Haptic Exoskeleton For Learning Piano

Final Report

Gordon Macshane
Kevin Wong
Zachary Xing
Stanley Zheng

ABSTRACT

We describe the implementation of a force-feedback exoskeleton hand that uses Passive Haptic Learning (PHL), a technique that improves subconscious motor function with robotic guidance, to improve form, precision, and muscle memory for finger-based tasks by guiding finger motion. Our system consists of two exoskeleton hands; one that records the finger motions of someone proficient in a motor task, and one that controls the finger motions of someone untrained in that task. We believe that mimicking expertise will speed up the development of muscle memory. We discuss previous studies related to our work on PHL, initial design decisions, our final product design, and results from our experimental learning trials.

TABLE OF CONTENTS

1. INTRODUCTION

Passive Haptic Learning is a new field of research that utilizes extrasensory information provided by robotic systems to enhance motor learning. While haptic systems that facilitate motor learning have existed since the 1970's, none have used "passive" learning. PHL takes advantage of the fact that human brain incorporates both feedback from the motor cortex and motor neurons from the muscles and skin when creating memories for performing motions in the cerebellum. This implies that extrasensory information provided to the motor neurons can influence muscle memory. Several types of haptic technology exist, but we choose to focus on force-feedback for reasons described in section 2A.

Force-feedback robotic systems have existed for around 50 years with applications in gaming, surgery, and muscle rehabilitation. The purpose of force-feedback is to provide extra-sensory information (force) that moves the user's limb to the correct course of action.

We propose a system that uses an exoskeleton hand to improve muscle memory by moving the user's finger motions for a complex task over many guided repetitions of that motion. We have specifically built the exoskeleton for playing piano, but this device can be generalizable to be used for many other finger motions.

The way the exoskeleton plays piano is by mimicking data received from a separate recorder glove, which records the finger motion of a proficient piano player. Force-feedback systems generally function by applying a small force that indicates to the user what direction his or her motion should follow. Our proposed system is different in that the exoskeleton moves the user's fingers by itself, while the user keeps his or her hand limp.

Other research on haptic (extra-sensory feedback) learning has focused primarily on vibrotactile feedback, which relays lower qualities and quantities of information than force-feedback. A small study done by Georgia Tech also used piano playing in a vibrotactile learning trial, so our project will serve as a benchmark for comparing the two methods [1].

A. Purpose

Learning how to play the piano is difficult. One needs to play the right notes at the right time, and have proper form to play well. It requires a lot of time and effort to practice. Passive learning systems could allow beginners to improve much faster than normally by teaching the correct movements from the beginning. We aim to make passive learning feasible with force-feedback that precisely moves the user's fingers through complex motions. While we have designed our exoskeleton specifically for piano playing, we believe the method could apply to any number of motor skills.

2. PREVIOUS WORK

A. Passive Haptic Learning

Vibrotactile learning is the easiest haptic technology to develop, because vibration motors are inexpensive and simple. However, the information conveyed is minimal with only one signal per motor at 2-3 frequencies. Previous work with vibrotactile haptics has focused on hand-based motion. In a small four person trial, a piano teaching glove vibrated the fingers of participants in the order of notes to simple 45-50 note songs. Participants practiced one song with the glove for 30 minutes and one song without the glove. After the practice, participants could play the right keys of the song taught with the glove near-perfectly, and the song practiced without the glove with an average of 5 more mistakes [1]. Our goal is to surpass this result, but our trials are designed differently to highlight the impact of more precise control.

This 2008 study proved the concept of Passive Haptic Learning, which led to more comprehensive studies of similar devices. One such study taught braille using PHL, which saw typing accuracy improvements of 60-70% over the control group in a trial of 22 participants with 8 hours total practice time [2]. Our goal is to show that force-feedback learning is feasible and can be applied to a multitude of motor skills as well.

B. Exoskeleton Hands

The main design factor that complicates exoskeleton hands is the degrees of freedom used for each finger. The metacarpophalangeal (MCP) joint, where the finger meets the palm, can engage in all four degrees of freedom; abduction, adduction, flexion and extension. In the proximal interphalangeal (PIP) joint and distal interphalangeal (DIP) joint only flexion and extension are possible.

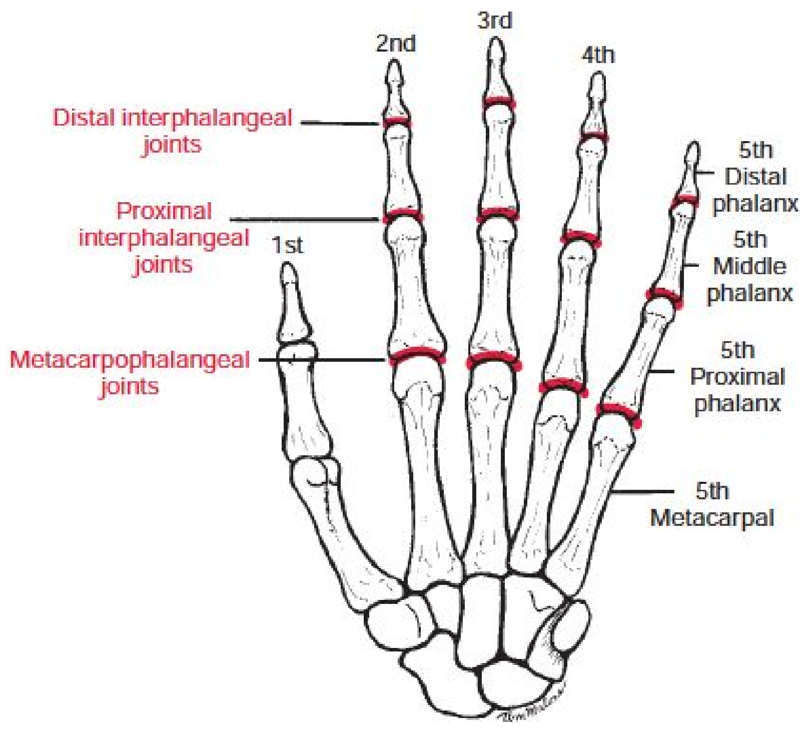

3rd
2nd
4th
Distal interphalangeal joints
5th Distal phalanx
Proximal interphalangeal joints
1st
5th Middle phalanx
Metacarpophalangeal joints
5th Proximal phalanx
5th Metacarpal

Figure 1: *Diagram of Finger Joints [3]*

Several exoskeleton hands have been developed with bidirectional (flexion and extension) control in multiple fingers. A team at Robotics Center-Ecole des Mines de Paris attained four degrees of freedom for the thumb and three degrees of freedom for the index finger, albeit with controls that operate only one degree of freedom at a time [5]. Further work by a team from the Technical University of Berlin designed a system where four degrees of freedom for each finger with bidirectional control was possible [4], but only control one degree of freedom at a time. In applications like muscle rehabilitation, the goal of an exoskeleton is to mimic the same degrees of freedom as the human body to restore complete range of motion. Implementations of exoskeleton hands for object interaction in Virtual Reality typically control only flexion, where the exoskeleton restricts the hand as it closes around an object.

3. PROJECT DESIGN

Our design for this project consists of two main devices, or gloves. The first device is able to record a user's finger movements. The second device is able to actuate the fingers of a separate user. To interface with each other, data are transferred and processed through the use of a microcontroller, personal computer, and SD card.

A. Recording Device

The recording device is a glove with sensors attached to it to monitor the bend angles of the user's joints. The user wears the glove during the recording session, and the glove transfers the recorded data to the microcontroller. The orientation of a person's fingers depends on the positioning of the arm, the wrist, the palm, and the bend of each joint on the finger. Therefore, in order to capture a perfect representation of the user's hand, the sensors would have to be able to detect the status of every relevant joint.

I. Early Design Decisions

We decided to only focus on finger movements for this project, because wrist motion is more static during piano playing and can be learned more easily by human players. The goal of the project is completely guiding the player's movements, but for this project we do not involve the entire arm. Even though side-to-side arm motion is important in piano playing, the creation and powering of an exoskeleton arm for that purpose would be beyond the scope of our project. Additionally, the songs we will test the glove on will be simple beginner songs that only require one octave, which will serve to minimize arm motion.

Similarly, this device will not record or control lateral finger movements (abduction and adduction). The main reason for this decision is that supporting lateral movement would overly complicate this project. Based on our research, effective methods for actuating fingers laterally do not exist. Our own design for actuating fingers laterally would most likely require a much larger device than we have planned for. The additional hardware necessary to support lateral movement actuation would also be complex to assemble and test. In addition, recording abduction and adduction would be difficult. Sensors would have to be placed in between each finger, and these sensors would have to be much more flexible than the flex sensors we wish to use. Incorporating the hardware will also make the recording device more bulky. For our project, which is largely a proof-of-concept experiment, supporting abduction and adduction is unnecessary. However, the main drawback is that this will also limit the amount of actions we can record and actuate well. Motions that require significant lateral movement, such as playing a stringed instrument, will not be captured by our device. However, since we are focusing on simple piano playing, which does not require significant lateral movement, our device will be well-suited for this purpose and motions like it; we believe only guiding the fingers' flexion and extension will yield sufficient improvements, based on a study in piano hand kinematics which emphasized the importance of consistency of the striking finger's joint rotation.

II. Sensors

In order to record the bend angles of each joint, we decided to use bend sensors, also known as flex sensors. The ideal sensor should be small enough to fit on a finger and span only one joint, it should provide accurate measurements with precision to within a few degrees per bent joint, and it should operate consistently without any hysteresis throughout the life of the device.

a. Predicted Challenges

The predicted challenges for the recording device were all related to the sensors. One potential challenge was determining how to mount the sensors onto the recording glove. For prototyping, we considered using tape, glue, or velcro. We ended up choosing velcro because it was stronger than tape and allowed the sensors' positions to be adjustable during prototyping. Another challenge that we foresaw was if the flex sensors were not accurate; they may not be precise or sensitive enough, or they may exhibit hysteresis after frequent bending. If accurate readings were not possible with flex sensors, our actuator device would not function correctly. If that were the case, other types of sensors would have had to be considered.

b. Initial Testing

Before we made the recording glove, we needed a way to test the sensors. Therefore, the sensors were tested on a simple bending rig, which consisted of a folding flap of a cardboard box and a digital protractor to record angles. The sensor was placed over the bend of the cardboard box, and the folding flap was used to bend the sensor in a controlled fashion. The radius of the bend of the box was a lot smaller that of a finger joint, but we thought this would be fine to test the accuracy and repeatability of the sensor.

