        self.write_attr(0x1d, b'\x01\x00')
        self.write_attr(0x19, b'\x01\x03\x00\x01\x00')
        self.write_attr(0x28, b'\x01\x00')
        self.write_attr(0x1d, b'\x01\x00')
        self.write_attr(0x19, b'\x01\x03\x01\x01\x00')

    def mc_end_collection(self):
        '''Myo Connect sends this sequence (or a reordering) when ending data collection
        for v1.0 firmware; this reenables arm and pose notifications, but
        doesn't disable raw data.
        '''

        self.write_attr(0x28, b'\x01\x00')
        self.write_attr(0x1d, b'\x01\x00')
        self.write_attr(0x24, b'\x02\x00')
        self.write_attr(0x19, b'\x01\x03\x01\x01\x01')
        self.write_attr(0x19, b'\x09\x01\x00\x00\x00')
        self.write_attr(0x1d, b'\x01\x00')
        self.write_attr(0x24, b'\x02\x00')
        self.write_attr(0x19, b'\x01\x03\x00\x01\x01')
        self.write_attr(0x28, b'\x01\x00')
        self.write_attr(0x1d, b'\x01\x00')
        self.write_attr(0x24, b'\x02\x00')
        self.write_attr(0x19, b'\x01\x03\x01\x01\x01')

    def vibrate(self, length):
        if length in xrange(1, 4):
            ## first byte tells it to vibrate; purpose of second byte is unknown
            self.write_attr(0x19, pack('3B', 3, 1, length))


    def add_emg_handler(self, h):
        self.emg_handlers.append(h)

    def add_imu_handler(self, h):
        self.imu_handlers.append(h)

    def add_pose_handler(self, h):
        self.pose_handlers.append(h)
```

```
        def add_arm_handler(self, h):
                self.arm_handlers.append(h)


        def on_emg(self, emg, moving):
                for h in self.emg_handlers:
                        h(emg, moving)

        def on_imu(self, quat, acc, gyro):
                for h in self.imu_handlers:
                        h(quat, acc, gyro)

        def on_pose(self, p):
                for h in self.pose_handlers:
                        h(p)

        def on_arm(self, arm, xdir):
                for h in self.arm_handlers:
                        h(arm, xdir)


def controlLogic(mode, gesture, confidence):
        global SLEEP; global WAITING; global DISPLAYCONTROL; global MOTORCONTROL;
        global REST; global FIST; global PIWI; global PIWO
        global duty; global curPi; global t_endWaiting; global t_30_SLEEP

        if ( mode == SLEEP ):

                if ( gesture == FIST ):
                        mode = WAITING
                        t_endWaiting = time.time() + 1                                                                                             # Reset the sleep timer once you leave SLEEP
                        print("SWITCHING MODE: WAITING\t\t\tConfidence Level: ", confidence)
                        t_30_SLEEP = time.time() + 30

        if ( mode == WAITING ):
                if ( time.time() >= t_30_SLEEP ):

                        mode = SLEEP
                        print("SWITCHING MODE: SLEEP")

                else:

                        # print("MODE = WAITING")
                        if ( time.time() > t_endWaiting ):
                                if ( gesture == FIST ):

                                        mode = SLEEP
                                        print("SWITCHING MODE: SLEEP\t\t\tConfidence Level: ",confidence)

                                elif ( gesture == PIWI ):

                                        mode = DISPLAYCONTROL
                                        print("SWITCHING MODE: DISPLAYCONTROL\t\t\tConfidence Level: ",confidence)
                                        t_endWaiting = time.time() + 1
                                        t_30_SLEEP = time.time() + 30

                                elif ( gesture == PIWO ):

                                        mode = MOTORCONTROL
                                        print("SWITCHING MODE: MOTORCONTROL\t\t\tConfidence Level: ",confidence)
                                        t_endWaiting = time.time() + 1
```

```
                        t_30_SLEEP = time.time() + 30

        if ( mode == DISPLAYCONTROL ):

            if ( time.time() >= t_30_SLEEP ):

                mode = SLEEP
                print("SWITCHING MODE: SLEEP")

            else:

                if ( time.time() > t_endWaiting ):
                    if ( gesture == FIST ):

