ECE 167 - Sensing and Sensor Technology
ASL Translation Glove Final Project

Alex Nivelle, Matthew Kaltman, Tim Kraemer

July 30, 2023

Contents

1 Abstract **3**

2 Problem/Motivation **3**

3 Bill of Materials **4**

4 Electromechanical Implementation **4**
- 4.1 Glove Construction 4
- 4.2 Circuit Construction 5
  - 4.2.1 Initial Attempts 5
  - 4.2.2 Final Implementation 6

5 Software Implementation **8**
- 5.1 Modified Binary Tree ADT 8
- 5.2 Custom Binary Tree Traversal Algorithm 9
- 5.3 Node.c Library & Other C Files 10
  - 5.3.1 Library functions 10
  - 5.3.2 Flexion Program File for PIC32 11
- 5.4 Software Filtering 12
  - 5.4.1 Moving Average Filter 12
  - 5.4.2 Initial Finger Calibration 12

6 Testing & Results **13**
- 6.1 Software Testing 13
- 6.2 Hardware Testing 13
- 6.3 Results 14

7 Limitations and Challenges **14**

8 Future Work **14**

9 Conclusion **15**

1 Abstract

This lab report presents the development of a glove-based American Sign Language (ASL) recognition system that has the capability of detecting specific ASL poses for different letters and numbers, and accurately output the corresponding value that is currently being signed. The system utilizes flex sensors integrated onto each finger of a glove to capture the flexion and extension positions of the fingers. By employing a modified binary tree structure combined with a custom traversal algorithm, the system efficiently stores and traverses through the possible signs based on the individual finger positions. The design and implementation of the glove system involve sensor integration, data acquisition, signal processing, feature extraction, and classification stages. Experimental results demonstrate the effectiveness of the glove system in accurately recognizing ASL signs, and achieving a high level of recognition accuracy.

2 Problem/Motivation

Deafness is a common disability that plagues society, and provides difficulties between individuals when it comes to communication. In addition, ASL is not a common language that individuals learn, which leads to communication issues between those who suffer speech impediments and those that do not understand sign language. This was one of the main motivations behind this project, as we were curious whether there was an ideal solution with the usage of embedded systems and sensing technologies. After careful consideration of design implementations, we concluded that the easiest way of realizing this idea was to use resistance flex sensors to track the position of each finger. We set the goal to be able to translate at least half of the English alphabet letters in addition to all numbers between 0 and 10. This goal was decided upon based on the amount of time given for the project, as well as inherent limitations that flex sensors have, however we will discuss and plan out additional implementations to show possibilities on how to build onto this project to ultimately reach the goal of being able to translate all letters in the alphabet, among all of the letters from 0 to 10.

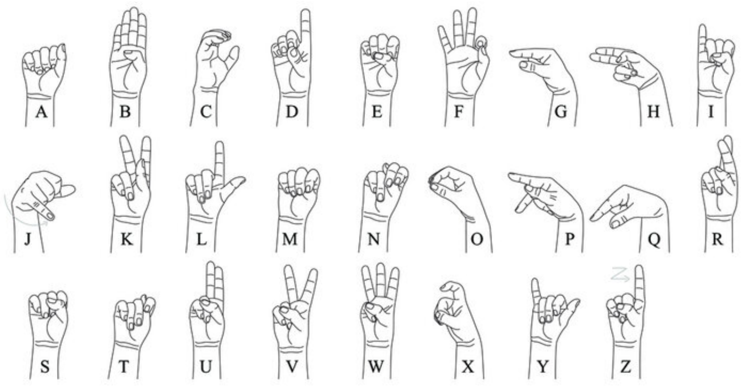

Figure 1: ASL Alphabet Signs (Provided By Adobe Stock)

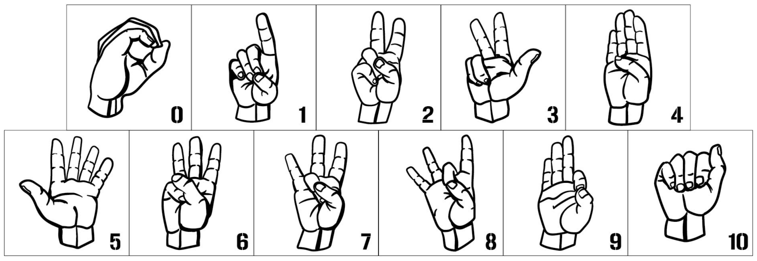

Figure 2: ASL Number Signs (Provided by Rintablee Graphics)

