2013

Mouse HCI Through Combined EMG and IMU

Timothy Forbes
University of Rhode Island, tcf686@gmail.com

MOUSE HCI THROUGH COMBINED EMG AND IMU

BY

TIMOTHY FORBES

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

2013

MASTER OF SCIENCE THESIS

OF

TIMOTHY FORBES

APPROVED:

Thesis Committee:

Major Professor Qing Yang

He Huang

Haibo He

Lutz Hamel

Nasser H. Zawia
DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
2013

ABSTRACT

Since the invention of the computer mouse interface there have been no major changes in its design. The way we interact with computers through keyboards and mice are still the same as they first were. Recent new technologies try to replace or emulate the mouse and each have their own disadvantages. Some require the use of surfaces, cameras, sensor bars, or tethering. Other technologies are gesture based, require training, and are not intuitive. Not only is there the need for a mouse interface that is intuitive and convenient, but advancements in the field of human computer interfacing, especially using electromyogram (EMG) sensors, further open doors. The objective of this project is to design and create a wearable computer mouse interface, independent of cameras, sensor bars, surfaces, or tethering, that is intuitive, fast, accurate and convenient. This is accomplished by combining the output of both EMG sensors and an inertial measurement unit (IMU) to control the cursor and clicking functions of a mouse. The goal is to be intuitive and accurate in order to overcome the shortcomings in existing technologies.

Many different muscles were examined as potential candidates for the EMG sensors. The output of each sensor was observed and several different sensor setups were chosen for focus. For each setup, at least one classifier configuration was designed in the attempt to obtain the highest testing accuracy. The angular velocity measurements of the z and y axis in the IMU were chosen to physically control the mouse cursor. Several different techniques were designed based on the different sensor setups and classifier configurations to obtain the highest overall system accuracy and fewest false positives.

Training data was collected in MatLab for processing. Time domain features were extracted and provided as input to a linear discriminant classifier. The implementation of the designs were programed in C++ for fast and efficient real-time performance. The setup and configuration with the highest training accuracy, highest testing accuracy, lowest false positive rate, and best overall IMU integration, was the setup with EMG sensors on the muscles: Extensor Carpi Radialis, Extensor Carpi Ulnaris, Flexor Carpi Ulnaris, Flexor Carpi Radialis, and Flexor Digitorum Superficialis (Near Wrist). One classifier took the input of all five channels for classification into a total of seven classes or actions: Rest, Up, Down, Left, Right, Left Click, and Right Click. This design took the least time to complete the accuracy program designed to evaluate the different configurations. This design also saw the lowest false positives.

Compared to a standard mouse or track pad, the best design still fell far short in performance. However, the common user has been using a standard mouse or track pad their whole life, and the need for time to practice with the device is expected. With practice, the design has the potential to match or come close to the accuracy of existing mice. With further automation of the training data collection and classifier training, the device would require little configuration to start using. Further development towards a commercial product would minimize the equipment required, the sensor setup time, and make the device no less convenient than a standard mouse.

One of the design goals at the forefront of this project, is the application and usability by hand amputees. Muscle locations are located in the forearm, and the IMU

device located on the back of the wrist or forearm, allows hand amputees and those with hand disabilities to operate the device.

ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Qing Yang, for the opportunity to complete this research. His advice, support and guidance helped me develop and carry out my research. Through his research grant was I was provided funding for graduate school, helping me to attend graduate school and allowing me to focus full time on my research. All of his advice and support have been greatly appreciated. I would like to thank my committee members as well, Dr. He Huang, Dr. Haibo He and Dr. Lutz Hamel, for providing advice when needed and reviewing my research. I would also like to thank Dr. Comerford for chairing my defense.

I would like to again thank Dr. He Huang and Dr. Haibo He for their guidance, advice, and motivation, through weekly meetings and other means, in completing my research. Numerous other individuals also provided assistance in my research. Everyone from the Laboratory for Integrating Neuron and Cyber: Dr. Ming Liu, Fan Zhang, Xiaorong Zhang, Lin Du, Ding Wang, and Robert Hernandez, assisted me in using lab equipment, code development, and feedback towards my design. I would also like to thank Dr. Frederick Vetter, the Electrical Engineering Graduate director, for his guidance and help in filling out all the required forms correctly.

None of this would have been possible without the continued support and encouragement of my friends and family. Notably, my parents, for their level of support and my roommate, Tom, for his patience and willingness to listen to me rant and complain along the way.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Since the invention of the computer mouse interface there have been no major changes in its design. The way we interact with computers through keyboards and mice are still the same as they first were. Recent new technologies try to replace or emulate the mouse and each have their own disadvantages. Some require the use of surfaces, cameras, sensor bars, or tethering. Some are also gesture based and therefore are not intuitive and require training. Not only is there the need for a mouse interface that is intuitive and convenient, but advancements in the field of human computer interfacing, especially using electromyogram (EMG) sensors, further opens doors for new advancements.

Advancements have been made in electromyography that allows readings to be taken from electrodes on the surface of the skin. Surface EMG sensors are easy and fast to apply to the skin. This technique is also much safer, allowing everyday users to measure muscle activity. These devices make human computer interface much more practical. However, not all muscles are located adjacent to the skin. Some are located underneath other muscles or bone, limiting the number of muscles available. Specifically, an EMG sensor measures the electrical activity in a muscle. The voltage measured is on the order of millivolts. Before processing on a computer, the data must first be converted from analog to digital. Each A/D device has an input voltage

requirement, usually around -5 to 5 volts. Thus, an amplifier is used to scale up the signals from the EMG sensors to meet this requirement.

With the popularity of both the personal computer and the mouse as the primary interface, several areas of the population are put at a disadvantage. The elderly and the disabled are often limited in their ability to use a mouse, and thus a computer. The new advancements of surface EMG sensors allow for the measurement of wrist and finger actions through muscles in the forearm. This gives hand amputees the potential to control a mouse through such an interface.

An inertial measurement unit (IMU) measures acceleration, angular velocity, and gravitational forces on all three axis by the use of accelerometers, gyroscopes and magnetometers. They are inexpensive and accurate in measuring motion, making them very practical. IMU's are very common in many technologies including aircraft, spacecraft, boats, satellites, and missiles. Because of their size, they can easily be included in embedded systems and small devices. They often include built in software to compensate for erroneous constant measured motion, called drift.

Many other technologies have been developed in the attempt to replace or emulate the functionality of a mouse. A paper from 2001 details a design based solely on an inertial measurement unit, where sufficiently small acceleration is considered cursor movement and larger movements are mapped to functions such as clicking (Lee). Another design is to detect and track user eye movement patterns and control mouse cursor movement (Miyoshi). The field of human control interface using electromyography is relatively new. Advancements in the field have improved the accuracy to the point where computer interfaces become practical, and feasible. EMG

signals have also been used in a variety of applications, including an EMG-based power assisted wheelchair (Oonishi), and neural controlled artificial arms (Englehart).

In order to classify actions within the EMG sensor data, pattern recognition techniques are required. Pattern recognition is the classification of input data to existing labels. In the case of this design, the labels are the separate wrist movements and finger clicking movements. Common methods for classifying EMG data are by time-domain analysis, where a set length of time of EMG data is analyzed for a pattern, and future data compared to those patterns. Specifically, common time-domain features are thresholds, zero crossings (crossing over from negative to positive or vice-versa), and slope sign changes. Common classifiers for EMG analysis are linear discriminant analysis, neural networks, fuzzy logic, and support vector machines (Al-Timemy; Khezri; Khushaba; et al). Each classifier must extract these features from the appropriate and relevant EMG sensors.

Design challenges focused around a computer mouse interface include accuracy, intuitiveness and convenience. A lack of any of these three makes the device impractical and inferior to the existent standard mouse design. Many devices are simply not accurate enough. The slightest inaccuracy becomes a major inconvenience to the user, considering a device as widely and commonly used as a mouse. If the proposed replacement to a mouse is not accurate, then a user may spend hours practicing with the device in order to achieve a level of accuracy that will still never match a conventional mouse. Lastly, convenience is a big factor, as a user will not want to use the device if it requires an hour setting up. In this paper, a novel human

computer interface is proposed by combining the output of both EMG sensors and an IMU device.

OBJECTIVE, HYPOTHESIS AND SCOPE

The objective of this thesis is to design and create a wearable computer mouse interface, independent of cameras, sensor bars, surfaces, or tethering, that is intuitive, fast, accurate and convenient. Combining the output of both EMG sensors and an IMU to control the cursor and clicking functions of a mouse should be intuitive and accurate, overcoming shortcomings in existing technologies, and also allowing hand amputees to operate the interface. Hand amputees are disadvantaged for both traditional mouse and typing actions. This project specifically focuses on the mouse interface.

Most existing technologies and designs listed in the review of literature section below are gesture based. One of the main goals of this design is to be intuitive, which is realized in the design of this project as a non-gesture based interface. Detecting normal mouse functions through EMG sensors is difficult. It is hard to distinguish a clicking action between each finger, and even wrist movements. This leads to most interfaces being gesture based, where gestures are chosen that are easily distinguishable from the rest. To overcome the difficulty in distinguishing these actions, the output from both the IMU and EMG sensors will be combined. How this is implemented will determine the balance between accuracy and intuitiveness.

The scope of this thesis is limited to a proof of concept. If the design meets all the desired objectives then the next step may be to develop a product. The realization of a product involves measures beyond the scope of this thesis, including but not limited to: acquiring greater sources of funding, and design and fabrication of hardware based on the prototype. Further testing and development would be required to translate the prototype to a product. The potential of a product was taken into account in the design of the prototype. EMG muscle locations were chosen for their ability to be incorporated in an arm band style sensor array, to allow for a convenient and fast setup process.

CHAPTER 2

REVIEW OF LITERATURE

The principal literature serves as both a timeline of computer mouse interfaces as well as the current research in human computer electromyography control. The history of mouse interfaces depicts the advantages and disadvantages of each device. The research in EMG measurement, control and interface, details existing classifying techniques, muscle target locations, accuracy and error rates, and progress in the field. The field of electromyography in human computer interfaces is a relatively new field, limiting both the research and locations available for research. The literature search was conducted in three primary areas: United States Patent and Trademark Office (USPTO) Patent and Patent applications, Institute of Electrical and Electronics Engineers (IEEE) Xplore articles and conference publications, and a general web search.

Patent and Patent Applications in the USPTO database revealed limited results for computer mouse interfaces, none of which involved EMG signals. Some of the patents detailed glove devices that used push buttons or pressure plates for clicking, accelerometers, track balls or lasers for controlling the cursor, and other buttons for either turning on and off the device or for gesture based controls. One details a glove with a mouse button on the fingertip and movement sensors/laser in the palm of the glove. It requires your hand to be on a table to work and transmits wirelessly over Bluetooth or infrared (Cheng). Another patent is of a glove with pressure plates on two

fingers for left and right mouse clicks. Scrolling buttons are located on the side of the index finger for use by the thumb. There is a side switch to turn off the device for using your hand for other things and a tracking device on the tip of one of the fingers (Bajramovic).

A general web search led to several more devices with the same variety of characteristics as the patent search. One device, called the G-speak, is a spatial operating environment. It is a user interface and networked system that allows users to control a variety of things, including controlling data and objects in 3D, based on gestures (Oblong Industries). It operates by use of several cameras that map your hands in 3D. The system is the glove interface from the movie Minority Report, created due to its popularity in the movie. The Mister gloves, a wireless USB gesture input system developed by two students from Cornell, is a gesture-based glove designed to replace the mouse and also include several other hotkeys or shortcuts. It uses accelerometers and gyroscopes to map gestures to mouse movements or hotkeys. The glove is wired to a base station that communicates wirelessly with the computer (Chen). Another device, called Leap Motion, is another gesture based device that replaces the mouse. It works by a motion sensor placed on the table underneath your hand, which maps your hand and fingers in 3D (Leap Motion).

Being a new area of research, myoelectric control and pattern recognition articles and papers are frequently published under IEEE Xplore. The first publication I found concerning mouse control and EMG signals is the 2001 publication “Mouse cursor control system using EMG”. Both cursor control and clicking were implemented using the neural network classifier for an accuracy of 70 percent. The

next publication, “A new means of HCI: EMG-MOUSE” from 2005, used a fuzzy min-max neural network as the classifier with an average recognition rate of 97 percent.

Several papers dealt solely with the classification of wrist and finger movements and not controlling a mouse. One paper used neural networks to identify a variety of movements including thumb flexion and extension. Their best recognition rate for several predefined gestures was 95 percent (Zhao). Another publication uses their own fuzzy linear discriminant analysis algorithm to classify several wrist, hand and forearm gestures, including hand open, hand close, wrist flexion and wrist extension. Their best recognition accuracy was 94 percent (Khushaba). “Frequency domain surface EMG sensor fusion for estimating finger forces” details how the use of an array of EMG sensors is better in detecting finger movements. They join the numerous inputs into a neural network classifier and show how it can increase the accuracy. The paper by Al-Timemy observes the effects of finger movement classification based on the force or effort exerted. Using linear discriminant analysis, they observed an accuracy of 84 percent through unconstrained force and 91 percent for constrained force. The 2011 paper, “Hand motion detection from EMG signals by using ANN based classifier for human computer interaction”, aims to classify the wrist movements up, down, left and right. A neural network classifier is used and produced a classification accuracy of 88 percent. Another 2011 paper by Khezri, tested a number of different techniques in tandem to produce a neuro-fuzzy classifier. He combined both time domain and frequency domain features to produce an average classification accuracy of 92 percent. Another paper used a variation of fuzzy linear

discriminant analysis to classify ten different hand and finger gestures. An accuracy of 91 percent was achieved through the use of only two electrodes (Khushaba).

The most recent paper found concerning EMG control for a mouse interface is the 2011 publication “A novel HCI based on EMG and IMU”. This paper is most similar to my proposed design. However, the design proposed in the paper uses the IMU solely for the cursor control and EMG solely for the clicking functions. The two technologies work independently of each other. The classifier used in this paper is also linear discriminant analysis. To achieve the mouse clicking functions, several finger and hand gestures were used. The accuracy reported for the clicking functions is 88 percent.

A brand new technology just announced and still in development, is a technology called Myo. It is an armband gesture based control interface for computers and potentially mobile devices. The device is currently available by presale and is expected to ship at the end of 2013. When released, the device will have an API that developers can use to create control for their own custom applications. Few details are available on the device through the FAQ section on its website. The device contains several EMG sensors and accelerometers. The output is combined in a method not specified to give the user accurate control. The device is said to have an on/off switch to prevent false positives and claims to have the ability to distinguish between each finger.

ANALYSIS OF LITERATURE

An important observation to note is how the accuracy is measured throughout the various publications. They report classification or recognition accuracy, which is the fact that the user is performing the action and not the amount or magnitude of that action. Because of this, many accuracies reported are deceptive when stated relative to the goal of a mouse interface. For example, recognizing the wrist up movement does not indicate how much the mouse cursor should move up. The use of an IMU, as realized in the last paper cited above, accomplishes this. However, that paper accomplishes both IMU cursor movement and clicking functions through the use of gestures. The movements performed are not the same as they would be using a conventional mouse. This makes the interface not intuitive and requires much learning and practice. A trend throughout the papers is the selection of the pattern recognition classifier. All papers use some version of linear discriminant analysis or neural network. Another useful fact to be noted, is that the use of a constraint improves recognition accuracy. Therefore, some sort of constraint may be used to improve the design proposed in this project.

Of all published work and products created, the design or device that is most similar to that proposed and tested in this project is the Myo device. This device was announced mere weeks before the writing of this thesis, and is not expected to fulfil preorders until the end of this year, 2013. While the technology is similar, the purpose of the device is still different from what is examined here. The application of the Myo appears to be app based. The promotional video shows a user waving their hand and a

music player switching to the next song, or a video game where the user's hand assumes the position of a gun and performs gestures to simulate firing and reloading. The design of the device seems to include the functionality necessary to act as a mouse interface, although, that is not its purpose and is never mentioned. As the release of the device is still far away, the description is vague. Furthermore, the intuitiveness is difficult to determine through the promotional video, and due to the fact that it is gesture-based. With all of that in mind, there remains a place for the design proposed in this project, while remaining unique.

CHAPTER 3

MATERIALS AND METHODS

There were many steps required to both design and test the system. To determine if the idea of a combined EMG and IMU mouse interface was feasible, both EMG and IMU data needed to be collected and analyzed. Based on the output, further designs could be developed and tested. The first was to determine the relevant and optimal data to use from the IMU. Secondly, the wrist and finger actions needed to be determined in order to determine the appropriate EMG muscle locations. For the EMG data, a feature extraction and pattern recognition algorithm must be chosen to classify the data. Once this is done, the method of combining the two outputs can be determined. Lastly, a method is required to determine the overall accuracy and intuitiveness of the design.

An inertial measurement unit measures acceleration, angular velocity and gravitational forces on all three axis, by the use of accelerometers, gyroscopes and magnetometers. The wireless IMU system in the lab is the Xsens MTx. It is made by Xsens and contains an API to read the data from within MatLab and C++. It is advertised to not have any drift, which is defined as a small constant movement measured when the device is stationary. The IMU is tethered to a battery powered pack that wirelessly communicates to a small antenna, connected to the computer via USB.

In the design, the IMU will be used to control the mouse cursor. Upon viewing the output from all three of the accelerometers, gyroscopes and magnetometers, the angular velocity output from the IMU matches physical wrist movements the best. The acceleration and magnetometers provide useful data but are not pertinent to cursor movements. The orientation of the IMU on the back of the wrist determines which axis are relevant. With the IMU on the back of the wrist and the arm flat, the IMU is oriented so that the z-axis runs vertical, the y-axis horizontal and perpendicular to the arm, and the x-axis parallel to the arm. Figure 3.1 visually depicts this. This results in z-axis gyroscope data translating to the mouse y-coordinate, and y-axis gyroscope translating to the mouse x-coordinate. X-axis data can be ignored. Figure 3.2 depicts this visually. The gyroscope data are measured as angular velocity. Both angular velocity and linear velocity affect the sensors in the same way, making the prospect of an IMU as part of a mouse cursor control system more versatile.

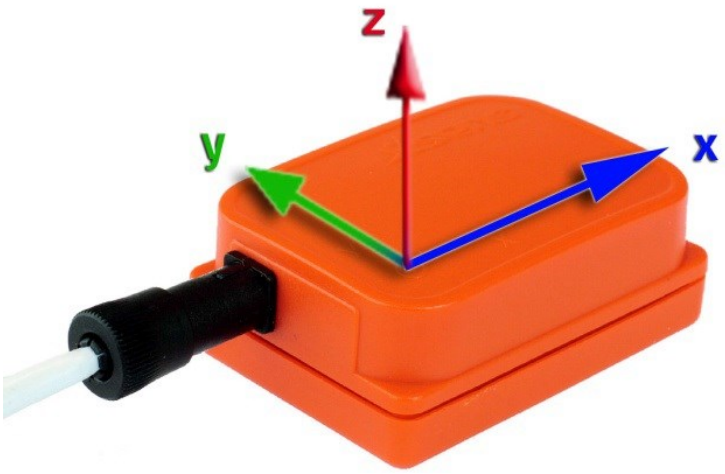

Figure 3.1. Xsens MTx IMU device with overlaid co-ordinate system. Source: Xsens MTi and MTx User Manual and Technical Documentation, page 9.

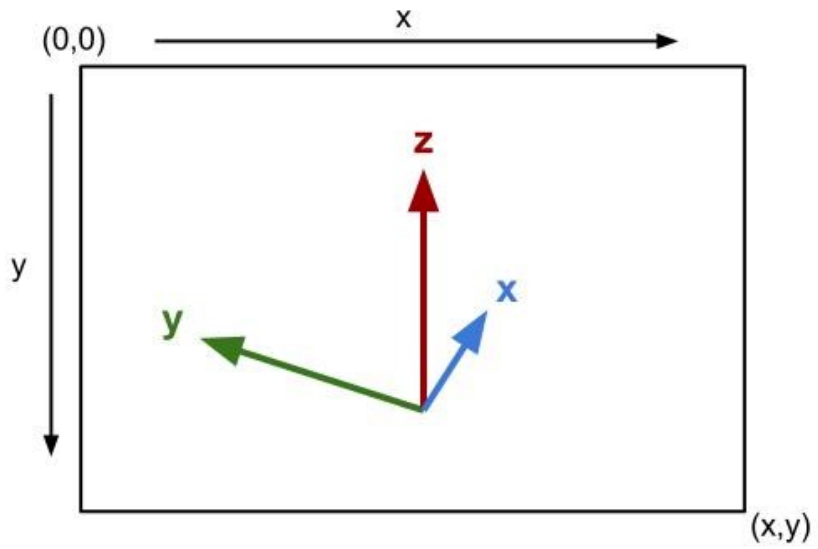

Figure 3.2. Xsens MTx co-ordinate system versus a computer screen co-ordinate system.

There are two EMG systems available in the lab, a wired and a wireless system. The EMG system most practical is the Trigno Wireless EMG system by Delsys. It consists of 16 wireless EMG sensors and a base receiver station. Each EMG sensor contains its own amplifier and also contains a three axis accelerometer. However, it has been previously determined that the gyroscopes from the Xsens IMU device provides the most accurate and relevant data. To read the EMG data, an analog to digital converter (A/D) is required. The A/D selected is the Measurement Computing USB-1616HS-BNC. It has 16 analog inputs and is capable of a one millisecond sampling rate. It also has both a MatLab and a C/C++ API. Both the Delsys EMG system and the Measurement Computing A/D come with software to view the EMG sensor output in real time. This is useful for identifying and verifying both muscle locations and verifying that the EMG sensors are working correctly. Through reading both IEEE articles and the book *Anatomical Guide For The Electromyographer: The Limbs And Trunk*, many different muscle locations were

chosen to be examined. Each targeted muscle must be detectable by the surface EMG sensors. Ideal locations are surface muscles with little to no crosstalk. Crosstalk is muscle activity in neighboring muscles that are detected through the EMG sensor on the targeted muscle. Each targeted muscle needs to have a low number of arm, wrist, and finger movements associated with it to prevent false classifications. Table 1 shows the muscles associated with the wrist to be examined. Table 2 shows the muscles associated with finger movements.

