HandsOn: Human Computer Interface Glove for Sign Language Translation

4BI6 Capstone Final Report - Group B07

Name	Student Number	Email
Bhavesh Kakwani	1162865	kakwanbt@mcmaster.ca
Bhavit Patel	1059825	patelb45@mcmaster.ca
Tom Yang	1063366	yangxy5@mcmaster.ca

Table of Contents

Abstract

A human computer interface glove was developed with the aim of translating sign language to text & speech. The glove utilizes nine flex sensors, seven touch capacitive sensors, and an inertial measurement unit to accurately capture hand gestures. All components were placed on the back side of the glove providing the user with full range of motion, and not restricting the user from performing other tasks while wearing the glove.

Machine learning algorithms were explored to perform classification of hand gestures. A support vector machine was successfully implemented to classify static hand gestures in real-time. A developer GUI application was created to sample sensor data; create and capture custom gestures; classify hand gestures with a support vector machine and output text & speech; and a hand animation to display the user's hand gestures. All of the GUI features were developed with multi-threading capabilities allowing real-time usage. The current results include translation of twenty-four letters and the numbers 0-9 from sign language to text & speech.

Background

Integrating people from deaf & mute communities into modern society, especially the modern workplace, is currently a major challenge. There is a major language barrier between users of Sign-language (signers or signing people) and those who do not understand it (non-signing people). Currently, the solution to this is to hire a Sign-language interpreter who will be present by the signer's side at all times. However, the employer often has to pay out of their pocket to hire this interpreter which results in it being more expensive than hiring another employee who does not have any communication barriers. Individuals who are deaf or mute are not only segregated in the modern workplace but also in everyday life causing them to live in their own separate communities. Worldwide there are 120 million deaf people who use Sign-language as their primary and most important means of communication. There is no single form of Sign-language as it varies from region to region; American Sign Language (ASL), British Sign Language (BSL), Indian Sign Language(ISL) are just a few of the different forms. ASL is often referred to as the fourth most common language in the United States [1].

There has been a rapid progression in medicine recently, leading to solutions for some deaf & mute individuals. For instance, there have been improvements in hearing aids and cochlear implants for the deaf and artificial voice-boxes for the mute with vocal cord damage. However, these solutions do not come without disadvantages and costs. Cochlear implants have even caused a huge controversy in the deaf community and many refuse to even consider such solutions. Therefore, we believe society still requires an effective solution to remove the communication barrier between deaf & mute individuals and non-signing people.

Our proposed solution and goal is to design a Human Computer Interface (HCI) device that can translate sign language, specifically ASL, to text and speech providing any deaf & mute individuals with the ability to effortlessly communicate with anyone. Sign language involves the use of gestures,

mainly specific hand shapes and movements, instead of sound to convey words and sentences. The idea is to design a device placed on a hand with sensors capable of capturing hand gestures and then transmitting the information to a processing unit which performs the sign language translation. The final product will also be able to efficiently teach sign language to any user since they would be able to receive instant feedback. We hope to be able to improve the quality of life of deaf and mute individuals with this device. During the development, we will also have the opportunity to learn a new language. We believe there is great potential in a device with these capabilities and know that it has far more applications in creating seamless gesture-based human-computer interfaces.

Current device solutions in the market are optic based such as LeapMotion. The disadvantage of using optical sensors is the light requirements necessary for function, and the limited space of use. Errors can arise from disturbances in the environment that obstruct the sensor's operating space. They are also not portable which is a necessity for sign language communication.

Design Requirements & Specifications

Our idea is to create a Human Computer Interface (HCI) wearable device that can translate sign language, specifically ASL, to text and speech providing the deaf & mute with the ability to effortlessly communicate with others. In order to accomplish the measurements for static gestures, we must design the device such that it is able to:

- Capture important hand gestures from the wearer of the device through various strategically placed sensors. The sensors should be able to detect:
    - Bending, Abduction, and Adduction of the Fingers. Extension and flexion of fingers must be able to be categorized into at least 3 modes to determine letters. Bending sensors are placed on the fingers and knuckles for more accurate classification but may be accomplished with just one sensor per finger. The sensor must be able to detect a bending range of 0-90° and have about a ±5° accuracy.

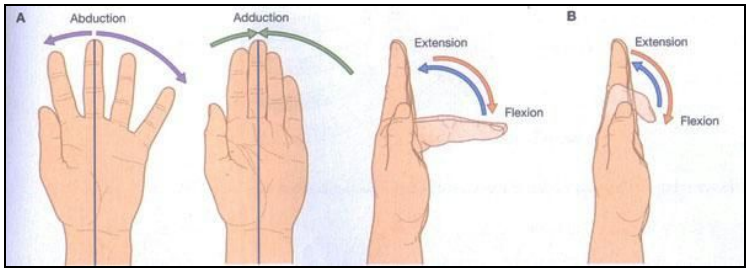

Figure 1: Motion of fingers (excluding thumb) [2]

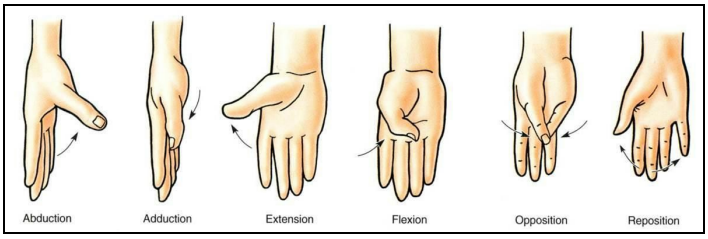

Figure 2: Motion of thumb [3]

- Roll, Pitch, and Yaw of the hand (ie. orientation of the hand). A precise measurement unit is not required for letter recognition due to the limited modes of orientation. A ±10° should be acceptable for classification of the letters. A very accurate orientation measurement is required however for advanced dynamic gestures as well as effective human computer interfacing.

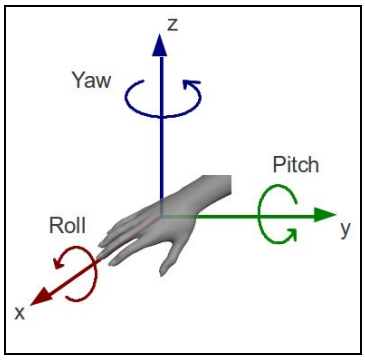

Figure 3: Orientation and motion of hand

- Contact detection to accurately distinguish between similar gestures such as ‘6’ and ‘W’.

- Position of the hand relative to a reference point on the body. Though not incorporated into our final design, it was proposed that the reference point be located near the chest in conjunction with the secondary left handed glove.

- Process the sensor measurements to accurately detect the performed hand gestures.

- Develop an ASL software database and processing algorithm to determine the hand gesture meaning and/or use a Machine Learning Algorithm to classify the hand gestures.

- Reproducibility of at least 90% for all gestures

- Utilize logic controls to determine when the user has completed signing a word and/or would like to pause from gesturing.

- Create the functionality for users to produce their own gestures and name them, as well as tailor the classification set to the user based on their gesture accents.

- Output text of the letter/number/word/phrase detected onto a computer in a GUI application for real time feedback.

- Utilize a text-to-speech algorithm or API to output the speech through a computer.

- Operate comfortably on the hand and minimize disruption to the functionality of the hand