3 Bill of Materials

Component	Quantity	Price per Unit
Thin Film Resistive Flex Sensor 4.33 in	3	$9.50
Adafruit Short Flex Sensor 3.053 in	2	$7.95
Long Finger Nylon Biker's Glove	1	Provided
Uno32 MicroController + I/O Shield	1	Provided
Pickit 3 MicroController Programmer	1	Provided
3x7cm Addicore Perfboard	1	$0.90
67 kΩ Resistor	4	Provided
47 kΩ Resistor	1	Provided
1μF Capacitor	5	Provided
MPLabX IDE	1	Free
Pickit3 Debugger	1	Provided
Hot-glue Gun & Glue Stick	1	Provided
3 inch Zipties	5	Provided

4 Electromechanical Implementation

4.1 Glove Construction

The translation glove consists of an array of 5 flex sensors, one for each digit. The prototype provides crude, yet stable connections of the sensor to the fingers of the glove, using electrical tape at the tips for a solid connection. There are rings on the glove constructed from zip-ties and paper in order to hold the sensors in place reliably, while also allowing the sensors to travel linearly up and down the hand as the fingers are flexed. Our team encountered a lot of problems with prototyping a reliable way to hold the sensors while still allowing that linear travel, and we feel that further improvements could be made through a custom printed sensor holder that could connect to light tension springs in order to pull the sensors back, preventing the sensors from folding in on themselves. A picture of the fully constructed prototype is provided below in Figure 3.

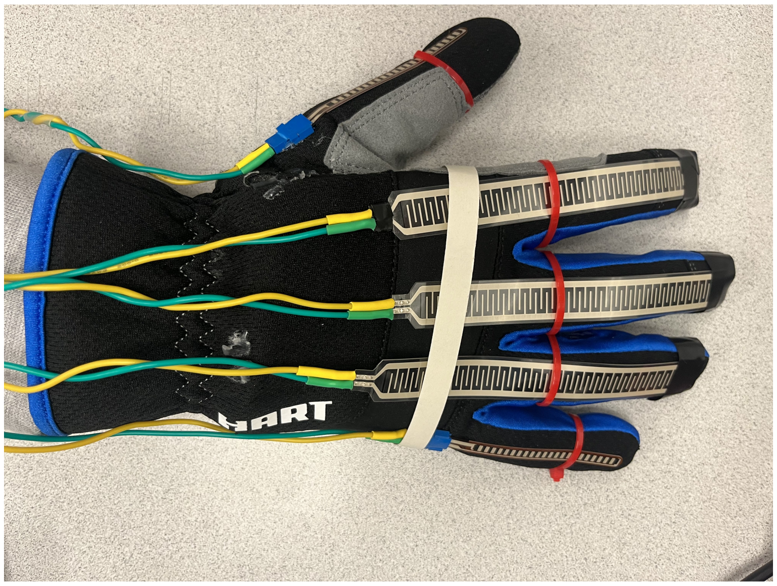

Figure 3: ASL Translation Glove Final Construction

4.2 Circuit Construction

The construction of the circuit for this glove centered around determining the resistance created by flexing and straightening each sensor, and finding a suitable circuit and component values in order to give us the widest range and sensitivity in order to accurately determine hand poses.

4.2.1 Initial Attempts

Our groups initial plan was to implement an instrumentation amplifier driven by a Wheatstone bridge with our flex sensors in place as the unknown resistance, R_4. The original circuit that was designed and prototyped is given below in figure 4.

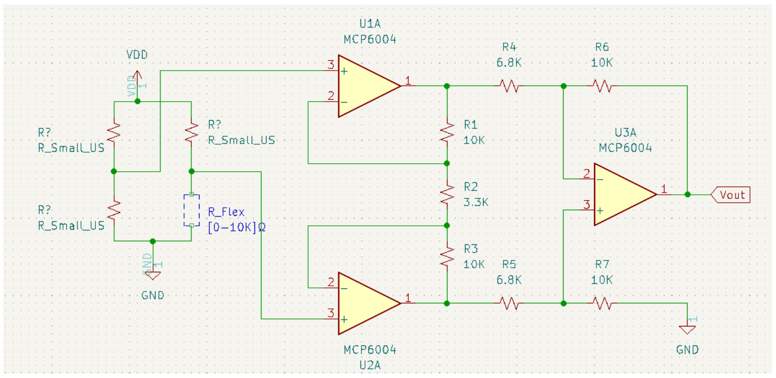

Figure 4: Prototype Circuit: Wheatstone bridge and instrumentation amplifier configuration