Table 3.1. List of muscles examined for the purpose of wrist movements.

Code	Muscle Name	Test Maneuver[1]	Test Maneuver for Right Wrist[2]
W1	Extensor Carpi Radialis	Dorsiflexion of wrist in radial deviation	Up and Left
W2	Flexor Carpi Ulnaris	Flexion of the wrist with ulnar deviation	Down and Right
W3	Flexor Carpi Radialis	Flexion of the wrist with radial deviation	Down and Left
W4	Extensor Carpi Ulnaris	Extension of the wrist with ulnar deviation	Up and Right

[1] information from *Anatomical Guide For The Electromyographer: The Limbs And Trunk*
[2] observed through testing

Table 3.2. List of muscles examined for the purpose of finger movements.

Code	Muscle Name	Test Maneuver[1]	Test Maneuver for Fingers[2]
F1	Flexor Digitorum Superficialis	Flex the proximal interphalangeal joint while the others are in hyper extension	Bend the middle knuckle of any finger
F2	Extensor Indicis Proprius	Extend finger with flexion of other fingers	Extend any finger
F3	Extensor Digitorum Communis	Extend metacarpophalangeal joints	Extend base knuckle of any finger
F4	Flexor Pollicis Longus	Flexion of the distal phalanx of thumb	Bend tip of thumb
F5	Abductor Pollicis Longus	Radial abduction of the thumb	Extend thumb outward
F6	Palmaris Longus	To cap the palm of the hand	Make a fist
F7	Flexor Digitorum Profundus	Flexion of the distal phalanges of digits	Bend tip of any finger

[1] information from *Anatomical Guide For The Electromyographer: The Limbs And Trunk*
[2] observed through testing

All observations for the four locations listed in Table 1 matched the given test maneuver and do so with little crosstalk. The ideal EMG sensor locations on all four muscles are also conveniently located a third the way down the arm from the elbow. The test maneuvers also correspond directly to the desired actions of the mouse cursor. The muscle locations and accuracy observed for the fingers in Table 2 were not as conclusive as those of the wrist muscles. The muscle location and accuracy determines what movement or gesture will be considered a mouse click. To be intuitive, the click gesture should resemble the click motion on a convention mouse. That eliminates the use of any muscle used for finger extension or thumb movement. Another reason to eliminate thumb muscles is due to the small size and difficult sensor placement for those muscles. Bending just the tip of a finger is not very feasible and making a fist is not very specific, making the Palmaris Longus and Flexor Digitorum Profundus muscles not practical. The Palmaris Longus is also too small of a muscle to accurately

read. The Flexor Digitorum Superficialis is the most relevant and optimal muscle to use. However, significant crosstalk was observed from the Flexor Carpi Radialis, the muscle associated with the down and left action by a right hand. To compensate for this, the location of the sensor can be moved up towards the wrist or down and to the outside of the arm near the elbow (Technical Note 101: EMG Sensor Placement 5).

There are five possible EMG sensor locations to use in our pattern recognition classifier. The five muscles are: Extensor Carpi Radialis, Extensor Carpi Ulnaris, Flexor Carpi Ulnaris, Flexor Carpi Radialis and Flexor Digitorum Superficialis. In order to know which action the user is performing, the output from either all or a select few of the sensors need to be measured and compared against predefined actions to form a prediction. There are numerous ways to combine these sensors and numerous predefined actions we can record. Each combination will have a different degree of accuracy. Before we can form the classifier, the data from the EMG sensors need to be quantified. This is called feature extraction and identifies properties or attributes in the data.

FEATURE EXTRACTION AND LINEAR DISCRIMINAT ANALYSIS

From the review of literature, it is shown that a variety of both feature extraction and pattern recognition classifiers are used to classify electromyography data. Common classifiers for EMG analysis are linear discriminant analysis, neural networks, fuzzy logic, and support vector machines. Feature extraction can take place in the time domain, feature domain or time-frequency domain. Requirements for the

chosen classifier are that of a computationally efficient classifier suitable for a real-time application such as a mouse interface. The design should not contain delay while being efficient enough to possibly be implemented in an embedded system. The simple linear discriminant analysis (LDA) classifier was used in this design due to its high classifier accuracy and computational efficiency compared to the other algorithms. Both the feature extraction and the LDA classifier used in the design are based on the algorithm proposed by Englehart and Hudgins in "A Robust, Real-Time Control Scheme for Multifunction Myoelectric Control," and equations detailed in "On Design and Implementation of Neural-Machine Interface for Artificial Legs."

The feature extraction consists of four time-domain features that were chosen for their low complexity. The four are: the number of zero crossings, the waveform length, the number of slope sign changes and the mean absolute value. Further descriptions and equations for extracting those features are given in Hudgin's paper, "A New Strategy for Multifunction Myoelectric Control." The time-domain features are extracted from every window length in the continuous stream of EMG signals. The size of the window length is chosen based on the events trying to be observed and over how long of a period they occur.

The purpose of discriminant analysis is to classify observed data to a defined class or label in which the posteriori probability is the greatest. The conditional probability is the probability of class *Cg* given the observed feature vector $\bar{f}$ is:

$$P(C_g|\bar{f}) = \frac{P(\bar{f}|C_g)P(C_g)}{P(\bar{f})}$$

given:

$$P(C_g) \neq 0$$

$$P(\bar{f}) \neq 0$$

where: G = number of classes
C_g = set of classes, where $g \in [1, G]$
$\bar{f}$ = feature vector of the given window of samples
$P(C_g)$ = priori probability
$P(\bar{f}|C_g)$ = likelihood (conditional probability)
$P(\bar{f})$ = probability of the observed feature vector $\bar{f}$

The user is capable of performing any action at any time, so we assume our priori probability to be the same for all classes. Thus, every class's covariance is equivalent, and the maximization of the posteriori possibility is:

$$C_g = \arg max_{C_g} \left\{ \bar{f}^T \Sigma^{-1} \mu_g - \frac{1}{2} \mu_g^T \Sigma^{-1} \mu_g \right\}$$

The linear discriminant function is defined as:

$$dC_g = \bar{f}^T \Sigma^{-1} \mu_g - \frac{1}{2} \mu_g^T \Sigma^{-1} \mu_g$$

where: μ_g = mean vector
Σ = common covariance matrix

Both the mean vector and common covariance matrix are calculated from a large amount of training data.

$$\mu_g = \frac{1}{K_g} \sum_{k=1}^{K_g} \bar{f}_{C_g,k}$$

$$\widetilde{\Sigma} = \frac{1}{G} \sum_{g=1}^{G} \frac{1}{K_g - 1} (F_g - M_g)(F_g - M_g)^T$$

where: K_g = number of observations in a class
$\bar{f}_{C_g,k}$ = the *k* observed feature vector in class *Cg*
F_g = feature matrix
M_g = mean matrix

$$F_g = \left[\bar{f}_{C_g,1}, \bar{f}_{C_g,2}, \ldots \bar{f}_{C_g,k}, \ldots, \bar{f}_{C_g,K_g}\right]$$

$$M_g = \left[\tilde{\mu}_g, \tilde{\mu}_g, \ldots, \tilde{\mu}_g\right]$$

F_g and M_g have the same number of columns. With the mean vector now known, the linear discriminant function above becomes:

$$dC_g = \bar{f}^T \Sigma^{-1} \tilde{\mu}_g - \frac{1}{2} \mu_g^T \Sigma^{-1} \tilde{\mu}_g$$

In real time testing, the observed feature vector $\bar{f}$ from each window of samples is inputted into the classifier above for each class. The prediction made, $\tilde{C}_g$, satisfies:

$$\tilde{C}_g = \arg max_{C_g} \left\{\tilde{d}_{C_g}\right\}, C_g \in \{C_1, C_2, \ldots, C_G\}$$

Furthermore, to eliminate sudden and limited incorrect decisions from the classifier, majority vote is used. The increased accuracy, however, comes at the cost of a delay in the response time. The size of the majority vote window will be varied and the resulting accuracy and delay results observed.

The system to improve accuracy and response time proposed in Englehart's paper is the overlapping window continuous decision stream. A large window of time is needed to extract features from the data, and to supply the classifier. This slows down the prediction rate of the classifier and hinders real time performance. The system proposed involves the window length, which is calculated every set interval, defined as the window increment. If the window size is 150 milliseconds, a window increment can be 50 milliseconds. Figure 3.3 shows the overlapping window scheme. The window increment in this design shall be 20 milliseconds, to ensure a quick and real-time decision stream. The window length will also be kept short, but can be varied and the accuracy of the classifier observed to determine the optimal length.

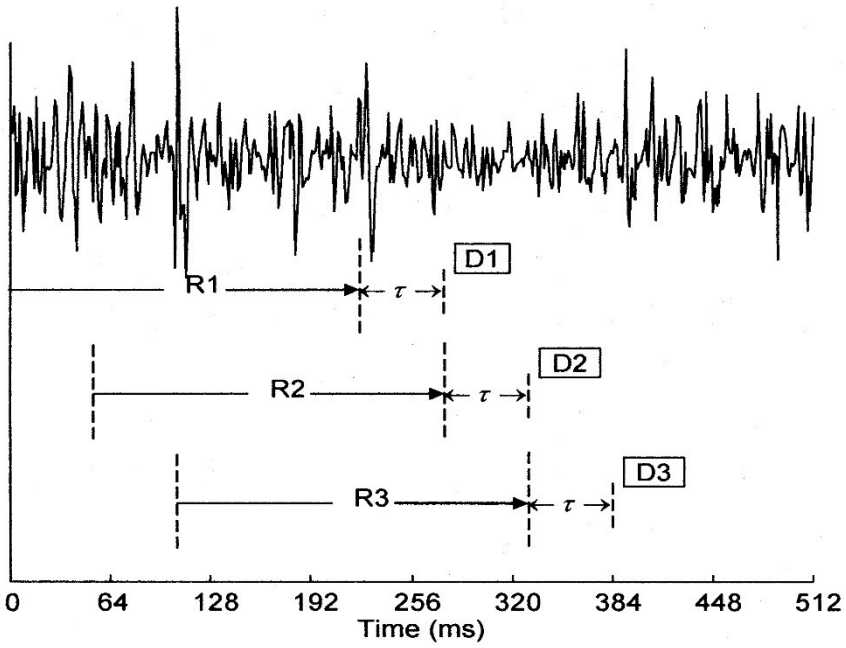

Figure 3.3. Overlapped window decision scheme. Every decision, D, is computed every window increment, τ, based on the previous window length of data, R (Englehart).

PATTERN RECOGNITION CLASSIFIERS AND COMBINED OUTPUT

The main concept in the design is to combine the sensor output from both the EMG and IMU sensors. The combination of both outputs will be used to control the mouse cursor while the mouse clicking functions will be controlled solely by the EMG output. One primary feature of the design is to detect and ignore false positives. False positives are defined as any action the user may perform in everyday life that is not meant to be translated to a mouse operation. Another objective of the design is to be accurate enough to be a replacement for the standard mouse; a goal not yet obtained

by any current design or device. A block diagram of the entire system is given in figure 3.4.

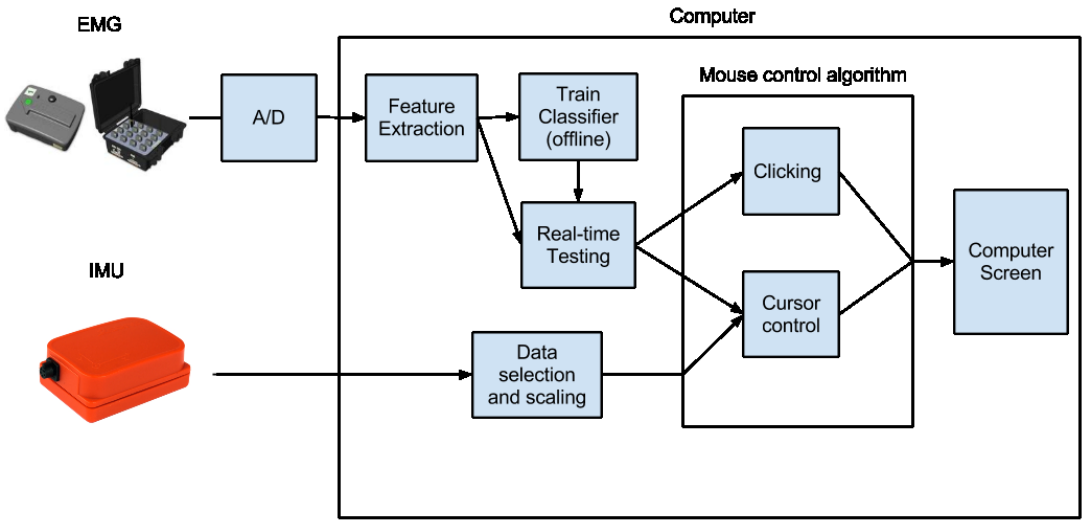

Figure 3.4. Block diagram of the entire system. This is an overview of the EMG and IMU mouse interface system. Values from Train Classifier (offline) are required in Real-time Testing, therefore Train Classifier is performed before Real-time Testing.

The mouse clicking functions will be implemented through solely EMG sensors. Both left and right click actions need to be identified. The functionality of pressing and releasing the mouse button, as it is known on a standard mouse, need to be separate functions. This enables drag and drop functionality. The EMG muscle locations selected for these functions need to differentiate between the index and middle finger, as well as be independent from wrist movements.

For mouse cursor movements, the IMU will be the primary input source. However, the IMU data are highly sensitive to movement, making it difficult to keep the cursor stationary when no movement is desired. Unintentional movement is

defined as a false positive. To counteract false positives, the EMG classification part of the system is introduced. The EMG is used to know which direction the user's wrist is pointing in order to know which direction they desire the IMU output to be going. The EMG part of the design can effectively be called an error or false positive detector.

The basic design is to combine the two outputs by only accepting the output from the IMU when the output of the EMG agrees with the IMU's direction. This prevents false positives and would allow the user to move their arm for everyday activities and not move the mouse cursor during such movements. Any movement above a certain threshold and below another threshold will be considered zero, or no movement. This prevents jitter, drift or any movements too small to be desired as cursor movements to register. Movements that are too large are ignored for the same reason.

Because the IMU is the primary input for cursor movements, and the EMG classifier works to prevent false positives, it leaves the specific design of the system very open. The accuracy and precision of just the EMG output is not as important as the final system. Creating too precise of an EMG classifier has the potential to decrease the accuracy and usability of the finished product. This results in flexibility in the choice of muscle location and the formation of classifiers. However, since click functionality is controlled solely by EMG, we need to ensure that the sensor and classification of clicks is as accurate as possible, and the sensor location takes priority over wrist muscle locations.

The four locations for the wrist each represent a different direction, and contain little crosstalk. However, they may not all be necessary. Depending on where the electrode is placed for the Flexor Digitorum Superficialis (F1), the crosstalk varies. If it is placed closer to the wrist, the crosstalk is minimal, if it is placed half way between the elbow and wrist, then there is crosstalk from the Flexor Carpi Ulnaris (W2), the muscle associated with the down and right action by a right hand. Therefore, I propose three different configurations for the EMG sensors. Each configuration and their corresponding classifier options are listed in Table 3. The first is all five locations, the four for the wrist and single sensor closer to the wrist for clicking functionality (Setup A, Figure 3.4). The second, is the clicking functionality sensor on F1 placed near the elbow, and only the following three wrist muscle locations: Extensor Carpi Radialis (W1), Extensor Carpi Ulnaris (W4), Flexor Carpi Radialis (W3), excluding W2 (Setup B, Figure 3.5). The direction of right and down are still covered by the other muscle locations. The third configuration aims to provide the most accuracy for clicking and the most relaxed of wrist direction classification. It is to have two electrodes for clicking, one on the F1 near the wrist, and another on the Extensor Digitorum Communis, hoping to use both at the same time to increase accuracy. The wrist classifier would only use two sensors, one on W1 and another on W2, for up and down, respectively (Setup C, Figure 3.6). However, the Extensor Digitorum Communis and W1 are located adjacent to each other, and contain a great amount of crosstalk. It is infeasible to place both sensors next to each other with the hope of obtaining distinguishable signals. Therefore, one sensor will be used on W1. Because

of the second sensor on the F1, classification accuracy for clicking should remain high.

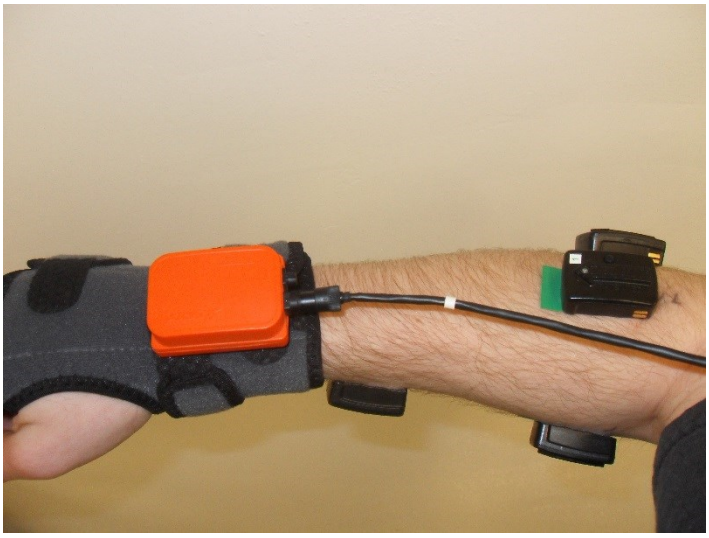

Figure 3.5. EMG configuration Setup A with IMU. From top to bottom: Extensor Carpi Ulnaris, Extensor Carpi Radialis, Flexor Digitorum Superficialis (Near Wrist), Flexor Carpi Radialis. Hidden: Flexor Carpi Ulnaris.

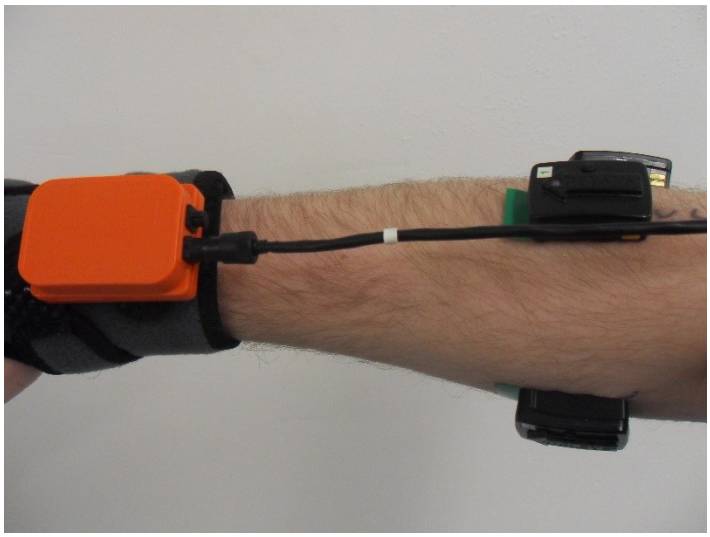

Figure 3.6. EMG configuration Setup B with IMU. From top to bottom: Extensor Carpi Ulnaris, Extensor Carpi Radialis, Flexor Carpi Radialis. Hidden: Flexor Digitorum Superficialis (Near Elbow).

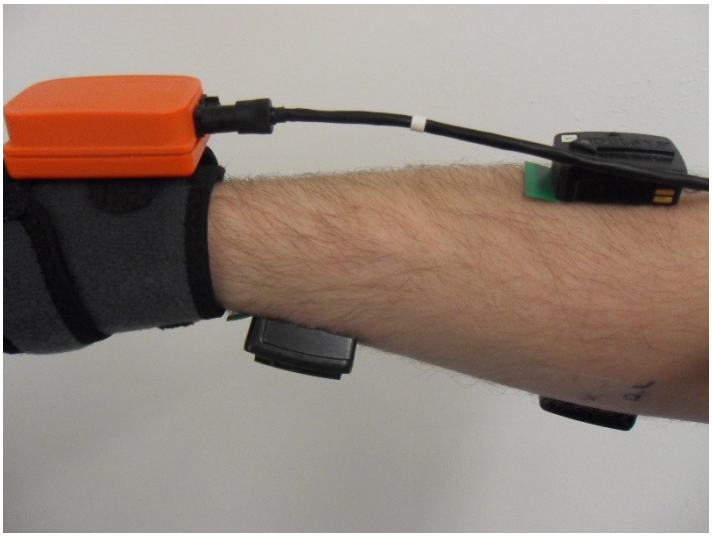

Figure 3.7. EMG configuration Setup C with IMU. From top to bottom: Extensor Carpi Radialis, Flexor Digitorum Superficialis (Near Wrist), Flexor Carpi Ulnaris.

Table 3.3. List of EMG sensor setups and configurations. Target muscles are given for each configuration. Possible actions detectable are given for each configuration.