At first, both ends of the sensor were taped to the box with the bend at 0 degrees, but we quickly ran into an issue we had anticipated. The first bend we made went smoothly, but when the rig was straightened back to 0 degrees, the sensor immediately buckled. This is an issue we had to deal with when designing and constructing the recording glove. Specifically, we wanted to eliminate the buckling, because it introduces more uncertainty in the measurements. The fix was actually quite simple. Instead of fixing both ends of the sensor in place, one is fixed, and the other is just strapped in place. This way, the sensor still bends with the finger, but can shift its position as necessary and won't buckle when straightened. The fix was implemented in the testing set-up by sticking a small piece of paper to the area of tape where the sensor passed under.

c. ThinPot Sensors

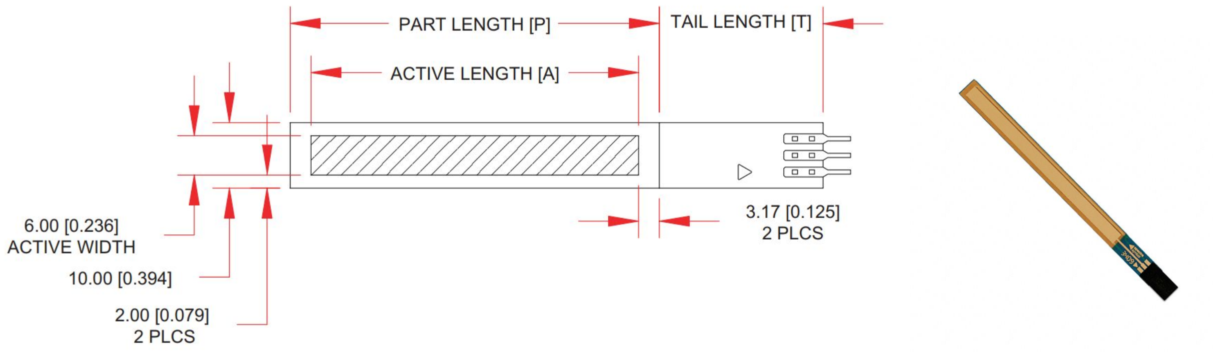


Figure 2: *ThinPot sensor (dimensions are in mm[in]) [9]*

The first sensors that we used were actually 25 mm long ThinPot sensors, not true flex sensors, but we did not realize it at the time. We first connected one to an ohmmeter, and observed that bending the sensor increased or decreased its resistance depending on which direction it was bent. In reality, the ThinPot sensor is more like a pressure sensor, varying resistance depending on the force experienced along the length of it. It is not too surprising that these sensors worked to a certain extent, since the bending would cause the top layer to press down on the bottom layer, but this is not the intended application. While one group member bent the sensor, another member recorded the angle at which it was bent along with the resistance measured by the ohmmeter. For each sensor, we repetitively bent the sensor from 0 degrees to either 45 or 90 degrees. In addition, we also varied the rate at which we bent the sensor (slow or fast bending). The results were not great.

The first thing we noticed was that different sensors of the same product did not operate at the same resistances; this was not too surprising though, because the datasheet stated the initial resistance to be 10kΩ ± 20%. The second thing we noticed was that the relationship between angle bent and resistance is fairly linear. The third thing we noticed was that each individual sensor seemed to be inconsistent; when the sensor was restored to 0 degrees, the resistance did not match the initial resistance. However, this may have been a result of poor/uncontrolled testing conditions (We later tried the same experiment and the observations did not vary as much). The first issue did not concern us, because we can easily map the resistances of different sensors to the same linear scale in the microcontroller. The second observation was desired. The third issue was problematic because it suggested that the data we collect may not be consistent with a particular finger position. This finding also suggested that the recording glove should be hard as well, so that the bending of the joint can be more consistent.

d. Flex Sensors

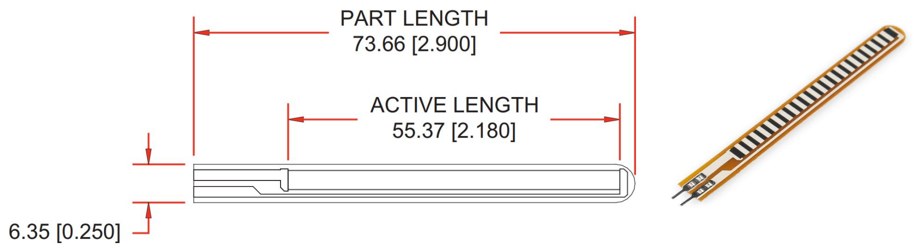


Figure 3: *Flex sensor (dimensions are in mm[in]) [10]*

After realizing the ThinPot sensors were not suitable for our application, we bought two flex sensors to test with. These sensors utilize conductive polymer ink to vary resistance with the severity of the bend [10]. A positive bend angle increases its resistance, while a negative bend angle decreases its resistance. The testing method was the same as mentioned above. We tested two separate flex sensors, with ten bending trials for each. The results, shown below for one sensor, were much better than before.

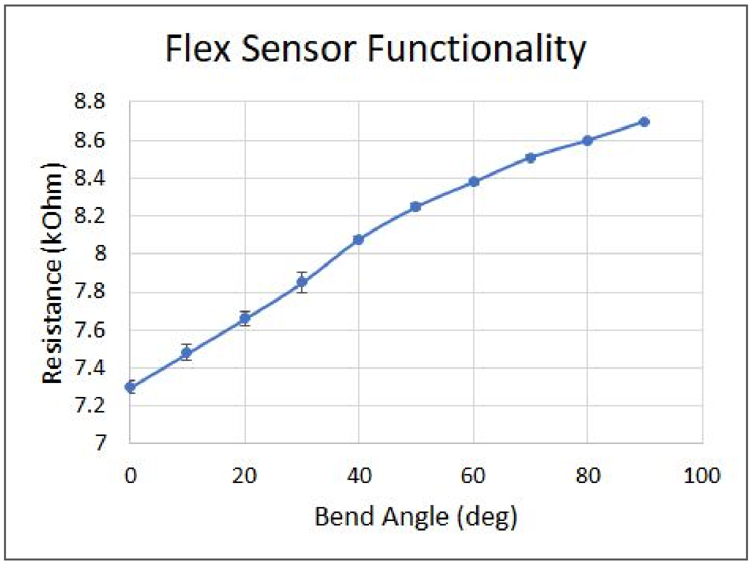


Figure 4: *Resistance vs. angle for one flex sensor, based on 10 trials.*

Once again, as expected, our first observation was that different sensors did not operate at the same resistances. We then observed that the sensors were very linear. Lastly, these flex sensors were very consistent; when the sensors were restored to their original positions at 0 degrees, the sensors' resistances were also restored to their original values. The first observation is not an issue, as long as the flex sensors are eventually mapped to the same linear scale. The second and third observations were desired. Due to these observations and how they align with our previously mentioned ideal sensor properties, we chose to continue with the project using these flex sensors for recording.

After choosing our sensors, we realized that the shortest sensors (2.2 inches) are too long for the fingers. The sensors on the glove run over more than a single joint. Due to this issue, another simplification to our project resulted; the measurements for the DIP and PIP joints of each finger were combined. This decision is justified by the fact that the motions of the two joints are highly correlated with each other [11]. Bending the DIP joint naturally causes bending in the PIP joint, thus the DIP and PIP joints could be actuated together with our device. By using one sensor for each pair of DIP and PIP joints, four sensors were eliminated from the final design, along with the need to process that data and actuate the DIP and PIP joints separately.

III. Glove Design

The most challenging part of the project during the first semester was finding a suitable way of attaching sensors to the hand to record movements. Even if we have accurate sensors, it won't matter if we can't accurately convey the motion of the hand to them. Similar to making sure the sensors output the same measurements every time it is bent to a certain degree, we need to make sure the glove moves the sensors the same way each time a finger is bent a certain way.

a. Initial Design

Initially, we planned to build a recording glove through the use of a soft glove and sensors lined on top of it. The idea behind this was so that the user of the recording glove would not be hindered by it. The design goal was that, even with the recording device equipped, the user would be able to perform all necessary finger motions with minimal resistance and maximum flexibility. Thus, the sensors would also have to be very flexible and mounted in a way as to not obstruct the user's movements.

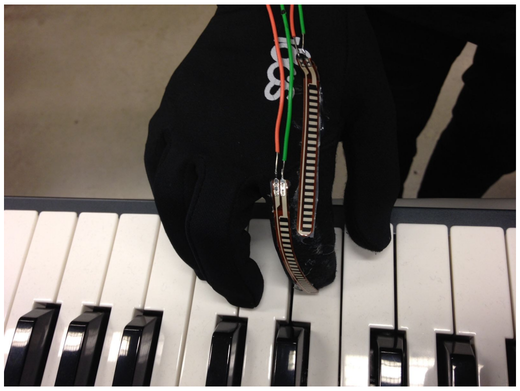


Figure 5: *Original prototype of the recording glove*

Most, if not all, recording gloves that we have seen use a soft glove as a base to attach sensors [11][12]. However, our original prototype of the soft recording glove caused concern.

The problem with soft gloves is that even if the sensor is securely attached, the glove can slide around on the hand, most notably on the knuckles, causing all sorts of unpredictable deviations. When testing our glove prototype, bending a finger caused the sensors above it to slide off the top of the joints and onto the side. Shifting along the length of the finger was one issue, because different sections of the sensor may change the resistance differently when bent the same way. However, side to side movement was an even bigger problem, because it slightly changed the radius of the bend, leading to varying measurements. Due to these problems, our soft glove prototype led to inaccurate and inconsistent readings from the sensor. In order to solve this problem, we had to consider other designs for the recording glove.

b. Exoskeleton Design Considerations

We want to build a glove that accurately transmits movement to the flex sensors, yet is flexible enough to not impede movement of the hand. Flexibility is where a soft glove excels, however due to its significant drawbacks, we need to consider alternatives. Naturally, a hard glove was the next design to evaluate. A hard glove is in many ways the opposite of a soft glove. It is much more restrictive than a soft glove, but from this lack of flexibility, it gains the accuracy we require. A hard glove only has the degrees of flexibility it is designed to have (assuming it is fabricated properly), minimizing the issue of sensors sliding up and down and side to side on the hand. However, the lack of flexibility means that the mechanical design of the glove is extremely important. If it is dimensioned incorrectly, and the joints of the glove do not line up with the joints of the hand, the user can't bend his fingers properly. Sizing is also a big issue. With a soft glove, there is a generous margin for finger length and girth, palm size, and overall length. With an exoskeleton, the glove has be compatible with the user's hand in every dimension. If the user has skinny fingers, the glove may be too loose. If he has big palms, the whole glove might not fit. If the user has long or short fingers, the joints won't line up. Different people might even have differently sized finger segments. What this means is that just having a couple set sizes of glove (small, medium, large) is not be sufficient for an exoskeleton glove. We need to custom fit gloves to the users, or design modular gloves with parts that can be switched out to accommodate the wide variety of hand sizes. Thankfully, three members of the group have very similarly sized hands, so the design process is simplified for preliminary testing.

c. Design Stages

Designing an entire exoskeleton glove from scratch would be extremely difficult, given that none of us on the team have very much experience with CAD software. Since we planned on 3D printing the glove, we started looking at 3D printing model websites. Surprisingly, there were not many models available for free or for purchase. One model, simply named "3D Printed Exoskeleton Hands" (see below), seemed promising at first, because it offered many degrees of freedom and did not seem to impede movement in demonstration videos. However, upon further inspection, we realized it only attached to the fingers at one point, the fingertip, and each finger seemed to use just a single U-joint. This is what gives the glove so much flexibility, however it would be insufficient for collecting the data we need. In addition, there is no obvious way to mount sensors, so this design was quickly eliminated.