Despite extensive simulations in Matlab and LT-Spice yielding favorable results where the signal would would range from [0-3.3]V depending on the flex sensors angle, the real world implementation of this configuration proved to be incredibly challenging. One major issue with the constructed Wheatstone configuration is that the sensors we had ordered were nowhere near the described performance on the seller's website. Whereas the seller had described a range from [0-10K]Ω, the actual performance of the sensor yielded a result of 3KΩ when fully flexed and a ranging resistance of [50K-500K]Ω when held completely straight. This proved to be an impassible problem under our team's time restraint and lack of familiarity with incredibly high ranging precision circuits as when the output signal was observed under a oscilloscope the signal would remain at 3.3V when completely straight, however any amount of flex in the finger caused the output to immediately drop to 0V. Because of the approximate 500% error in the given sensors and the massive difficulty of replicating the circuit in a small form-factor perfboard, the team felt it was best to move on to a different approach for the hardware configuration.

4.2.2 Final Implementation

After turning away from the precision-based Wheatstone/In-Amp circuit our team decided to follow the same concept provided in the Flex-Sensor lab. The base design of our circuit was a voltage divider in order to obtain an output voltage that varied with each finger's pose in order for us to reliably determine pose over the uC32's ADC. This base schematic is provided below in figure 5.

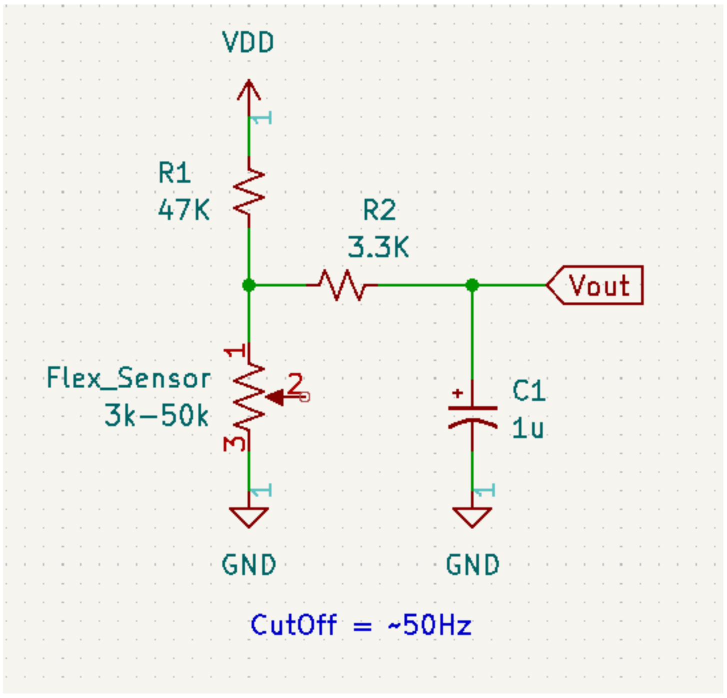

Figure 5: Final Circuit: Voltage Divider fed into Low-Pass

The circuit shown above was designed to get the greatest range of voltages out of the voltage divider while providing some smoothing to the incoming signal as it went to the ADC. Our team was able to maintain a maximum resistance of 50KΩ when mounting the sensors so a 47KΩ resistor was found to be sufficient for the final circuit.