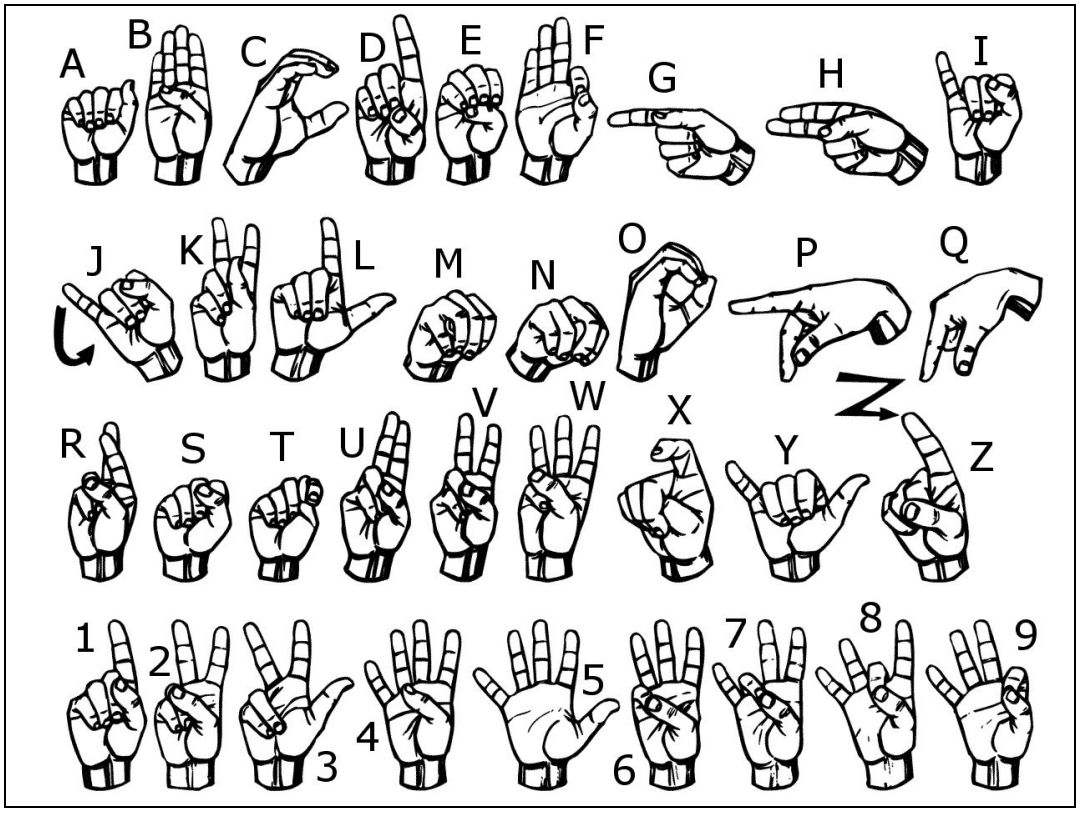

Figure 4: American Sign Language (ASL) Alphabet and Numbers [4]

Figure 4 above shows the scope of ASL gestures that our group has accomplished. The degree of abduction is not relevant for the above gestures. It is identified that fingers and knuckles have a few distinct modes that they operate in which are flexed, extended, and partially bent. Additionally, the roll, pitch, and yaw of the hand also operates in a few modes as well which are:

i. the palm of the hand facing away from the body and fingers pointing upwards,
ii. the palm facing to the left and fingers pointing up,
iii. the palm facing to the left and fingers pointing forward, and
iv. the palm facing toward the body with the fingers pointing left.

Movement of the hand for letters 'J' and 'Z' are always from one mode to another. To maintain accuracy of the system, we did not find sufficient static gestures for words that could develop a coherent sentence and kept our scope to letters and numbers. The user is able to add additional static gestures at their leisure to the machine learning database as long as the gesture has a sensor reading that is determined to be in a different mode and they are able to reproduce the gesture.

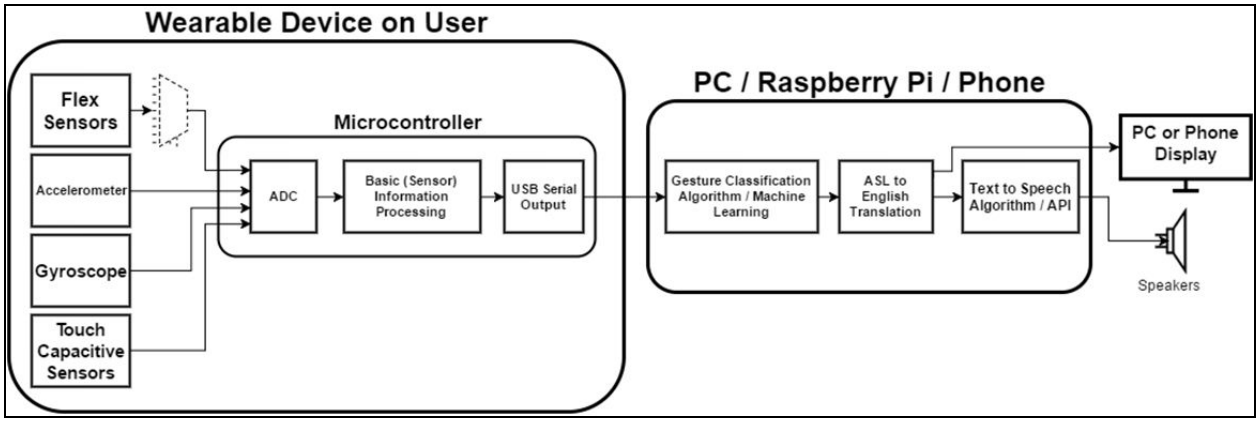

Figure 5: Block diagram for our proposed HCI device for Sign-language translation

The block diagram for our device is shown above in Figure 5. An Inertial Measurement Unit (IMU) is required for accelerometer and gyroscope readings which is placed on the back of the hand to determine hand orientation and movement. 9 Flex sensors are used for detecting extension and flexion. 7 Capacitive touch sensors are included for adduction and abduction. These sensors are fed to a microcontroller which is connected to a laptop via USB. On the computer, arduino software is used to relay serial port information into the main program that operates a GUI, gesture classification algorithm, text to speech, hand model visual feedback, and other controls. Sensor readings are displayed on the GUi and the interpretation of hand movements are shown on a hand model that mimics hand orientation and finger movement. The gestures are converted their respective text and displayed on the screen in sequential order before forming a complete word and delivering speech.

With the scope of this project defined above, it is important to note the scope that would be required for commercialization (i.e. outside the scope of this project). The commercial version will need to have a sleek design with fast response time, low-production cost, a system for updates to be installed onto the glove, and a calibration process for different users. Achieving these goals requires customized components, for example replacing the microcontroller with a custom-designed circuit that has an ASIC and possibly a small microcontroller used in combination. It also requires heavy optimization of the classification and translation algorithms. Given the timeframe of this project, we chose to put these features out of our scope and we will pursue them after achieving the goals of this project.

Hardware Implementation

To realize the design requirements, several hardware components were considered during prototyping. Much of the hardware was selected due to their ease of use, cost, and accuracy. The Teensy 3.2 microcontroller was selected for its various advantages in size, power, and price. Most importantly, it had the required number of analog inputs for the sensors, despite having to access the bottom side of the device. The flex sensors selected are the Spectra Symbol 2.2” due to their market availability and documentation. The alternative flex sensor size is 4.5” but to optimize accuracy, we decided to place two flex sensors on each fingers except the pinky at the knuckle joint and on the phalanges. In fact, testing showed that flexion and extension was captured to ±5° accuracy. Touch capacitive sensors were simply created with copper tape by taking advantage of the built in touch

sense analog pins of the teensy and the user's hand as the capacitor. The BNO055 IMU was selected for convenience as it produces the computation necessary for linear acceleration and gyroscopic measurements. The roll, pitch and yaw measurements were found to be roughly ±2° accuracy which is far beyond sufficient for our sign language requirements. Another selling point was the availability of a magnetometer that could be used for absolute positioning and the potential for relative positioning. We were able to calculate euler angles through the quaternion measurements to solve the problem of gimbal lock.

Ideally, a PCB would be the most organized and compact way of implementing the circuit board. Instead, it was accomplished using a perfboard for its thin thickness and dimensions. Since the bottom side of the teensy was required for additional analog pins, careful planning of the perfboard allowed us to fit the circuit in an agreeable manner on the hand. Space was left to add in the requested HC-06 bluetooth module which did not arrive in time for incorporation onto the device. A colour-coded wiring paradigm was used to help identify components.

The right handed glove selected was an ordinary cotton glove that is widely available. It was important that the glove was skin tight to improve flex sensor accuracy while maintaining a level of comfort so that it could be worn for extended periods of time. The glove's size was optimal for one group member but could be worn by all for testing purposes. Discussions were made to purchase a higher tech glove that is more comfortable to the user such as those for arthritis patients. To get around the touch sense requirements, slits were cut at certain fingertips so that they could make contact with the copper foils. It was also discussed that a material such as those used on touch screen gloves could be an adequate replacement for our fingertips. Our intention initially was to make the glove aesthetically pleasing but due to time limitations we kept the design and wiring used in the latest prototype. Initial prototypes used velcro to keep the breadboard circuit on the back of the hand and held the flex sensors with strands of sew. It was envisioned that pockets across the fingers would be used to hold the flex sensors in place while allowing enough space for flexion. The final decision was to use double sided tape to secure the sensors while sewing any extra pockets of air down that could prove to be an issue.