Setup	# Sensors	Target Muscles	Configuration	Actions Classified
A	5	Extensor Carpi Radialis, Extensor Carpi Ulnaris, Flexor Carpi Ulnaris, Flexor Carpi Radialis, Flexor Digitorum Superficialis (Near Wrist)	1	Rest, Up, Down, Left, Right, Left Click, Right Click
			2	Rest, Up, Down, Left, Right, Up-Left, Up-Right, Down-Left, Down-Right, Left Click, Right Click
			3	Rest, Up, Down, Left, Right, Left Click, Right Click
B	4	Extensor Carpi Radialis, Extensor Carpi Ulnaris, Flexor Carpi Radialis, Flexor Digitorum Superficialis (Near Elbow)	1	Rest, Up, Down, Left, Right, Left Click, Right Click
C	3	Extensor Carpi Radialis, Flexor Carpi Ulnaris, Flexor Digitorum Superficialis (Near Wrist), Extensor Digitorum Communis	1	Rest, Up, Down, Left Click, Right Click

With these three setups we can form a set of classifiers for each to see which one performs best. For Setup A, we have several options for pattern recognition configurations. The first is to have two classifiers. One for all wrist movements and the other for all clicking actions (Configuration 1). The set of actions are: Rest, Up, Down, Left, Right, Left Click and Right Click. Having two classifiers enables the user to simultaneously click while moving the cursor. The second option is to have three classifiers, one for clicking and two for the wrist (Configuration 2). One wrist classifier will output the decision for up and down and the other for left and right. Both classifiers also have a rest or no action decision. This two classifier design for the wrist combines to not only output the standard five classes, up, down, left, right and rest, but also the combinations of them, up-right, up-left, down-right and down-left. To achieve this, each muscle location must be identified for only one of the up, down, left and right actions, in order to prevent one muscle from interfering with the other classifier. The accuracy within each of the two classifiers may be 100 percent but the accuracy across both may be low. The third option is to have one single classifier for all wrist movements and clicking (Configuration 3). The set of actions for this option is the standard set of rest, up, down, left, right, left click and right click. This will provide the most accuracy but have several disadvantages as well. Only one action can be performed at a time, meaning the user cannot click while moving. This eliminates the ability to drag and drop.

For Setup B, we do not have four independent EMG sensors for each wrist direction. Because of this, we can only implement Configuration 1 from Setup A, with just the three EMG sensors for the wrist. For Setup C, the standard set of actions do

not apply. The actions for Setup C are: Rest, Up, Down, Left Click and Right Click. Similar to Setup B, Setup C is limited to Configuration 1, with only the wrist actions of Rest, Up and Down. This configuration includes two sensors for clicking and only three classes for wrist movements. This helps prevent false positives in the clicking classifier, and simplifies the wrist direction classifier to obtain higher overall accuracy with the IMU.

For each setup and configuration, the accuracy and percent error will be calculated to determine the optimal configuration. Each setup and configuration will be tested in real time to determine the accuracy when combined with the IMU output. To help improve the accuracy of the design, limit the effort required by the user, and prevent hyper-extension of the wrist, a simple wrist or glove type of constraint will be used. The current restraint used is the Ace Tek Zone wrist brace. However, to meet the goal of being convenient to wear and use, it must not interfere with any everyday action the user might want to perform.

The fusion method for combining the EMG and IMU output varies based on the sensor setup and classifier configuration. For all cases, the IMU output is only accepted when the direction of the output agrees with the direction classified from the EMG data. The following directions given correspond to wrist/hand movements, rather than the coordinates of a computer screen, where the y-axis is inverted. For Setup A, Configuration 1: When the prediction is the direction Up, IMU output is only accepted if the y direction is positive/up and the absolute value of the slope is greater than 1, forming a v-shaped region of accepted values. Figure 3.7 depicts the data fusion and the slope values accepted. The same actions are used in Setup A,

Configuration 3, and Setup B, Configuration 1, making the fusion method the same. For Setup A, Configuration 2: For the actions of Up, Down, Left and Right, the fusion method is the same as Configuration 1. However, the actions of Up-Left, Up-Right, Down-Left and Down-Right are different, and overlap regions of the original actions. When the prediction is the direction Up-Left, IMU output is only accepted if the y direction is positive/up and the x direction is negative/left, forming a square shaped region of accepted values. Figure 3.8 depicts the data fusion of Setup A, Configuration 2. For Setup C, Configuration 1: There is only two actions of use, Up and Down. When the prediction is the direction Up, IMU output is only accepted if the y direction is positive/up. When the prediction is the direction Down, IMU output is only accepted if the y direction is negative/down. Any x value is accepted in either case. Figure 3.9 depicts the data fusion of Setup C, Configuration 1.

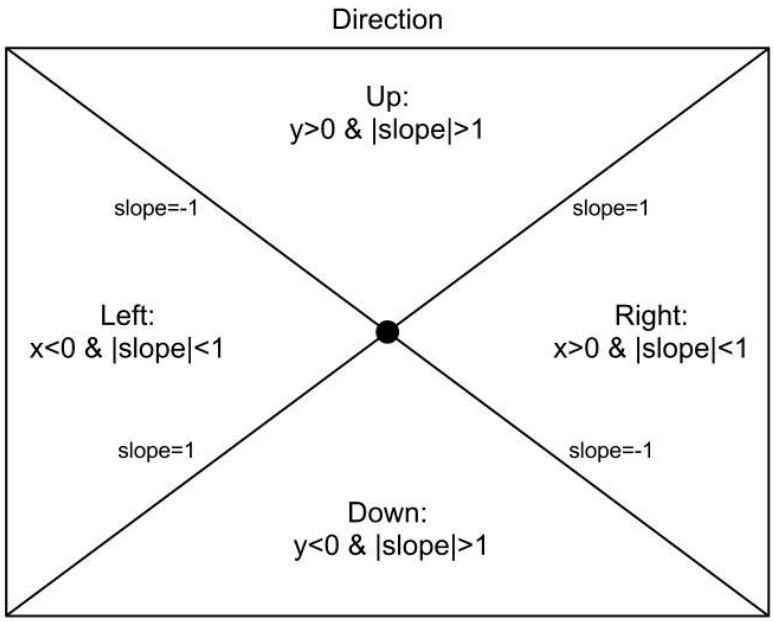

Figure 3.8. Fusion Method: Setup A, Configuration 1, Setup A, Configuration 3 and Setup B, Configuration 1. If the conditions for each region are met by the x and y values from the IMU, then the input is accepted for cursor control.

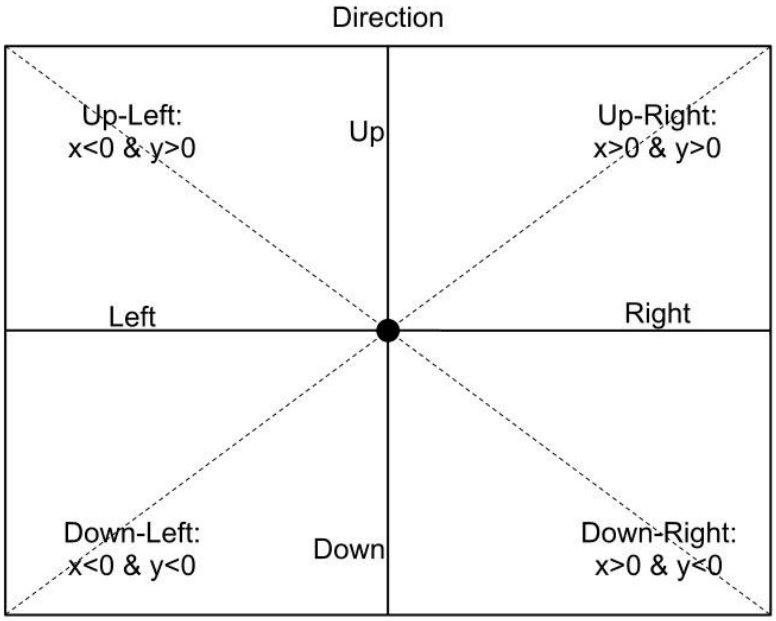

Figure 3.9. Fusion Method: Setup A, Configuration 2. Up-Left, Up-Right, Down-Left and Down-Right actions are added on top of the logic for Setup A, Configuration 1, shown in Figure 3.7. If the conditions for each region are met by the x and y values from the IMU, then the input is accepted for cursor control.

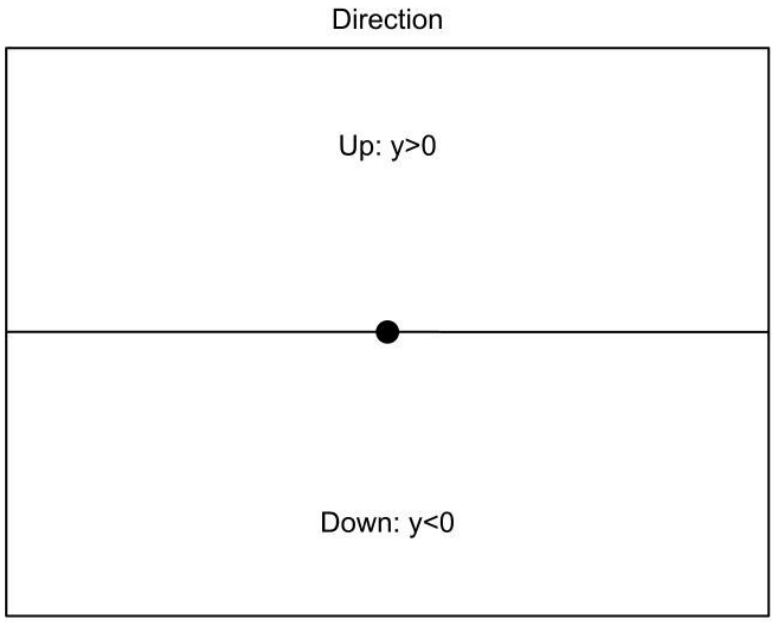

Figure 3.10. Fusion Method: Setup C, Configuration 1. If the conditions for each region are met by the x and y values from the IMU, then the input is accepted for cursor control.

IMPLEMENTATION

The program can be divided into two sections, training and testing. The training portion of the program collects EMG training data, extracts features, builds the LDA pattern recognition classifiers and saves off the classifier values. The second portion of the program implements the mouse interface in real time. It collects EMG data, classifies the EMG data, and uses the predictions along with the IMU data to control the mouse. For simplicity and ease of use, the training portion of the program is implemented in MatLab. The proof of concept was originally developed completely in MatLab, and later the real time mouse interface was ported to C++. The advantage of developing the program in MatLab, is the ability to use built in functions, including Matrix manipulation functions, the ability to easily plot data, and debug code. C++ is chosen for the real time mouse interface because it is much more efficient, free to use and develop in, and one step closer to an embedded system implementation. Both the A/D and the IMU device contain APIs for MatLab and C/C++.

The Measurement Computing A/D samples the EMG sensors at 1000Hz, every millisecond. The Xsens IMU system samples the IMU at 100Hz, every 10 milliseconds. The window increment for the EMG data has been chosen to be 20ms. Thus, the IMU must be sampled twice per EMG prediction. The window length will be determined by accuracy and delay in classification, and will initially be set to 60ms.

In MatLab, the A/D is called by using the built in data acquisition toolbox. An analog input is defined and number of channels and sample rate set. Data is collected by a simple function call that returns the newest window increment size of data in the

A/D buffer. The MatLab code for data collection, feature extraction and the LDA training algorithm are given and detailed in Appendix A. In C++, the size of the buffer in the A/D must be allocated by specifying the multiplying the number of channels with the window length size. Another FIFO buffer is set up, first in first out, to store each window increment up to the size of the window length, to be used later in classification. This is done by repeatedly calling a function that returns the index, or the current number of samples collected per window length, to know that another window increment has been collected, and it can be copied into the FIFO buffer. For each window increment where a full window length of data are available, the data are passed to the LDA classifier, explained in a previous section. The feature extraction, LDA classifier, and majority vote function are written in C, with efficiency being the priority. The size of the majority vote decision stream is set to 20, and is varied to observe what size delivers the optimal accuracy per delay. The decision outputted by the majority vote function for each classifier forms the EMG portion of the program.

As mentioned earlier, the pertinent data from the IMU are the z-axis gyroscope and y-axis gyroscope, translating to the mouse y-coordinate and x-coordinate, respectively. The axis data proportionately corresponds intuitive hand movements to cursor movements. All that is needed is appropriate scaling of the values. The level of scaling in order to match hand movements appropriately to the screen depends on both the screen size and the distance of the user to the screen. The units of the IMU data are meters per second. An appropriate scaling factor found for the y-axis is 36, and x-axis 42. This forms the IMU portion of the program.

The function calls and API used in the Real Time C++ implementation are those of Windows, using the Windows.h header file. The function calls are not complex, and would not involve much work to port to a different operating system such as Linux. When updating the x and y coordinates of the mouse, we use the API to get the current mouse coordinates. Doing this allows other mouse input devices and programs to be used concurrently, the same way commercial mice operate. The clicking and functionality operates as follows: The last clicking action is stored in order to know what the next appropriate action to be taken is. For example, if the action is left mouse click, the left mouse press function is called. If the last action was a left mouse click and the next action is rest, then the left mouse release function is called. This concludes how the mouse functionality works. The code for the C++ real-time implementation of the design are given and detailed in Appendix B.

DETERMINATION OF OVERALL ACCURACY

Once the prototype is developed there needs to be a method of measuring the testing accuracy and percent error. While there are ways to measure the training accuracy of the pattern recognition classifiers independently, there is no easy way to measure the accuracy of the final design. For this a separate program was developed to measure the accuracy. The program measures the elapsed time it takes for the user to perform a set number of clicks at predefined locations. A graphical user interface was created with circles located in the window and a predefined order in which the user clicks inside them. Along with the elapsed time, the number of incorrect clicks

performed during the trial is also recorded. There is a total of nine clicks the user must make and each click outside the specified circle will count as an incorrect click. The user will perform the task with a standard mouse and then with the EMG and IMU mouse interface. The elapsed time and incorrect number of clicks will then be compared. The program was written in C++ with the Win32 API. Figure 3.10 shows a screenshot of the program. Figure 3.11 shows a screenshot of the program while it is running. Appendix C contains the code for the program.

The goal of the project is to be intuitive and convenient. This also means that setup time as well as ability to wear the interface while performing everyday tasks is a factor in the accuracy and effectiveness of the design. However, the scope of this project is limited to a proof of concept, so setup time and ability to wear the device are not as heavily weighted in the accuracy of the design. Furthermore, an observation of false positives when wearing the device and performing everyday tasks, such as typing on a keyboard or writing with a pencil, will be noted towards the overall accuracy of the device.

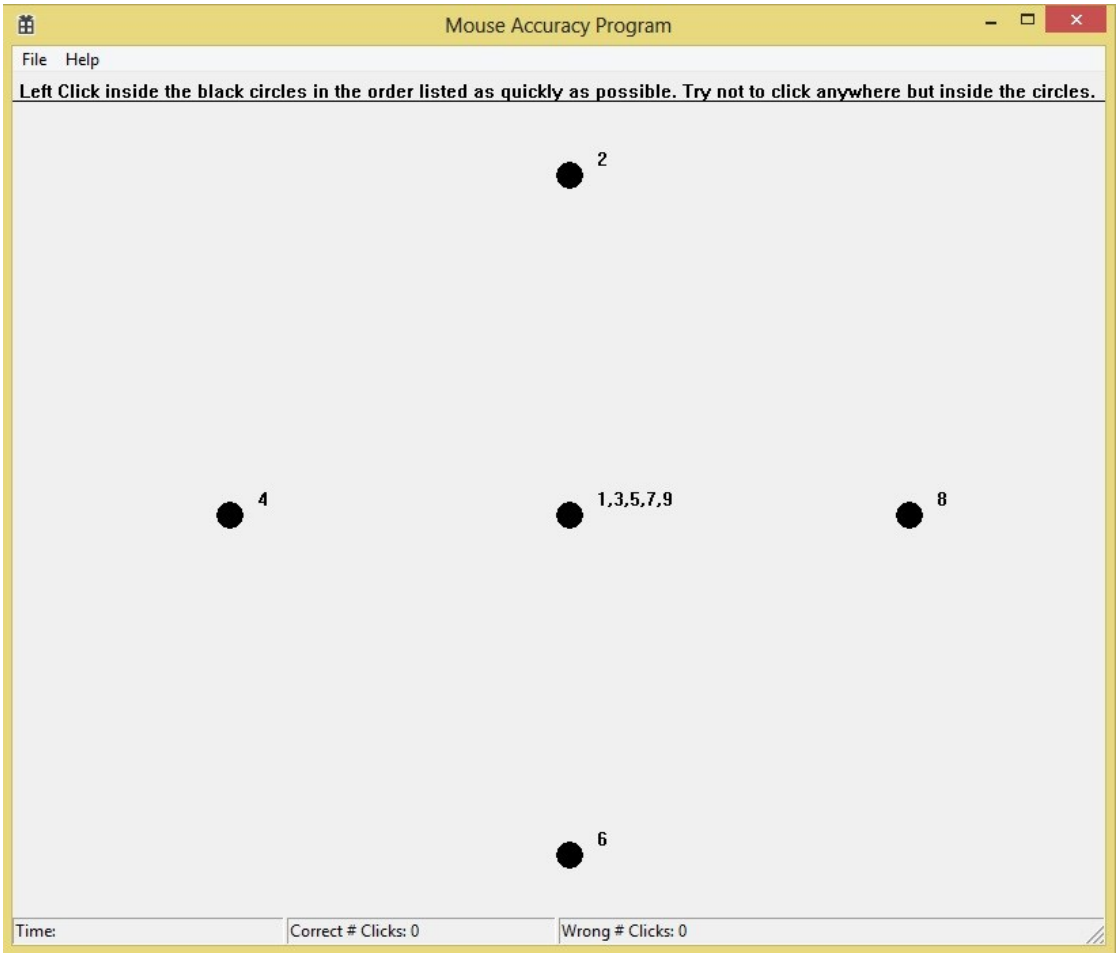

Figure 3.11. Screenshot of the Mouse Accuracy Program. The user must click within the black circles in the predefined order given. The time it takes to perform all nine clicks is showed in the bottom left corner, while the number of correct and incorrect clicks is given to the right.

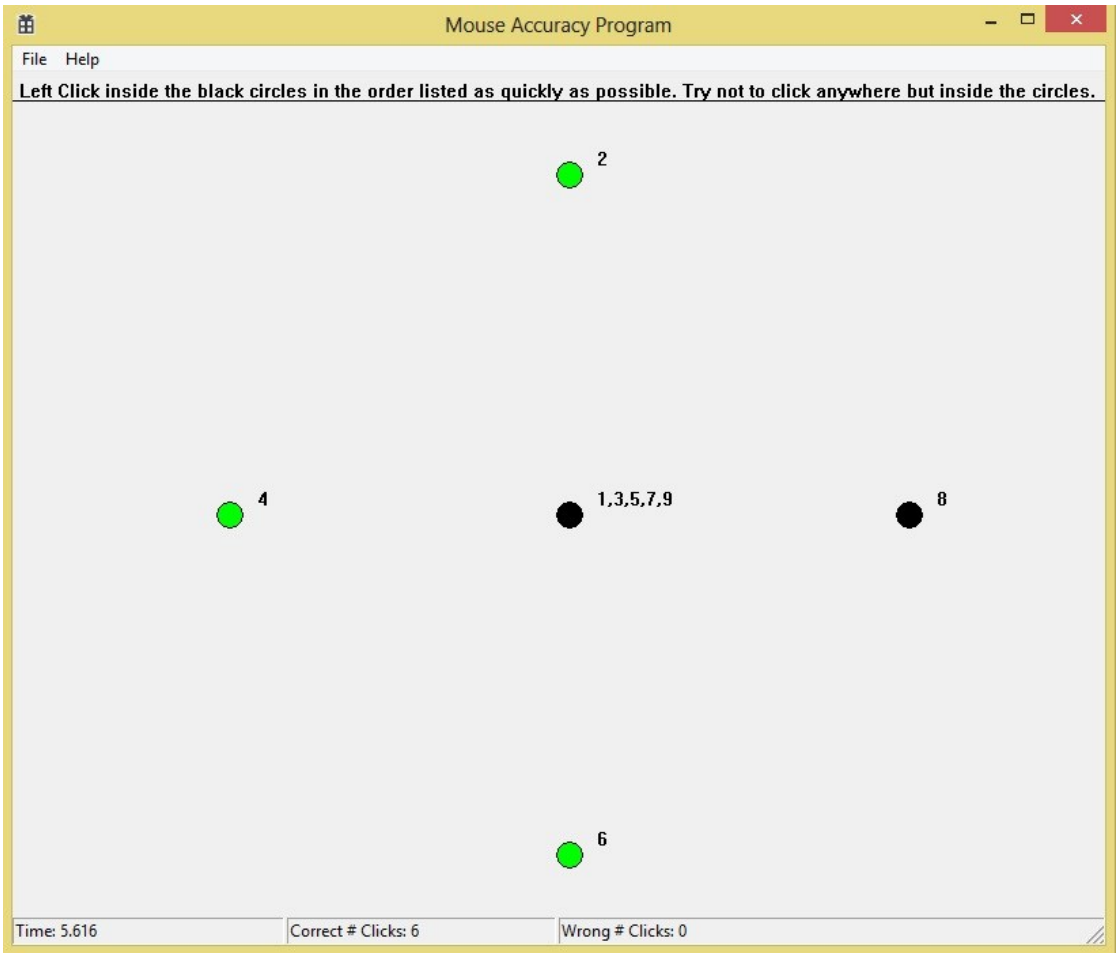

Figure 3.12. Screenshot of the Mouse Accuracy Program while running. The circles turn green when clicked to help indicate to the user that they have clicked inside the circle correctly. The current time elapsed is displayed in real time in the bottom left corner, while the number of correct and incorrect clicks is given to the right.