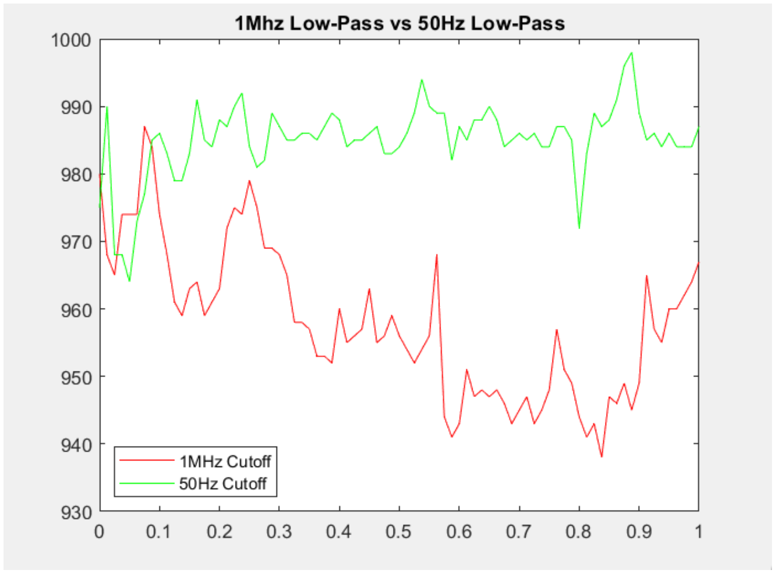

Figure 6: 1Mhz cutoff (Red) vs 50Hz cutoff (Green)

Shown above in figure 6 is a comparison of our initial 1Mhz cut-off vs our 50Hz cut-off frequency. Upon further inspection the 1Mhz has a a range of about 70mV whereas the 50Hz cut-off frequency yields about 50mV range which is enough of a difference to validate the use of the lower frequency cutoff. This cut-off frequency still allowed for noise but when a finger was flexed and straightened quickly the signal was lost in the RC rise and fall time so a shorter time constant was avoided.

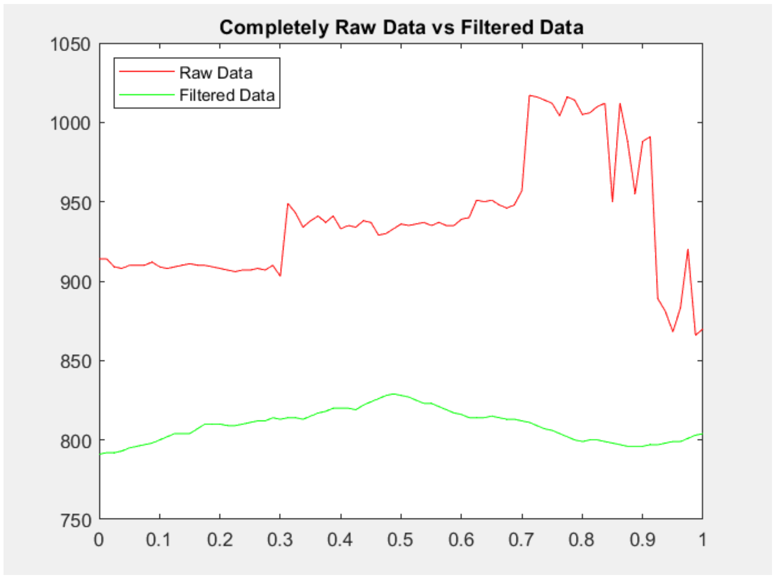

Figure 7: Raw Data (Red) vs Hardware/Software Filtered Data (Green)

The final circuit implemented for this glove was nowhere near the initially anticipated configuration. Due to sensors being broken upon arrival from Amazon and some ADC ranges ranging at values as small as 10 on the uC32's ADC our team had to scramble and collect spare flex sensors to replace the faulty sensors on the thumb and pinky. A comparison of the completely raw sensor reading vs the same sensor ran through a voltage divider and low-pass filter with additional software smoothing in the form of a moving average filter is provided above and helps to visually quantify our teams improvement as time progressed.

5 Software Implementation

5.1 Modified Binary Tree ADT

To adequately represent as many finger flexion configurations as possible in a clean and concise method using solely the flex sensors, we chose to create a binary tree to store all possible ASL letter and number hand positions.