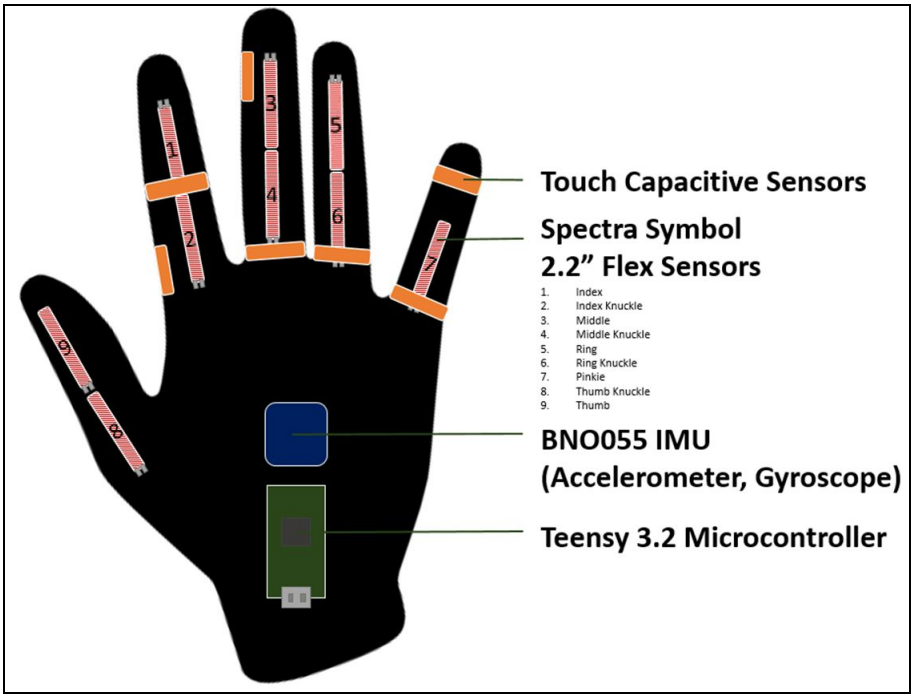

Figure 5: Location of hardware components on the right hand.

Figure 5 shows the approximate locations of the hardware on the device. The sensors must be placed in a way as to not make contact with each other causing short circuits which and disruption of measurement readings. Electrical tape was necessary to provide insulation for our sensors. Flex sensors cover the knuckles and phalanges of all the fingers except the pinky as they are the joints most relevant for finger flexion. The accelerometer and gyroscope had to be placed either on the back of the hand or the palm to follow the hand’s reference plane. Since our device was connected via USB to the computer, it was optimal to have the female USB port point away from the rest of the hand as to not obstruct movement and sensor placement.

Hand Gesture Classification

To translate the hand gestures to letters and numbers, the data from the sensors needs to be mapped to its corresponding letter or number in sign language. This was accomplished using a machine learning algorithm. The reason for using machine learning for classification, as opposed to conditional (if/else) statements, is that it gives the user the ability to tailor the gesture classification to their unique style of signing. The user can train the machine learning classifier by showing it examples of different gestures, and saving each gesture with the corresponding letter or number attached.

To implement the machine learning there are 3 important considerations: **features**, **targets**, and the **type of classifier** used. What follows is a discussion of the implementation of these considerations for this project.

Features & Targets

In machine learning, features are the set of independent variables extracted from the raw data. The linear combination of all the features is known as the feature space. Features are the inputs to the classifier whereas targets are the decision output. In this project, the features are extracted from the hardware sensor inputs and the targets are the letters/numbers in sign language.

On the glove, there are 23 inputs from the hardware sensors:

- Analog values from 9 flex sensors
- Analog values from 7 touch sensors
- 4 quaternion values from the gyroscope
- 3 orthogonal acceleration values from the accelerometer

The raw data needs to be preprocessed to turn it into a set of useful features. A useful feature is one in which there is considerable difference between the clusters of data, so that the machine can easily separate one state from another. For example, it should easily determine whether the touch sensor is being touched or not. Secondly, the whole feature space should be laid out so that there is a large difference between *all* the target clusters. In this way the classifier will choose the correct target (letter/number) based on the features with high accuracy.

This strategic laying out of the feature space is accomplished using *feature scaling*. Features can be scaled by multiplying them by a scalar value to give them a certain level of importance in the classifier. A higher scaling value results in more importance being given to that feature.

The sensor inputs from the glove were preprocessed as follows:

- Analog values from flex sensors were linearly mapped to angles in degrees, based on the bending of the finger joint. The joint bending angles typically lie in the range 0°-90°. These joint-bending angles provide **9** features for the feature space.

- Analog values from touch sensors were mapped to 0 or 1, where 1 means the touch sensor is being touched by the user. However, the difference between 0 and 1 is very small which means the classifier will give this feature very little importance. Hence, feature scaling was implemented here with a scale factor of 1000. The classifier receives a value of 0 for when the touch sensor is at rest, and 1000 for when the sensor is being touched. This provides **7** more features for the feature space.

- The quaternions from the gyroscope indicate the direction in which the user's hand is pointing. For gesturing letters and numbers in ASL it is not necessary to know the precise direction in which the hand is pointing. All that is necessary is to know whether the hand is pointing *up*, *forward/sideways*, or *down*. These would be assigned values of +1, -1, and 0 respectively. Again, the differences between these values is too small so feature scaling will be

implemented with a scaling factor of 100. Hence, the classifier receives a value of 100 when the hand points up, -100 when the hand points forward/sideways, and 0 when the hand points down. This provides **1** important feature for the workspace.

- The acceleration readings from the accelerometer are not necessary for discerning ASL gestures, because all the gestures for letter and numbers are static and don't involve movement of the hand. So the acceleration values are not sent to the classifier. However, it is important to know the acceleration in order to determine whether the hand is moving or stationary, in order to discern between when the user is holding a sign gesture as opposed to transitioning between signs. Hence, the acceleration values will be compared against a threshold value, using an if/else statement, to determine whether or not the program should try to classify the gesture at that specific moment.

The above features give a total of 9+7+1=**17 features** for the feature space. The classifier must take these features and assign them to the alphabet and numbers. The alphabet and numbers combined give a total of 26+10=**36 targets** for the classifier to resolve. Graphically, this is represented by 36 clusters of data in a 17-dimensional feature space. An appropriate classification algorithm is needed for this layout of the data.

Classifier Type

To classify the gestures, the classifier needs to *accurately* and *quickly* separate the target clusters based on the provided features. Additionally, it should be *flexible* enough so that it can mould to any user's style of sign language. For high speed of classification, the classifier should have low computational intensity. For high accuracy and for flexibility, it needs to have a method of separating clusters that is universally applicable no matter where the gesture clusters are in the feature space.

To summarise the above points, the classifier should be simple in implementation while providing a complex target separation with high accuracy. For these reasons, a **Support Vector Machine (SVM)** approach was taken for classifying the hand gestures. SVM separates the target clusters using a linearized hyperplane, as shown in Figure 6. The hyperplane is placed to provide maximum margin between the support vectors; support vectors are the data points at edge of each target cluster. Since the separation is linear, SVM performs very fast. Also, because the feature space is large (17 dimensions!) for only 36 targets, it will provide high accuracy and flexibility with a low risk of overfitting the data.

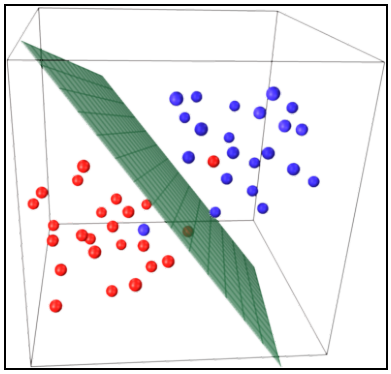

Figure 6: Target cluster separation using SVM hyperplane [5]

The SVM is implemented on the gestures in pseudo-realtime; that is, the program is constantly classifying the user's hand gesture after certain time periods and under certain conditions. The steps for this operation are outlined below:

1. Collect a set of samples from all the sensors using a moving time window.
2. Calculate the mean of the sensor values in this time window.
3. Preprocessing and feature scaling of the sensor data to turn it into feature space.
4. If the hand is not moving (mean acceleration is less than the threshold), then use the SVM classifier to predict the hand gesture. {Else, go to step 1}.
5. Output the letter/number corresponding to the output from the classifier.
6. Small time delay, then go to step 1.

Software Implementation

Teensyduino

The Teensy 3.2 microcontroller has the ability to use Arduino code and libraries. As a result, "Teensyduino" was used to implement the code on the Teensy to perform sensor data collection and processing. Since this project does not involve electrophysiological signals, a sampling rate of 50Hz was selected for all sensor data (quaternions, linear acceleration, flex sensors, and touch capacitive sensors). The sampling rate could be increased but this is a reasonable starting point for the sensor data that is required.

As mentioned in the hardware implementation, the flex sensors involve a simple voltage divider circuit. Each flex sensor was tested individually for linearity between finger bending and voltage output. The linearity was good enough to create a linear mapping from 0°-90° of finger flexion. Each flex sensor was individually mapped for best performance. The output voltage from the flex sensor circuit (represented as a decimal corresponding to the 12bit ADC output) was measured for 0°-90° bending on each joint. Teensyduino allows the use of "analogRead()" to read the voltage output from the flex sensor circuit which is performed every 20ms.