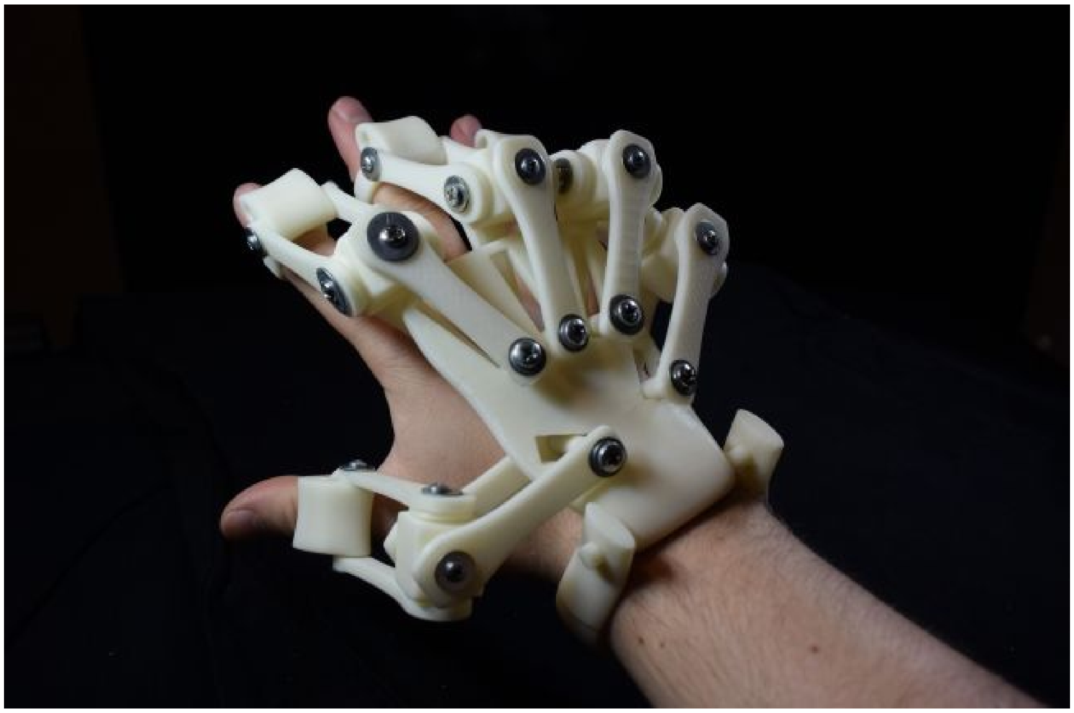


Figure 6: *3D Printed Exoskeleton Hand [13]*

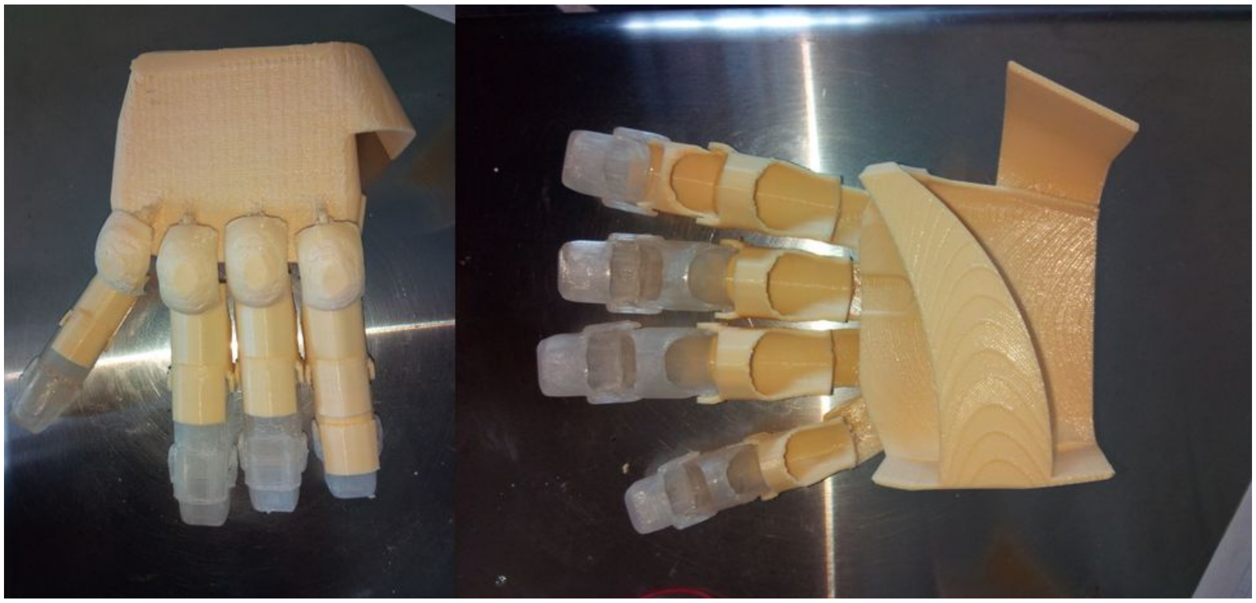


Figure 7: *Exo Glove [14]*

The second design, called the "Exo Glove", is more suited for our design over the first. It has the correct number of joints, and if the knuckle and and joint shields are removed, it would not be difficult to attach sensors. However, the fully enclosed design on finger segments will hinder movement, and it looks a bit clunky. It would also be a lot more difficult to size the glove for differently sized hands.

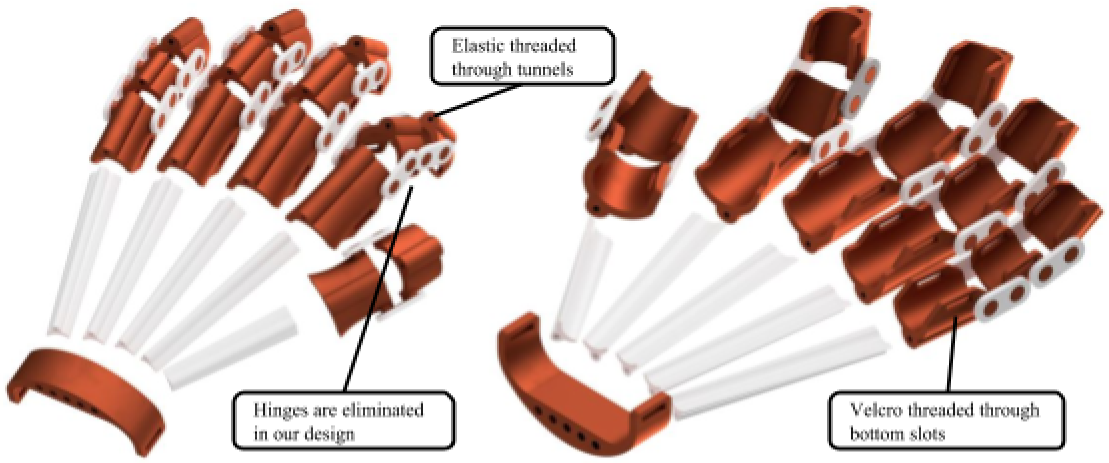

Elastic threaded through tunnels
Hinges are eliminated in our design
Velcro threaded through bottom slots

Figure 8: *Galacta Hand [25]*

The next design, called the "Galacta Hand" was the design we thought was the most promising and we felt was best suited for our application. This glove was designed as a strength training device. The fingers of the glove only cover the top of the user's fingers, and is strapped on using Velcro or an elastic band. An elastic cord is threaded through the tube on the top of the fingers, to provide resistance when they are flexed. This design is an excellent base to start with, but it has several issues we needed to fix. The most glaring issue is the lack of a shell for the body of the hand, which was not necessary in its original application but is for ours. In this regard, the Exo Glove is superior, and we have taken inspiration from it when designing the piece for the palm.

A second issue that needed to be addressed with this design was with the joints. The fingers use a system of two joints for each joint of the finger. At first, we thought this would give a degree of freedom that wasn't entirely necessary, although it might have helped with relaxing constraints on the dimensioning. The assumption was that the two joints would equally split the total bend, or possibly unequally split the bend if the glove didn't fit correctly. We also thought the system of two joints might allow for a twisting motion depending on the tolerances in the 3D printing process.

In reality, the system of joints proved to be much more imperfect. We quickly realized that the pegs that the joints rotate around were too short, and any sideways force would cause the joint to fall apart. While this issue can be solved in the design, there was another inherent flaw in the design. Originally, each hinge was the same length for each joint. In order to properly fit the hand, hinges needed to be different lengths. But even with properly sized hinges, the joints did not bend same way as the joints in our hands do. This, combined with the rough 0.3 mm tolerances of the pieces printed in the Makerspace lab, meant that the joints did not move smoothly, producing a lot of resistance for the user. We eventually realized that for the recording glove, joints are not necessary at all, and are not necessary for the actuation glove either. Strapping the segments of the glove individually to the fingers significantly simplifies the design constraints and the restrictiveness of the glove. Without joints, the exoskeleton can sit on the hand in the most comfortable position for the user, and there is no joint hinge dimension that can cause interference between the movement of the hand and the movement of the exoskeleton. It does not affect the accuracy of the recording either, as long as the pieces are securely attached to the user’s hand.

A third issue with the design is that the pieces are strapped onto the hand. This is actually a plus in some regards, such as flexibility and modularity, but it presents a challenge for the actuation glove. Since we've decided to use a cable system (discussed in the next section), there need to be attachments on the bottom of the glove, and we cannot just attach the cable to the velcro strap. There are two potential solutions for this. The first one is to design a piece that will snap in place, using the holes for the strap as attachment points. This will reduce range of motion, as will any design that encloses the finger, but depending on the movement recorded, that may not be an issue. The solution that we first tried is to have a simple piece that is threaded onto the strap, which the cable will be attached to. This is the simpler solution, however it led to some issues with actuation accuracy, which will be discussed later.