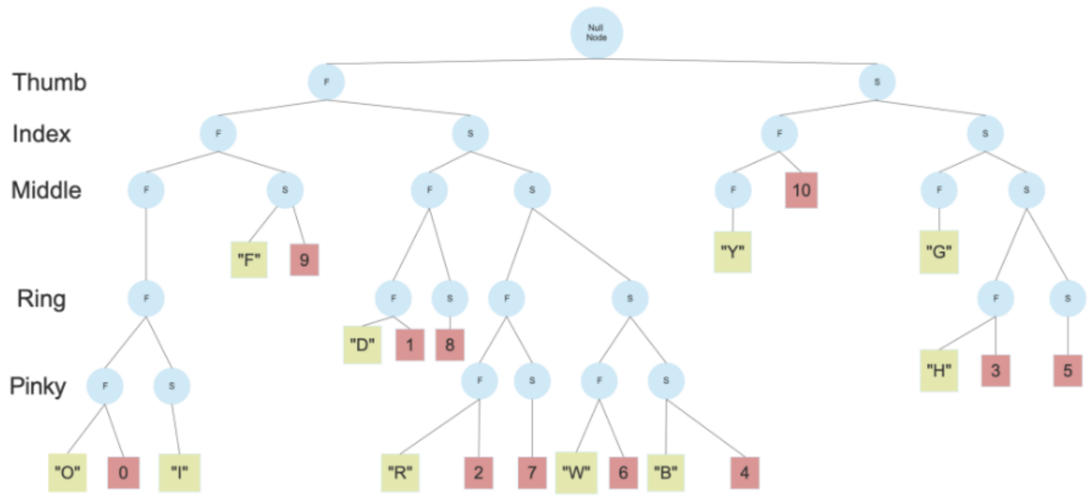

Figure 8: Modified Finger Position Binary Tree

Above is our own design and implementation of the modified binary tree ADT (Abstract Data Type). “Flexed” and “Straight” nodes are displayed with circular shapes, while letter and number nodes are represented with square shapes. The tree has 5 active layers, each representing a finger. Above these layers sits a dummy node, where the traversal starts. The layers start with the thumb, and move towards the pinky finger. It is simple to see that each letter/number has its own unique pathway from the `Null` node at the top. It is also worth noting that all of the numbers and letters listed in this tree are all signs that can be completely distinguished with only 2 finger positions, Flexed and Straight.

The tree nodes are defined in code as follows:

```
typedef struct Node {
    uint8_t numChildren;    // Number of children this node has
    struct Node **children; // An array of pointers to the actual children
    uint8_t isNum;          // Whether or not this node contains a number
    Flexion_t flex;         // The finger position this node corresponds to
    char *val;              // The numerical value of this node
    uint8_t isChar;         // Whether or not this node contains a letter
    char *letter;           // The corresponding letter
} Node_t;
```

5.2 Custom Binary Tree Traversal Algorithm

Traveling through the tree requires a simple traversal algorithm that ends once the search reaches a letter or number node. This was simple to envision, though we went through several iterations (including utilizing recursion) before deciding on the final method. The general process is as follows:

```
Traversal(Node, positions[5])
    for i in every level in the tree (including dummy node)
        if current node type is number
            return node number
        else if current node type is letter and switch is not in number mode
            return node letter
        else
            if positions[i] is STRAIGHT
                current node is updated to be straight child node
            else if positions[i] is FLEXED)
                current node is updated to be flexed child node
```

The traversal starts at the input node (which by design should only ever be the dummy node mentioned previously), and assuming the node is not a letter or number, it checks the finger flexion corresponding to the *next* node. This can be reasoned from the fact that we use a dummy node. The dummy node has no corresponding finger; we check `positions[0]` to determine the initial path through the thumb, but there are still 5 levels in the tree, and we must check all of them. However, we do not have to worry about indexing outside of the `positions` array, as the final level of the tree will always contain a letter or number.

5.3 Node.c Library & Other C Files

In addition to the custom tree and traversal algorithm, we designed our own library to contain the ADT, movement function, among other functions needed to build the tree: add new children nodes, delete nodes, add number values, add character values, etc.

5.3.1 Library functions

`char *GetVal(Node_t node, Flexion_t *positions);`

`GetVal()` is our custom traversal algorithm, returns a value (if there is one) given a list of finger positions. Returns `null` for invalid signs.

`uint8_t AddChild(Node_t *parent, Node_t *child);`

`AddChild()` simply adds a new child node to a corresponding parent. This is used when creating the initial tree, and takes in as input a pointer to a parent node and a pointer to a created child node.

`uint8_t SetValue(Node_t *node, char *val);`

`SetValue()` sets the number value of a node and updates it to be a number node.

`uint8_t SetLetter(Node_t *node, char* *letter);`

`SetLetter()` sets the letter value of a node and updates it to be a letter node.

`uint8_t SetFlex(Node_t *node, Flexion_t flex);`

`SetFlex()` just resets the value of the flex state for a given node. Takes in as input a pointer to a node and a new flex state.

`Node_t DefaultInit();`

`DefaultInit()` creates a node with default initialization state: Straight finger, memory for one child, no character value, etc. Allocates the space for one child pointer and sets that memory to 0.