Another reason the Teensy 3.2 microcontroller was chosen was due to the built-in “Touch Pins”. After wiring a piece of copper foil to the pin, the internal pull-up resistance and pin capacitance allow using “touchRead()” to calculate the capacitance through an RC time constant calculation . When not touched, the value was less than 400 and would jump to 6000 when fully touched. Note that the units for these values were not tested because the only necessary information was whether the sensor was touched or not touched. A threshold value of 2000 was used to distinguish between the two states.

The Adafruit BNO055 class library was used to interact with the IMU as it involved a lot of tedious register reading and writing. The class was modified to include a method for reading some settings and the IMU mode. The absolute IMU orientation mode (called NDOF mode) uses the magnetometer to find the Earth’s magnetic North and set that to the yaw=0 orientation. However, in order to easily sync the orientation of the hand model animation and the glove, the IMU mode was changed to relative orientation. The relative orientation mode (called IMU mode) sets the “0” orientation to the initial orientation of the glove when it is powered on. As a result, the user can start with their hand flat and pointing into the screen to easy sync with the animation. The IMU has its own processor to perform sensor fusion of the accelerometer, gyroscope, and magnetometer for better accuracy and less drift. In the relative orientation mode, the IMU outputs data at 100Hz. The appendix shows more details about the various outputs of the IMU.

The serial output begins with initialization and debug data for the Teensy and IMU. After that, all of the sensor data (IMU calibration data, quaternions, linear acceleration, 9 flex sensor values, 7 touch capacitor values) is printed to serial output every 20ms. Note that quaternions were chosen over Euler Angles to prevent an issue known as gimbal lock. The Euler angles were calculated through the quaternions if required (such as for the hand animation which will be discussed below). The figure below shows a snapshot of the serial output.

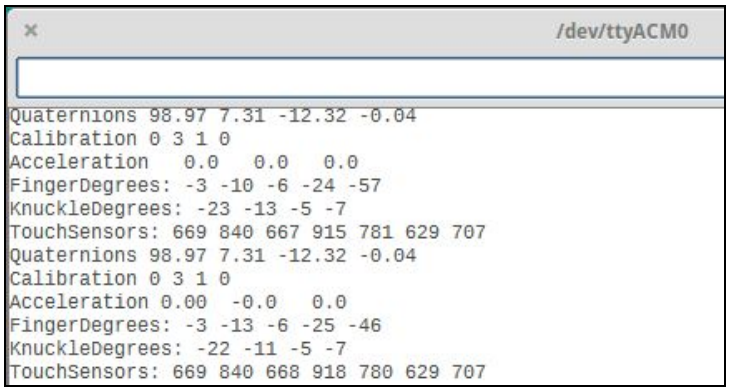

Figure 7: Output from Teensy in the serial monitor

Python

In the early stages of design, Processing was used to create a hand model animation. This animation allowed very fast and intuitive debugging of the quaternions and Euler angles. However, it quickly became apparent that Processing would not be useful to perform any other features such as

gesture classification. As a result, the hand model animation was migrated over to Python so all features could be in the same programming language.

The goal was to not only implement software for real-time classification of sign language, but to also create a developer application. This developer application would allow the user (or developer) to easily view all sensor data and hopefully allow easy development for new applications of the HCI glove. In the end, a fully functional developer GUI application was created which will be discussed later in this section.

Since the project consists of 1,904 lines of Python code (not including some additional Python scripts and the mouse + game control scripts), the appendix will contain only a brief description of each of the files and functions/classes contained within those files. To view the full code, a link to a GitHub page will be provided.

Console Application

To begin, HandsOn.pseudoMain(delay, debugFlag,ttsFlag) is a console menu application that was created to perform capturing of hand gesture data to a file, reading a training file and fitting an SVM, real-time gesture classification, and text-to-speech output. To perform everything in real-time, the program required multi-threading. Therefore, HandsOn.pseudoMain(delay,debugFlag,ttsFlag), HandsOn.parseSerialHandData(ser), and Animation.py were all designed to run in their own threads using the Python Thread class. The diagram below shows the general layout of the console application.

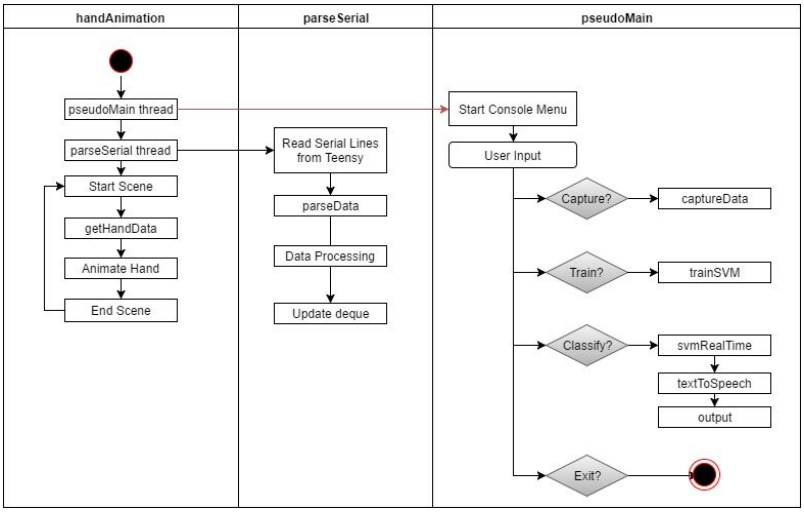

Figure 8: Console Application Software Diagram

Serial Data Parser & Collections.Deque

HandsOn.parseSerialHandData(ser) opens a serial communication port from the Teensy, reads each line, and parses the line to assign each item to the appropriate variables. Note that the Teensy sends a block of sensor data information every 20ms thus the Python program must be efficient in data collection. Also, each thread requires the usage of some of these variables at the same time.

Therefore, “Collections.deque” structures were utilized for the real-time moving window data collection. It was decided that a 500ms moving window was sufficient for accurate data although it could be reduced to speed up the sign-language translation. A “Collections.deque” structure was created for each sensor and each variable from the IMU. They allow O(1) moving window data collection due to the ability to instantly add a new element to the end of the list and remove the oldest one. They are also thread-safe which allows all 3 threads to use them at the same time. The “share_var.py” file instantiates these deque structures and some global variables to be used by all threads. Although global variables are bad practice, structures and classes to contain these variables kept causing errors with the multithreading and was not worth debugging for the purpose of this capstone project.

Tools.py for Data Processing and File IO

The “Tools.py” file contains multiple functions for processing & organizing the hand sensor data along with any additional functions required. The appendix contains a full list of all functions in the file with a description of the inputs, outputs, and usage of each function. It contains Tools.printHandDataToFile(fileName,str_handIdentifier) which writes the moving window mean of the flex sensor and touch capacitive sensors as well as a number specifying the orientation of the hand to a specified. This data is saved with a user-input gesture label so that the file can be used as a training file for the machine learning algorithm. This allows multiple training files so each user can have their own calibrated data, as well as creating custom gestures and labels. The means are calculated using Tools.DequeMean(deq) and Tools.DequeMeanList(deqList). Tools.ListToCSstr(list) then converts these lists to comma separated strings that can be easily printed to file in an organized format. Tools.readHandDataFromFile(fileName) reads and parses the hand data from a file into an *nx1* list of targets and *nxm* list of *m* features for each target. Currently the number of features is 17 (9 flex sensor means, 7 touch capacitor means, and one number corresponding to an orientation).