                        mode = WAITING
                        print("SWITCHING MODE: WAITING\t\t\tConfidence Level: ",confidence)
                        t_endWaiting = time.time() + 1
                        t_30_SLEEP = time.time() + 30

                    elif ( ( curPi == 0 ) and ( gesture == PIWI ) ):

                        curPi = display.switchDisplay()
                        print("Switching to Camera 2")
                        t_endWaiting = time.time() + 1
                        t_30_SLEEP = time.time() + 30

                    elif ( ( curPi == 1 ) and ( gesture == PIWO ) ):

                        curPi = display.switchDisplay()
                        print("Switching to Camera 1")
                        t_endWaiting = time.time() + 1
                        t_30_SLEEP = time.time() + 30

        if ( mode == MOTORCONTROL ):

            if ( time.time() >= t_30_SLEEP ):

                mode = SLEEP
                print("SWITCHING MODE: SLEEP")

            else:

                if ( time.time() > t_endWaiting ):
                    ''' Select which slave to control '''
                    if ( curPi == 0 ):

                        curPi_name = "emgPi_1"
                        currentMotor = 0

                    elif ( curPi == 1 ):

                        curPi_name = "emgPi_2"
                        currentMotor = 1

                    ''' Check Gesture '''
                    if ( gesture == PIWI ):                                                                                                    # Pan Clockwise

                        if (duty[curPi] <= 70):

                            duty[curPi] += 10

```

```
                                                ssh_string = "ssh " + curPi_name + " 'python /home/pi/scripts/moveMotor.py  " + str(duty[curPi]) + " 0 0' &"
                                                os.system(ssh_string)

                                        elif ( ( duty[curPi] > 70 ) and ( duty[curPi] < 80 ) ):
                                                duty[curPi] = 80
                                                ssh_string = "ssh " + curPi_name + " 'python /home/pi/scripts/moveMotor.py  " + str(duty[curPi]) + " 0 0' &"
                                                os.system(ssh_string)
                                                print("Motor is at limit.")

                                        t_endWaiting = time.time() + 1
                                        t_30_SLEEP = time.time() + 30


                                elif ( gesture == PIWO ):                                                                                                       # Pan Counter Clockwise

                                        if ( duty[curPi] >= 30 ):

                                                duty[curPi] -= 10
                                                ssh_string = "ssh " + curPi_name + " 'python /home/pi/scripts/moveMotor.py  " + str(duty[curPi]) + " 1 0' &"
                                                os.system(ssh_string)

                                        elif ( ( duty[curPi] < 30 ) and ( duty[curPi] > 20 ) ):

                                                duty[curPi] = 20
                                                ssh_string = "ssh " + curPi_name + " 'python /home/pi/scripts/moveMotor.py  " + str(duty[curPi]) + " 1 0' &"
                                                os.system(ssh_string)
                                                print("Motor is at limit.")

                                        else:
                                                print("Motor is out of range. Cannot rotate CCW")

                                        t_endWaiting = time.time() + 1
                                        t_30_SLEEP = time.time() + 30

                                elif ( gesture == FIST ):

                                        mode = WAITING
                                        print("SWITCHING MODE: WAITING\t\t\t\tConfidence Level: ", confidence)
                                        t_endWaiting = time.time() + 1

        return mode


def getConfidence(realTimeData, calData):

        matchCounter = 0

        '''
                calibrated: 823
                actual:         832
                result:         10 + 2 + 3 = 15

                calibrated: 781
                actual:         832
                result:         7

                calibrated: 231
                actual:         832
                result:         1 + 6 + = 7
        '''

```