`uint8_t DeleteChildren(Node_t *node);`

`DeleteChildren()` frees all memory in a node's array of children. This function at the moment is not necessary since the program on the PIC32 runs indefinitely and we never have a need to delete nodes, however it might be useful in the future to be able to delete singular child nodes.

5.3.2 Flexion Program File for PIC32

`Flexion_Test.c`

`Flexion_Test.c` is the main program file for the Uno32 microcontroller, which we uploaded and ran using MPLABX. Here, we separate 3 different processes: Straight finger calibration, Flexed finger calibration, and then actual data reading plus tree traversal. Additionally there are a few helper functions that make a few mundane tasks simpler.

`unsigned int moving_avg_calc(unsigned int arr[WINDOW_SIZE])`

This function calculates the average of a given moving average filter array. WINDOW_SIZE is variable, and is defined as a global variable.

`void ReadFingers(unsigned int readings[5])`

`ReadFingers()` processes all connected AD pins and stores the raw data values associated with each finger into the input readings argument.

The actual main function follows the following pseudo code:

```
main()
    initialize board, AD
    add AD1-AD5 to trackable AD pins

    create binary tree

    while(1)
        if(calibrated)
            check if new data is ready
            add new data to individual finger moving averages
            for(all fingers)
                if moving average > flexed threshold
                    finger[i] = FLEXED
                else
                finger[i] = STRAIGHT
            print(GetValue(dummy node, fingers values))
            shift moving average arrays
        else if(not calibrated)
            start straight finger calibration
            add finger data to straight finger data array

            finish straight finger calibration, start flexed calibration

            add finger data to flexed finger data array
            finish flexed finger calibration

            flexed threshold = straight + flexed / 2

            calibrated = 1
```

The data sampling rate varies depending on the current process. Flex sensor data is sampled every half second during calibration phase, while data is sampled every tenth of a second when actively surveying for ASL signs. This allows for ample data, while minimizing errors when printing data to the serial terminal.

5.4 Software Filtering

5.4.1 Moving Average Filter

Half of the curriculum for ECE 167 is filtering and data calibration. Even though hardware filtering was implemented using low-pass filters for signal smoothing, quantization with a high resolution causes fluctuations in the digital converted data read from the AD pins, which can cause issues in threshold calculations. This is why we implemented a simple, yet effective software filtering method of moving average filters. Sequential data is read in and added into a set-size array, replacing the oldest data point with the newest sampled data point. Then, the average over the whole array is calculated, and is used as the filtered data point at that given point in time. This allows us to effectively filter out small perturbations in the data stream, and provide us with a smooth new data stream. In addition it also allows us to mute huge outlier spikes which might potentially ruin thresholds.

5.4.2 Initial Finger Calibration

Another issue that we faced was the fact that different hands caused varying ranges in voltages being read from the flex sensor voltage divider, most likely due to the fact that everyone's hands and fingers have different shapes and difference sizes, which cause the flex sensor to shift or bend slightly. To fix this issue, we implemented a calibration procedure that always takes place at the beginning of the program, which effectively measures a stream of data for around 5 seconds for both a flexed and straight position of each finger. This stream is average to give a singular flexed/straight value for each finger, and defines a general movement range, with which we find the midpoint and set this as the effective threshold between a flexed position and a straight position for each finger.

6 Testing & Results

6.1 Software Testing

Testing of our project was iterative. One of the first pieces of code written was the Node library, and as such it was one of the first components we tested. To do so, we created a simple test harness that didn't run on the Uno32, in which we first tested the ability to create nodes, the ability to add a node as a child, and the ability to give each node a character value. The testing went smoothly, with a few hiccups as we remembered how to properly use memory in C. We also at this point wrote the `DeleteChildren()` function, as we were worried about memory leaks. Upon using `valgrind` to check, however, we discovered that the code did not feature memory leaks even if we didn't deliberately free the memory allocated to the children.