CHAPTER 4

RESULTS AND DISCUSSION

For all five muscles (W1, W2, W3, W4 and F1), for a total of six locations (F1 near wrist and F1 near elbow), training data were collected. Data were collected for seven actions, Rest, Up, Down, Left, Right, Left Click and Right Click. Data were also collected to verify the EMG sensors and locations as well as provide a visual tool for understanding the output. In Figures 4.1 and 4.2, each EMG channel is offset to better present the data, for both wrist and finger actions. Each figure represents data taken for Setup A. By observing the data, the test maneuvers for each muscle, as provided by the literature, are proven. For muscle F1, the EMG sensor contains motion artifacts. Motion artifacts are noise in the EMG sensor generated by movement, either physical movement or vibrations, or movement against the skin. This is generated by the wrist brace, which is located right next to the sensor. For future data collection and testing, the sensor was moved slightly farther away from the wrist brace. For finger clicking actions, some crosstalk is detected in W2, which was anticipated. However, F1 is a prominent muscle for these actions compared to the others. A short spike in the voltage is observed for brief left and right clicking actions. Left press and right press represent continuously holding out the left and right click actions, respectively. A clear difference between the voltage amplitudes of the two actions is observed.

The actual training data recorded for use in the training of the LDA classifiers were taken by holding out each action for several seconds. For each action, features were extracted and used for training the classifiers. The training error is calculated for each classifier of each setup and configuration. The training error is calculated by the percentage of points that are incorrectly classified, and fall on the wrong side of the decision boundary formed by the classifier. Table 4.1 shows the training error for each classifier in each Setup and configuration, as well as ratings of the testing errors and false positives. Depending on the classifier configuration, the training error has little significance to our real-time accuracy. For Setup A, Configuration 2, there are a total of three classifiers. Each classifier has a very high accuracy. However, crosstalk between the muscles in the different classifiers can result in wrong classifications in real-time, something that the training error does not reflect. However, for Setup A, Configuration 3, there is only one classifier for all actions. Thus, the training error highly reflects what the real-time testing error will be. Furthermore, the testing error of the EMG classifiers still does not reflect the overall accuracy of the system when combined with the IMU output.

Table 4.1. EMG Training and Testing Accuracy. Testing Accuracy and False Positive Rating are based on rating scales. The numbers listed are given by the user of the system during testing.

Setup	Configuration	Classifier	Training Accuracy (%)	Testing Accuracy[1]	False Positive Rating[2]
A	1	Wrist	99.82	5	1
		Click	97.53	3	4
	2	Wrist Up/Down	99.04	5	2
		Wrist Left/Right	99.59	5	2
		Click	97.53	3	4
	3	All	99.94	5	1
B	1	Wrist	99.3	4	1
		Click	91.93	3	5
C	1	Wrist	99.04	5	1
		Click	98.35	3	4

[1] Classifier Recognition Accuracy Rating Scale: 0 – Not accurate at all. 1 – Barely accurate. 2 – Accurate less than half the time. 3 – Accurate over half the time. 4 – Accurate almost all the time. 5 – Perfect.
[2] False Positive Rating Scale: 0 – No false positives. 1 – Hardly noticeable false positives, little to no inconvenience. 2 – Noticeable false positives. Minor inconvenience. 3 – Considerable false positives, Definite inconvenience. 4 – Great many false positives. Almost useless. 5 – Useless.

In real-time, each setup and configuration was tested. The real-time EMG decision was observed while each action is performed to note the accuracy and occurrence of false positives. The accuracy of the classifier is defined as how often it classifies the action correctly. False positives are defined as incorrect actions given for that classifier when the actions of a different classifier are performed. There is no way of calculating the testing error, as the class the action belongs to is decided by the user in real-time. Therefore, a rating system was developed, as listed in Table 4.1. Each setup and configuration is also tested and evaluated for accuracy when combined with the IMU output. The results are listed in Table 4.2.

Table 4.2. Testing Accuracy and False Positive Rating for the combined EMG and IMU system. Results from the Mouse Accuracy Program are also given. Testing Accuracy, False Positive Rating, and IMU Combined Accuracy are based on rating scales. The numbers listed are given by the user of the system during testing.

Setup	Configuration	Completion Time (s)	# Wrong Clicks[1]	Testing Accuracy[2]	False Positive Rating[3]	IMU Combined Accuracy[4]
A	1	64.794	17	4	2	4
	2	75.179	17	4	3	5
	3	54.724	2	5	1	4
B	1	N/A	N/A	3	5	4
C	1	65.606	29	4	4	3
Mouse[5]	N/A	6.39	0	5	0	N/A
Touchpad[6]	N/A	10.548	0	5	0	N/A

[2] The number of wrong clicks is generated by the number of times the user clicks outside of the specified circle during testing.

[2] Classifier Recognition Accuracy Rating Scale: 0 – Not accurate at all. 1 – Barely accurate. 2 – Accurate less than half the time. 3 – Accurate over half the time. 4 – Accurate almost all the time. 5 – Perfect.

[3] False Positive Rating Scale: 0 – No false positives. 1 – Hardly noticeable false positives, little to no inconvenience. 2 – Noticeable false positives. Minor inconvenience. 3 – Considerable false positives, Definite inconvenience. 4 – Great many false positives. Almost useless. 5 – Useless.

[4] IMU Combined Output Accuracy Rating Scale: 0 – Useless. 1 – Hard to control, almost useless. 2 – Inconvenient or challenging to use. 3 – Minor difficulties in usability. 4 – Almost no difficulties in usability. 5 – Perfectly fluid integration.

[5] Logitech G700 laser mouse.

[6] Dell Studio 1535 built in track/touchpad.

In table 4.2, the testing accuracy is the average of all classifiers for each EMG configuration. The false positives rating is an evaluation of false positives when performing every day, non-mouse related actions, as well as incorrect decisions in one classifier while performing the action belonging to another classifier. Specifically, false positives were observed when the actions of typing on a keyboard and writing with a pencil on paper were performed. The results from the Mouse Accuracy Program are also listed along with the accuracy rating for the system combined with the IMU. The IMU accuracy is based on several factors, as noticed when testing the different configurations. For Setup A, Configuration 2, the accuracy of the individual classifiers led to the seamless integration of the IMU. Large and small movements were easy to produce. In most configurations, small IMU movements were difficult to produce, making it hard to click within the small black circles in the Mouse Accuracy Program. This greatly increased the time to complete the test.

Although the IMU integration for Setup A, Configuration 2 was very accurate, there was a significant number of false positives, especially related to clicking. False positives do not have the ability to be compensated for through the IMU because clicking is controlled solely through EMG. Thus, the usability of the overall configuration is low. Common false positives include the actions down or left producing the decision of down-left. This limits the range of IMU input accepted and hinders the overall performance. Another false positive is the crosstalk between the action down and clicking. The former often produces the latter decision concurrently.

Setup A, Configuration 1 is similar to Configuration 2. The false positives observed through the actions down, and left click and right click remain. However, the

false positives surrounding wrist directions are eliminated by the single wrist classifier. Unfortunately, it was observed that small movements with the IMU were more difficult to produce than in Configuration 2, lowering the IMU integration accuracy.

Lastly, for Setup A, we have Configuration 3. Unexpectedly, this configuration was the most accurate of any setup and configuration. Having one classifier eliminates almost all false positives, as seen with only two incorrect clicks in the accuracy program, but unfortunately removes the drag and drop functionality. Clicking cannot occur concurrently with wrist movements. While the accuracy is much higher, small movements with the IMU are still observed to be difficult.

Setup B involved the most false positives of any configuration, making the design useless. Specifically, the number of clicking false positives, combined with the lower accuracy of the wrist classifier, resulted in the incompletion of the mouse accuracy program. Setup C also involved a large amount of false positives. Furthermore, using the one classifier for solely up and down actions, resulted in relaxed IMU integration. Because it is so easy to obtain an up or down prediction from the classifier, the mouse cursor was constantly moving in directions not desired. Despite the IMU output reflecting hand and thus cursor movements seamlessly, the simple classifier design results in enough false positives to render this design practically useless.

A common observation throughout the configuration is the difficulty in small IMU movements. At the instant the wrist would change direction and the classifier would change to predict the correct action, the wrist was still slightly moving. This movement would frequently result in the IMU registering a small movement

immediately upon wrist direction change. This would result in the mouse cursor jumping slightly in the direction of the new action. This hindered the performance during the Mouse Accuracy test. To solve this, the IMU needs to be moved slightly further up the arm. Another general observation made is the false positives through typing on a keyboard and writing with a pencil. The configurations with multiple classifiers, such as Setup A, Configuration 2, resulted in the most false positives while performing these actions. Setup A, Configuration 3, resulted in the fewest false positives during these actions. While typing, the most common false positive observed was clicking, which is a result of pressing down with your fingers on the keyboard. This is a slight problem and is seen less in Setup A, Configuration 3.

In the end, Setup A, Configuration 3, the simplest design with just one classifier, was the most accurate. Most importantly, it had the fewest false positives. The completion times for the Mouse Accuracy program are still high compared to that of the conventional mouse and touchpad. However, the times recorded for each configuration were that of the first attempt. Minor practice with the device would result in an immediate drop in completion time. The scaling factor by which the IMU output is adjusted by in order to control the mouse cursor can also be modified to better fit the user. Small changes can be made to the design to better fit the user after sufficient practice has been completed, and possible improvements identified.

Other factors that contribute to the overall accuracy include the window length and size of the majority vote function. The classifier accuracies for all configurations for both training and testing were high, thus the window length was kept constant at 60 milliseconds. It was observed that the ability to double click was not possible. The

recognition of a click required too much time in order to detect a second click soon enough. In order to enable double clicking, the size of the majority vote window can be reduced. This, however, will increase the number of false positive clicks. Furthermore, the most accurate configuration, Setup A, Configuration 3, only has one classifier. Therefore, the majority vote window size cannot be changed just for clicking. This would increase the false positives for all actions, detrimental to a positive feature of this configuration.

Small, yet considerable factors in the overall success and usability of the design are setup and preparation time. How long it takes to put on the EMG sensors, and how long it takes to record training data and train the classifiers is important. However, the time it takes to apply the EMG sensors, would greatly be reduced in any final product fabrication. A design such as an arm band that just wraps around the forearm with no adhesive, would improve setup time. Comparable to other electromyography applications, recording data and training classifiers takes minimal time. Partly automated, the procedure in this design takes only minutes.

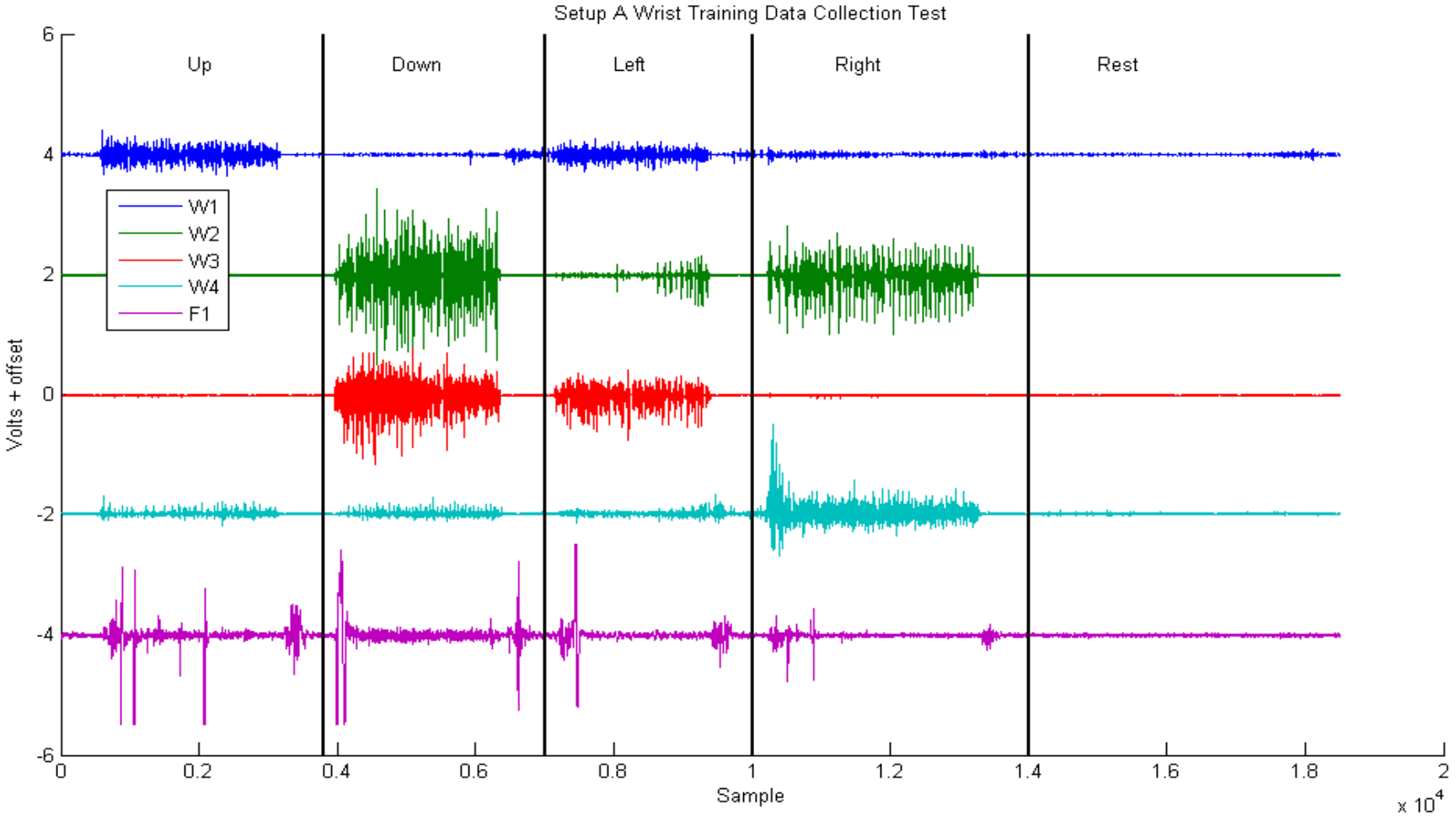

Figure 4.1. Setup A Wrist Training Data Collection Test – Channels offset. Each channel is offset to better preview each channel. Motion artifacts are present in F1. They were generated by the wrist brace being worn, which was located next to the sensor. For future data collection and testing, the sensor was moved slightly farther away from the wrist brace.

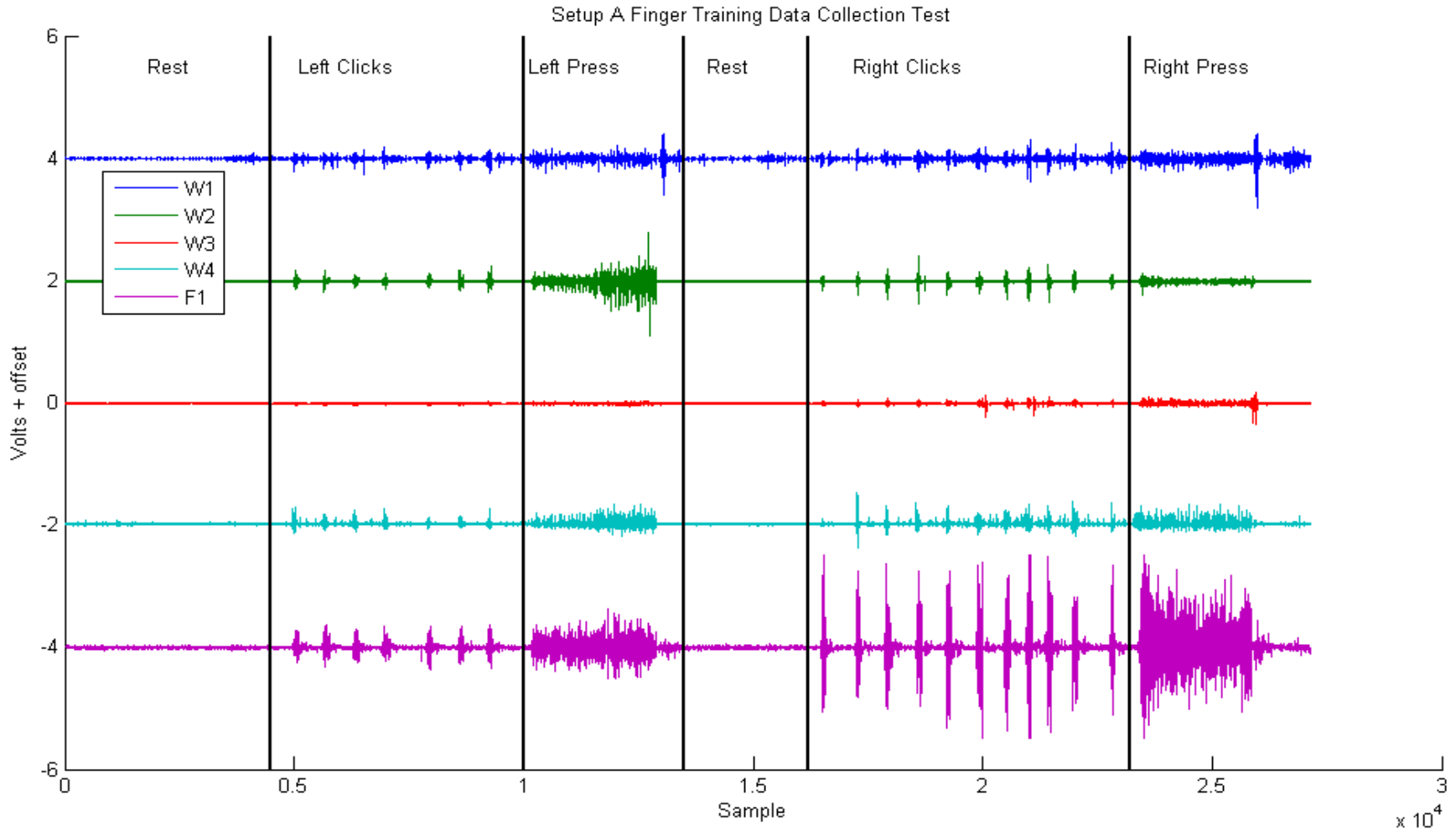

Figure 4.2. Setup A Finger Training Data Collection Test – Channels offset. Each channel is offset to better preview each channel.

DESIGN IMPROVEMENTS

Now that Setup A, Configuration 3 has been identified as the most accurate, several improvements can be made to the initial design. To enable the drag and drop functionality lost to the single classifier design, free control over the mouse cursor with the IMU can be allowed when a click action is detected and held. To more quickly identify changes in actions, the size of the majority vote window can be decreased, allowing for better IMU integration. Also, this would enable the ability to perform the double click action, which is lost when the transition from rest to left click, rest and back again, takes too long. Another proposed improvement is to enable small IMU movements when the classifier is predicting the Rest action. Small movements can either be defined as small values of angular velocity, or all values of angular velocity with the scaling factor greatly reduced. This allows the user to more easily narrow in on specific screen locations with the mouse cursor. To allow for easier transitions between directions, the accepted IMU input boundaries per action can be widened. For example, for the action Up, instead of only accepting input values resulting in the absolute value of the slope being greater than one, the allowed slope can be when greater than 3/4. For the actions Left and Right, the slope accepted can be less than 4/3, instead of one. This results in overlapping regions, and will prevent the temporary stop of the cursor at slope equal to one, while the user transitions from Up to Left, for example. The new data fusion design is shown in figure 4.3. A more advanced possible improvement is to continuously retake training data and

recomputed the classifier, to ensure consistent accuracy and account for any shift or change in the EMG sensors.

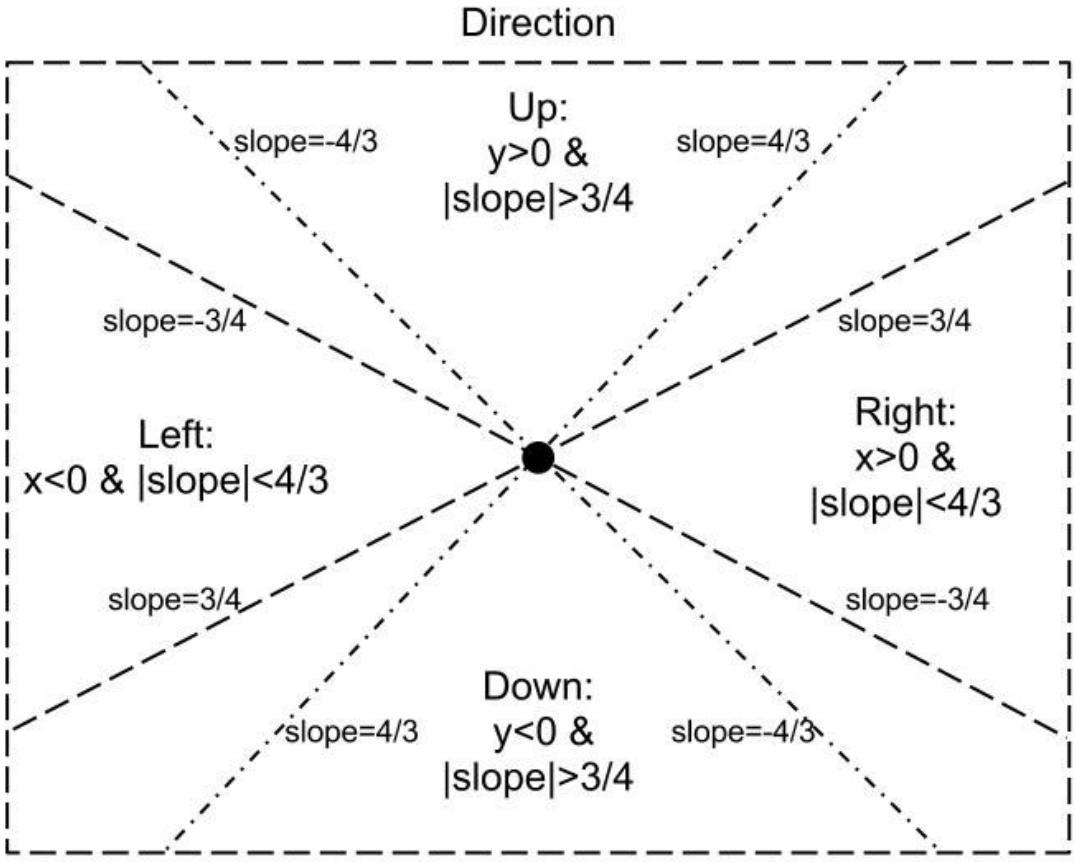

Figure 4.3. New Fusion Method: Setup A, Configuration 3. Overlapping regions of accepted IMU input. If the conditions for each region are met by the x and y values from the IMU then the input is accepted for cursor control.