```
        if (realTimeData[0] == calData[0]):
                matchCounter += 10
        if (realTimeData[0] == calData[1]):
                matchCounter += 7
        if (realTimeData[0] == calData[2]):
                matchCounter += 3

        if (realTimeData[1] == calData[0]):
                matchCounter += 4
        if (realTimeData[1] == calData[1]):
                matchCounter += 6
        if (realTimeData[1] == calData[2]):
                matchCounter += 2

        if (realTimeData[2] == calData[0]):
                matchCounter += 2
        if (realTimeData[2] == calData[1]):
                matchCounter += 3
        if (realTimeData[2] == calData[2]):
                matchCounter += 4

        return matchCounter


'''
        If the gesture is the same as the last one, increment the counter. If the gesture is different from the last gesture,
        update the variable, lastGesture, and reset the counter. This allows us to wait for n counts of the same gesture before
        considering a gesture valid.
'''
def confirmGesture(gesture):
        global CONFIRM_COUNTER

        if ( confirmGesture.lastGesture != gesture ):
                confirmGesture.flag = False

        if ( confirmGesture.counter < CONFIRM_COUNTER ):
                confirmGesture.counter += 1
                confirmGesture.flag = False
        else:
                confirmGesture.lastGesture = gesture
                confirmGesture.counter = 0
                confirmGesture.flag = True


        return confirmGesture.flag

confirmGesture.flag = False                                                                                                                 # static variable initialization for the above function
confirmGesture.counter = 0
confirmGesture.lastGesture = REST

if __name__ == '__main__':

        m = MyoRaw(sys.argv[1] if len(sys.argv) >= 2 else None)                                                      # this has to come first, and proc_emg() second (see below)

        def proc_emg(emg, moving, times = []):                                                                                      # data is sent in packets of two samples at a time. I *think* we only save half of
these
                global calMode; global emg_data
                global fistCalData;   global piwiCalData;   global piwoCalData;

                emg = list(emg)                                                                                                                           # convert tuple to list
                emg_data = emg
```

```
    if ( mode == GETTINGCALDATA ):                                                                                               # write calibration data to a global array

        if (calMode == FIST):
            fistCalData.append(emg_data)
        if (calMode == PIWI):
            piwiCalData.append(emg_data)
        if (calMode == PIWO):
            piwoCalData.append(emg_data)

'''
    INITIALIZATION
    this code is only executed once
'''
m.add_emg_handler(proc_emg)
m.connect()
global GETTINGCALDATA; global CALIBRATING; global SLEEP; global WAITING; global DISPLAYCONTROL; global MOTORCONTROL;
global REST; global FIST; global PIWI; global PIWO; global calMode; global curPi; global CONFIRM_COUNTER;

os.system("python displayControl.py &")                                                                                     # initializes the display on every run

confidenceArray = []

curPi = 0
gesture = REST
isResting = 0

BUFFER_SIZE = 100                                                                                                                         # size of circular buffer
emg_buffer = ringBuffer(BUFFER_SIZE)
counter = 0                                                                                                                                       # counter

CONFIDENCE_LEVEL = 10                                                                                                             # allows for tuning. Max = 20. Min = 0. See getConfidence()
CONFIRM_COUNTER = 150                                                                                                             # number of samples of same gesture required to confirm a gesture
SENSITIVITY = 75                                                                                                                          # upper and lower threshold = minValueFromCal +/- SENSITIVITY

NUM_CALS = 4                                                                                                                              # this is always 1 greater than the number of calibrations

CALIBRATION_SIZE = 500
n = CALIBRATION_SIZE
CSVFILE = "./adityaCal.csv"                                                                                                       # file to write/read calibration data from
minValueFromCal = 9999                                                                                                                    # initially an arbitrarily large value
iWantToCal = 0                                                                                                                            # set to '1' when switching users or when recalibration is needed
calibrateFlag = 1

if ( iWantToCal == 1 ):
    mode = GETTINGCALDATA
else:
    mode = SLEEP                                                                                                                          # skip GETTINGCALDATA and CALIBRATING states

os.system("ssh emgPi_1 'python /home/pi/scripts/initMotor.py 50' &")                            # The ampersand is essential here. If this does not run in the background ...
os.system("ssh emgPi_2 'python /home/pi/scripts/initMotor.py 50' &")                            # the bluetooth protocol fails and the system is frozen.

print("MOTORS INITIALIZED")
os.system("clear")

while True:                                                                                                                                       # run the program indefinitely, or until user interruption
    m.run()

    emg_buffer.append(emg_data)

    if (counter >= BUFFER_SIZE * 2):                                                                                        # there was an undiagnosed issue with 7 null data points causing havoc.
```