Tools.py for Quaternions

Some hand gestures are similar in terms of finger positions but differ in their orientation. Sending the quaternions to the machine learning algorithm would not be useful as it contains 4 values that are separated by the SVM hyperplane. As a result, we performed some quaternion math to determine the exact orientation of the hand and assign that to a single number that classifies its orientation. Quaternions consist of 4 numbers which can be thought of as a 3D vector (x,y,z) along with the amount of rotation about that vector (w). Quaternion multiplication is given by (1.1). In order to determine the exact orientation of the hand, a vector corresponding to the direction the hand/glove is pointing needs to be rotated by the quaternion. Since we are starting with the glove pointing in the computer screen, this corresponds to the vector To perform the quaternion rotation of this vector, it must be post-multiplied by the quaternion, and pre-multiplied by the conjugate (**)** of the quaternion as shown by (1.2). This quaternion math implemented by Tools.q_mult(q1,q2), Tools.qv_mult(q1,v1), and Tools.q_conjugate(q). The direction is assigned by Tools.QuatToDir(qW, qX, qY, qZ).

$$
\begin{aligned}
q_1 \times q_2 &= w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k} \\
w &= w_1 * w_2 - x_1 * x_2 - y_1 * y_2 - z_1 * z_2 \\
x &= w_1 * x_2 + x_1 * w_2 + y_1 * z_2 - z_1 * y_2 \\
y &= w_1 * y_2 + y_1 * w_2 + z_1 * x_2 - x_1 * z_2 \\
z &= w_1 * z_2 + z_1 * w_2 + x_1 * y_2 - y_1 * x_2
\end{aligned} \tag{1.1}
$$

$$
p = q \times v \times q^* \tag{1.2}
$$

Hand Animation

The hand animation was migrated to Python and implemented using OpenGL and Pygame. It consists of multiple rectangular prisms placed at specific coordinates to create a hand model. Each finger consists of two rectangular prisms (except the pinky) which allows the flex sensor values to be used directly on each rectangular prism to animate the finger bending. The touch capacitive sensors cause the appropriate finger to glow. Finally, the quaternions are converted into Euler angles to allow 3D rotation of the frame to visualize the hand orientation. The animation runs at 30fps and runs in its own thread.

Machine Learning

A support vector machine was used with one-vs-all classifiers. This was implemented using the scikit-learn library in Python which contains modules for multiple different types of machine learning algorithms. Specifically, the svm.SVC module was utilized with the C=100, kernel='rbf', and gamma=0.0001. These parameters were determined by inputting a training file and cross-validation file into a script that sweeps combinations of these parameters and outputs the ones with the best accuracy. However, that script determined the best parameter as C=1. The C controls the soft-margin parameter with a lower value allowing the hyperplane to contain more 'errors' which can be useful to allow the SVM to ignore/minimize the effect of outliers. However, we noticed this was causing issues with the wrong letter being classified when two gestures were fairly similar. As a result, we increased it to 100 which provides much better results.

The user must first train the SVM by specifying the training file. The program reads and parses the training file as explained in the Tools for Data Processing and File IO section. During real-time classification, a delay parameter (with default delay of 0.5s) was added to prevent classifying the same gesture multiple times by accident. The linear acceleration is examined by Tools.isMoving() and Tools.LinAccelMoving() to determine if the user is moving. In that case, the program waits until the user has stopped moving before attempting to classify the static hand gesture. In the future, this feature could be used to determine when to use dynamic gesture classification (ex. with Hidden Markov Models) or use the SVM for static gesture classification.

Text-to-Speech

The text-to-speech was accomplished using the python module “pyttsx”. The predicted letters are saved until the user puts his hand down to indicate that the word has been spelled out. The text-to-speech is then performed on that word before continuing the classification.

Python GUI Application

The next step was to create the developer GUI application to easily display all sensor data, display the hand animation, capture gestures, train classifiers, and classify in real-time. The GUI was created with PyQt5 which allows using the Qt Creator framework to design the GUI layout and convert it to Python code. Note that this only allowed easily designing the button and window placements. The functionality still had to be coded. The console application was designed to be modular which allowed many of the functions to be reused in the GUI application. The figure below illustrates the various features of the GUI. The GUI itself can be seen in the Results & Discussion section.

“HandsOn_GUI_Layout.py” contains the python code for the GUI layout with buttons, displays, layout windows, etc. “HandsOn_GUI_main.py” contains the classes and methods for the functionality of the GUI: DevApp, SerialParse, ClassifyRealTime, HandAnimation. To perform multi-threading, each thread required its own class which inherits from the QThread class. One of the major difficulties with the GUI was creating “SIGNALS” and “SLOTS” to communicate between the threads and the main GUI. However, in the end we managed to create separate threads for the GUI, serial parsing of hand data, real-time classification, and the hand animation. The appendix contains a description of each of the classes and methods with the entire code provided in the GitHub repository.

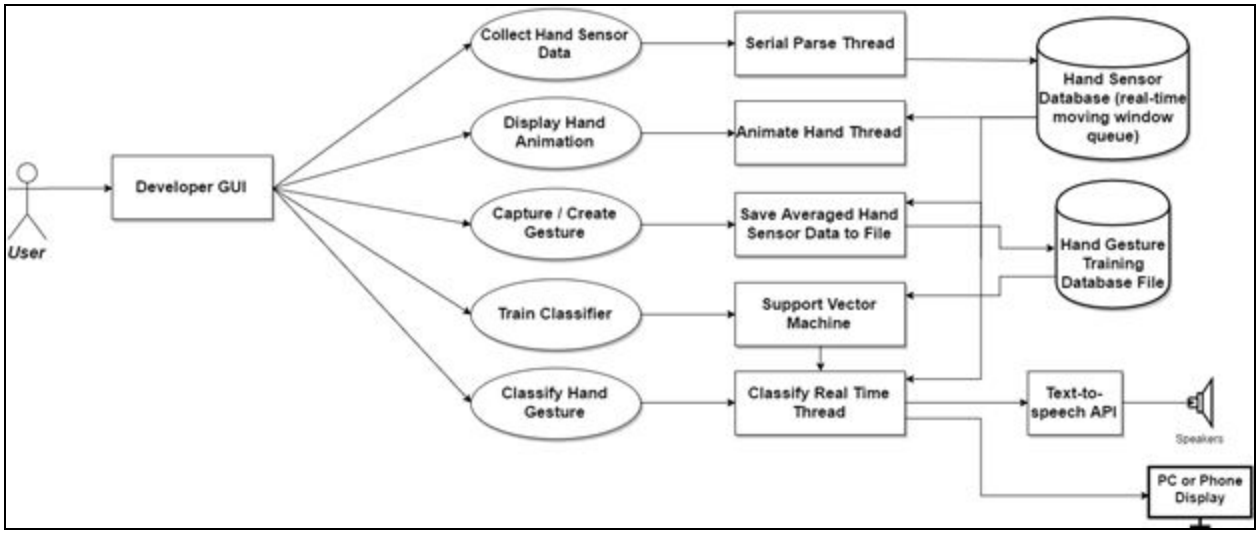

Figure 9: GUI Application Software Diagram

Results & Discussion

Hardware Prototypes

The first major component of the project was to create the hardware based on our design. The hardware prototypes are shown in Figure 10. The first prototype was made on a breadboard to test out the accuracy of the flex sensors, accuracy of the IMU measurements and user comfort while wearing the glove. The flex sensor and IMU readouts were very accurate, and the glove was light and comfortable despite the breadboard on the back of the hand. It was confirmed that a smaller form factor of this prototype would be viable for consumer use.

The second prototype included the capacitive touch sensors, which were added to the Teensy by accessing the surface mount solder pads on the underside of the microcontroller; this was not possible with the breadboard version. The flex sensor and IMU readouts were very accurate as before, and the glove was even more comfortable due to the light perfboard, thinner glove, and holes for the fingertips. Furthermore, the touch sensors were highly responsive with a large dynamic range between being at rest and being touched.

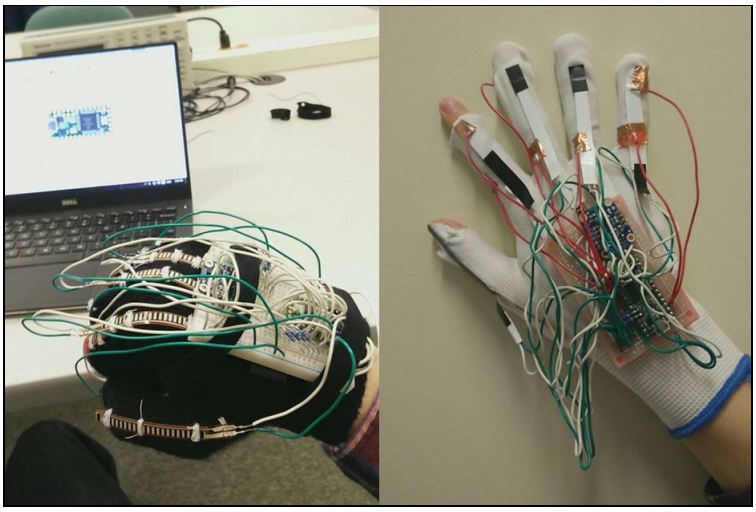

Figure 10: (Left) Breadboard prototype, (Right) Final perfboard soldered prototype

Sensor Jitter

When viewing the OpenGL hand animation, it was visibly clear that the flex sensor values had a high frequency noise component causing the animated fingers to jitter on screen. This can be attributed to micromotions of the user's hand as well as the thermal noise that can be observed in any resistor. The jitters were approximately ±5° in magnitude, which wouldn't affect the sign language classification. Furthermore, the program averages the readings over a *moving window of 25 samples* and the noise is *zero mean*, so the improvement in SNR from averaging is:

$$\frac{SNR_{avg}}{SNR} = 20 \times log(25) = 28.0dB$$

Hence there were no issues with noise in the sensor data. However, the smoothness of the hand animation is an important concern for user-experience. The solution is to add capacitors in parallel with the flex sensors, thus making a passive lowpass filter that attenuates the noise. This was not feasible with the limited space on the perfboard, but it will be implemented in the PCB based prototype using surface mount capacitors.