Despite it's issues, The Galacta Hand was a great foundation to start from. In addition to removing the hinges and the circular extrusions that supported them, the tunnel on top of the glove was modified to accommodate the flex sensors by replacing the circular tunnel with a rectangular one. At first, the tunnel was sized to match the dimensions of the flex sensor, however, in the final design, the tunnel is sized up to accommodate overlapping flex sensors. As for sizing, the original design was too narrow to fit our hands, and too long in some segments. Unfortunately, the Galacta Hand did not come as a modifiable SolidWorks file; it was only provided as a fully dimensioned model. We were only able to scale each piece up and down on each axis, however this was sufficient for resizing.

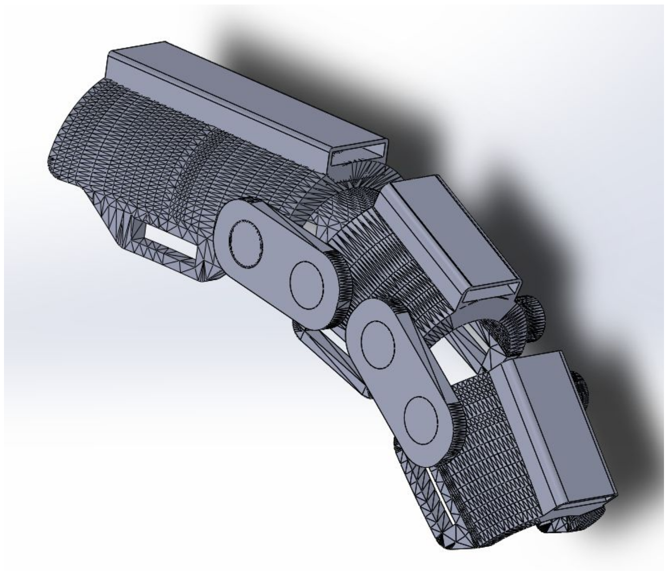


Figure 9: *Design of a finger with hinges.*

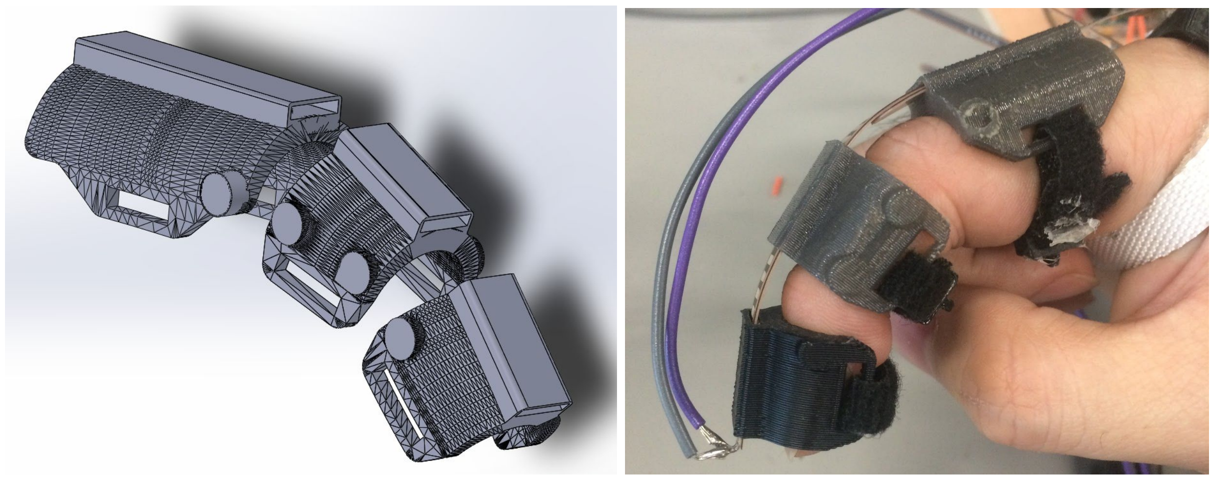


Figure 10: *Final design. Pegs cannot be completely removed in the model, so they are sanded down after printing.*

d. Palm

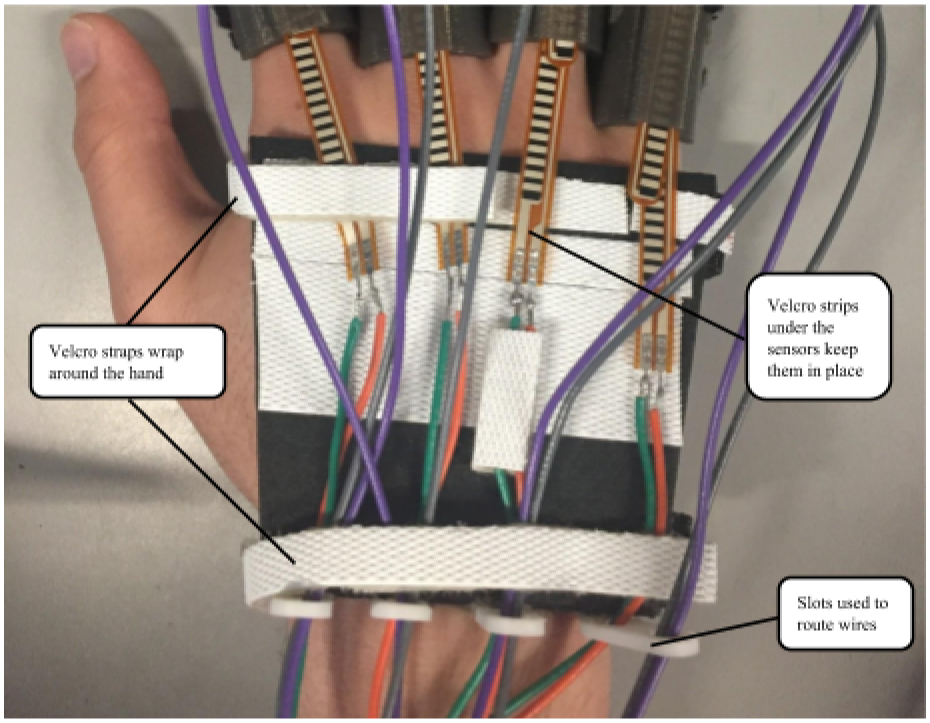

Velcro straps wrap around the hand
Velcro strips under the sensors keep them in place
Slots used to route wires

Figure 11: *Palm of the recording glove*

The previous design was useful only for four fingers of the hand. We had to design from scratch components that would work for the palm. The palm's design was necessary for holding the first flex sensors going through the first segment of the fingers. We utilized velcro strips to place the flex sensors, so that we may be able to make adjustments if necessary. The positioning depends greatly on how the palm sits on the users hand. The palm is a SolidWorks design that looks like a trapezoid viewed from above. We took measurements for Kevin's hand for its design, even though it sits well on multiple hand sizes. The four measurements are: the distance from the outside of the forefinger knuckle to the pinky knuckle, the forefinger knuckle to the wrist, the pinky knuckle to the wrist, and the length of the wrist. The measurements for Kevin's hand were 3.5 in, 3 in, 3 in, and 2.5 in respectively. The top is flat so the flex sensors can sit evenly with the entrance to the finger tunnels. The bottom is curved to accommodate the natural bend in the palm; each side falls about half an inch from the crux of the middle finger knuckle to the outside of the palm. Two strap holes are affixed to the right side of the glove, which Velcro straps go through to attach on the top. This also serves to help hold the wire in place. Additionally, several slots are affixed to the back of the palm, for the wires from each sensor to be seperated. The first sensors that come from the MCP joint go through the bottom, and the second sensors that cover the DIP and the PIP go through the top hole.

e. Final Design

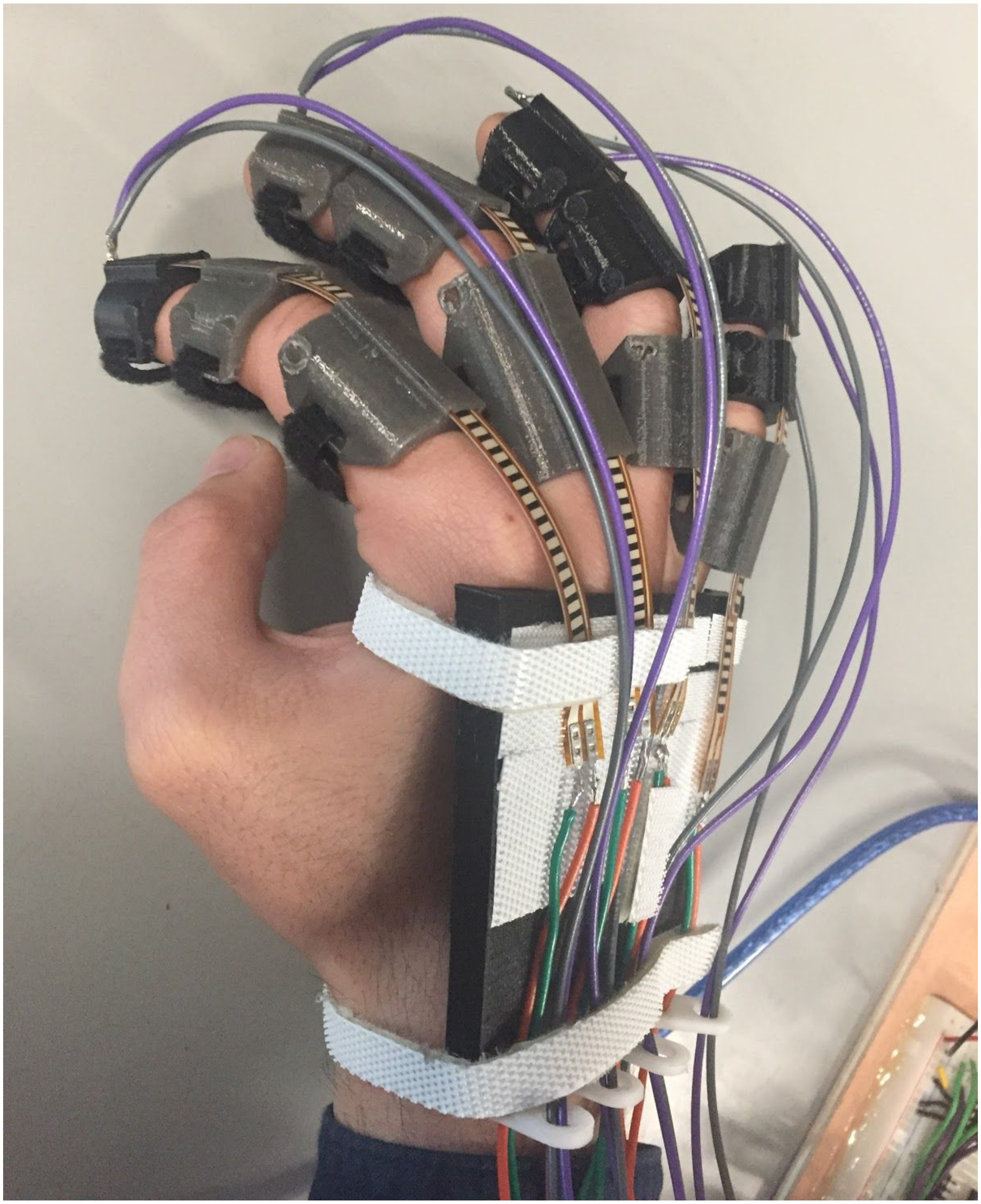


Figure 12: *Overview of recording glove*

The final iteration of the recording glove design featured 3D printed half shells for each segment of the finger, attached using velcro straps. Flex sensors were placed in channels above each finger. One sensor records the movement in the MCP, and one sensor measures the combined movement of the PIP and DIP. Lateral movement is not supported, given the limited flexibility of the sensors, however, detecting lateral motion is not within the scope of the project since we only plan on teaching simple, five key tunes.