```
                                                                                                                        # this ensures that those are gone before proceeding

average = emg_buffer.getAvg()                                                           # average value of each sensor in the buffer. [ 1 x 8 ]

bufferAvg = np.mean(np.array(average))                                                  # average value of the whole buffer. type: float, [1 x 1]

maxGrouping = emg_buffer.getMaxGrouping()

if ( mode >= SLEEP ):                                                                          # where the main gesture detection and control happens

    if ( calibrateFlag == 1 ):                                                                 # load saved cal data
        with open(CSVFILE, 'rb') as csvfile:                                             # Example: [ 7, 6, 1]; [ 4, 2, 5]; [ 0, 2, 7]; [ 157.6, 157.6, 157.6]
            CalReader = csv.reader(csvfile, delimiter=',')
            i = 0
            for row in CalReader:
                savedCalData = np.genfromtxt(CSVFILE, delimiter=',')

        print("Calibration Data: \n", savedCalData)
        print("MODE = SLEEP")
        calibrateFlag = 0
        fistGrouping = savedCalData[0]
        piwiGrouping = savedCalData[1]
        piwoGrouping = savedCalData[2]
        minValueFromCal = savedCalData[3,1]

    fistConfidence = getConfidence(maxGrouping, fistGrouping)
    piwiConfidence = getConfidence(maxGrouping, piwiGrouping)
    piwoConfidence = getConfidence(maxGrouping, piwoGrouping)

    confidenceArray = [fistConfidence, piwiConfidence, piwoConfidence]

    maxMatch = np.argmax(confidenceArray)                                               # index of the gesture that returned the most confidence
    maxConfidence = confidenceArray[maxMatch]                                    # confidence level of the most confident gesture

    if ( ( bufferAvg >= ( minValueFromCal + SENSITIVITY ) ) ):
        if ( maxMatch == 0 ) and ( fistConfidence >= CONFIDENCE_LEVEL) :

            if ( confirmGesture(FIST) ):                                                # if we saw FIST for n times
                gesture = FIST
                print("\tFIST CONFIRMED\t\t\t\tConfidence Level: ", fistConfidence)

                isResting = 0

        elif ( maxMatch == 1 ) and ( piwiConfidence >= CONFIDENCE_LEVEL ):

            if ( confirmGesture(PIWI) ):                                                # if we saw PIWI for n times
                gesture = PIWI
                print("\tPIWI CONFIRMED\t\t\t\tConfidence Level: ", piwiConfidence)

                isResting = 0

        elif ( maxMatch == 2 ) and ( piwoConfidence >= CONFIDENCE_LEVEL ):

            if ( confirmGesture(PIWO) ):                                                # if we saw PIWO for n times
                gesture = PIWO
                print("\tPIWO CONFIRMED\t\t\t\tConfidence Level: ", piwoConfidence)

                isResting = 0
        else:
            if ( confirmGesture(REST) ):                                                # if we saw REST for n times
                gesture = REST
```

```
                    print("\n\n\tMOTION DETECTED BUT NO GESTURE MATCH: REST ASSUMED")
                    print("\n\tMinimum Accepted Confidence: ", CONFIDENCE_LEVEL)
                    print("\tFIST Confidence: ",fistConfidence, "\tPIWI Confidence: ",piwiConfidence, "\tPIWO Confidence: ",piwoConfidence)
                    print("\tStill in mode: ", modeString[mode])
                    print("\n\n")

        elif ( (bufferAvg < (minValueFromCal - SENSITIVITY)) ): #isResting or

            #print("REST CONFIRMED")
            gesture = REST
            isResting = 1

        #else:

            # print("UNKNOWN")
            # print("Sensitivity: ", SENSITIVITY)
            # print("minValueFromCal: ", minValueFromCal)
            # print("Buffer average: ", bufferAvg)

        mode = controlLogic(mode, gesture, maxConfidence)                                          # get new mode