After implementing the new regions and the allowing of small cursor movements during both the "Rest" and "Left Click" actions, the intuitiveness and ease of use increased. Specifically, not only are small movements now allowed, but their scaling factor is reduced, to make small precise movements easier. After several trials of the mouse accuracy program, an improved time of 20.38 seconds and only four wrong clicks was measured.

CHAPTER 5

CONCLUSION

The objective of this project was to design and create a wearable computer mouse interface, independent of cameras, sensor bars, surfaces, or tethering, that is intuitive, fast, accurate and convenient. The means of doing so was to combine the output of both EMG sensors and an IMU to control the cursor and clicking functions of a mouse. The goal is to be intuitive and accurate, in order to overcome the shortcomings in existing technologies.

Many different muscles were examined as potential candidates for the EMG sensors. The output of each sensor was observed and several different sensor setups were chosen for focus. For each setup, at least one classifier configuration was designed in the attempt of obtaining the highest testing accuracy. The angular velocity measurements of the IMU z and y axis were chosen to control the physical mouse cursor. Several different fusion techniques were designed based on the different sensor setups and classifier configurations to obtain the highest overall system accuracy, and fewest false positives.

Training data was collected in MatLab for processing. Time domain features were extracted and provided as input to a linear discriminant classifier. The implementation of the designs were programed in C++, for fast and efficient real-time performance. The setup and configuration with the highest training accuracy, highest testing accuracy, lowest false positive rate and best overall IMU integration was the

setup with EMG sensors on the muscles: Extensor Carpi Radialis, Extensor Carpi Ulnaris, Flexor Carpi Ulnaris, Flexor Carpi Radialis and Flexor Digitorum Superficialis (Near Wrist). One classifier took the input of all five channels for classification into a total of seven classes or actions: Rest, Up, Down, Left, Right, Left Click, and Right Click. A training accuracy of 99.94% was obtained and a high testing accuracy was also observed. This design took the least time of all designs to complete the accuracy program designed to evaluate the different configurations. This design also saw the lowest false positives, with only two wrong clicks observed in the accuracy program.

Compared to a standard mouse or track pad, the best design still fell far short in performance. However, the common user has been using a standard mouse or track pad their whole life, and the need for time to practice with the device is expected. With practice, the design has the potential to match or come close to the accuracy of existing mice. With further automation of the training data collection and classifier training, the device would require little configuration to start using. Further development into a commercial product would minimize the equipment required, as well as the sensor setup time, to make the device no less convenient than a standard mouse.

One of the design goals at the forefront of this design, was the application and usability by hand amputees. Muscle locations are located in the forearm, and the IMU device located on the back of the wrist or forearm, allows hand amputees and those with hand disabilities to operate the device.

As with any design, there are many improvements and new ideas that can be tested and added to increase the accuracy and further minimize false positives. The proof of concept of this project was a success, and is part of a bigger and growing field of research. The applications and potential of this design and research are large.

FURTHUR DEVELOPMENT

There will always be more features to add and different methods to try in an attempt to increase accuracy and usability. Further functionality can be added to emulate mouse wheel scrolling or middle clicking. As computers and device slowly switch to a more gestured based interface, such as swiping to scroll vertically or horizontally, could be added.

The original plan was for the design to be implemented on an embedded system. With an embedded system, all calculations would no longer be performed on the actual computer. This would allow the device to work as a normal mouse, simply sending the mouse related commands to the computer. The requirements for the embedded system are as follows: be able to run C++ programming code, be able to communicate wirelessly through Bluetooth, contain an A/D, have an interface for EMG sensor signals, and contain an IMU. Research was done and determined that all functionality is met with the combination of the Gumstix Overo Air COM computer-on-module board and the Gumstix RoboVero LPC1769 microcontroller board.

Once the embedded system implementation is completed, custom hardware could be manufactured to turn the board into a fully functional commercial product: small, stylish, easy to set up and use, and inexpensive.

APPENDICES

APPENDIX A: MATLAB TRAINING AND FEATURE EXTRACTION

EMG_data_collection.m:

This file uses MatLab's data acquisition toolbox to initialize the A/D. The input is set to the Measurement Computing A/D, and sample rate set to 1000Hz. A simple GUI is set up to specify the file name of the file to save the training data in, and a start and stop button.

```
%% Real-time pattern recognition training data collection
% Tim Forbes
% 03/14/2013
%%%%%%%%%%%%%

clc
ANALOG.EMG=1:5; % EMG channel numbers

% create analog input object for Measurement Computing A/D
ai = analoginput('mcc',0);
% add channels, -1 to account for channel 1 being port 0 on the A/D
addchannel(ai, ANALOG.EMG-1);

% set ai sample rate to 1000Hz
set(ai,'SampleRate',1000);
set(ai, 'SamplesPerTrigger', inf);
ai.LoggingMode = 'Memory';
% set to start running manually on 'start(ai)'
ai.TriggerType = 'Immediate';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Simple GUI for specifying a filename and saving the data
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
scrsz = get(0, 'ScreenSize');
figure('Position', [1 1 scrsz(3) scrsz(4)],'Color','k');

% Filename text box
h0 = uicontrol('Style','edit','String','Save file name',...
    'Position', [900 400 450 200], 'FontSize',26);

% Start Data Collection Button
h1 = uicontrol('Style', 'pushbutton', 'String',...
    'Start Data Collection', 'Position', [150 600 700 300],...
    'FontSize',50,'Callback', 'start(ai);');

% Stop Data Collection Button
% Get all the A/D data once we click the stop button
```

```
h2 = uicontrol('Style', 'pushbutton', 'String',...
    'Stop Data Collection', 'Position', [150 200 700 300],...
    'FontSize',50,'Callback',...

'stop(ai);ANALOG.raw=getdata(ai,ai.SamplesAvailable);name=get(h0,''St
ring'');comm=[''save '', name, ''
ANALOG''];eval(comm);ANALOG.raw=[];');
```

LDA_Train_SetupA_Configuration3.m:

This MatLab file contains the code necessary to train the LDA classifier for Setup A, Configuration 3. There is a separate file for each classifier for each configuration. The only differences between each file are which .mat data files are loaded, and which channels are sent to the cal_feature_fusion method. The .mat files contain the training data for all 5 channels for each individual action. The function cal_feature_fusion, shown below, extracts and returns the time domain features from each file. The features are returned in a matrix, where features are extracted for the size of a window length, for every window increment. The time domain features are then supplied into the LDA training algorithm. The error is calculated, and the weights for the LDA classifier are saved to a text file.

```
%% Real-time pattern recognition training procedure
% Tim Forbes
% 03/14/2013
%%%%%%%%%%%%%
clear;
clc;

file(1).name = dir('SetupARest.mat');
file(2).name = dir('SetupAUp.mat');
file(3).name = dir('SetupADown.mat');
file(4).name = dir('SetupALeft.mat');
file(5).name = dir('SetupARight.mat');
file(6).name = dir('SetupALeftClick.mat');
file(7).name = dir('SetupARightClick.mat');

for j=1:length(file)
    Classifier_train(1,j).feat=[];
end
%Window Length
WLen=60;
%Window Increment
WInc=20;

for j=1:length(file)
    for i=1:length(file(j).name)
        index = strfind(file(j).name(i).name,'.mat');
        load (file(j).name(i).name(1:index-1));
        disp(file(j).name(i).name);
        %pass in all 5 channels for Setup A Configuration 3
        feature_Matrix=cal_feature_fusion(ANALOG.raw(:,1:5),...
```

```
            WLen,WInc);

        Classifier_train(1,j).feat=...
            [Classifier_train(1,j).feat,feature_Matrix.Train];
    end
end

TrainData.F=[];
TrainClass.F=[];

for j=1:length(file)
    TrainData.F=[TrainData.F,Classifier_train(1,j).feat];
    TrainClass.F=[TrainClass.F,...
        j*ones(1,size(Classifier_train(1,j).feat,2))];
end

N = size(TrainData.F,1);
Ptrain = size(TrainData.F,2);   %training data size

sc = std(TrainData.F(:));
%training data + noise
TrainData.F =  TrainData.F + sc./1000.*randn(size(TrainData.F));

K = max(TrainClass.F); % number of classes

%%-- Compute the means and the pooled covariance matrix --%%
C = zeros(N,N);
for l = 1:K;
    idx = find(TrainClass.F==l);
    Mi(:,l) = mean(TrainData.F(:,idx)')';
    C = C + cov((TrainData.F(:,idx)-Mi(:,l)*ones(1,length(idx)))');
end

C = C./K;
Pphi = 1/K;
Cinv = inv(C);

%%-- Compute the LDA weights --%%
for m = 1:K
    Wg.Train(:,m) = Cinv*Mi(:,m);
    Cg.Train(:,m) = -1/2*Mi(:,m)'*Cinv*Mi(:,m) + log(Pphi)';
end
% Compute Training Error
Atr = TrainData.F'*Wg.Train + ones(Ptrain,1)*Cg.Train;
errtr = 0;
AAtr = compet(Atr');
errtr = errtr + sum(sum(abs(AAtr-ind2vec(TrainClass.F))))/2
netr = errtr/Ptrain
PeTrain = 1-netr

disp('Done');

Wg_all = Wg;
Cg_all = Cg;
```

```
save Tim_LDA_SetupA_All.mat Wg_all Cg_all;
convert_to_text_file('Tim_LDA_wrist.txt',...
    Cg_all.Train,Wg_all.Train,4);
```

cal_feature_fusion.m:

Because the function for extracting time domain features only process one window length at a time, this file contains the code to increment through the training data and pass one window length at a time into the tdfeats_emg function.

```
%% Calculate the time domain features for the classifier
% Tim Forbes
% 03/14/2013
%%%%%%%%%%%%%

function feature_Matrix=cal_feature_fusion(EMGdata,WLen,WInc)
    Index=1:WLen;
    Feature_raw=[];

% Iterate through the training data, one window increment at a time
    while Index(end)<=size(EMGdata,1)
% pass in a window length of data to the feature extraction algorithm
        [Features, Property, Feature_EMG] = ...
            tdfeats_emg(EMGdata(Index,:),1,WLen);

        Feature_raw=[Feature_raw,Feature_EMG];
        Index=Index+WInc;
    end

    feature_Matrix.Train=Feature_raw(:,:);
end
```

tdfeats_emg.m

This MatLab file contains the code necessary to extract time domain features from a set of EMG data. It extracts zero crossings, turns (slope changes), waveform length, and mean absolute value. It is based on a papers published by Englehart and Hudgins.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TDFEATS   Compute all time domain features
%           and the mean features [Hudgins 1991]
%
% (c) Kevin Englehart, 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [Features, Property,Feature_emg] =...
    tdfeats_emg(DataSet,Nframe,inc)

DEADZONE = 0.025;
```

```
SCALE_ZC = 15;
SCALE_MAV = 2;
MAV_SIZE = 100;
ruler = 1/inc;
rulersq = ruler^2;
lscale = inc/40;
tscale=(inc/40.0)*10;

Ntotal = size(DataSet,1);
Nsig = size(DataSet,2);
DataSet = DataSet - ones(Ntotal,1)*mean(DataSet);

for SigNum = 1:Nsig
    clear mav dmav turns zero_count len;
    for frame = 1:Nframe
        zero_count(frame) = 0;
        maxvalue(frame) = 0;
        len(frame) = 0;
        f_len(frame) = 0;
        mav(frame) = 0;
        dmav(frame) = 0;
        turns(frame) = 0;
        signallevel(frame) = 0;

        index = 1 + (frame-1)*inc;
        range = index:index+inc-1;
        mav(frame) = mean(abs(DataSet(range,SigNum)));
        signallevel(frame) = std(DataSet(range, SigNum));

        flag1 = 1;
        flag2 = 1;
        for i = 1:inc-2;
            idx = index + i;
            fst = abs(DataSet(idx-1,SigNum));
            mid = abs(DataSet(idx,SigNum));
            lst = abs(DataSet(idx+1,SigNum));

% Compute Zero Crossings
            if ((DataSet(idx,SigNum)>=0 ...
                    & DataSet(idx-1,SigNum)>=0)...
                    | (DataSet(idx,SigNum)<=0 ...
                    & DataSet(idx-1,SigNum)<=0))
                flag1 = flag2;
            elseif ((mid<DEADZONE) & (fst<DEADZONE))
                flag1 = flag2;
            else
                flag1 = (-1)*flag2;
            end
            if (flag1~=flag2);
                zero_count(frame) = zero_count(frame) + 1;
                flag1 = flag2;
            end
% Compute Turns (Slope Changes)
            if((mid>fst & mid>lst) | (mid<fst & mid<lst))
% turns threshold of 15mV (i.e. 3uV noise)
                if((abs(mid)-abs(fst))>0.015 ...
```

```
                        | (abs(mid)-abs(lst))>0.015)
                            turns(frame)=turns(frame)+1;
                end
            end
% Compute Waveform Length
            len(frame) = len(frame) + ...
                sqrt(((fst-mid)/20.0)^2 + rulersq);
        end
    end

% Scale the features to normalize for the neural network
    zero_count = (zero_count./SCALE_ZC)*40/inc;
% scaling based on 40 ms
    mav = mav/SCALE_MAV;
    len = (len-1)./lscale;
    turns = turns/tscale;

    sd = std(dmav(:));
    Features(:,SigNum) = [mav(:)' len(:)' zero_count(:)' turns(:)'];
    Property(:,SigNum) = signallevel;
end
Feature_emg=[];
for kk=1:size(Features,2)
    Feature_emg=[Feature_emg;Features(:,kk)];
end
end
```

APPENDIX B: C++ REAL-TIME IMPLEMENTATION

RealTime_EMGandIMU.cpp:

This is the main file for the Real-time implementation of the EMG IMU mouse interface. It is the code unique to Setup A, Configuration 3. This file is organized in the following way: First, the LDA classifier values are loaded from text files that were created and exported from MatLab. Next, the A/D for the EMG channels is initialized, followed by the IMU device. Then our main loop initializes data collection, collects data, classifies the EMG data, and combines the output of both EMG and IMU to form our design.

Both the A/D and IMU initialization uses the API that comes with the devices. The code is based off the example files and documentation that also accompany the device and device software. The size of the buffer in the A/D must be allocated by specifying the number of channels and the window length size. Another FIFO buffer is set up, first in first out, to store each window increment up to the size of the window length, to be used later in classification. This is done by repeatedly calling a function that returns the index, the current number of samples collected, to know that another window increment has been collected, and it can be copied into the FIFO buffer. When the buffer is full the index wraps around to the beginning and continues collecting data. For each window increment where a full window length of data are available, the data are passed to the LDA classifier.

The feature extraction, LDA classifier, and majority vote function are written in C, with efficiency their focus. The code for feature extraction and the LDA classifier is shown below: EMG_PR.cpp. The size of the majority vote decision stream is set to 20. The decision outputted by the majority vote function for each classifier forms the EMG portion of the program.

The pertinent data from the IMU are the z-axis gyroscope and y-axis gyroscope, translating to the mouse y-coordinate and x-coordinate, respectively. The units of the IMU data are meters per second. The scaling factor used for the y-axis is 36, and x-axis 42. This forms the IMU portion of the program.

Combining the data from the LDA classifier and IMU device is done according to figure 4.5. IMU data is only accepted when the slope agrees with the EMG action. The regions of accepted values from the IMU overlap the other regions. The absolute value of the slope is computed and for actions Up and Down, slopes larger than 3/4 are accepted. For actions Left and Right, slopes less than 4/3 are accepted. This forms the EMG and IMU fusion portion of the program.

The mouse function calls and API used in the Real Time C++ implementation is that of Windows, using the Windows.h header file. When updating the x and y coordinates of the mouse, we use the API to get the current mouse coordinates. Doing this allows other mouse input devices and programs to be used concurrently, the same way commercial mice operate. The clicking and functionality operates as follows: The last clicking action is stored in order to know what the next appropriate action to be taken is. For example, if the action is left mouse click, the left mouse press function is

called. If the last action was a left mouse click and the next action is rest, then the left mouse release function is called. This concludes how the mouse functionality works.

For debugging purposes, a command prompt window displays the IMU output along with the LDA classifier decision. Some of the functions towards the end of the file set the command window cursor to enable printing over existing text, to avoid continuous scrolling as new data come in. When the program is terminated, the device API functions are called to release and free the A/D and IMU.

```
//     Tim Forbes
//     03-17-2013
//     EMG and IMU with click in C++

// General Includes
#include <iostream>
#include <fstream>
#include <Windows.h>
#include <stdio.h>
#include <string>
#include <list>
#include <iomanip>
#include <conio.h>
#include <fstream>
#include <deque>
#include <vector>
#include <sstream>

// IMU Includes
#include <objbase.h> // Needed for COM functionality
#include "xsens_cmt.h"

// EMG A/D includes
#include "stdafx.h"
#include "cbw.h"

// IMU function declarations
void doHardwareScan();
void doMtSettings(void);
void clrscr(void);
void writeHeaders(void);
void gotoxy(int x, int y);
void exitFunc(void);

// EMG function declarations
int LDA_test(float *TestData,int window_size, int channels, int classes, float
*Wg, float *Cg);

////////////////////////
// General variables //
////////////////////////

#define TRUE 1
#define FALSE 0

////////////////////////
// A/D API variables //
////////////////////////
```

```
#define NUMCHANS  5
#define NUMPOINTS 60
static const int WINDOW_LEN = NUMPOINTS;        // window size: 60 ms
static const int WINDOW_INC = 20;                       // window increment 20ms
static const int RATE = 1000;                   // sampling rate Hz (samples
per second)
int Row, Col, i;
int  BoardNum = 0;
int Options;
long PreTrigCount, TotalCount, Rate, ChanCount;
short ChanArray[NUMCHANS];
short ChanTypeArray[NUMCHANS];
short GainArray[NUMCHANS];
int ULStat = 0;
short Status = IDLE;
long CurCount;
long CurIndex=0, DataIndex=0;
WORD *ADData;
float RevLevel = (float)CURRENTREVNUM;
int LastBuffer=3, BufferAvailable=FALSE, FullWindowLengthAquired=FALSE;

///////////////////////////////////
// EMG classification variables //
///////////////////////////////////
static const int BUFFER_SIZE = NUMCHANS * WINDOW_INC;
static const int MAJORITY_VOTE_WIN = 20;

static float
Channel_1_Data[WINDOW_LEN],Channel_2_Data[WINDOW_LEN],Channel_3_Data[WINDOW_LEN
],Channel_4_Data[WINDOW_LEN],Channel_5_Data[WINDOW_LEN];
static float
WInc_Channel_1_Data[WINDOW_INC],WInc_Channel_2_Data[WINDOW_INC],WInc_Channel_3_
Data[WINDOW_INC],WInc_Channel_4_Data[WINDOW_INC],WInc_Channel_5_Data[WINDOW_INC
];
static float Feature_EMG[5*WINDOW_LEN];

std::string allMode[]    ={"Rest","Up","Down","Left","Right", "Left Click",
"Right Click"};
std::string last_action = "Rest";

// **Don't know the size of Wg exactly so just allocat a huge array
float *Cg_all = (float*)malloc(7*sizeof(float));
float *Wg_all = (float*)malloc(300*sizeof(float));

////////////////////////
// IMU API variables //
////////////////////////
#define KEY "5BCCB-41190-3720A-7FFAB"
#define EXIT_ON_ERROR(res,comment) if (res != XRV_OK) { printf("Error %d
occurred in " comment ": %s\n",res,cmtGetResultText(res)); exit(1); }
long instance = -1;
CmtOutputMode mode;
CmtOutputSettings settings;
unsigned short mtCount = 0;
CmtDeviceId deviceIds[256];
XsensResultValue res;
//structs to hold data.
CmtCalData caldata;
```

```
////////////////////////
// IMU cursor control variables //
////////////////////////
int errorCalc = FALSE;
int newx=0;
int newy=0;
float nextx=0;
float nexty=0;
float xSum=0;
float ySum=0;
int errorCounter = 0;
float xFixError = 0;
float yFixError = 0;
int SENSITIVITY = 12;
double yScale = 3.0 * SENSITIVITY;
double xScale = 3.5 * SENSITIVITY;
// For returning the current cursor coordinates
POINT cursorPos;
// Variables for sending the click function
INPUT Input={0};

////////////////////////////////////////////////////////////////////////
// MajorityVote
//
// Called from ClassifyEmgAndImu once we have our prediction
// Looks at a predefined number of previous class predictions and takes the
most common
//  This accounts for single wrong predictions that occur inside several
correct predictions
int MajorityVote(int *DecisionStream, int Nclass) {
// Microsoft Visual studio does not support VLA's (Variable length arrays) so I
have to hard code this
int classes[7] = {0,0,0,0,0,0,0};
int streamSize = sizeof(DecisionStream) / sizeof(int);
// Create array classes and tally them up
for (int j=0; j<streamSize; j++) {
classes[DecisionStream[j]] = classes[DecisionStream[j]] + 1;
}
int indexOfMax = 0;
int max = 0;
// Find the max
for (int j=0; j<Nclass; j++) {
if (classes[j] > max) {
indexOfMax = j;
max = classes[j];
}
}
return indexOfMax;
}

////////////////////////////////////////////////////////////////////////
// extract_numbers
//
// Helper method for ReadInLDAValues - parses strings of numbers into a vector
void extract_numbers(std::string input_str, float* num)
{
std::istringstream input(input_str);
```