B. Actuating Exoskeleton

There has been extensive work done in the field of powered exoskeletons, and there are many possible ways to actuate the fingers. For simplicity, we can divide actuation methods into two categories: motors located on the hand, and motors located away from the hand. The former is probably the more obvious way to approach the problem. Since we need control over individual joints, the two designs we can consider are motors positioned on the finger coincidental to each joint, or a four bar mechanism (shown in figure X) [8]. However, both these designs require very precise mechanical design in order to work properly, a challenge we do not feel qualified to handle with no mechanical engineers in the group. In addition, these designs require the use of very small motors, which are inherently weak. Given these limitations, we decided to pursue an actuation method in the second category, with the motors located away from the hand. By far the most popular design in this category is the cable mechanism. With a cable mechanism, size constraint on motors is eliminated since they will not be located on the user's hand. Mechanical design is simple as well, since we only need

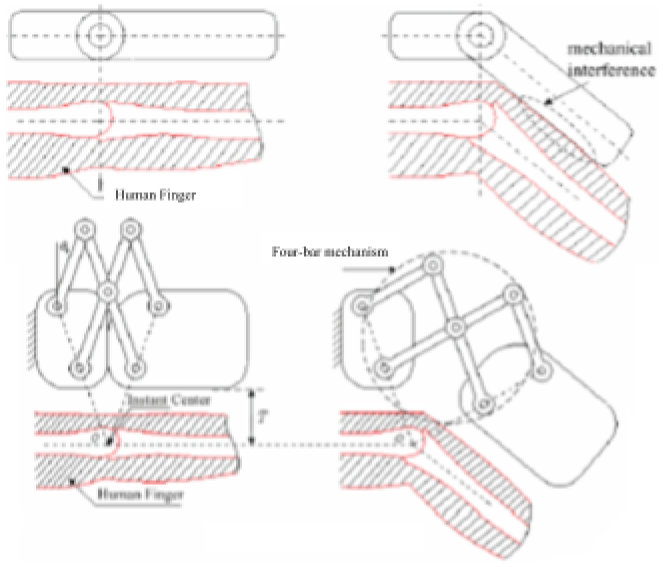


Figure 13: *A simple hinge placed above a joint cannot properly actuate it due to mechanical interference. A four bar mechanism, which shares a center of rotation with the joint, is necessary.*

to attach actuation cables to various points on the hand. In terms of specific designs, there are many possibilities. We can have cables mounted above and below the fingers which control extension and flexion, respectively. Alternatively, we can attach cables below the finger that control only flexion, paired with a more passive method of extension. Similarly, we can attach cables above each finger, paired with a more passive method of flexion. Ultimately, we decided to use a system of cables mounted under each finger for flexion and elastic cords attached above each finger for extension.

If we were to use just one cable per finger, attached to the fingertip, all three joints in the finger would bend in unison. However, this system of one cable cannot mimic many finger motions, such as those found in piano playing. Typically when striking a key, the MCP joint flexes, while the DIP and PIP joints extend. Thus, we need a system that can control each joint independently. We could use three cables per finger, one for each joint. This would allow for fine control, however, after examining the anatomy of the hand, we realized that this level of control is not necessary. The DIP and PIP joints are actually controlled by the same tendon [15], therefore, we can do the same in our exoskeleton. The cable system can be simplified to use only two cables per finger - one that actuates the MCP joint, and one that actuates both the DIP and PIP joints.

I. Glove Design

The body of the exoskeleton is realized through the use of a 3D printed hard glove. Actuation is achieved using a system of cables stepper motors and elastic bands. Pretensioned elastic bands run along the top of the hand, and are used to extend the fingers. The cables are attached to the bottom of the hand to flex the fingers. The cables we use are called bowden cables, and are a used to transmit mechanical force. It consists of a longitudinally incompressible sheath surrounding a freely moving inner cable. The ends of the sheath are fixed to the two components between which force is transmitted, in our case, the stepper motor and the hand. At rest, the fingers are held in a fully extended position. As the the cables are pulled, the fingers bend, stretching out the elastic. When cable is released, the elastic pulls the finger back into an extended position.

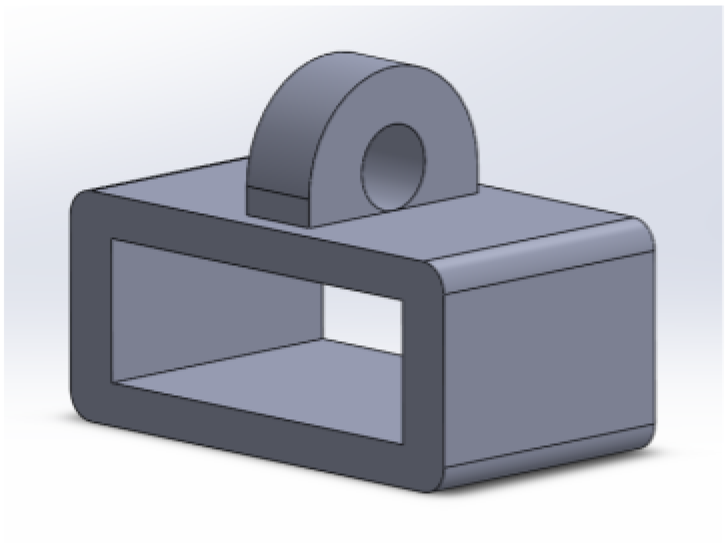


Figure 14: *Cable anchor*

Initially, the actuation glove was adapted from the recording glove with two small changes. First, a part was added to the bottom of each finger segment. These parts are simple 3D printed loops (Figure X) threaded onto the velcro strap, and are used to attach the cables (Figure Y). On the tops of the fingers, the slots meant for the flex sensors on the recording glove are the perfect size to thread elastic bands through. Since an attachment point on the palm is required for the bowden cables, a new hand piece with a solid palm was designed (Figure X). The end of the sheaths that controls the MCP are glued to the palm, and its cable is attached to the first segment of the finger. The sheath that controls the DIP and PIP joints is glued to the first segment of the finger, and it's cable is attached to the fingertip (Figure XX).

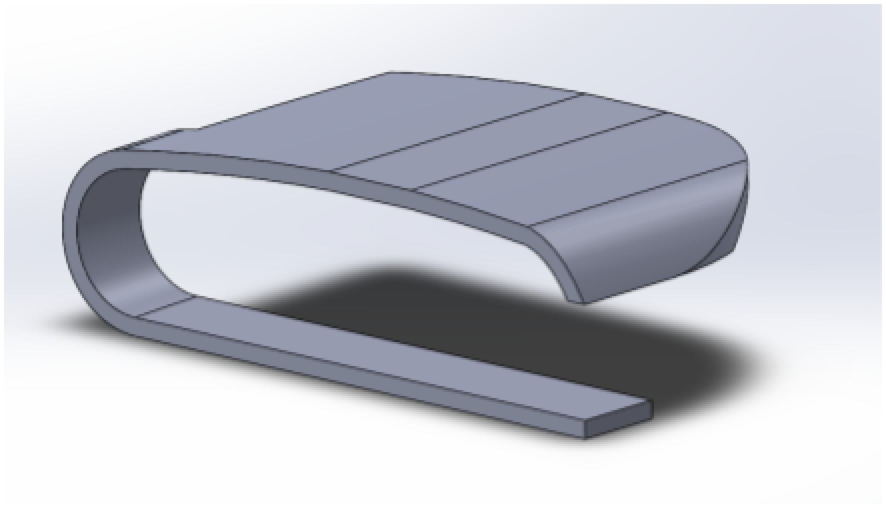


Figure 15: *New palm piece*

However, an issue with the design of the glove became apparent upon initial testing. When the cables try to flex the fingers, what they are essentially doing is pulling its two attachment points closer to each other. However, most of the cables are anchored to velcro straps, which are not rigid. The force of