    '''
        CALIBRATION
        note: this can probably be put into a function later. Maybe not all of it, but enough that it becomes a little easier to follow
    '''
    if ( ( mode == GETTINGCALDATA ) and ( calMode < NUM_CALS ) ):

        if (n >= CALIBRATION_SIZE):

            n = 0                                                                                                                   # reset calibration timer
            print("Cal Mode = " + gestureString[calMode])
            print("Hold a " + gestureString[calMode] + " until told otherwise")
            calMode += 1
            # time.sleep(2)                                                                                      # WARNING: THIS BREAKS THE CODE! # sleep to give user time to switch to next gesture

        n += 1
        if (bufferAvg < minValueFromCal):                                                                        # this gets the minimum 8-sensor average from the time that calibration was run
            minValueFromCal = bufferAvg                                                                                  # it sets the threshold that separates gestures from resting.

    else:

        if ( calibrateFlag == 1 ):
            mode = CALIBRATING

        gesture = REST
        mode = controlLogic(mode, gesture, 0)


    if ( mode == CALIBRATING ) :

        print("mode = CALIBRATING")
        fistCal = Calibrate()
        fistGrouping = fistCal.getMaxGrouping(fistCalData)

        piwiCal = Calibrate()
        piwiGrouping = piwiCal.getMaxGrouping(piwiCalData)

        piwoCal = Calibrate()
        piwoGrouping = piwoCal.getMaxGrouping(piwoCalData)

        minValueFromCalArray = [minValueFromCal,minValueFromCal,minValueFromCal]
```

```

                with open(CSVFILE, 'w') as csvfile:
                    writer = csv.writer(csvfile)
                    writer.writerow(fistGrouping)
                    writer.writerow(piwiGrouping)
                    writer.writerow(piwoGrouping)
                    writer.writerow(minValueFromCalArray)

                calibrateFlag = 0
                mode = SLEEP

                print("Fist Group: ", fistGrouping)
                # print(fistCalData)
                print("Piwi Group: ", piwiGrouping)
                # print(piwiCalData)
                print("Piwo Group: ", piwoGrouping)


        else:                                                                                                                                     # Runs until data is guaranteed to be good

            counter += 1
            # print(counter, "Data contains null values\n")


```

B. calibrate.py

```
'''
AUTHOR: Aditya Patel
DATE CREATED: 2018-04-08
LAST MODIFIED: 2018-04-09
PLATFORM: Raspberry Pi 3B, Raspbian Stretch Released 2017-11-29
PROJECT: EMG Human Machine Interface
ORGANIZATION: Bradley University, School of Electrical and Computer Engineering
FILENAME: calibrate.py
DESCRIPTION:
    Calibration class. Create a unique Calibrate() object in the parent function for each gesture.
    Primary use of this class is to get the top three sensor groupings in the calibration data,
    i.e., the three trios of consecutive sensors with the highest average EMG value.

    Ex:
        If the calibration data is:
            [ 99    100    32    03    14    16    42    95 ]

        index: 0      1     2     3     4     5     6     7

        The top three sensor groups would be, in order,
            [ 701, 670, 012 ]

        Giving the return:
            [ 7, 6, 0]
'''

import numpy as np

class Calibrate():

    def __init___(self):
```

```

        self.size()
        self.data = []
        self.sums = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        self.avg = []
        # self.avg = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])                           # this has to be a numpy array to use the 'divide' function below
        self.groupingAvg = []

    '''
        Sets the class variable, data, equal to the calibration data.
    '''
    def setData(self, calData):

        self.data = calData

    '''
        Computes an average down each column of data.
        writes to self.avg, 1 x 8 array containing average value of each sensor
    '''
    def getAvg(self):

        self.sums = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]                              # reset sums to prevent it from accumulating forever. This is NOT elegant or efficient
        self.avg =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
        for r in range(0,len(self.data)):
            for c in range(0,8):
                self.sums[c] = self.sums[c] + self.data[r][c]

        np.divide(self.sums, float(len(self.data)), out = self.avg)                       # compute the average by dividing the sums by the size of the data array
        #return self.avg