Classification Accuracy

Figure 11 shows the capabilities of HandsOn as demoed on poster day. Based on this sample of letter classification, an estimate of the accuracy can be made. Out of all the words shown, there is one mistake in classification in the third row: it says “REX” instead of “RED” because the classifier outputted X instead of D that time. All the other words in the screenshot are correct, and there are a total of 48 letters. The accuracy is then:

$$Classifier\ accuracy = \frac{47}{48} \times 100 = 97.9\%$$

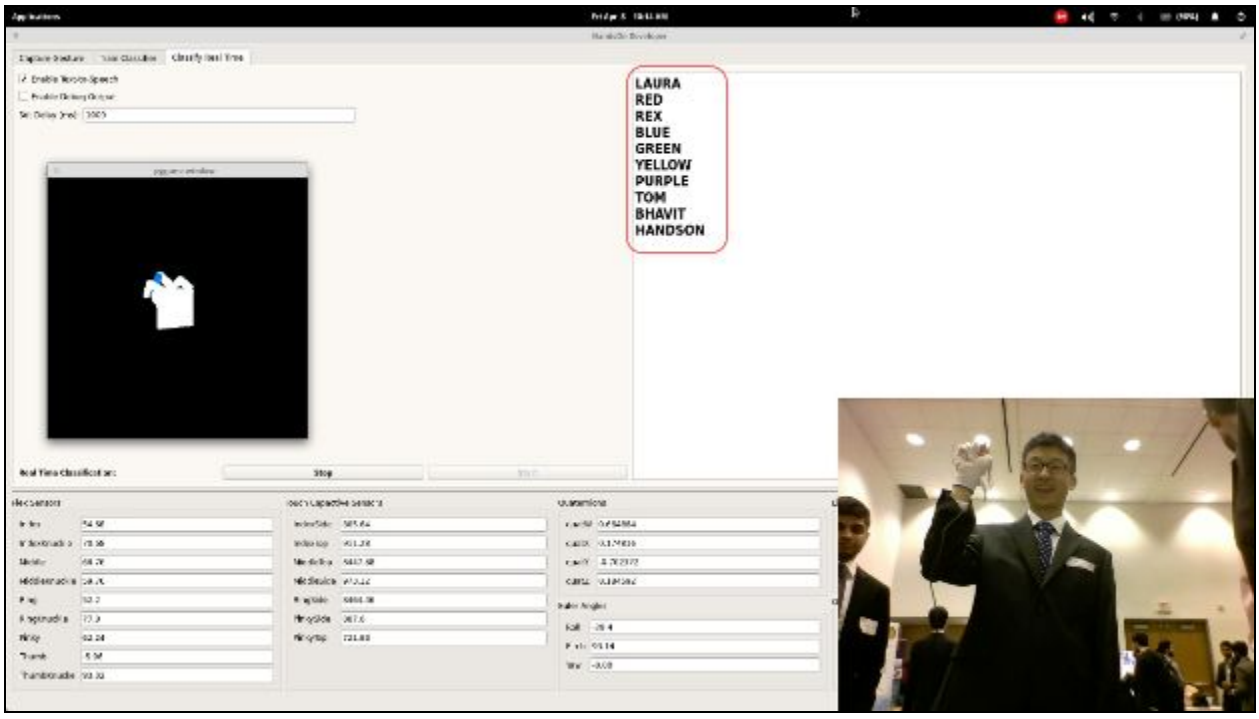

Figure 11: HandsOn GUI showing sign language classification on poster day

Classifier Training

An important consideration for achieving high accuracy of the classifier is the amount of training required. There is a tradeoff between number of training sets and classifier accuracy, and it is

important from the user-experience perspective that high accuracy can be achieved with minimal training. For the demo shown in Figure 11, the number of training sets for each letter was **5**. For 26 letters, this equals 26x5=**130** total sets. Hence the time needed for training is:

$$Time\ per\ training\ set = num\ of\ samples \times sampling\ period = 25 \times 20ms = 500ms$$

$$Miscellaneous\ time\ spent\ (user\ typing\ in\ letter\ and\ clicking\ mouse\ button) \approx 3s$$

$$Training\ time = 130 \times (0.5s + 3s) = 455s = 7.58\ min$$

Hence it would take the user around 7.5 minutes to set up the glove for their specific style of signing, to achieve approximately 98% classification accuracy. This is quite a remarkable result for a small training time, mostly due to the large number of dimensions in the feature space. However, modern apps and devices are very competitive in speed and user experience so it is desirable to improve these statistics for maximum user satisfaction. This can possibly be accomplished using a large database of training sets from thousands of users and pre-training the classifier on that database. Alternatively, there can be an adaptive training algorithm that automatically improves itself every time the user makes a gesture.

Conclusion & Future Improvements

Our group had very high aspirations for this HCI glove. As a result, there are many future improvements that have already been considered. There are already plans to continue working on this project to see how far it can go. Hopefully, it will have some marketable value – whether that is as a sign language product or as a general HCI product.

One of the major improvements to the glove would be aesthetics. For the purpose of the capstone project, we only had time to focus on the functionality of the glove. However, aesthetics are a major part of marketing of the glove and an area that can be significantly improved. One of the first steps would be to create a PCB to replace the perfboard and minimize the number of wires. Also, conductive thread could be used to wire the flex sensors and touch capacitive sensors although the effectiveness of this thread still needs to be tested.

Another major improvement would be to reduce the cost. The bulk of the cost was due to the flex sensors. The flex sensor technology is the same as the ones used in the Nintendo Power glove but it still costs $12/sensor for a total of $108 per glove. There are a few guides for custom-made flex sensors, some of which are fabric-based. These would allow easy integration into the glove and would be aesthetically pleasing. The disadvantage is that it likely has reduced accuracy/resolution and linearity.

The glove was meant to be wireless to prevent any movement restriction so that would be a must in the next iteration of the device. Bluetooth and battery are fairly straightforward additions.

Absolute position information is the next step in this project – and probably the most challenging. The easiest way to obtain absolute position information with accuracy is to use an optical-based approach. However, the whole idea was to create an alternative to optical such that the glove can be used without having to face a camera, and could be used as a portable device. Finding the position from the accelerometer is an option, but it contains drift error due to the double integration that accumulated very quickly. There are ways to compensate/minimize the drift error which need to be explored. The glove would constantly have to be recalibrated to a known position thus could be annoying to the user if it occurred too often.

Once absolute position information is obtained, gestures involving movement can be explored. SVMs are not useful for these types of gestures but we could use another form of machine learning. In our research thus far, Hidden Markov Models (HMMs) seem to be a reasonable starting point to explore classification of dynamic gestures. Another glove could then be created to allow classification of any word or phrase in ASL.

Before investing the time and money into such a massive continuation of the project, it would be important to gather feedback from the deaf and mute communities on the device. It may be possible that they would not prefer to have such a device. In that case, a greater emphasis could be placed on creating a general purpose device to be applied to gaming, and design (3D modelling, CAD software, etc.).

As demonstrated in the Results & Discussion, as well as on ECES Expo day, the goals for this project were clearly met. The device accurately classified ASL letters and numbers and performed text-to-speech on the spelled word. There are many plans to continue improving the device with the goal of hopefully entering the market one day.

References

[1] Harington, Tom. "Sign Language Tags: Deaf, Faq ." *ASL: Ranking and Number of Users*. N.p., 1 May 2010. Web. 19 Oct. 2015. <http://libguides.gallaudet.edu/content.php?pid=114804&sid=991835>.