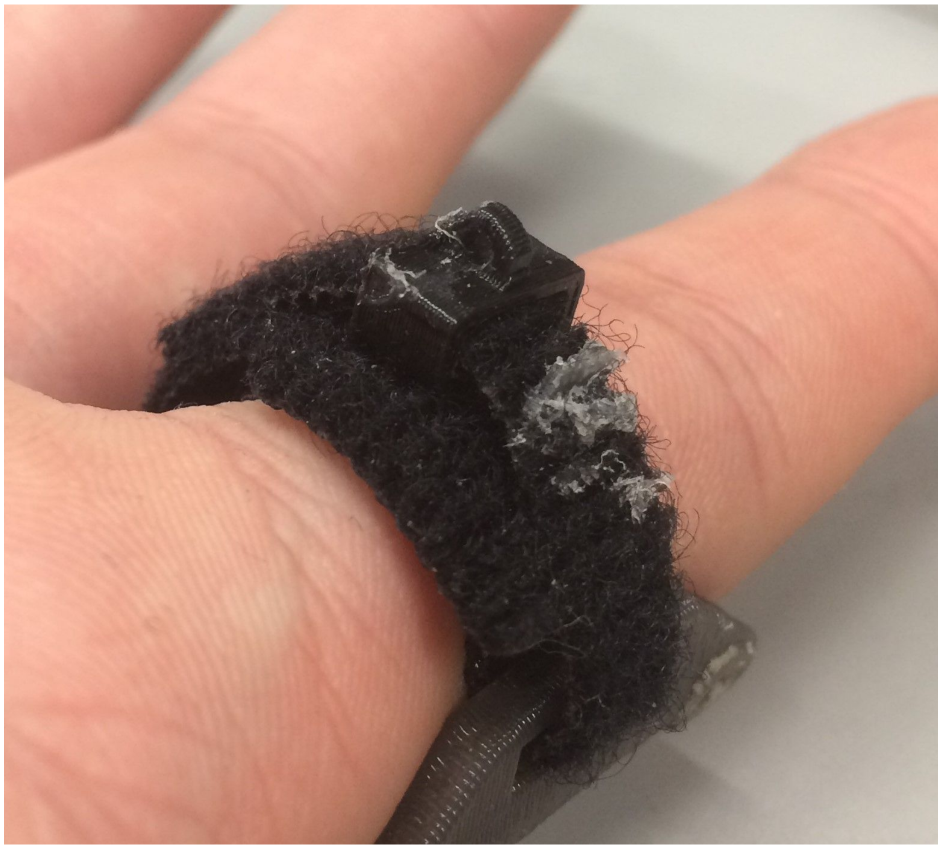


Figure 16: *Cable anchor, threaded onto velcro strap*

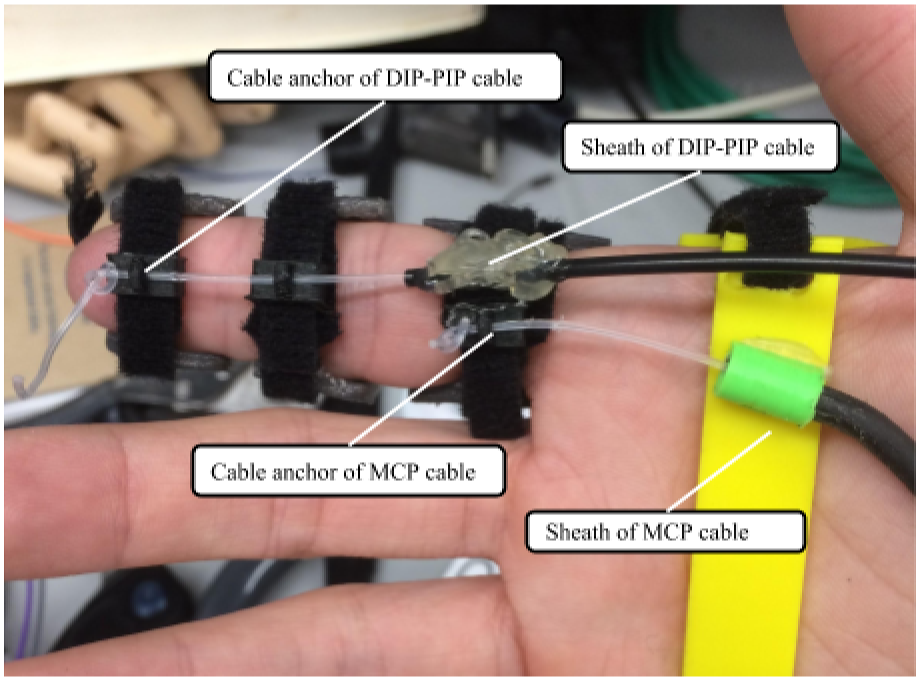


Figure 17: *Cable connections on actuation glove*

actuation deflects the velcro up and down the finger, by up to 1 cm (Figure 17) on the index finger. The MCP joint requires about a 3 cm pull to reach 90° bend, so this translates to an error of more than 30° in actuation. This inconsistency is one of the problems we wanted to avoid by using a hard glove. So we had to go back to the drawing board for the actuation exoskeleton.

To solve this, we abandoned the recording exoskeleton design and created a set of rings to replace them (Figure X). These rings have slots for elastic bands built in, like the previous design, but more importantly have a circular tunnel along the bottom, providing a completely rigid cable anchor point. Each ring is sized carefully to fit each finger segment of the wearer's hand for a snug fit to prevent shifting along the finger. When viewed from the size, the rings have a trapezoidal shape, small on the bottom and large on the top. The large area contacting the finger on top prevents the ring from changing pitch along the finger, and the narrow bottom allows for a wider range of motion when bending the finger. The arrangement of cables does not change from the previous exoskeleton design. With this design, the pitching along the finger is not eliminated, however it is minimized.