Once we had confirmed the ability to create nodes, add children, and assign characters to nodes, it was time to test our tree, and traversal through it. We continued in the test harness code, and manually created the entire tree (a process we hope to fine-tune or avoid in the future). We then created a sample set of finger positions, and ran `GetVal()` on them. Our first test proved successful, returning the correct number, which was very exciting. Our second test, however, broke. After running the code a couple more times and examining it with `gdb`, we discovered an issue with the way children were accessed. We originally accessed children by their position in an array according to the finger flexion: Straight or Flexed, which was assigned 0 or 1 through an enum. However, some nodes only had one child node, which could be either a Straight or Flexed node. When this occurred, and we tried to index into the second child, we naturally broke the tree. To get around this, finger positions in the array were instead checked against each child node's flexion. We deemed this okay since there were at most two children to a node, but if more were to be added this process would likely have to be adjusted. Once this was fixed, however, the tree worked well, and we were able to move on to building the glove.

6.2 Hardware Testing

Hardware testing could only be done once the software worked flawlessly. At first we tested only the numbers, as they were the simplest to sign, and provided us with a good idea as to how accurate the glove was. Some issues we initially faced with the hardware design had to do with the original prototype of the glove. The rings we had made were sometimes too tight or too loose, which caused the flex sensors to slide about or stay flexed when moving your fingers. Several rounds of testing with various iterations led us to create these stability rings using zip-ties as they: are adjustable, did not cause much friction, and stayed in place. Occasionally the sensors will still get stuck when moving our fingers after being in the flexed position, however, simply moving them back to their original positions had no impact on the actual measurements and readings.

In addition the glove went through stress testing with several components, such as the adhesive connections of the sensors to the glove, the connection of the brace rings, and the structural stability of the soldered wires and perf-board through regular usage. Because of the wear-and-tear testing, we found the best method of adhesion on a nylon fabric is indeed hot-glue.

6.3 Results

After thorough testing of our hardware and software, we landed at the conclusion that our glove works as intended, with little to no issues. We went through all letters and numbers several times throughout the design and testing phases, including at the end once prototyping was done, and our glove was able to accurately detect every letter and number we had originally planned for with no measurement drift, missed, or incorrect values. Effectively, we ended up with a recognition rate of 100%. Our original intent was to widen the breadth of recognizable characters in a more efficient way and hopefully recognize half of the alphabet and all numbers. We succeeded in improving upon the past attempt, however, we fell short of the half-way mark and were only able to recognize 10 letters with the limitations that were present.

7 Limitations and Challenges

Navigating the path towards implementing our solution was not without its fair share of limitations and challenges, which occurred throughout the design process.

One main issue that caused much headache were the flex sensor themselves. Not all sensors are created equal, and the ones that we acquired had notable problems. Testing each sensor revealed that they had a tolerance level of around 500% compared to the manufacturer's specifications. Measurements would range from 50kΩ to 500kΩ in resistance, which caused inconsistencies in digitizing the flexion level of each sensor.

Another Challenge that we worked on was the fact that not everyone's hands are the same size. When using a standard sized glove, this causes the fingers to bend or stretch the sensors in a unique way with each hand, causing completely different ranges when digitizing the voltage divider readings. To combat this issue, we implemented a 10 second calibration phase at the beginning of the program, which measures the average flexion value when the fingers are straight and when they are flexed, and then find the midpoint, creating unique flexion thresholds for each user.

Lastly comes the issue of scalability, which is addressed in the next section. Our current model is capable of only handling around 10 letters. This is achieved when simply quantifying the finger position into two states, flexed or straight. We have attempted adding a third position "half flexed", however due to the wide variation in sensor output, it was practically impossible to get solid readings in the correct position. Additionally, introducing a new position would create a massive binary tree, while only adding a few more characters into what we are able to read. Effectively, our current method is not sustainable for increasing the range of detectable characters.

8 Future Work

If given more time to work on this project, we would consider experimenting with more efficient data structures, as well as the addition of more sensors (ie IMU or vibration-piezo sensor for movement, touch sensor for similarly posed gestures, etc.). Even more advanced, we would implement a machine learning model, and train it with data from the flexion sensors at each position to make accurate predictions without the need for complex data structures or additional sensors.

A complete project would entail being able to detect complete ASL signs rather than just letters, and having a glove that is simple to wear, and easy to calibrate so that anyone, no matter the size of their hand, can use it.

9 Conclusion

This project was a great learning experience for our team in the process of creating an overarching project plan as well as deepening our understanding of the intricacies involved in the integration between electromechanical systems and software. This implementation of translation between ASL and text, while a novelty, does not seem to be an idea worth continuing as it impractical to require those who are hard of hearing to always wear a glove for translation as there are already apps and other devices that can accomplish the same base task with much less technical complexity.