[2] Motion of fingers (excluding thumb). Digital Image. <http://www.outlanderanatomy.com/wp-content/uploads/2015/06/finger-movements-KLS-edited.jpg>

[3] Motion of thumb. Digital Image. <https://www.studyblue.com/notes/note/n/anatomy-exam/deck/4771081>

[4] Finger Spelling. Digital Image. <http://www.lifeprint.com/asl101/fingerspelling/images/abc1280x960.png>

[5] "What is the relation between the number of Support Vectors and training data and classifiers performance?," *Stack Overflow*. http://stackoverflow.com/questions/9480605/what-is-the-relation-between-the-number-of-support-vectors-and-training-data-and

[6] Sheet, Data. "BNO055 Intelligent 9-axis Absolute Orientation Sensor." (n.d.): n. pag. Bosch, Nov. 2014. Web. Dec. 2015. <https://cdn-shop.adafruit.com/datasheets/BST_BNO055_DS000_12.pdf>

Appendix

BNO055 IMU [6]

" The BNO055 is a System in Package (SiP), integrating a triaxial 14-bit accelerometer, a triaxial 16-bit gyroscope with a range of ±2000 degrees per second, a triaxial geomagnetic sensor and a 32-bit cortex M0+ microcontroller running Bosch Sensortec sensor fusion software, in a single package. The corresponding chip-sets are integrated into one single 28-pin LGA 3.8mm x 5.2mm x 1.1 mm housing. For optimum system integration the BNO055 is equipped with digital bidirectional I2C and UART interfaces. The I2C interface can be programmed to run with the HID-I2C protocol turning the BNO055 into a plug-and-play sensor hub solution for devices running the Windows 8.0 or 8.1 operating system. "

The IMU can output the following data:

1. Orientation (Euler Vector, 100Hz)
   Three axis orientation data based on a 360° sphere
2. Orientation (Quaternion, 100Hz)
   Four point quaternion output for more accurate data manipulation
3. Angular Velocity Vector (100Hz)
   Three axis of 'rotation speed' in rad/s
4. Acceleration Vector (100Hz)
   Three axis of acceleration (gravity + linear motion) in m/s^2
5. Magnetic Field Strength Vector (20Hz)
   Three axis of magnetic field sensing in micro Tesla (uT)
6. Linear Acceleration Vector (100Hz)
   Three axis of linear acceleration data (acceleration minus gravity) in m/s^2
7. Gravity Vector (100Hz)
   Three axis of gravitational acceleration (minus any movement) in m/s^2
8. Temperature (1Hz)
   Ambient temperature in degrees Celsius

3.6.3 **Fusion Output data rates**

Table 3-14: Fusion output data rates

BNO055 Operating Mode	Data input rate			Algo calling rate	Data output rate			
	Accel	Mag	Gyro		Accel	Mag	Gyro	Fusion data
IMU	100Hz	NA	100Hz	100Hz	100Hz	NA	100Hz	100Hz
COMPASS	20Hz	20Hz	NA	20Hz	20Hz	20Hz	NA	20Hz
M4G	50Hz	50Hz	NA	50Hz	50Hz	50Hz	NA	50Hz
NDOF_FMC_OFF	100Hz	20Hz	100Hz	100Hz	100Hz	20Hz	100Hz	100Hz
NDOF	100Hz	20Hz	100Hz	100Hz	100Hz	20Hz	100Hz	100Hz

Software

As mentioned in the Software section, there was 1,904 lines of code for the sign language project (not including some additional scripts for optimization and mouse UI control). Therefore, all of the code will not be provided in this document. All the code is open source and available on GitHub: https://github.com/bhavesh-k/hands-on

An overview of the files and functions contained within each one is given below with the code removed.

HandsOn_GUI_main.py

```
"""
HandsOn_GUI_main
Authors: Bhavit Patel, Bhavesh Kakwani, Tom Yang
Date Created: March 2016
Developer GUI Application using multi-threading to capture gestures; setup machine learning as gesture classifier; perform real-time gesture classification
while parsing serial data from an HCI glove. Also plots all input signals and displays a hand animation using OpenGL in separate threads
"""

class DevApp(QtWidgets.QMainWindow, HandsOn_GUI_Layout.Ui_MainWindow, QtCore.QObject):
  """ HandsOn Developer GUI Application """

  def __init__(self):
        """ Constructor method """

  def _initButtons(self):
        """ Initialize all buttons and connect them to appropriate methods """

  def SetGestOutFile(self):
    """ Sets file name and opens a file used to save gesture data with a user-input gesture label. Outputs contents of file in GUI"""

  def CaptureGesture(self):
    """ Captures gesture to file with a user-input gesture identifier and flex, touch, quaternion sensor data """

  def SetTrainFile(self):
    """ Sets file name and opens a file used to train machine learning classifiers. Outputs contents of file in GUI """

  def TrainClassifier(self):
    """ Trains the classifier using the training examples in "self.trainFileName" """

  def StartClassifyThread(self):
    """ Instantiates and runs a QThread class to perform the gesture classification continuously in real time """

  def EndClassifyThread(self):
```

```
        """ Terminates Classify thread """

    def UpdatePredictionDisplay(self, predList):
        """ Updates GUI display after signal emitted by ClassifyRealTime thread indicating that a gesture has been
predicted """

    def LogSession(self, predList):
        """ Logs the classify real time session with predicted gesture and input data """

    def StartSerialParseThread(self):
        """ Instantiates and runs a QThread class to perform the serial data parsing continuously """

    def UpdateSensorDisplay(self):
        """ Updates GUI display after signal emitted by ParseSerial thread indiciating sensor values have been updated
"""

    def StartAnimateThread(self):
        """ Instantiates and runs a QThread class to perform the hand animation """

    def EndAnimateThread(self):
        """ Terminates Hand Animation Thread """


class SerialParse(QtCore.QThread):
    """ Threading class using QThread to perform parsing of serial data containing flex sensor, touch sensor, and IMU
data """
    # Signal to communicate with main program class. signal will emit and run a method we connect it to in the main
program class
    sig_UpdateData = QtCore.pyqtSignal()

    def __init__(self):
            """ Constructor method """
        super(SerialParse, self).__init__()

    def __del__(self):
        self.wait()

    def runTest(self):
        """ Simple test for SerialParse threading, multidimensional lists of deque objects, and updating GUI without glove
data """

            def run(self):
        """ Opens serial port and parses data. Emits signal to update GUI sensor display values after sensor data in
share_var has been updated """


class ClassifyRealTime(QtCore.QThread):
    """ Threading class using QThread to perform real-time classification of hand gestures """
    # Signal to communicate with main program class. signal will emit and run a method we connect it to in the main
program class
```

```
    sig_PredictedGest = QtCore.pyqtSignal(list)

    def __init__(self, trainedClassifier, delay, debugFlag, ttsFlag):
            """ Constructor method """

    def _initTTS(self):
            """ Text-to-speech constructor method """

    def __del__(self):
        self.wait()

    def run(self):
            """ Performs real-time classification and text-to-speech output """


class HandAnimation(QtCore.QThread):
    """ Threading class using QThread to perform hand animation using OpenGL """

    def __init__(self):
            """ Constructor method """
        super(HandAnimation, self).__init__()

    def __del__(self):
        self.wait()

    def run(self):
            """ Performs hand animation with use of Animation.drawHand() and Pygame """
```

HandsOn_GUI_Layout.py

```
# Form implementation generated from reading ui file 'HandsOn_GUI_Layout.ui'
# Created by: PyQt5 UI code generator 5.5.1

class Ui_MainWindow(object):
    """ Class to setup the GUI Layout and place all buttons, layout windows, text display, etc. """

    def setupUi(self, MainWindow):
            """ Places all UI elements """

            def retranslateUi(self, MainWindow):
            """ Scales and retranslates all UI elements when window is resized """
```