    '''
        Function to compute the average value of each of the eight groupings of three sensors.
        @return none, only writes to class variable, groupingAvg
                [ 123    234    345    456    567    678    781    812]
                where each of these is the average of the sensors
    '''
    def getGroupingAvg(self):

        self.groupingAvg = [0, 0, 0, 0, 0, 0, 0, 0]

        for startIndex in range(0, 8):
            sum = 0

            for i in range(startIndex, startIndex + 3):

                if ( i > 7 ):
                    i %= 8                              # if i exceeds the range of the data, do the modulus operator. This allows for groupings 781 and 812 to work.
                sum += self.avg[i]
                i += 1

            self.groupingAvg[startIndex] = sum / 3.0


    '''
        Function to compute the three highest sensor groups.
        @return maxGrouping, [1 x 3] integer list of the index of the top three sensor groups in the calibration data
        @example maxGrouping = myCalibrationObject.getMaxGrouping(gestureCalibrationData) --> maxGrouping: [ 6, 5, 0]
```

```
      '''
      def getMaxGrouping(self,calData):

            maxGrouping = [0, 0, 0]
            self.setData(calData)
            self.getAvg()
            self.getGroupingAvg()

            array = np.array(self.groupingAvg)                          # REFERENCE: https://stackoverflow.com/questions/5284646/rank-items-in-an-array-using-python-numpy
            temp = array.argsort()
            ranks = np.empty_like(temp)
            ranks[temp] = np.arange(len(array))

            maxGrouping[0] = int( np.where(ranks == 7)[0] )
            maxGrouping[1] = int( np.where(ranks == 6)[0] )
            maxGrouping[2] = int( np.where(ranks == 5)[0] )

            return maxGrouping

'''
      Used for Testing/Debugging Purposes
'''
if __name__ == '__main__':

      t1 = []
      #t2 = [0, 1, 2, 3, 4, 5, 6, 7]
      t2 = [100.0, 100.0, 7.0, 7.0, 0.0, 0.0, 0.0, 100.0]

      t1.append(t2)
      t1.append(t2)
      t1.append(t2)
      cal = Calibrate()
      maxGrouping = cal.getMaxGrouping(t1)


      print(maxGrouping)
```

C. ringBuffer.py

```
'''
AUTHOR: Aditya Patel and Jim Ramsay
DATE CREATED: 04/01/2018
LAST MODIFIED:  2018-04-09
PLATFORM: Raspberry Pi 3B, Raspbian Stretch Released 2017-11-29
PROJECT: EMG Human Machine Interface
ORGANIZATION: Bradley University, School of Electrical and Computer Engineering
FILENAME: ringBuffer.py
DESCRIPTION:
      Class that implements a ring/circular buffer to hold the emg data. It stores data until full, then
      overwrites the oldest element every time. It also has a method to take an average of the last n
      data points.

KNOWN FLAW:
```

```
      Instead of only ignoring the oldest element when computing the average, I flush the entire buffer.
      Then the full n-length sum is taken. This is grossly inefficient, but was not found to be a bottleneck
      in implementation. Thus, it was ignored.

EDIT HISTORY:
      20180409 -- Added functions to compute groupingAvg and maxGrouping. The groupings are as follows:

                              [ 123    234    345    456    567    678    781    812 ]

                        In main, this allowed us to calculate the three groupings with the highest average sensor value.
'''
import numpy as np

class ringBuffer:
      def __init__(self,size_max):                                                                                # Constructor
            self.max = size_max
            self.data = []
            self.sums = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
            self.avg = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])                             # this has to be a numpy array to use the 'divide' function below
            self.full = False
            self.groupingAvg = []
      class __Full:                                                                                                           # sub-class that implements a full buffer

            def append(self, x):
                  self.data[self.cur] = x                                                                               # append an element, overwriting the oldest one
                  self.cur = (self.cur + 1) % self.max                                                            # cycle 'cur' from 0 to self.max

            def getAvg(self):
                  self.sums = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]                                 # reset sums to prevent it from accumulating forever. This is NOT elegant or
efficient
                  for r in range(0,self.max):
                        for c in range(0,8):
                              self.sums[c] = self.sums[c] + self.data[r][c]

                  np.divide(self.sums, float(self.max), out = self.avg)                               # compute the average by dividing the sums by the size of the data array
                  return self.avg

            def get(self):                                                                                                    # return list of elements from oldest to newest
                  return self.data[self.cur:] + self.data[:self.cur]