```
std::string number;
int k = 0;
while (std::getline(input, number, ','))
{
std::istringstream iss(number);
float i;
iss >> i;
num[k] = i;
k++;
}
}

////////////////////////////////////////////////////////////////////////
// ReadInLDAValues
//
// Reads in the LDA values from a text file that we calculated during training
void ReadInLDAValues(std::string fileName, float *Cg, float *Wg) {
std::string stringLine;
std::ifstream infile;
infile.open(fileName);
while(!infile.eof())
{
getline(infile,stringLine);
if (stringLine == "Cg") {
while(!(stringLine == "Wg")) {
if(!(stringLine == "Cg")) {
extract_numbers(stringLine, Cg);
}
getline(infile,stringLine);
}
}

if (stringLine == "Wg") {
while(!infile.eof()) {
if(!(stringLine == "Wg")) {
extract_numbers(stringLine, Wg);
}
getline(infile,stringLine);
}
}
}
infile.close();
}

////////////////////////////////////////////////////////////////////////
// InitializeEmgSystem
//
// Overhead and Function calls that detect, configure and prepare some
variables for EMG A/D collection
int InitializeEmgSystem() {
/* Variable Declarations */

SetPriorityClass (GetCurrentProcess(),REALTIME_PRIORITY_CLASS);
SetThreadPriority (GetCurrentThread(),THREAD_PRIORITY_TIME_CRITICAL);

/* Declare UL Revision Level */
ULStat = cbDeclareRevision(&RevLevel);
```

```
ADData = (WORD*)cbWinBufAlloc(NUMPOINTS * NUMCHANS);
if (!ADData)    /* Make sure it is a valid pointer */
{
printf("\nout of memory\n");
exit(1);
}

/* Initiate error handling
Parameters:
PRINTALL :all warnings and errors encountered will be printed
DONTSTOP :program will continue even if error occurs.
Note that STOPALL and STOPFATAL are only effective in
Windows applications, not Console applications.
*/
cbErrHandling (PRINTALL, DONTSTOP);

/* load the arrays with values */
ChanArray[0] = 0;
ChanTypeArray[0] = ANALOG;
GainArray[0] = BIP5VOLTS;

ChanArray[1] = 1;
ChanTypeArray[1] = ANALOG;
GainArray[1] = BIP5VOLTS;

ChanArray[2] = 2;
ChanTypeArray[2] = ANALOG;
GainArray[2] = BIP5VOLTS;

ChanArray[3] = 3;
ChanTypeArray[3] = ANALOG;
GainArray[3] = BIP5VOLTS;

ChanArray[4] = 4;
ChanTypeArray[4] = ANALOG;
GainArray[4] = BIP5VOLTS;

return 0;
}

////////////////////////////////////////////////////////////////////////
// InitializeImuSystem
//
// Overhead and Function calls that detect, configure and then put the device
in measurement mode
int InitializeImuSystem() {

res = XRV_OK;

// Set exit function
atexit(exitFunc);

// lets create the Xsens CMT instance to handle the sensor(s)
printf("Creating an XsensCMT instance\n");

char serialNumber[] = KEY;
if (strcmp(serialNumber,"b8r6RCoGjQJVsytwUMo8WCRiJiVCCdoL11cCj4HqnaKPHtTn") ==
0)
```

```
printf("Warning: Using the demo key as a serial code will limit CMT
functionality to 1000 calls. Enter your own serial code for unlimited CMT
functionality.\n");

instance = cmtCreateInstance(serialNumber);
if (instance != -1)
printf("CMT instance created\n\n");
else {
printf("Creation of CMT instance failed, probably because of an invalid serial
number\n");
exit(1);
}

// Perform hardware scan
doHardwareScan();

//clear screen present & set the output mode.
clrscr();
mode = CMT_OUTPUTMODE_CALIB;

// Set device to mode input settings
doMtSettings();

// Wait for first data item(s) to arrive. In production code, you would use a
callback function instead (see cmtRegisterCallback function)
Sleep(20);

//get the placement offsets, clear the screen and write the fixed headers.
clrscr();
writeHeaders();

return 0;
}

//////////////////////////////////////////////////////////////////////////
// collectIMUdata
//
// Grabs the next packet of IMU data and grabs only the 2 values we want
//  We only want the Y and Z Gryoscope data, which translate to the X and Y
screen coordinates, respectively
// Averages the first 100 values to create a drift constant that we subtract
from every data value in the future, to account for drift in the device
//  **Means you need to keep the IMU device as still as possible during the
first 100 values (1 second)
void collectIMUdata() {

if (res == XRV_OK)
{
//get the bundle of data
res = cmtGetNextDataBundle(instance);

res = cmtDataGetCalData(instance, &caldata, deviceIds[0]);

gotoxy(0,4);
printf("%6.2f\t%6.2f", -caldata.m_gyr.m_data[2], -caldata.m_gyr.m_data[1] );
gotoxy(0,13);

// Grab the x and y values we want for the mouse cursor
```

```
// invert them to make the correspond to a graph, where Up-Left is positive for
both values
// We will adjust for up being a negative direction for a computer screen later
nextx = -caldata.m_gyr.m_data[2];
nexty = -caldata.m_gyr.m_data[1];
}

////////////////////////////////////////
// Average the 100 points to calculate the device drift
// Later the drift adjustment is subtracted from the measured values
if (!errorCalc) {
xSum = xSum + nextx;
ySum = ySum + nexty;
if (errorCounter==100) {
xFixError = xSum/errorCounter;
yFixError = ySum/errorCounter;
errorCalc = TRUE;
gotoxy(0, 6);
printf("xFixError = %f", xFixError);
gotoxy(0, 7);
printf("yFixError = %f", yFixError);
gotoxy(0,13);
}
gotoxy(0, 5);
printf("%d", errorCounter);
errorCounter++;
}
}

////////////////////////////////////////////////////////////////////////////
// ClassifyEmgAndImu
//
// Contains all code and local variables necessary for the actual reading in of
EMG A/D data and classification
void ClassifyEmgAndImu() {

////////////////////////////////////////////
// EMG data collection and classification //
////////////////////////////////////////////

// Classification local variables
int Decision_stream[MAJORITY_VOTE_WIN];
// set all to zero
for (i=0; i<MAJORITY_VOTE_WIN; i++) {
Decision_stream[i] = 0;
}
int decisionCounter = 0;
int TestPredict;
int Test_pro;

Input.type = INPUT_MOUSE;

/* Collect the values with cbDaqInScan() in BACKGROUND mode, CONTINUOUSLY
Parameters:
BoardNum    :the number used by CB.CFG to describe this board
ChanArray[] :array of channel values
ChanTypeArray[] : array of channel types
GainArray[] :array of gain values
```

```
ChanCount    :the number of channels
Rate         :sample rate in samples per second
PretrigCount:number of pre-trigger A/D samples to collect
TotalCount   :the total number of A/D samples to collect
ADData[]     :the array for the collected data values
Options      :data collection options  */
double MaxTime=0;

ChanCount = NUMCHANS;
PreTrigCount =0;
TotalCount = NUMPOINTS * NUMCHANS;
Rate = RATE;                                                              /*
sampling rate (samples per second) */
Options = CONVERTDATA + BACKGROUND + CONTINUOUS;         /* data collection
options */

// Function to start collecting EMG A/D data
ULStat = cbDaqInScan(BoardNum, ChanArray, ChanTypeArray, GainArray, ChanCount,
&Rate, &PreTrigCount, &TotalCount, ADData, Options);

if (ULStat == NOERRORS) {
Status = RUNNING;
}

/* During the BACKGROUND operation, check the status */
while (Status==RUNNING)
{
/* Check the status of the current background operation
// Parameters:
// BoardNum  :the number used by CB.CFG to describe this board
// Status    :current status of the operation (IDLE or RUNNING)
// CurCount  :current number of samples collected
// CurIndex  :index to the last data value transferred
// FunctionType: A/D operation (DAQIFUNCTION)*/

ULStat = cbGetStatus (BoardNum, &Status, &CurCount, &CurIndex,DAQIFUNCTION);

/* Check the current status of the background operation */
/* if the DAQ is RUNNING then get one window increment of data */
if (Status == RUNNING)
{
if (CurIndex>=BUFFER_SIZE-1 && CurIndex<2*BUFFER_SIZE-1 && LastBuffer==3)
{
LastBuffer = 1;
BufferAvailable = TRUE;

for (i=0;i<WINDOW_INC;i++)
{
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i], &WInc_Channel_1_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+1],
&WInc_Channel_2_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+2],
&WInc_Channel_3_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+3],
&WInc_Channel_4_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+4],
&WInc_Channel_5_Data[i]);
}
```

```
}

if (CurIndex>=2*BUFFER_SIZE-1 && CurIndex<3*BUFFER_SIZE-1 && LastBuffer==1)
{
LastBuffer = 2;
BufferAvailable = TRUE;

for (i=0;i<WINDOW_INC;i++)
{
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+BUFFER_SIZE],
&WInc_Channel_1_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+(BUFFER_SIZE+1)],
&WInc_Channel_2_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+(BUFFER_SIZE+2)],
&WInc_Channel_3_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+(BUFFER_SIZE+3)],
&WInc_Channel_4_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+(BUFFER_SIZE+4)],
&WInc_Channel_5_Data[i]);
}
}

if ((CurIndex>=0 && CurIndex<BUFFER_SIZE-1 && LastBuffer==2) ||
(CurIndex==3*BUFFER_SIZE-1 && LastBuffer==2))
{
LastBuffer = 3;
BufferAvailable = TRUE;
FullWindowLengthAquired = TRUE;

for (i=0;i<WINDOW_INC;i++)
{
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+(2*BUFFER_SIZE)],
&WInc_Channel_1_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+(2*BUFFER_SIZE+1)],
&WInc_Channel_2_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+(2*BUFFER_SIZE+2)],
&WInc_Channel_3_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+(2*BUFFER_SIZE+3)],
&WInc_Channel_4_Data[i]);
cbToEngUnits(BoardNum, BIP5VOLTS, ADData[NUMCHANS*i+(2*BUFFER_SIZE+4)],
&WInc_Channel_5_Data[i]);
}
}

if (BufferAvailable==TRUE)
{
BufferAvailable = FALSE;

//Shift data up by one window increment to clear space for new buffer
for (i=WINDOW_INC;i<NUMPOINTS;i++)
{ //20 goes into 0...59 goes into 39
Channel_1_Data[i-WINDOW_INC] = Channel_1_Data[i];
Channel_2_Data[i-WINDOW_INC] = Channel_2_Data[i];
Channel_3_Data[i-WINDOW_INC] = Channel_3_Data[i];
Channel_4_Data[i-WINDOW_INC] = Channel_4_Data[i];
Channel_5_Data[i-WINDOW_INC] = Channel_5_Data[i];
}
```

```
for (i=NUMPOINTS-WINDOW_INC;i<NUMPOINTS;i++)
{ //goes in from 40 to 59
Channel_1_Data[i] = WInc_Channel_1_Data[i-NUMPOINTS+WINDOW_INC];
Channel_2_Data[i] = WInc_Channel_2_Data[i-NUMPOINTS+WINDOW_INC];
Channel_3_Data[i] = WInc_Channel_3_Data[i-NUMPOINTS+WINDOW_INC];
Channel_4_Data[i] = WInc_Channel_4_Data[i-NUMPOINTS+WINDOW_INC];
Channel_5_Data[i] = WInc_Channel_5_Data[i-NUMPOINTS+WINDOW_INC];
}

/***************************************************/
/********** Perform Prediction *********************/
/***************************************************/

// IMU data collection
for  (i=0; i<(WINDOW_INC/10); i++) {
collectIMUdata();
}

for (i=0;i<WINDOW_LEN;i++)
{
Feature_EMG[5*i] = Channel_1_Data[i];
Feature_EMG[5*i+1] = Channel_2_Data[i];
Feature_EMG[5*i+2] = Channel_3_Data[i];
Feature_EMG[5*i+3] = Channel_4_Data[i];
Feature_EMG[5*i+4] = Channel_5_Data[i];
}

TestPredict = 1;

if (FullWindowLengthAquired) {
TestPredict = LDA_test(Feature_EMG, WINDOW_LEN, 5,
sizeof(allMode)/sizeof(allMode[0]), Wg_all, Cg_all);
}

// Build decision stream and -1 to adjust for...whatever
Decision_stream[decisionCounter] = TestPredict-1;
// modulus function: decisionCounter will count from 0 to MAJORITY_VOTE_WIN
repeatedly
decisionCounter = ++decisionCounter % MAJORITY_VOTE_WIN;

////////////// Majority Vote ////////////////
Test_pro = MajorityVote(Decision_stream, sizeof(allMode)/sizeof(allMode[0]));

////////////////////////////////////////////////
//////// Display Class Labels in Window ////////
////////////////////////////////////////////////

gotoxy(0, 11);
std::cout << allMode[Test_pro] << "\t\t";
gotoxy(0, 13);

//////////////////////////////////////
//////// Call Mouse Functions ////////
//////////////////////////////////////

// Compare last action to current to know which type of click to perform
// Double if statements covers the possibility of going straight from Left
Click to Right click
```

```
//  - Originally, if that transistion happened, then we would "right press
down", having never "left release up",
//            and clicks either wouldn't register or things start to get messed
up
if (last_action == "Left Click" && allMode[Test_pro] != "Left Click") {
Input.mi.dwFlags  = MOUSEEVENTF_LEFTUP;
SendInput( 1, &Input, sizeof(INPUT) );
last_action = "Rest";
}
else if (last_action == "Right Click" && allMode[Test_pro] != "Right Click") {
Input.mi.dwFlags  = MOUSEEVENTF_RIGHTUP;
SendInput( 1, &Input, sizeof(INPUT) );
last_action = "Rest";
}
if (last_action != "Left Click" && allMode[Test_pro] == "Left Click") {
Input.mi.dwFlags  = MOUSEEVENTF_LEFTDOWN;
SendInput( 1, &Input, sizeof(INPUT) );
last_action = "Left Click";
}
else if (last_action != "Right Click" && allMode[Test_pro] == "Right Click") {
Input.mi.dwFlags  = MOUSEEVENTF_RIGHTDOWN;
SendInput( 1, &Input, sizeof(INPUT) );
last_action = "Right Click";
}

// Adjust IMU data for drift
if (errorCalc == TRUE) {
nextx = nextx - xFixError;
nexty = nexty - yFixError;
}
// If too small set equal to zero
if (nextx<0.03 && nextx>-0.03) {
nextx = 0;
}
if (nexty<0.03 && nexty>-0.03) {
nexty = 0;
}
// If too large set equal to zero
if (nextx<-3 && nextx>-3) {
nextx = 0;
}
if (nexty<-3 && nexty>3) {
nexty = 0;
}

// Get the current positon before updating it
// This insures that another mouse device can be used at the same time (this is
how a normal mouse work)
GetCursorPos(&cursorPos);
newx = (int) cursorPos.x;
newy = (int) cursorPos.y;

float slope;
// Just to prevent the rare divide by zero error and set the slope arbitaritly
greater than 1 if so
if (nextx != 0) {
slope = abs(nexty/nextx);
}
```

```
else {
slope = 5;
}
// Code to only allow IMU cursor movement in a direction that agrees with the
EMG prediction
// In here we adjust for the direction up being a negative y value on a
computer screen
if (allMode[Test_pro] == "Up") {
if (nexty >= 0 && slope >= 3/4) {
newx = newx + nextx * xScale;
newy = newy - nexty * yScale;
}
}
else if (allMode[Test_pro] == "Down") {
if (nexty <= 0 && slope >= 3/4) {
newx = newx + nextx * xScale;
newy = newy - nexty * yScale;
}
}
else if (allMode[Test_pro] == "Left") {
if (nextx <= 0 && slope <= 4/3) {
newx = newx + nextx * xScale;
newy = newy - nexty * yScale;
}
}
else if (allMode[Test_pro] == "Right") {
if (nextx >= 0 && slope <= 4/3) {
newx = newx + nextx * xScale;
newy = newy - nexty * yScale;
}
}
// Make very small movements easy without a wrist direction
else if (allMode[Test_pro] == "Rest" || allMode[Test_pro] == "Left Click") {
if (nextx < 0.3 && nexty < 0.3) {
newx = newx + nextx * xScale/4;
newy = newy - nexty * yScale/4;
}
}

// Windows Mouse cursor function call
SetCursorPos(newx, newy);
nextx = 0;
nexty = 0;
//}
/*************************************************/
/******** End Perform Prediction ******************/
/*************************************************/
}//End If Buffer Available
}//End IF DAQ RUNNING
}//End IF KBHIT and RUNNNING

// Release any mouse action that is incomplete, such as left or right has been
pressed but not released yet
if (last_action == "Left Click") {
Input.mi.dwFlags  = MOUSEEVENTF_LEFTUP;
SendInput( 1, &Input, sizeof(INPUT) );
}
else if (last_action == "Right Click") {
```

```
Input.mi.dwFlags  = MOUSEEVENTF_RIGHTUP;
SendInput( 1, &Input, sizeof(INPUT) );
}

/* The BACKGROUND operation must be explicitly stopped
Parameters:
BoardNum    :the number used by CB.CFG to describe this board
FunctionType: A/D operation (DAQIFUNCTION)*/
ULStat = cbStopBackground (BoardNum,DAQIFUNCTION);

cbWinBufFree((HGLOBAL)ADData);
}

////////////////////////////////////////////////////////////////////////
// main
//
// Just calls other functions
int main(int argc, char* argv[])
{
ReadInLDAValues("Tim_LDA_all.txt",Cg_all, Wg_all);
InitializeEmgSystem();
InitializeImuSystem();
ClassifyEmgAndImu();

// If we get here we either exited by pressing a key or we got an error
printf("Data collection terminated.");

// Close IMU instance
cmtClose(instance);

free(Cg_all);
free(Wg_all);

return 0;
}

////////////////////////////////////////////////////////////////////////
// doHardwareScan
//
// Checks available COM ports and scans for MotionTrackers
void doHardwareScan()
{
XsensResultValue res;
CmtPortInfo portInfo[256];
uint32_t portCount = 0;

printf("Scanning for connected Xsens devices...");
res = cmtScanPorts(portInfo, &portCount, 0);
EXIT_ON_ERROR(res,"cmtScanPorts");
printf("done\n");

if (portCount == 0) {
printf("No MotionTrackers found\n\n");
exit(0);
}

for(int i = 0; i < (int)portCount; i++) {
printf("Using COM port %d at %d baud\n\n",
```

```
(long) portInfo[i].m_portNr, portInfo[i].m_baudrate);
}

printf("Opening ports...");
//open the port which the device is connected to and connect at the device's
baudrate.
for(int p = 0; p < (int)portCount; p++){
res = cmtOpenPort(instance, portInfo[p].m_portNr, portInfo[p].m_baudrate);
EXIT_ON_ERROR(res,"cmtOpenPort");
}
printf("done\n\n");

//get the Mt sensor count.
printf("Retrieving MotionTracker count (excluding attached Xbus Master(s))\n");
res = cmtGetMtCount(instance,&mtCount);
EXIT_ON_ERROR(res,"cmtGetMtCount");
printf("MotionTracker count: %i\n\n",mtCount);

// retrieve the device IDs
printf("Retrieving MotionTrackers device ID(s)\n");
for(unsigned int j = 0; j < mtCount; j++ ){
res = cmtGetMtDeviceId(instance, &deviceIds[j], j);
EXIT_ON_ERROR(res,"cmtGetDeviceId");
printf("Device ID at index %i: %08x\n",j,(long) deviceIds[j]);
}

// make sure that we get the freshest data
printf("\nSetting queue mode so that we always get the latest data\n\n");
res = cmtSetQueueMode(instance,CMT_QM_LAST);
EXIT_ON_ERROR(res,"cmtSetQueueMode");
}

////////////////////////////////////////////////////////////////////////////
// doMTSettings
//
// Set user settings in MTi/MTx
// Assumes initialized global MTComm class
void doMtSettings(void)
{
XsensResultValue res;

// set sensor to config sate
res = cmtGotoConfig(instance);
EXIT_ON_ERROR(res, "cmtGotoConfig");

unsigned short sampleFreq;
res = cmtGetSampleFrequency(instance, &sampleFreq, deviceIds[0]);

// set the device output mode for the device(s)
printf("Configuring your mode selection");
for (int i=0; i < mtCount; i++) {
if (cmtIdIsMtig(deviceIds[i])) {
res = cmtSetDeviceMode(instance, mode,settings, sampleFreq, deviceIds[i]);
} else {
res = cmtSetDeviceMode(instance, mode & 0xFF0F, settings, sampleFreq,
deviceIds[i]);
}
EXIT_ON_ERROR(res, "setDeviceMode");
```

```
}

// start receiving data
res = cmtGotoMeasurement(instance);
EXIT_ON_ERROR(res, "cmtGotoMeasurement");
}

//////////////////////////////////////////////////////////////////////////
// writeHeaders
//
// Write appropriate headers to screen
void writeHeaders()
{
gotoxy(0,0);
printf ("Press any key to quit.");
gotoxy(0,1);
printf("IMU data: x and y angular velocity");
gotoxy(0, 3);
printf("Gyr X\t Gyr Y");
gotoxy(17, 4);
printf("(rad/s)");
gotoxy(0, 10);
printf("Class\t");
gotoxy(0, 13);
}

//////////////////////////////////////////////////////////////////////////
// clrscr
//
// Clear console screen
void clrscr()
{
#ifdef WIN32
CONSOLE_SCREEN_BUFFER_INFO csbi;
HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
COORD coord = {0, 0};
DWORD count;

GetConsoleScreenBufferInfo(hStdOut, &csbi);
FillConsoleOutputCharacter(hStdOut, ' ', csbi.dwSize.X * csbi.dwSize.Y, coord,
&count);
SetConsoleCursorPosition(hStdOut, coord);
#else
int i;

for (i = 0; i < 100; i++)
// Insert new lines to create a blank screen
putchar('\n');
gotoxy(0,0);
#endif
}

//////////////////////////////////////////////////////////////////////////
// gotoxy
//
// Sets the cursor position at the specified console position
//
// Input
```

```
//      x     : New horizontal cursor position
//   y : New vertical cursor position
void gotoxy(int x, int y)
{
#ifdef WIN32
COORD coord;
coord.X = x;
coord.Y = y;
SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);
#else
char essq[100];             // String variable to hold the escape sequence
char xstr[100];             // Strings to hold the x and y coordinates
char ystr[100];             // Escape sequences must be built with characters

/*
** Convert the screen coordinates to strings
*/
sprintf(xstr, "%d", x);
sprintf(ystr, "%d", y);

/*
** Build the escape sequence (vertical move)
*/
essq[0] = '\0';
strcat(essq, "\033[");
strcat(essq, ystr);

/*
** Described in man terminfo as vpa=\E[%p1%dd
** Vertical position absolute
*/
strcat(essq, "d");

/*
** Horizontal move
** Horizontal position absolute
*/
strcat(essq, "\033[");
strcat(essq, xstr);
// Described in man terminfo as hpa=\E[%p1%dG
strcat(essq, "G");

/*
** Execute the escape sequence
** This will move the cursor to x, y
*/
printf("%s", essq);
#endif
}

////////////////////////////////////////////////////////////////////////////
// exitFunc
//
// Closes cmt nicely
void exitFunc(void)
{
// if we have a valid instance, we should get rid of it at the end of the
program
```

```
if (instance != -1) {
// Close any open COM ports
cmtClose(instance);
cmtDestroyInstance(instance);
}
}
```

EMG_PR.cpp:

This file contains the C implementation of the time domain feature extraction, LDA training algorithm, and LDA test algorithm. It also contains all the matrix manipulation functions necessary for those algorithms. Credit: Xiaorong Zhang.