Tools.py

```
"""
Tools
Authors: Bhavit Patel, Bhavesh Kakwani, Tom Yang
Date Created: March 2016
```

```
Contains multiple functions and tools for processing hand sensor data obtained
from parseSerialData() and stored in global variables or deque objects in share_var
"""

def UpdateDequeData():
    """ Updates the collections.deque instances of all hand sensor data (flex sensor, touch sensors, quaternion, euler
angles, lin accel) """

def DequeMean(deq):
    """ Returns the mean as a float of a collections.deque instance
        Input: deq - collections.deque instance (ex. flexIndexFingerCollect, qWCollect, etc.)
        Output: mean - mean for the collections.deque instance  """

def DequeMeanList(deqList):
    """ Returns the means for a list of collections.deque instances
        Input: deqList - list of collections.deque instances (ex. flexCollectList, touchCollectList, etc.)
        Output: meanList - list of current means for the collections.deque instances  """

def ListToCSstr(list):
    """ Returns a comma separated string consisting of values in input list """

def FlexCurrDataList():
    """ Returns the current instananeuous flex sensor values in the form of a list """

def FlexCurrDataStr():
    """ Returns the instananeous Flex Sensor values in the form of a string separated by ","s """

def FlexMeanDataList():
    """ Returns the mean of collected Flex Sensor data as a list """

def FlexMeanDataStr():
    """ Returns the mean of collected Flex Sensor data in the form of a string separated by ","s """

def TouchMeanDataList():
    """ Returns the mean of collected Touch Sensor data as a list """

def TouchMeanDataStr():
    """ Returns the mean of collected Touch Sensor data in the form of a string separated by ","s """

def TouchMeanBoolList():
    """ Returns a boolean int (0 or 1) from the means of collected Touch Sensor data as a list
        0 if TouchMean < threshold (ie. touch sensor has not been touched)
        1 if TouchMean > threshold (ie. touch sensor has been touched) """

def TouchMeanBoolStr():
    """ Returns a boolean int (0 or 1) from the means of collected Touch Sensor data in the form of a string separated
by ","s
        0 if TouchMean < threshold (ie. touch sensor has not been touched)
        1 if TouchMean > threshold (ie. touch sensor has been touched) """
```

```
def QuatMeanDataList():
   """ Returns the mean of collected Quaternion data as a list """

def QuatMeanDataStr():
   """ Returns the mean of collected Quaternion data in the form of a string separated by ","s """

def QuatCurrDataStr():
   """ Returns the current Quaternion data in the form of a string separated by ","s """

def EulerCurrDataStr():
   """ Returns the current Euler Angle values in the form of a string separated by ","s """

def LinAccelCurrDataStr():
   """ Returns the current Linear Acceleration values in the form of a string separated by ","s """

def LinAccelMeanDataList():
   """ Returns the mean of collected Linear Accelreation data as a list """

def LinAccelMeanDataStr():
   """ Returns the mean of collected Linear Acceleration data in the form of a string separated by ","s """

def LinAccelMoving():
   """ Determines a mean of the absolute values of the previous 250ms of accelerometer data. This is to be used by
isMoving to determine if the user is moving their hand """


def isMoving():
   """ Returns a boolean if the user is moving determined with the collected linear acceleration """

def QuatToEuler(q0, q1, q2, q3):
   """ Euler Angle calculation from quaternions """

def EulerToDir(roll,pitch,yaw):
   """ Convert Euler angles to hand direction (down, up, or other) """

def QuatToDir(qW, qX, qY, qZ):
   """ Convert quaternions to hand direction (down, up, or other) """

def q_conjugate(q):
   """ Returns conjugate of input quaternion """

def q_mult(q1, q2):
   """ Performs quaternion multiplication """

def qv_mult(q1, v1):
   """ Performs vector rotation with quaternion """

def printInstHandDataToFile(fileName, str_handIdentifier):
   """ Inputs a letter/number/gesture identifying the hand animation being performed
      Writes the identifier with the instantaneuous hand data to a file
```

```
    Inputs:     fileName - specify entire filename with extension (ie. 'test.csv')
    Outputs:    writes to file
  """

def printHandDataToFile(fileName, str_handIdentifier):
  """ Inputs a letter/number/gesture identifying the hand animation being performed
    Writes the identifier with the appropriate hand data to the file specified
    Inputs:     fileName - specify entire filename with extension (ie. 'test.csv')
    Outputs:    writes to file
  """

def readHandDataFromFile(fileName):
  """ Reads and parses the hand data saved in a file
    Stores the identifier in numpy array "signTarget"
    Stores the hand data in numpy array "signFeatures"
    Inputs:     fileName - specify entire filename with extension (ie. 'test.csv')
    Outputs:    signTarget - nx1 array of targets/identifiers/labels
            signFeatures - nxm array of m features for each of the n targets
  """
```

share_var.py

```
"""
share_var
Authors: Bhavit Patel, Bhavesh Kakwani, Tom Yang
Date Created: 2016

Variables for instantaneuous hand sensor data and collections.deque objects for storing real-time moving window of sensor data
Global to allow usage by multiple functions. Deques are thread-safe allowing multi-threading with GUI, serial parsing, and animation
"""
## Global variables to be used by various functions
global alt, temp, sysCal, gyroCal, accelCal, magCal, \
    flexIndexFinger, flexIndexKnuckle, \
    flexMiddleFinger, flexMiddleKnuckle, \
    flexRingFinger, flexRingKnuckle, \
    flexPinkyFinger, flexThumb, flexThumbKnuckle, \
    touchIndSide, touchIndTop, touchMidSide, touchMidTop, \
    touchRing, touchPinkySide, touchPinkyTop, \
    qW, qX, qY, qZ, \
    roll, pitch, yaw, direction, \
    accelX, accelY, accelZ
## Deque objects for fast and efficient real-time data collection. Also thread-safe to allow usage with GUI, animation, and various functions
maxNumSamples = 25 #Set moving window length
# Flex Sensors
flexIndexFingerCollect = collections.deque([0], maxNumSamples)
flexIndexKnuckleCollect = collections.deque([0], maxNumSamples)
flexMiddleFingerCollect = collections.deque([0], maxNumSamples)
```

```
flexMiddleKnuckleCollect = collections.deque([0], maxNumSamples)
flexRingFingerCollect = collections.deque([0], maxNumSamples)
flexRingKnuckleCollect = collections.deque([0], maxNumSamples)
flexPinkyFingerCollect = collections.deque([0], maxNumSamples)
flexThumbCollect = collections.deque([0], maxNumSamples)
flexThumbKnuckleCollect = collections.deque([0], maxNumSamples)
flexCollectList = [flexIndexFingerCollect, flexIndexKnuckleCollect, flexMiddleFingerCollect, \
                   flexMiddleKnuckleCollect, flexRingFingerCollect, flexRingKnuckleCollect, \
                       flexPinkyFingerCollect, flexThumbCollect, flexThumbKnuckleCollect]
# Touch Capacitive Sensors
touchIndSideCollect = collections.deque([0], maxNumSamples)
touchIndTopCollect = collections.deque([0], maxNumSamples)
touchMidSideCollect = collections.deque([0], maxNumSamples)
touchMidTopCollect = collections.deque([0], maxNumSamples)
touchRingCollect = collections.deque([0], maxNumSamples)
touchPinkySideCollect = collections.deque([0], maxNumSamples)
touchPinkyTopCollect = collections.deque([0], maxNumSamples)
touchCollectList = [touchIndSideCollect, touchIndTopCollect, touchMidTopCollect, touchMidSideCollect, \
          touchRingCollect, touchPinkySideCollect, touchPinkyTopCollect]
# Linear Acceleration
accelXCollect = collections.deque([0], maxNumSamples)
accelYCollect = collections.deque([0], maxNumSamples)
accelZCollect = collections.deque([0], maxNumSamples)
accelCollectList = [accelXCollect, accelYCollect, accelZCollect]
# Quaternions
qWCollect = collections.deque([0], maxNumSamples)
qXCollect = collections.deque([0], maxNumSamples)
qYCollect = collections.deque([0], maxNumSamples)
qZCollect = collections.deque([0], maxNumSamples)
quatCollectList = [qWCollect, qXCollect, qYCollect, qZCollect]
# Euler Angles
rollCollect = collections.deque([0], maxNumSamples)
pitchCollect = collections.deque([0], maxNumSamples)
yawCollect = collections.deque([0], maxNumSamples)
eulerCollectList = [rollCollect, pitchCollect, yawCollect]
# Multidimensional list containing all deque objects for easier processing
sensorCollectList = [ flexCollectList, touchCollectList, accelCollectList, quatCollectList, eulerCollectList ]
```

HandsOn.py

```
"""
HandsOn
Authors: Bhavit Patel, Bhavesh Kakwani, Tom Yang
Date Created: January 2016
Version 1

Also contains pseudoMain() which is a console application to save gesture data to file,
train SVM from data in a file, and classify gestures in real time (also with text-to-speech output).
"""
```

```
def pseudoMain(delay, debugFlag, ttsFlag):
    """ Console menu application for capturing hand gestures, training SVM, and classifying gestures """
    print("-----------------------------------------------")
    print("                    HandsOn                    ")
    print("-----------------------------------------------")
    print("MENU:")
    print("1. Capture Hand Gesture Data to File")
    print("2. Train SVM with Hand Gesture Data From File")
    print("3. Load SVM from File")
    print("4. Predict Hand Gestures")
    print("5. Exit\n")

def parseSerialHandData(ser):
    """ Parses serial data continuously and assigns to appropriate global variables """

def parseLineData(line):
    """ Parses serial line and assigns to appropriate global variables """
```

Animation.py

```
"""
Animation
Authors: Bhavit Patel, Bhavesh Kakwani, Tom Yang
Date Created: January 2016
Version 1

OpenGL real-time hand animation. Multi-threading with parseSerialData and pseudoMain
to run the entire HandsOn application
"""

def drawHand():
    """Draws the 3D hand using euler angles and finger joint angles"""

def main():
    """Sets up the PyGame window, view angles, aspect ratio, and continuously performs   hand animation"""
```