            '''
                  Function to compute the average value of each of the eight groupings of three sensors.
                  @return none, only writes to class variable, groupingAvg
                              [ 123    234    345    456    567    678    781    812]
                              where each of these is the average of the sensors
            '''
            def getGroupingAvg(self):

                  self.groupingAvg = [0, 0, 0, 0, 0, 0, 0, 0]

                  for startIndex in range(0, 8):
                        sum = 0

                        for i in range(startIndex, startIndex + 3):

                              if ( i > 7 ):
```

```
                                    i %= 8                                                                                          # if i exceeds the range of the data, do the 
modulus operator. This allows for groupings 781 and 812 to work.
                              sum += self.avg[i]
                              i += 1

                        self.groupingAvg[startIndex] = sum / 3.0

            '''
                  Function to compute the three highest sensor groups.
                  @return maxGrouping, [1 x 3] integer list of the index of the top three sensor groups in the ring buffer
                  @example maxGrouping = myRingBuffer.getMaxGrouping() --> maxGrouping: [ 6, 5, 0]
            '''
            def getMaxGrouping(self):

                  maxGrouping = [0, 0, 0]

                  self.getAvg()
                  self.getGroupingAvg()

                  array = np.array(self.groupingAvg)                                                         # REFERENCE: https://stackoverflow.com/questions/5284646/rank-
items-in-an-array-using-python-numpy
                  temp = array.argsort()
                  ranks = np.empty_like(temp)
                  ranks[temp] = np.arange(len(array))

                  maxGrouping[0] = int( np.where(ranks == 7)[0] )
                  maxGrouping[1] = int( np.where(ranks == 6)[0] )
                  maxGrouping[2] = int( np.where(ranks == 5)[0] )

                  return maxGrouping


      def append(self,x):                                                                                         # append an element to the end of the buffer until it 
is full
            self.data.append(x)

            if len(self.data) == self.max:
                  self.cur = 0
                  self.full = True
                  self.__class__ = self.__Full                                                               # Permanently change class from not full to full

      def get(self):                                                                                              # return list of elements from oldest to newest
            return self.data


'''
      Used for Testing/Debugging Purposes
'''
if __name__=='__main__':
      x = ringBuffer(3)
      print "average: ", x.avg
      print "sums: ", x.sums
      emg1 = [1,1,1,1,1,1,1,1]
      emg2 = [2,2,2,2,2,2,2,2]
      emg3 = [3,3,3,3,3,3,3,3]
      x.append(emg1); x.append(emg2); x.append(emg3);
      x.append(emg3); x.append(emg3);x.append(emg3);
```

```
      average = x.getAvg()

      print "Average: ", average

```

D. common.py

```
import struct

def pack(fmt, *args):
    return struct.pack('<' + fmt, *args)

def unpack(fmt, *args):
    return struct.unpack('<' + fmt, *args)

def text(scr, font, txt, pos, clr=(255,255,255)):
    scr.blit(font.render(txt, True, clr), pos)

```

E. displayControl.py

```
'''
AUTHOR: Aditya Patel and Jim Ramsay
DATE CREATED: 2018-03-31
LAST MODIFIED:
PLATFORM: Raspberry Pi 3B, Raspbian Stretch Released 2017-11-29
PROJECT: EMG Human Machine Interface
ORGANIZATION: Bradley University, School of Electrical and Computer Engineering
FILENAME: videoControl.py
DESCRIPTION:
      Script to control the display of the two different camera feeds.
NOTE:
      There is no logic needed in the switchDisplay() function, as this is a binary system. The main function will contain the logic.

REFERENCES:
      [1] https://stackoverflow.com/questions/279561/what-is-the-python-equivalent-of-static-variables-inside-a-function

'''

import os                                                                                 # used to execute shell commands
from time import sleep

def init():

      os.environ['DISPLAY'] = ":0"                                            # allows us to launch GUI applications and control the mouse. Limited to the scope of the call of
this function.
      os.system("killall firefox-esr")
      os.system("nohup firefox http://169.254.184.5/html/ &")           # 'nohup' -- ignore HANGUP signals generated by firefox (there are a ton)
      sleep(15)
      os.system("nohup firefox http://169.254.13.230/html/ &")
      sleep(2)
      os.system("xdotool key ctrl+Tab")                                       # Cycle tab to the "Restore session" tab that always comes up
      os.system("xdotool key ctrl+w")                                               # Close the tab
      sleep(3)
      os.system("xdotool key F11")                                            # launch browser in full screen
      sleep(3)
      os.system("xdotool mousemove 600 200 &")                          # enlarge camera 1
```