```
// EMG_PR.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdlib.h>
#include <string.h>
#include"time.h"
#include "math.h"
#include "common.h"


float mean(float *src, int length)
{
    int i;
    float mean=0;
    float sum=0;
    for(i=0;i<length;i++)
    {
        sum+=*src;
        src++;
    }
    mean = sum/length;
    return mean;
}


void swap (float *a,float *b)
{
    float c=*a;
    *a=*b;
    *b=c;
}

int compare_float(float f1, float f2)
{
    float precision = 0.000001f;
    if (((f1 - precision) < f2) && ((f1 + precision) > f2))
    {
        return 1;//same
    }
```

```
    else
    {
        return 0;//different
    }
}
int inv(float *A,int n)
{
    int i,j,k;
    float d;
    int *JS,*IS;
    JS = (int*)malloc(n*sizeof(int));
    IS = (int*)malloc(n*sizeof(int));
    for (k=0;k<n;k++)
    {
        d=0;
        for (i=k;i<n;i++)
            for (j=k;j<n;j++)
            {
                if (fabs(A[i*n+j])>d)
                {
                    d=(float)fabs(A[i*n+j]);
                    IS[k]=i;
                    JS[k]=j;
                }
            }

        if(d+1.0==1.0)
        {
            goto out1;
        }
        if (IS[k]!=k)
            for (j=0;j<n;j++)
                swap(&A[k*n+j],&A[IS[k]*n+j]);

        if (JS[k]!=k)
            for (i=0;i<n;i++)
                swap(&A[i*n+k],&A[i*n+JS[k]]);

        A[k*n+k]=1/A[k*n+k];
        for (j=0;j<n;j++)
            if (j!=k) A[k*n+j]=A[k*n+j]*A[k*n+k];

       for (i=0;i<n;i++)
            if (i!=k)
                for (j=0;j<n;j++)
                    if (j!=k) A[i*n+j]=A[i*n+j]-A[i*n+k]*A[k*n+j];

        for (i=0;i<n;i++)
            if (i!=k) A[i*n+k]=-A[i*n+k]*A[k*n+k];

    }
    for (k=n-1;k>=0;k--)
    {
        for (j=0;j<n;j++)
            if (JS[k]!=k) swap(&A[k*n+j],&A[JS[k]*n+j]);
            for (i=0;i<n;i++)
                if (IS[k]!=k) swap(&A[i*n+k],&A[i*n+IS[k]]);
    }
```

```
out1:
    free(JS);
    free(IS);
    return 0;
}

void mulAB(float *A,float *B,float *C,int am,int an,int bn)
{
    int i,j,l,u;

    for (i=0; i<am; i++)
        for (j=0; j<bn; j++)
        {
            u=i*bn+j; C[u]=0.0;
            for (l=0; l<an; l++)
                C[u]=C[u]+A[i*an+l]*B[l*bn+j];
        }
    return;
}

void Transpose(float *A,float *B,int m,int n)
{
    int i,j;

    for (j=0; j<n; j++)
        for (i=0; i<m; i++)
          {
              B[j*m+i]=A[i*n+j];
          }
    return;
}

void transpose_1(float *f, int row, int col)
{
    int i,j;
    float *tmp = (float*)malloc(row*col*sizeof(float));
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        {
            tmp[row*j+i] = f[col*i+j];
        }
    }
    memcpy(f,tmp,row*col*sizeof(float));
    free(tmp);
}

void addition(float *A,float *B,float *C, int m,int n)
{
    int i,j;

    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
          {
              C[i*m+j]=A[i*m+j]+B[i*m+j];
          }
    return;
```

```
}
void substract(float *A,float *B,float *C, int m,int n)
{
    int i,j;

    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
          {
              C[i*m+j]=A[i*m+j]-B[i*m+j];
          }
    return;
}

//Covariance matrix, A:input, B:output
void cov(float *A,float *B,int row,int col)
{
    int i,j;
    float *tmp;
    float *tmp1;
    float *t_tmp1;
    float sum=0;
    tmp=(float*)malloc(col*sizeof(float));
    tmp1=(float*)malloc(row*col*sizeof(float));
    t_tmp1=(float*)malloc(col*row*sizeof(float));

    memset(tmp,0,col*sizeof(float));
    memset(tmp1,0,row*col*sizeof(float));
    memset(t_tmp1,0,row*col*sizeof(float));
    //calculate mean of each column
    for (i=0; i<col; i++)
    {
        sum = 0;
        for (j=0; j<row; j++)
        {
            sum+=A[j*col+i];
        }
        tmp[i]=sum/row;
              //printf("%f\n",tmp[i]);
    }

    //substract matrix A with mean
    for (i=0; i<col; i++)
    {
        for (j=0; j<row; j++)
        {
            tmp1[j*col+i]=A[j*col+i]-tmp[i];
            //float tmptmp = tmp1[j*n+i];
        }
    }
    //multiply tmp1 with transpose of tmp1
    Transpose(tmp1,t_tmp1,row,col);
    mulAB(t_tmp1,tmp1,B,col,row,col);
    //calculate mean of each element of B
    for (i=0; i<col; i++)
    {
        for (j=0; j<col; j++)
        {
            B[j*col+i]=B[j*col+i]/(row-1);
```

```
        }
    }
    free(tmp);
    free(tmp1);
    free(t_tmp1);
    return;
}

//calculate compet of transpose matrix of A
void compet_t(float *A, float *B, int m, int n)
{
    float *tmp;
    int idx;
    float max;
    int i,j;
    tmp=(float*)malloc(m*n*sizeof(float));
    for(i=0;i<m;i++)
    {
        max=0;
        idx=0;
        for(j=0;j<n;j++)
        {
            if(A[i*n+j]>max)
            {
                max=A[i*n+j];
                idx=i*n+j;
            }
            tmp[i*n+j]=0;
        }
        tmp[idx]=1;
    }
    Transpose(tmp,B,m,n);
    free(tmp);
}

void ind2vec(int *A, float *B,int m, int n)
{
    int i,j;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            B[i*n+j]=0;
        }
    }
    for(i=0;i<n;i++)
    {
        B[(int)(A[i]*n+i)]=1.0;
    }

}

void vec2ind(float *A, float *B,int m, int n)
{
    int i,j;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
```

```
        {
            if(A[i*n+j]==1.0)
                B[j]=(float)i;
        }
    }
}
//------------- Variables -------------------------------------------------

/*
 % This function is for the computation of only one channel
 %input: DataSet(Ntotal X # of windows) e.g.(60X100) , Nframe 1, inc Ntotal
 %output: Features(# of features in one channel X # of windows) e.g. (4X100)
 % Run this function for 16 times (16 channels) to form a 64X100 matrix as
 % the input of LDA
 */
//void tdfeats_H(float *DataSet,int DataSet_Row,int DataSet_Column, int
Nframe,float *v_zero_count, float *v_mav, float *v_len, float *v_turns)
void tdfeats_H(float *DataSet1,int DataSet_Row,int DataSet_Column, int Nframe,
float *Features)
{
    int TRAINDATA_ROW = 4*DataSet_Column;

       float ruler=0;
    float rulersq;
    float lscale;
    float tscale;
    int inc;

    int Ntotal;
    int Nsig;

    int i,j;
    float sum=0;
    float *mean;
       float *DataSet;

    float zero_count;
    float len;
    float mav;
    float turns;
    int index;
    int range;
    float sum_sig;
    int flag1;
    int flag2;
    int idx;
    float fst;
    float mid;
    float lst;
    int SigNum;

    inc=DataSet_Row;
    ruler=1/(float)inc;
    rulersq=ruler*ruler;
    lscale=(float)inc/40;
    tscale=((float)inc/40)*10;
```

```
    Ntotal=DataSet_Row;
    Nsig=DataSet_Column;


    mean=(float*)malloc(1*DataSet_Column*sizeof(float));
       DataSet=(float*)malloc(1*DataSet_Row*DataSet_Column*sizeof(float));

    for(i=0;i<DataSet_Column;i++)
    {
        sum=0;
        for(j=0;j<DataSet_Row;j++)
        {
            sum=sum+DataSet1[j*DataSet_Column+i];
        }
        mean[i]=sum/DataSet_Row;//mean of a sigal (column)
    }

    for(i=0;i<DataSet_Row;i++)
    {
        for(j=0;j<DataSet_Column;j++)
        {
            DataSet[i*DataSet_Column+j]=DataSet1[i*DataSet_Column+j]-mean[j];
        }
    }

    free(mean);

    for(SigNum=0;SigNum<Nsig;SigNum++)
    {
        zero_count=0;
        len=0;
        mav=0;
        turns=0;

        index=0;
        sum_sig=0;
        for(range=index;range<index+inc;range++)
        {

sum_sig=sum_sig+(float)fabs(DataSet[range*DataSet_Column+SigNum]);//sum of a
signal (a column)
        }
        mav=sum_sig/inc;//mean of a signal

        flag1=1;
        flag2=1;
        for(i=1;i<inc-1;i++)//inc=DATASET_ROW
        {
            idx=i;    //idx=2:DATASET_ROW-1
            fst=(float)fabs(DataSet[(idx-
1)*DataSet_Column+SigNum]);//DataSet[0,0]:DataSet[DATASET_ROW-3,0]
            mid=(float)fabs(DataSet[(idx)*DataSet_Column+SigNum]);
//DataSet[1,0]:DataSet[DATASET_ROW-2,0]

lst=(float)fabs(DataSet[(idx+1)*DataSet_Column+SigNum]);//DataSet[2,0]:DataSet[
DATASET_ROW-1,0]

            //% Compute Zero Crossings
```

```
            if(((DataSet[(idx)*DataSet_Column+SigNum]>=0) &&(DataSet[(idx-
1)*DataSet_Column+SigNum]>=0))
                ||((DataSet[(idx)*DataSet_Column+SigNum]<=0) &&(DataSet[(idx-
1)*DataSet_Column+SigNum]<=0)))
            {
                flag1=flag2;//if
(DataSet[1,0]>=0&&DataSet[0,0]>=0)||(DataSet[1,0]<=0&&DataSet[0,0]<=0)
            }else
            {
                if((mid<DEADZONE)&&(fst<DEADZONE))//approximately zero
                {
                    flag1=flag2;
                }
                else
                {
                    flag1=(-1)*flag2;
                }
            }
            if(flag1!=flag2)
            {
                zero_count=zero_count+1;
            }
            //% Compute Turns (Slope Changes
            if(((mid>fst)&&(mid>lst))||((mid<fst)&&(mid<lst)))
            {
                //% turns threshold of 15mV (i.e. 3uV noise)
                if((fabs(mid)-fabs(fst))>DEADZONE_TURN||(fabs(mid)-
fabs(lst))>DEADZONE_TURN)
                {
                    turns=turns+1;
                }
            }
            //% Compute Waveform Length
            //len=len+(float)sqrt(((fst-mid)/20.0)*((fst-mid)/20.0)+rulersq);
            len=len+(float)sqrt(((fst-mid)/20.0)*((fst-
mid)/20.0)+rulersq);//rulersq=(1/DATASET_ROW)^2
        }

        //% Scale the features to normalize for the neural network

        zero_count=(zero_count/SCALE_ZC)*40/inc;

        //% scaling based on 40 ms

        mav=mav/SCALE_MAV;
        len=(len-1)/lscale;
        turns=turns/tscale;
        Features[4*DataSet_Column*Nframe+SigNum] = mav;
        Features[4*DataSet_Column*Nframe+DataSet_Column+SigNum] = len;
        Features[4*DataSet_Column*Nframe+DataSet_Column*2+SigNum] = zero_count;
        Features[4*DataSet_Column*Nframe+DataSet_Column*3+SigNum] = turns;

    }
       free(DataSet);
}


/*
```

```
% input: TrainData(combined 16 Feature matrices e.g. 4X100)e.g. 64X100
%        TestData (one window) e.g. 64X1
%        TrainClass 1X100
%        TestClass 1X1
*/
void LDA_train(float *TrainData,int *TrainClass, float * Wg, float * Cg, int*
length, int classes, int channels)
{
       int TRAINDATA_ROW = 4*channels;
    int i,j,n;

    float stdd;

    int K;

    int *idx;   //????
    int idx_length;
    float *Mi;
    float *C;
    float *cov_matrix;
    float sum;
    float *tmp_matrix;
    int m;

    float Pphi;

    float *Cinv;

    float *Mi_i;
    float *Wg_i;
    float *tmp2;
    float *tmp3;

    float tmpc;
    srand(1);

    K= classes;

    idx=(int*)malloc(WINDOW_NUM*sizeof(int));
    Mi=(float*)malloc(TRAINDATA_ROW*K*sizeof(float));
    C=(float*)malloc(TRAINDATA_ROW*TRAINDATA_ROW*sizeof(float));
    tmp_matrix=(float*)malloc(TRAINDATA_ROW*WINDOW_NUM*sizeof(float));
    cov_matrix=(float*)malloc(TRAINDATA_ROW*TRAINDATA_ROW*sizeof(float));

    memset(C,0,TRAINDATA_ROW*TRAINDATA_ROW*sizeof(float));
    memset(Mi,0,TRAINDATA_ROW*K*sizeof(float));
    memset(tmp_matrix,0,TRAINDATA_ROW*WINDOW_NUM*sizeof(float));

    for(i=0;i<K;i++)
    {
        memset(tmp_matrix,0,TRAINDATA_ROW*WINDOW_NUM*sizeof(float));
        //mean(TrainClass,idx);
        //calculation for one class
        for(m=0;m<TRAINDATA_ROW;m++)
        {
            sum=0;
            j=0;
            while(j<length[i])
```

```
            {
                sum+=TrainData[m+j*TRAINDATA_ROW+i*length[i]*TRAINDATA_ROW];
                j++;

            }
            Mi[m*K+i]=sum/length[i];
            j=0;
            while(j<length[i])
            {

tmp_matrix[m+j*TRAINDATA_ROW]=TrainData[m+j*TRAINDATA_ROW+i*length[i]*TRAINDATA
_ROW]-Mi[m*K+i];
                j++;
            }
        }

        cov(tmp_matrix,cov_matrix,length[i],TRAINDATA_ROW);
        addition(C,cov_matrix,C,TRAINDATA_ROW, TRAINDATA_ROW);

    }
    //printf( "lda 2\n");
    //C = C./K;
    for (i=0; i<TRAINDATA_ROW; i++)
    {
        for (j=0; j<TRAINDATA_ROW; j++)
        {
            C[j*TRAINDATA_ROW+i]=C[j*TRAINDATA_ROW+i]/K;

            tmpc = C[j*TRAINDATA_ROW+i];
        }
    }
    for(i=0;i<4;i++)
    {
        printf("\nC[%d]\n",i);
        for(j=0;j<4;j++)
            printf("%.3g\t",C[i*TRAINDATA_ROW+j]);
        printf("\n");
    }

    Pphi=1/(float)K;

    Cinv=(float*)malloc(TRAINDATA_ROW*TRAINDATA_ROW*sizeof(float));
    memcpy(Cinv,C,TRAINDATA_ROW*TRAINDATA_ROW*sizeof(float));
    //addition(C,C, Cinv,TRAINDATA_ROW,TRAINDATA_ROW);
    inv(Cinv,TRAINDATA_ROW);
    for(i=0;i<4;i++)
    {
        printf("\nCinv[%d]\n",i);
        for(j=0;j<4;j++)
            printf("%.3g\t",Cinv[i*TRAINDATA_ROW+j]);
        printf("\n");
    }

    free(tmp_matrix);
    free(cov_matrix);

    Mi_i=(float*)malloc(TRAINDATA_ROW*1*sizeof(float));
    Wg_i=(float*)malloc(1*TRAINDATA_ROW*sizeof(float));
```

```
    tmp2=(float*)malloc(1*TRAINDATA_ROW*sizeof(float));
    tmp3=(float*)malloc(1*1*sizeof(float));

    mulAB(Cinv,Mi,Wg,TRAINDATA_ROW,TRAINDATA_ROW,K);

    for(i=0;i<K;i++)
    {
        for(j=0;j<TRAINDATA_ROW;j++)
        {
            Wg_i[j]=Wg[j*K+i];
                    Mi_i[j]=Mi[j*K+i];
        }
        //Transpose(Mi_i,tmp1,TRAINDATA_ROW,1);
        //mulAB(tmp1,Cinv,tmp2,1,TRAINDATA_ROW,TRAINDATA_ROW,TRAINDATA_ROW);
        mulAB(Mi_i,Wg_i,tmp3,1,TRAINDATA_ROW,1);
        //Cg[i]=(float)((-1/2)*tmp3[0]+log(Pphi));
        Cg[i]=(float)((-0.5)*tmp3[0]);
    }


    //printf( "lda 4\n");
    free(idx);
    free(Mi);
    free(C);
    free(Cinv);

    free(Mi_i);
    free(Wg_i);
    free(tmp2);
    free(tmp3);

}

int LDA_test(float *TestData,int window_size, int channels, int classes, float
*Wg, float *Cg)
{
       int TRAINDATA_ROW = 4*channels;
       int i,j,m;
       float *Feature_test;
       float *tmp;
       float *tmp1;
       float maxdata=-999999.0;
       int test_decision = 1;

       Feature_test = (float*)malloc(4*channels*classes*sizeof(float));
       tmp=(float*)malloc(1*classes*sizeof(float));
       tmp1=(float*)malloc(1*classes*sizeof(float));

       tdfeats_H(TestData,window_size,channels, 0, Feature_test);
       // Not needed for LDA
       //feature_normalization_apply(Feature_test,xmean,xstd);
       mulAB(Feature_test,Wg,tmp,1,TRAINDATA_ROW,classes);
       addition(Cg,tmp,tmp1,1,classes);

       for(j=0;j<classes;j++){
              if(tmp1[j]>maxdata)
              {
                     maxdata=tmp1[j];
```

```
                    test_decision=j+1;
                }
        }

        free(Feature_test);
        free(tmp);
        free(tmp1);
        return test_decision;
}
```

common.h:

Header file for EMG_PR.cpp

```
#pragma once

#define DEADZONE     0.025//0.025
#define DEADZONE_TURN       0.015//0.015
#define SCALE_ZC     15
#define SCALE_MAV    2

#define MAV_SIZE     100
#define DATASET_ROW         60//120                    //Ntotal
#define DATASET_COLUMN    3//100          //# of windows
//#define CHANNEL           2
#define WINDOW_SIZE 60
#define WINDOW_NUM  500
#define CLASS 3
```

APPENDIX C: MOUSE ACCURACY PROGRAM

Mouse_accuracy_program.cpp

This is the single file for the C++ Mouse Accuracy Program. The Windows Win32 API is used to generate the graphic user interface. The program creates structures defined as a circle, containing the radius, coordinates of the origin, and the color. Every circle is drawn to the screen, along with text instructions and a status bar. When the user first clicks inside the middle circle, the timer starts. The order in which the circles are to be clicked is predefined. If the user clicks outside the correct circle, then the counter of incorrect clicks is incremented. When the user clicks inside the correct circle, it turns green. In the case of the center circle, it turns back to black once the next circle is clicked. When the center circle is clicked for the last time, the timer stops. Throughout the process the timer counts like a stopwatch. To be more precise in knowing if the user has clicked inside a circle, rather than checking to see if the cursor is within the square bounds of size 2 times the radius, the distance of the cursor from the center of the circle is computed and compared against the radius. If the distance is less than the radius, then the user has clicked inside the circle. There are also options in the menu bar to reset the program, exit, or display an about message box. The program generates the graphical interface shown in figure 3.10.