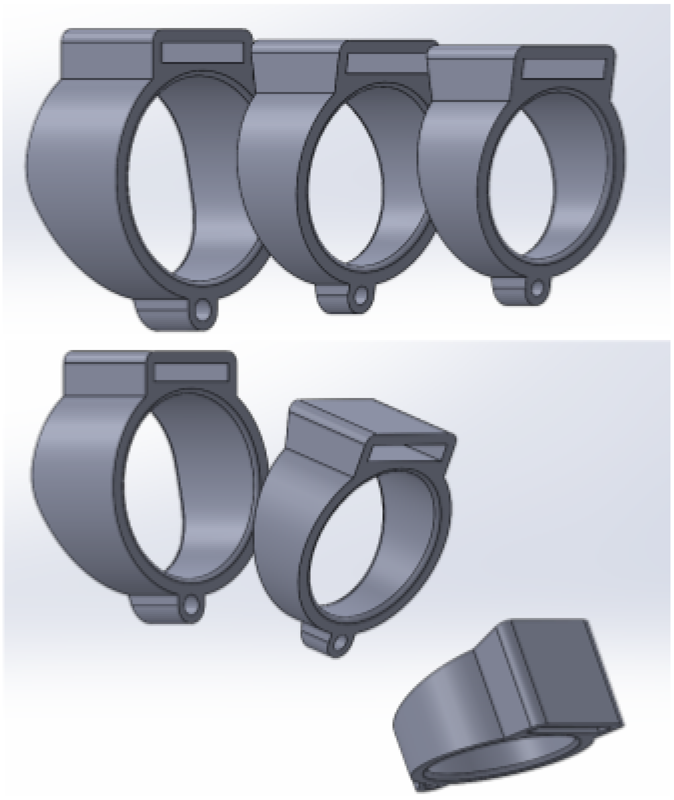


Figure 18: *Rings*

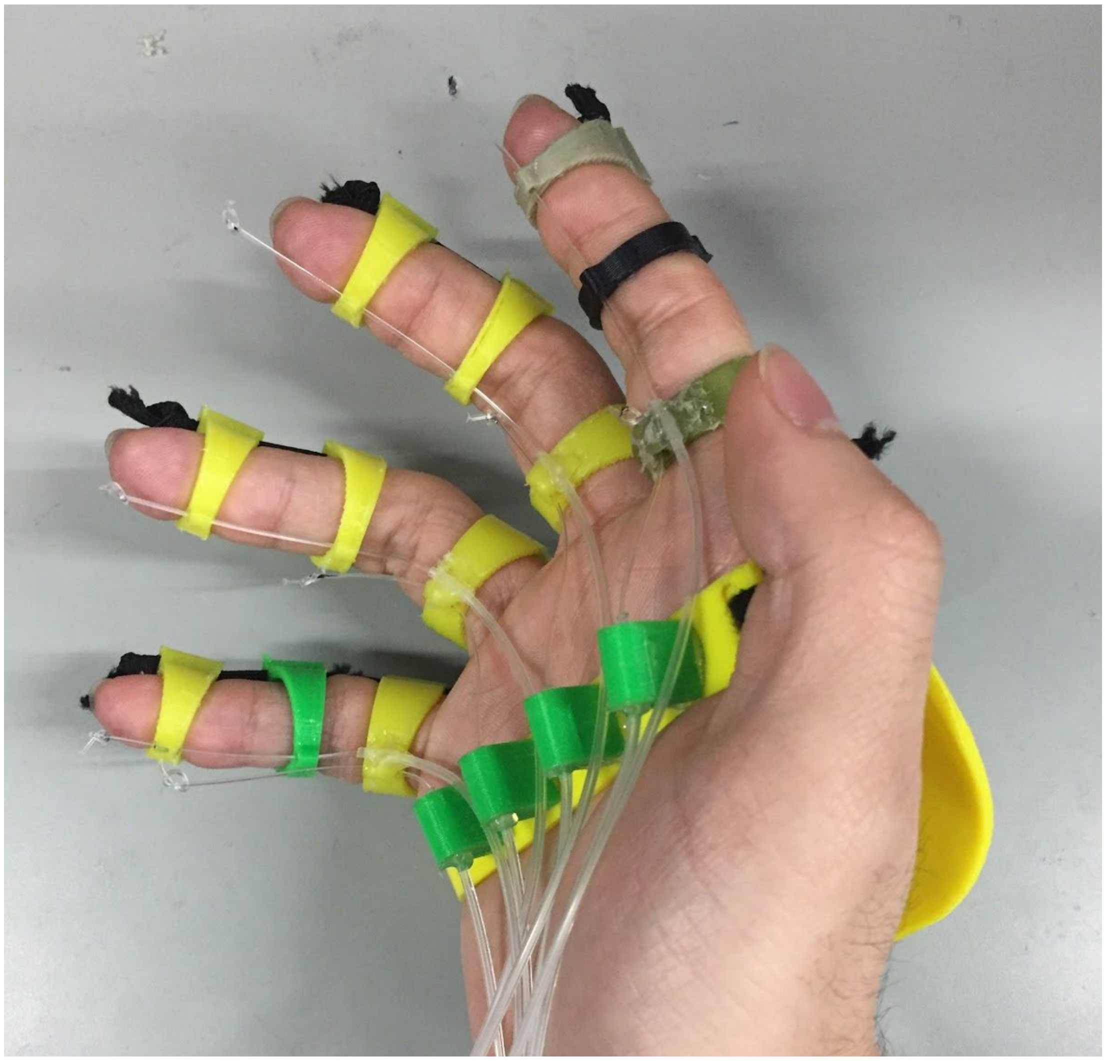


Figure 19: *The assembled actuation exoskeleton, three-quarter view*

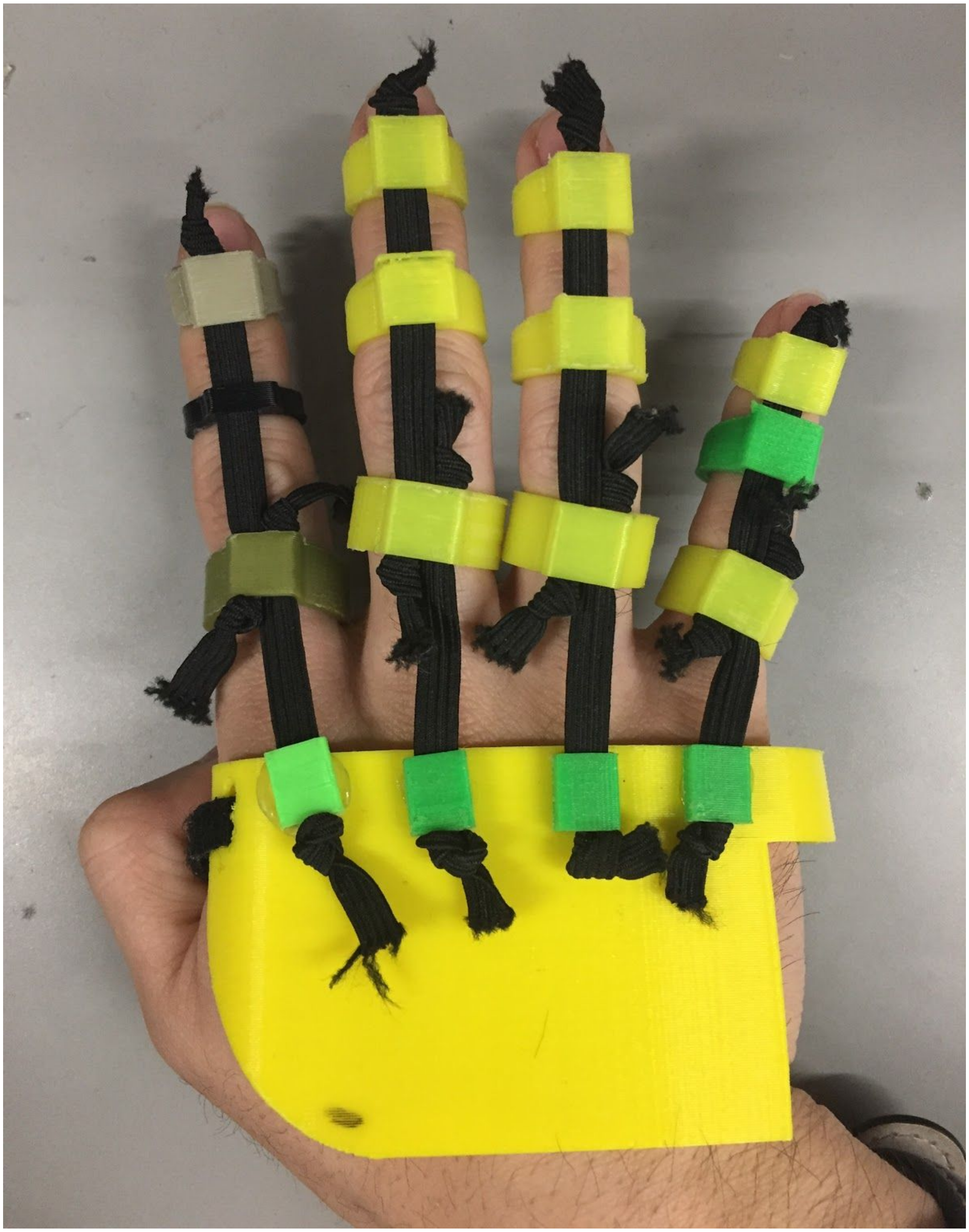


Figure 20: *The assembled actuation exoskeleton, viewed from the top.*

II. Choosing Motors

When choosing which motors to use, we considered torque, speed, precision and accuracy, power, cost, and form factor. We narrowed down our options to servo motors and stepper motors because of their precision control [16]. Servo motors have internal position regulation (closed-loop feedback) and are geared to lower speeds, which result in very precise position control. Stepper motors move in discrete increments by using magnetic fields and do not operate on feedback systems. In general, servo motors are smaller in size and have less torque than stepper motors. Also, stepper motors have the advantage of a high holding torque; when the motor is not moving but is still powered, it will hold its position firmly. However, the torque significantly decreases as a function of speed.

We bought a few servo motors and stepper motors to test. The servo motors proved to be a bad option for our project because they did not hold their positions very well; the user's hands would affect the position of the motors and create cause for recalibration. We decided on using stepper motors because they were able to handle this a bit better, which outweighed the downside of their larger physical size.

There are many different types of stepper motors [17]. When comparing unipolar and bipolar stepper motors, we chose to use bipolar stepper motors because they tend to have more torque. In addition, we want stepper motors that have a fairly large step count; 200 steps per revolution is sufficient for the precision we require when actuating the fingers. In theory, a lower step count could be used, but a gear system would be required to gain the precision we desire. The tradeoffs to a high step count are lower speed and lower torque, but this is not a problem for us. There are also a wide variety of shaft types, although this feature is not important to us because we can procure 3D-printed parts that will easily interface with whatever shaft we have. In terms of power usage, efficiency is not important to us, especially since our project is largely a proof-of-concept design.

The motors we are using are bipolar stepper motors rated for 12 V and 0.35 A [18]. Each step rotates the motor 1.8 degrees, equating to 200 steps per revolution. They exhibit 20 Ncm of holding torque; the torque decreases as the speed increases. We originally tried using smaller stepper motors, but they were not strong enough to pull on the user's fingers [19]. We could also use stronger motors, but they would be larger and heavier. In addition, we do not want the motors to be too strong, in order to allow the user to safely resist the motors.

III. Motor Mounts

A box was constructed to provides a solid mount for the bowden cables as well as house the spools to give the cables the necessary range of motion (Figure XY). First, we had to determine an appropriate spool size to use. If the spool is too small, the motor might not be able to turn it quickly enough to replicate motions at their recorded speed. If the spool is too big, it sacrifices precision, and may not be able to exert enough torque. Since the MCP joints require about 3-4 cm of linear movement to reach 90°, we decided to use spools with 2 cm diameters to balance speed, torque and precision. At first, the design consisted of just the holder, motor and spool. When the motor turns, the cable spools and unspools. When it unspools, the idea is that the elastic would pull on the cable from the exoskeleton side, keeping the cable taut. However, in reality it didn't work so well; the motor and spool try to push the cable through the sheath, but the cable is pushed out of place. To solve this, a shell was designed to go around the spool to guide the cable in the right direction.

While the motors have no trouble overcoming the force of the elastic bands to move the user's fingers, there is added resistance near the upper end of the range of motion due to interference between the exoskeleton and the bunching up of skin on the palm of the hand. The resistance becomes too high for the motor to overcome at angles greater than 80°, and the motor sometimes slips, not completing some steps. We could make the spool smaller, although this could cause an issue with

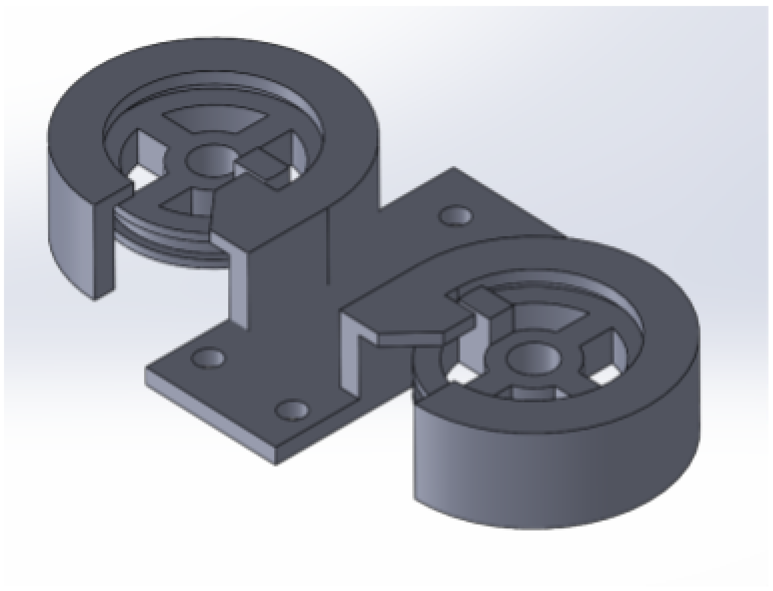


Figure 21: *Spool and shell, with mechanical stop*

speed, or we could buy a more powerful motor. Regardless, we do not want to force any motion for the user's safety. This problem could be solved in further iterations of the exoskeleton, but it is not an issue for replicating piano playing, since high degree bends are not seen. However, it is problematic if does happen, because stepper motors do not have position sensors. If a motor slips, there is not way to programmatically account for it, and the motor will go into negative degrees when returning to 0. To prevent this, we have incorporated a mechanical stop at 0° (Figure X). If slippage occurs during actuation and the finger does not fully extend immediately, there will be an error in the actuation until the finger is fully extended to a bend of 0°. Currently there is no solution to this, but it does not occur if we only actuate piano key presses.