```
      sleep(0.1)
      os.system("xdotool click 1")
      sleep(0.1)
      os.system("xdotool key ctrl+Tab")
      sleep(3)
      os.system("xdotool click 1")                                            # enlarge camera 2
      os.system("xdotool key ctrl+Tab")
      os.system("xdotool mousemove 0 500")                                    # Move mouse out of the way
      print("DISPLAY CONFIGURED")

def switchDisplay():

      switchDisplay.display ^= 1                                                    # toggle variable between 0 and 1
      os.system("xdotool key ctrl+Tab")

      return(switchDisplay.display)

switchDisplay.display = 0                                                           # initialize static variables

# Initialize
if __name__=='__main__':
      init()

      # sleep(1)
      # switchDisplay()
      # print("Attempting to change display")
      # sleep(5)
      # print("Attempting to change display")
      # switchDisplay()
```

F. moveMotor.py

```
'''
AUTHOR: Aditya Patel and Jim Ramsay
DATE CREATED: 2018-04-05
LAST MODIFIED: 2018-04-07
PLATFORM: Raspberry Pi 3B, Raspbian Stretch Released 2017-11-29
PROJECT: EMG Human Machine Interface
ORGANIZATION: Bradley University, School of Electrical and Computer Engineering
FILENAME: moveMotor.py
DESCRIPTION:
      Script to move the connected servo motor to a specific duty cycle.

NOTE:

      CW  --> INCREASE PWM
      CCW --> DECREASE PWM

      20% --> +90 deg
      80% --> -90 deg

USAGE:

      ssh emgPi_1 'python /home/pi/scripts/moveMotor.py PWMdir isLastTime'
```

```

      Command Line Arguments:

            PWMdir            integer      0 --> CW         1 --> CCW   others --> not recognized, do nothing
            isLastTime  boolean     True --> execute IO cleanup         False --> do not cleanup, run normally

'''

import RPi.GPIO as IO
import sys
import time

global initialDuty
initialDuty = 50

class Motor:

      global initialDuty
      def __init__(self, PWMdirection, startPWM):

            IO.setwarnings(False)
            IO.setmode(IO.BOARD)

            IO.setup(12, IO.OUT)                                              # set GPIO pins to Output mode
            self.p = IO.PWM(12,350)                                           # set pin 12 to 350Hz pwm output
            self.PWMdir = PWMdirection
            self.duty = startPWM
            self.p.start(startPWM)

      def ccw(self):
            if (self.duty >= 20) and (self.duty <= 70) :
                  self.duty += 10
                  # print("ccw")
                  # print(self.duty)
                  self.p.ChangeDutyCycle(self.duty)
                  time.sleep(0.01)

      def cw(self):
            if ( self.duty >= 30 ) and ( self.duty >= 80 ) :
                  self.duty -= 10.0
                  # print("cw")
                  # print(self.duty)
                  self.p.ChangeDutyCycle(self.duty)
                  time.sleep(0.01)

      def cleanup(self):
            self.p.stop()
            IO.cleanup()

if __name__ == '__main__':

      # moduleName = sys.argv[0]
      startPWM = float(sys.argv[1])
      PWMdir = int(sys.argv[2])
      isLastTime = int(sys.argv[3])

      # print("PWM Direction: ", type(PWMdir))
```

```

    mtr = Motor(PWMdir, startPWM)
    time.sleep(1)
    # if (PWMdir == 0):
          # mtr.cw()
          # time.sleep(1)
    # elif (PWMdir == 1):
          # mtr.ccw()
          # time.sleep(1)
    # if (isLastTime):
          # mtr.cleanup()


    # t_end = time.time() + 10

    # if (isLastTime == 1):
                # mtr.cleanup()

    # while (time.time() < t_end):
          # if (PWMdir == 0):
                # mtr.cw()
          # elif (PWMdir == 1):
                # mtr.ccw()


```