```
// Mouse_accuracy_program.cpp
//

#include "stdafx.h"
#include "Mouse_accuracy_program.h"

#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst;                                                    // current
instance
TCHAR szTitle[MAX_LOADSTRING];                                  // The title bar
text
TCHAR szWindowClass[MAX_LOADSTRING];                    // the main window class
name
LPCWSTR programTitle = L"Mouse Accuracy Program";

// Forward declarations of functions included in this code module:
ATOM                        MyRegisterClass(HINSTANCE hInstance);
BOOL                        InitInstance(HINSTANCE, int);
LRESULT CALLBACK     WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK     About(HWND, UINT, WPARAM, LPARAM);
void Reset(HWND);
void InsideCircle(HDC hdc, int iPosX, int iPosY);

HWND hStatus;
HWND hWnd;

HBRUSH blackBrush = CreateSolidBrush(RGB(0, 0, 0));
```

```
HBRUSH greenBrush = CreateSolidBrush(RGB(0, 255, 0));

int timerStarted = 0;
int correctClicks = 0;
int wrongClicks = 0;

std::string stemp; // Temporary buffer for converting ints to strings
std::wstring wstemp; // Temporary buffer for converting strings to LPCWSTR
strings
LPCWSTR result;

const int ID_TIMER = 1;
unsigned int startTime;
unsigned int stopwatch;
unsigned int milliseconds;
unsigned int seconds;

struct Circle {
	int x;
	int y;
	int radius;
	std::string color;
};

Circle center;
Circle top;
Circle bottom;
Circle left;
Circle right;
int RADIUS = 10;

const int BALL_MOVE_DELTA = 2;

typedef struct _BALLINFO
{
	int width;
	int height;
	int x;
	int y;

	int dx;
	int dy;
}BALLINFO;

BALLINFO g_ballInfo;
HBITMAP g_hbmBall = NULL;
HBITMAP g_hbmMask = NULL;

std::wstring s2ws(const std::string& s)
{
 int len;
 int slength = (int)s.length() + 1;
 len = MultiByteToWideChar(CP_ACP, 0, s.c_str(), slength, 0, 0);
 wchar_t* buf = new wchar_t[len];
 MultiByteToWideChar(CP_ACP, 0, s.c_str(), slength, buf, len);
 std::wstring r(buf);
 delete[] buf;
 return r;
```

```
}

void InitCircles()
{
	int windowWidth = 820;
	int windowHeight = 650;
	int DISTANCEFROMCENTER = 250;

	center.x = windowWidth/2;
	center.y = windowHeight/2;
	center.radius = RADIUS;
	center.color = "Black";
	top.x = windowWidth/2;
	top.y = windowHeight/2-DISTANCEFROMCENTER;
	top.radius = RADIUS;
	top.color = "Black";
	bottom.x = windowWidth/2;
	bottom.y = windowHeight/2+DISTANCEFROMCENTER;
	bottom.radius = RADIUS;
	bottom.color = "Black";
	left.x = windowWidth/2-DISTANCEFROMCENTER;
	left.y = windowHeight/2;
	left.radius = RADIUS;
	left.color = "Black";
	right.x = windowWidth/2+DISTANCEFROMCENTER;
	right.y = windowHeight/2;
	right.radius = RADIUS;
	right.color = "Black";
}

void DrawCircles(HDC hdc)
{
	// Draw All Circles and accompanying text
	SetBkMode(hdc, TRANSPARENT);
	// Center
	if (center.color == "Green")
		SelectObject(hdc, greenBrush);
	else
		SelectObject(hdc, blackBrush);
	Ellipse(hdc, center.x-RADIUS, center.y-RADIUS, center.x+RADIUS,
center.y+RADIUS);
	TextOut(hdc, center.x+2*RADIUS, center.y-2*RADIUS, TEXT("1,3,5,7,9"), 9);
	// Top
	if (top.color == "Green")
		SelectObject(hdc, greenBrush);
	else
		SelectObject(hdc, blackBrush);
	Ellipse(hdc, top.x-RADIUS, top.y-RADIUS, top.x+RADIUS, top.y+RADIUS);
	TextOut(hdc, top.x+2*RADIUS, top.y-2*RADIUS, TEXT("2"), 1);
	// Bottom
	if (bottom.color == "Green")
		SelectObject(hdc, greenBrush);
	else
		SelectObject(hdc, blackBrush);
	Ellipse(hdc, bottom.x-RADIUS, bottom.y-RADIUS, bottom.x+RADIUS,
bottom.y+RADIUS);
	TextOut(hdc, bottom.x+2*RADIUS, bottom.y-2*RADIUS, TEXT("6"), 1);
	// Left
```

```
	if (left.color == "Green")
		SelectObject(hdc, greenBrush);
	else
		SelectObject(hdc, blackBrush);
	Ellipse(hdc, left.x-RADIUS, left.y-RADIUS, left.x+RADIUS, left.y+RADIUS);
	TextOut(hdc, left.x+2*RADIUS, left.y-2*RADIUS, TEXT("4"), 1);
	// Right
	if (right.color == "Green")
		SelectObject(hdc, greenBrush);
	else
		SelectObject(hdc, blackBrush);
	Ellipse(hdc, right.x-RADIUS, right.y-RADIUS, right.x+RADIUS,
right.y+RADIUS);
	TextOut(hdc, right.x+2*RADIUS, right.y-2*RADIUS, TEXT("8"), 1);

}

int APIENTRY _tWinMain(_In_ HINSTANCE hInstance,
                       _In_opt_ HINSTANCE hPrevInstance,
                       _In_ LPTSTR    lpCmdLine,
                       _In_ int       nCmdShow)
{
	UNREFERENCED_PARAMETER(hPrevInstance);
	UNREFERENCED_PARAMETER(lpCmdLine);

	MSG msg;
	HACCEL hAccelTable;
	InitCircles();

	// Initialize global strings
	LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
	LoadString(hInstance, IDC_MOUSE_ACCURACY_PROGRAM, szWindowClass,
MAX_LOADSTRING);
	MyRegisterClass(hInstance);

	// Perform application initialization:
	if (!InitInstance (hInstance, nCmdShow))
	{
		return FALSE;
	}

	hAccelTable = LoadAccelerators(hInstance,
MAKEINTRESOURCE(IDC_MOUSE_ACCURACY_PROGRAM));

	// Main message loop:
	while (GetMessage(&msg, NULL, 0, 0))
	{
		if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
		{
			TranslateMessage(&msg);
			DispatchMessage(&msg);
		}
	}

	return (int) msg.wParam;
}

//
```

```
//  FUNCTION: MyRegisterClass()
//
//  PURPOSE: Registers the window class.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
       WNDCLASSEX wcex;

       wcex.cbSize = sizeof(WNDCLASSEX);

       wcex.style                  = CS_HREDRAW | CS_VREDRAW;
       wcex.lpfnWndProc     = WndProc;
       wcex.cbClsExtra             = 0;
       wcex.cbWndExtra             = 0;
       wcex.hInstance              = hInstance;
       wcex.hIcon                  = LoadIcon(hInstance,
MAKEINTRESOURCE(IDI_MOUSE_ACCURACY_PROGRAM));
       wcex.hCursor         = LoadCursor(NULL, IDC_ARROW);
       wcex.hbrBackground   = (HBRUSH)(COLOR_WINDOW);
       wcex.lpszMenuName    = MAKEINTRESOURCE(IDC_MOUSE_ACCURACY_PROGRAM);
       wcex.lpszClassName   = szWindowClass;
       wcex.hIconSm         = LoadIcon(wcex.hInstance,
MAKEINTRESOURCE(IDI_SMALL));

       return RegisterClassEx(&wcex);
}

//
//   FUNCTION: InitInstance(HINSTANCE, int)
//
//   PURPOSE: Saves instance handle and creates main window
//
//   COMMENTS:
//
//        In this function, we save the instance handle in a global variable
and
//        create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   //HWND hWnd;

   hInst = hInstance; // Store instance handle in our global variable

   hWnd = CreateWindow(szWindowClass, programTitle, WS_OVERLAPPEDWINDOW,
      100, 50, 820, 700, NULL, NULL, hInstance, NULL);

   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}
```

```
//
//  FUNCTION: WndProc(HWND, UINT, WPARAM, LPARAM)
//
//  PURPOSE:  Processes messages for the main window.
//
//  WM_CREATE - creates the timer and the status bar on the bottom
//  WM_COMMAND       - process the application menu
//  WM_LBUTTONDOWN - handles left mouse clicks
//  WM_PAINT  - Paint the main window
//     WM_TIMER      - Handles the timer
//  WM_DESTROY       - post a quit message and return
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
	int wmId, wmEvent;
	PAINTSTRUCT ps;
	HDC hdc;

	std::ostringstream convertMilli;
	std::ostringstream convertSeconds;
	LPCTSTR instructions;
	int iPosX;
	int iPosY;
	RECT rect;
	int windowWidth;
	int windowHeight;

	switch (message)
	{
	case WM_CREATE:
	{
		HFONT hfDefault;
		HWND hEdit;
		HWND hTool;
		UINT ret;

		int statwidths[] = {200, 400, -1};

		// Create Status bar
		hStatus = CreateWindowEx(0, STATUSCLASSNAME, NULL,
			WS_CHILD | WS_VISIBLE | SBARS_SIZEGRIP, 0, 0, 0, 0,
			hWnd, (HMENU)103, GetModuleHandle(NULL), NULL);

		SendMessage(hStatus, SB_SETPARTS, sizeof(statwidths)/sizeof(int),
(LPARAM)statwidths);
		SendMessage(hStatus, SB_SETTEXT, 0, (LPARAM)L"Time: ");
		SendMessage(hStatus, SB_SETTEXT, 1, (LPARAM)L"Correct # Clicks:
0");
		SendMessage(hStatus, SB_SETTEXT, 2, (LPARAM)L"Wrong # Clicks: 0");

		// Set the Timer
		ret = SetTimer(hWnd, ID_TIMER, 10, NULL);
		if(ret == 0)
			MessageBox(hWnd, TEXT("Could not SetTimer()!"),
TEXT("Error"), MB_OK | MB_ICONEXCLAMATION);
	}
	break;
	case WM_COMMAND:
```

```
            wmId    = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
            {
            case IDM_ABOUT:
                  DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd,
About);
                  break;
            case IDM_RESET:
                  Reset(hWnd);
                  break;
            case IDM_EXIT:
                  DestroyWindow(hWnd);
                  break;
            default:
                  return DefWindowProc(hWnd, message, wParam, lParam);
            }
            break;
      case WM_LBUTTONDOWN:
            hdc = GetDC(hWnd);
            iPosX = LOWORD(lParam);
            iPosY = HIWORD(lParam);
            InsideCircle(hdc, iPosX, iPosY);
            ReleaseDC(hWnd, hdc);
            break;
      case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            // TODO: Add any drawing code here...

            instructions = L"Left Click inside the black circles in the order
listed as quickly as possible. Try not to click anywhere but inside the
circles.";

            if(GetClientRect(hWnd, &rect))
            {
                  windowWidth = rect.right - rect.left;
                  windowHeight = rect.bottom - rect.top;
            }

            SetBkMode(hdc, TRANSPARENT);
            // Draw Top text and line
            TextOut(hdc, 5, 5, instructions, 130);
            MoveToEx(hdc, 0, 20, NULL);
            LineTo(hdc, windowWidth, 20);

            DrawCircles(hdc);

            EndPaint(hWnd, &ps);
            break;
      case WM_TIMER:
            if (timerStarted) {
                  stopwatch = clock() - startTime;
                  convertMilli << stopwatch % 1000;
                  convertSeconds << floor(stopwatch / 1000);
                  stemp = "Time: " + convertSeconds.str() + '.' +
convertMilli.str();
                  wstemp = s2ws(stemp); // Temporary buffer is required
```

```
                        result = wstemp.c_str();
                        SendMessage(hStatus, SB_SETTEXT, 0, (LPARAM)result);
                }

                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        default:
                return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}

// Message handler for about box.
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
        UNREFERENCED_PARAMETER(lParam);
        switch (message)
        {
        case WM_INITDIALOG:
                return (INT_PTR)TRUE;

        case WM_COMMAND:
                if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
                {
                        EndDialog(hDlg, LOWORD(wParam));
                        return (INT_PTR)TRUE;
                }
                break;
        }
        return (INT_PTR)FALSE;
}

// Message handler for reset box.
void Reset(HWND hDlg)
{
        // Reset circle colors
        center.color = "Black";
        top.color = "Black";
        bottom.color = "Black";
        left.color = "Black";
        right.color = "Black";
        // Repaint
        HDC hdc = GetDC(hWnd);
        RECT rcClient;
        GetClientRect(hWnd, &rcClient);
        DrawCircles(hdc);
        ReleaseDC(hWnd, hdc);

        // Reset Timer
        timerStarted = 0;
        SendMessage(hStatus, SB_SETTEXT, 0, (LPARAM)L"Time: ");

        // Reset Click Counters
        wrongClicks = 0;
        correctClicks = 0;
        SendMessage(hStatus, SB_SETTEXT, 1, (LPARAM)L"Correct # Clicks: 0");
```

```
	SendMessage(hStatus, SB_SETTEXT, 2, (LPARAM)L"Wrong # Clicks: 0");
}

void InsideCircle(HDC hdc, int iPosX, int iPosY)
{
	int centerDistance = sqrt(pow(abs(iPosX-center.x), 2) + pow(abs(iPosY-
center.y), 2));
	int topDistance = sqrt(pow(abs(iPosX-top.x), 2) + pow(abs(iPosY-top.y),
2));
	int bottomDistance = sqrt(pow(abs(iPosX-bottom.x), 2) + pow(abs(iPosY-
bottom.y), 2));
	int leftDistance = sqrt(pow(abs(iPosX-left.x), 2) + pow(abs(iPosY-
left.y), 2));
	int rightDistance = sqrt(pow(abs(iPosX-right.x), 2) + pow(abs(iPosY-
right.y), 2));

	if (centerDistance <= RADIUS && correctClicks == 0 && !timerStarted) {
		center.color = "Green";
		timerStarted = 1;
		correctClicks++;
		startTime = clock();
	}
	else if (centerDistance <= RADIUS && correctClicks == 8) {
		center.color = "Green";
		correctClicks++;
		timerStarted = 0;
	}
	else if (centerDistance <= RADIUS && (correctClicks == 2 || correctClicks
== 4 || correctClicks == 6)) {
		center.color = "Green";
		correctClicks++;
	}
	else if (topDistance <= RADIUS && correctClicks == 1) {
		top.color = "Green";
		center.color = "Black";
		correctClicks++;
	}
	else if (bottomDistance <= RADIUS && correctClicks == 5) {
		bottom.color = "Green";
		center.color = "Black";
		correctClicks++;
	}
	else if (leftDistance <= RADIUS && correctClicks == 3) {
		left.color = "Green";
		center.color = "Black";
		correctClicks++;
	}
	else if (rightDistance <= RADIUS && correctClicks == 7) {
		right.color = "Green";
		center.color = "Black";
		correctClicks++;
	}
	else if (timerStarted) {
		wrongClicks++;
		std::ostringstream convert2;
		convert2 << wrongClicks;
		stemp = "Wrong # Clicks: " + convert2.str();
		wstemp = s2ws(stemp); // Temporary buffer is required
```

```
                result = wstemp.c_str();
                SendMessage(hStatus, SB_SETTEXT, 2, (LPARAM)result);
        }

        std::ostringstream convert;
        convert << correctClicks;
        stemp = "Correct # Clicks: " + convert.str();
        wstemp = s2ws(stemp); // Temporary buffer is required
        result = wstemp.c_str();
        SendMessage(hStatus, SB_SETTEXT, 1, (LPARAM)result);

        DrawCircles(hdc);
}
```

BIBLIOGRAPHY

Ahsan, M.R., M.I. Ibrahimy, and O.O Khalifa. "Hand motion detection from EMG signals by using ANN based classifier for human computer interaction." *Modeling, Simulation and Applied Optimization (ICMSAO), 2011 4th International Conference on* April 2011: 1-6.

Al-Timemy, Ali, Guido Bugmann, Nicholas Outram. "A Study of the Effect of Force Control on the Performance of a Myoelectric Finger Flexion Recognition Algorithm." *ICABB 2010: 1st International Conference on Applied Bionics and Biomechanics* Oct. 2010.

Bajramovic, Mark B. "Computer mouse on a glove." US Patent 7737942. 14 October 2005.

Chen, Sean, Evan Levine. "Mister Gloves - A Wireless USB Gesture Input System." *Cornell University*. 2010. 22 August 2012. <https://courses.cit.cornell.edu/ee476/FinalProjects/s2010/ssc88_egl27/index.html>.

Cheng, Wen-Lung. "Magic Finger Mouse." US Patent 20100328205. 29 June 2009.

Delsys Incorporated. "Technical Note 101: EMG Sensor Placement." Delsys Incorporated. 7 Dec. 2007. < http://www.delsys.com/Attachments_pdf/TN101%20-%20EMG%20Sensor%20Placement-web.pdf>.

Delsys Incorporated. "Trigno Wireless System User's Guide." Delsys Incorporated. 6 Feb. 2013. <http://www.delsys.com/Attachments_pdf/Trigno%20%20Wireless%20System%20Users%20Guide%20(MAN-012-2-6).pdf>.

Englehart, K., and B. Hudgins, "A Robust, Real-Time Control Scheme for Multifunction Myoelectric Control," *IEEE Transactions on Biomedical Engineering* 50 July 2003: 848-54.

"G-Speak." *Oblong Industries*. Oblong Industries, Inc. 2012. 22. August 2013. <http://oblong.com/what-we-do/g-speak>.

Hudgins, B., P. Parker, and R. N. Scott, "A New Strategy for Multifunction Myoelectric Control," *IEEE Transactions on Biomedical Engineering* 40 Jan 1993: 82-94.

Itou, T., M. Terao, J. Nagata, and M. Yoshida. "Mouse cursor control system using EMG." *Engineering in Medicine and Biology Society, 2001. Proceedings of the 23rd Annual International Conference of the IEEE* 2 2001: 1368-1369.

Khezri, M., M. Jahed, "A Neuro–Fuzzy Inference System for sEMG-Based Identification of Hand Motion Commands," *Industrial Electronics, IEEE Transactions on* 58, May 2011: 1952-1960.

Khushaba, Rami N., Adel Al-Jumaily, and Ahmed Al-Ani. "Fuzzy discriminant analysis based feature projection in myoelectric control." *Engineering in Medicine and Biology Society, 2008. EMBS 2008. 30th Annual International Conference of the IEEE* August 2008: 5049-5052.

Khushaba, R.N., S. Kodagoda, Dikai Liu; G. Dissanayake. "Electromyogram (EMG) based fingers movement recognition using Neighborhood Preserving Analysis with QR-decomposition." *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2011 Seventh International Conference on* Dec. 2011: 1-6.

Kim, Jong-Sung, Huyk Jeong, and Wookho Son. "A new means of HCI: EMG-MOUSE." *Systems, Man and Cybernetics, 2004 IEEE International Conference on* 1 March 2005:100-104.

"Leap Motion." *Leap Motion*. Leap Motion, Inc. 2012. 8 Jan. 2013. <https://leapmotion.com/product>.

Lee, Seungbae, Gi-Joon Nam, Junseok Chae, Hanseup Kim, and A.J. Drake. "Two-dimensional position detection system with MEMS accelerometer for mouse applications." *Design Automation Conference, 2001. Proceedings* 2001: 852-857.

Measurement Computing Corporation. "USB-1616HS-BNC User's Guide." Measurement Computing Corporation. 13 Dec. 2012. <http://www.mccdaq.com/PDFs/manuals/USB-1616HS-BNC.pdf>.

Miyoshi, T., A. Murata. "Input device using eye tracker in human-computer interaction," *Robot and Human Interactive Communication, 2001. Proceedings. 10th IEEE International Workshop on* 2001: 580-585.

Oonishi, Y., S. Oh, and Y. Hori. "A New Control Method for Power-Assisted Wheelchair Based on the Surface Myoelectric Signal." *Industrial Electronics, IEEE Transactions on* 57 2010: 3191-3196.

Perotto, Aldo O., Edward F. Delagi. *Anatomical Guide For The Electromyographer: The Limbs And Trunk.* Springfield, Illinois. Charles C Thomas. 2005.

Potluri, C., P. Kumar, M. Anugolu, A. Urfer, S. Chiu, D.S. Naidu, and M.P. Schoen. "Frequency domain surface EMG sensor fusion for estimating finger forces." *Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE* Sep. 2010: 5975-5978.

Thalmic Labs. “Myo.” *Myo*. Thalmic Labs. 21 Feb. 2013. 11 March. 2013. <https://getmyo.com/>.

Wikipedia contributors. "Electromyography." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 6 Mar. 2013. Web. 11 Mar. 2013. <http://en.wikipedia.org/wiki/Electromyography>.

Wikipedia contributors. "Inertial measurement unit." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 26 Feb. 2013. Web. 11 Mar. 2013. < http://en.wikipedia.org/wiki/Inertial_measurement_unit>.

Xiong, Anbin, Yang Chen, Xingang Zhao, Jianda Han, and Guangjun Liu. "A novel HCI based on EMG and IMU." *Robotics and Biomimetics (ROBIO), 2011 IEEE International Conference on* Dec. 2011: 2653-2657.

Xsens Technologies B.V. “MTi and MTx User Manual and Technical Documentation.” Xsens Technologies B.V. 15 Oct. 2010. <http://www.xsens.com/images/stories/products/manual_download/MTi_and_MTx_User_Manual_and_Technical_Documentation.pdf>.

Zhang, Xiaorong, Yuhong Liu, Fan Zhang, Jin Ren, Y. Sun, Qing Yang and He Huang. "On Design and Implementation of Neural-Machine Interface for Artificial Legs." *Industrial Informatics, IEEE Transactions on* 8 May 2012: 418,429.

Zhao, Jingdong, Li Jiang, Hegao Cai, Hong Liu, and G. Hirzinger. "A Novel EMG Motion Pattern Classifier Based on Wavelet Transform and Nonlinearity Analysis Method." *Robotics and Biomimetics, 2006. ROBIO '06. IEEE International Conference on* Dec. 2006:1494-1499.