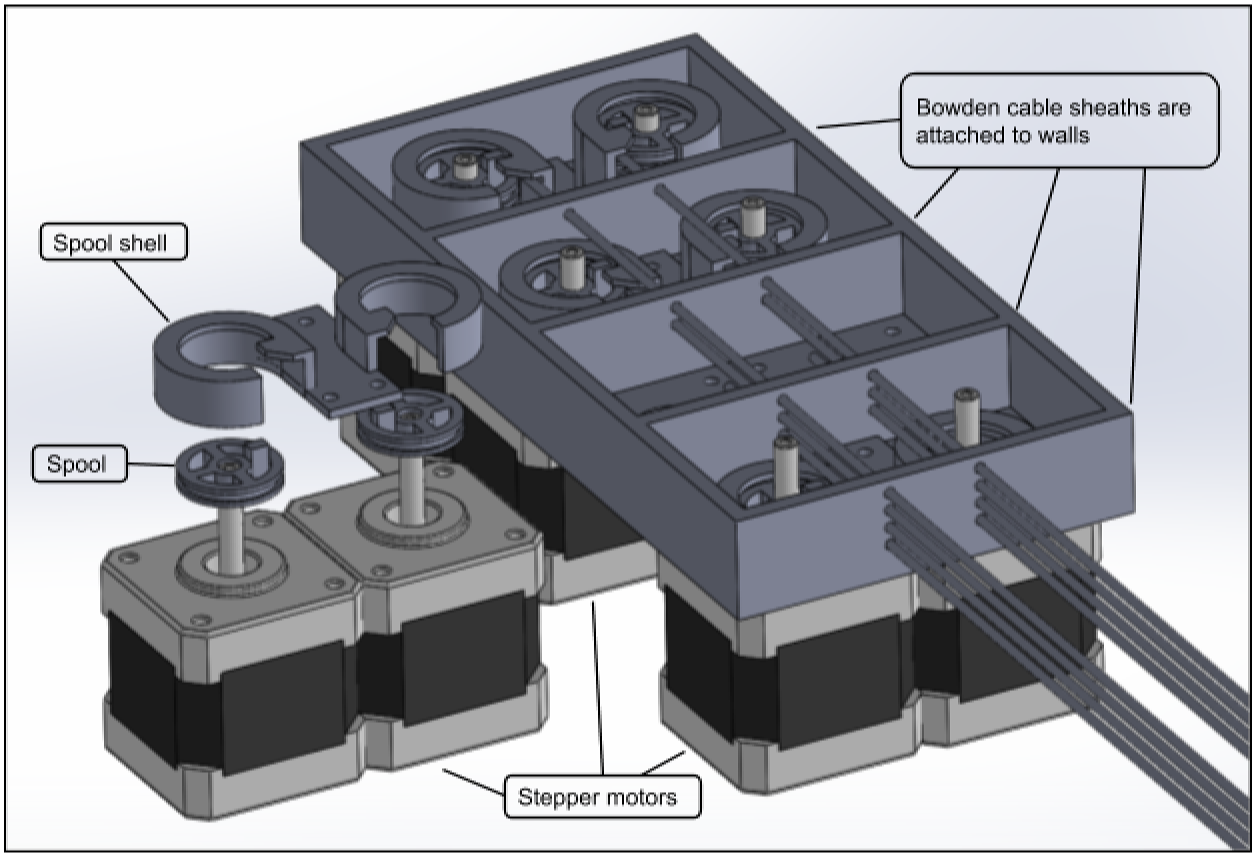


Figure 22: *Complete motor Assembly*

C. Processing Device

In order to interface between the recording device and the actuator device, we need to use a microcontroller. The microcontroller operates in three separate modes, one during recording, one during processing, and one during actuating.

In the recording mode, the microcontroller primarily serves to read the data coming from the sensors attached to the recording glove. This data must be collected in real time. There must also be a mechanism for storing the data for later use. Ideally, this raw information is sufficient to properly operate the actuator device. Otherwise, some sort of error correction must be done while the data is being processed. In addition, the microcontroller needs to properly read information from all of the sensors at once. It must gather the data at a sufficient rate in order to capture the recording user's motions accurately. If the sampling rate is too low, the stored data will not be enough to reproduce the user's actions. If the sampling rate is too high, this leads to unnecessary processing during data storage or actuation.

In the actuating mode, the microcontroller primarily serves to control the motors of the exoskeleton. It must read the stored data line by line, and map each value to a desired bend angle. The microcontroller must then command the actuator device to move until it detects that the device is in the desired position. The microcontroller must be able to actuate all of the fingers simultaneously in order to recreate the recorded action.

I. Arduino

For this project, we initially chose to use an Arduino Uno (in our final design, we switched to an Arduino Mega for reasons discussed later). This is because our needs are not particularly demanding or special in terms of recording and processing. For our purposes, the Arduino has been set up to be used with a computer. It can read values from the flex sensors, and it can store the data on an SD card.

In order to program the Arduino, the computer must have the proper drivers installed, as well as the Arduino IDE or Web Editor. The Arduino Web Editor was the tool we initially chose because the source code is saved automatically on the cloud, and it eliminated the need to set up an IDE on a specific computer. However, we later switched to using the IDE because it has more features and does not require an internet connection. The Arduino is powered by and communicates through a USB connection to the computer. Connections made between the sensors and the microcontroller are implemented through a breadboard for prototyping.

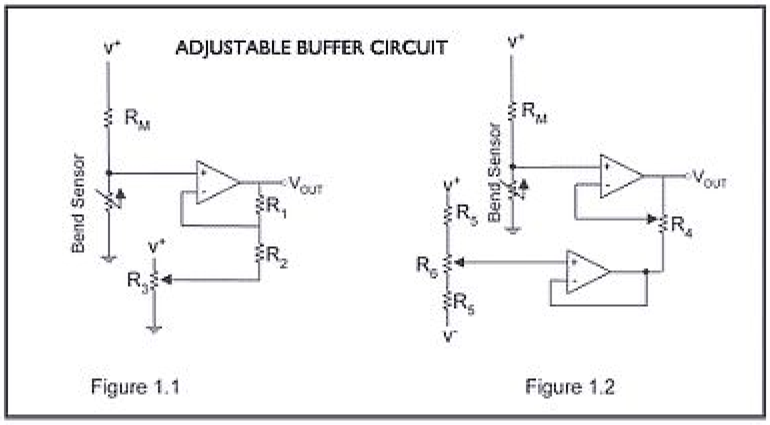
The Arduino programming language is largely based off of the C++ programming language. However, instead of one “main” function, the microcontroller’s code is comprised of a setup function and a loop function. The setup function gets called once as soon as the Arduino is powered on, or when the reset button is pressed. Then the loop function gets continuously called as long as the Arduino is on. Due to this structure, separate programs need to be loaded onto the Arduino for recording and actuating, or entirely separate Arduinos may be used. An alternative would be to implement a physical switch on the breadboard to determine the microcontroller’s operating mode. In our final design, the latter method was chosen to make the user experience as simple as possible; one rotating switch is used to select the mode of operation, and another switch is used to start or stop the operation.

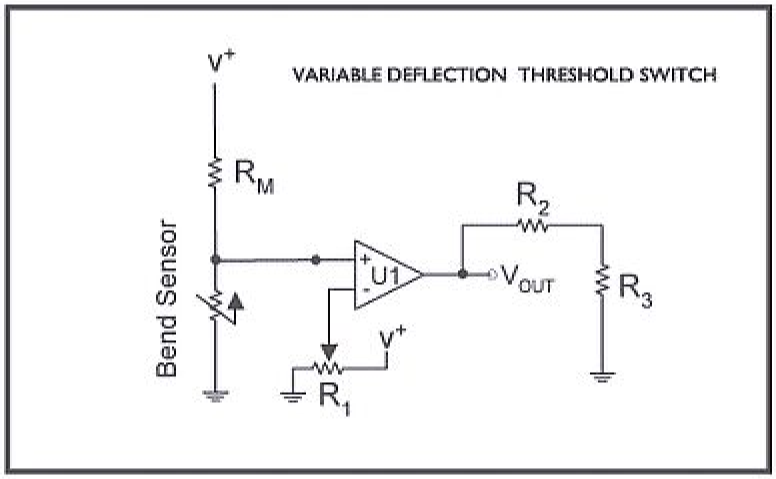
II. Testing the Sensors

To first test the functionality of the microcontroller, a program was written to read one analog input. A voltage divider was created using the flex sensor and an arbitrary 10kΩ resistor, and this was attached to the microcontroller’s analog pin for reading. The voltage reading is stored as an integer between 0 and 1023, with 0 representing 0V and 1023 representing 5V. This means that the value can easily be converted to volts, degrees bent, or any other arbitrary unit. By bending the flex sensor, the reading of the Arduino changed accordingly. As confirmed through our previous experimentation, the value changed linearly with respect to the amount of bending by the sensor. Thus, no complicated function is needed to convert the raw data to something useful for the exoskeleton.

After receiving multiple flex sensors, it was observed that their resistances were all about 25kΩ (as per the datasheet [10]). Therefore, the 10kΩ resistor was replaced with a 24kΩ resistor, the closest resistor value that was readily available in the lab, in the voltage divider for maximum variation in the readings.

A voltage divider is only one configuration that can be used to measure data from the flex sensor; several other methods involving operational amplifiers are possible. In the simplest implementation, an op amp can be used as an impedance buffer with unity gain. A potentiometer can be used to make the buffer adjustable. In addition to this, one can create a variable deflection threshold switch. A third possibility is to create a resistance to voltage converter. The schematics for these three implementations are shown below.





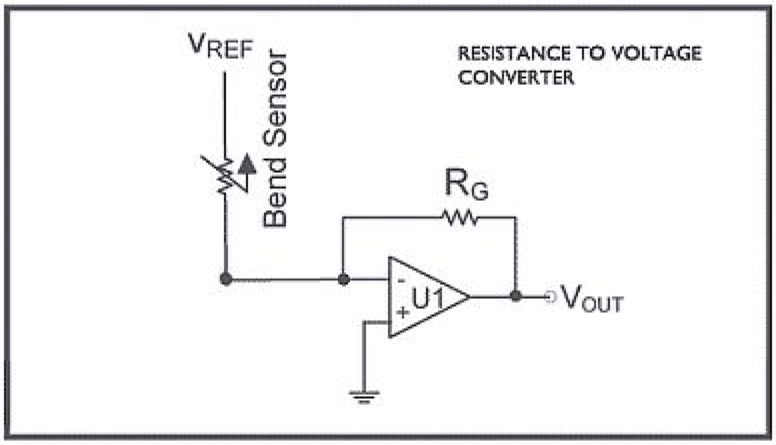


Figure 23: *Schematics for different methods of reading the flex sensor. [10]*

According to the datasheet for the flex sensor, the resistance to voltage converter can be used in situations where one wants an output at a low degree of bending. However, this setup is not necessary for our project; we have observed that the microcontroller can consistently detect a change of 1 degree in the flex sensor using the voltage divider configuration, which is sufficient resolution for our project. The variable deflection threshold switch can be used to create a Boolean output. In essence, this turns the sensor into a switch. This design is also not needed for our project. Even if we were to require a switch, the microcontroller can be programmed to achieve the same functionality. Therefore, the voltage divider configuration is what we use in our final design.

Since reading the output of the voltage divider occurs in the loop function, the microcontroller automatically samples the value as frequently as possible. For testing purposes, a delay of 1 second was implemented. In our actual implementation, the delay is shortened to 50 ms, thus establishing a frequency of 20 Hz. Based on our observations, 20 Hz is a suitable rate for polling the position of the finger joints; it is fast enough to capture quick, sharp changes in the bend of a finger joint, but does not needlessly record extra information.

III. Recording Multiple Sensors

After testing the microcontroller's functionality for one flex sensor, the next step was to implement the ability to read multiple sensors. Our recording glove has 10 flex sensors attached to it, and the ideal microcontroller must be able to store data from all of them simultaneously. However, the microcontroller can only process data serially, so actual parallel reading is impossible. Despite this, using a microcontroller is still viable as long as the delay between each read is unnoticeable. In addition to this limitation, the microcontroller does not have enough analog input pins for all the sensors. The first proposed solution was to simply use another microcontroller. However, this would require coordination between the two microcontrollers when recording, storing, and actuating. Instead, an easier solution was discovered; use a 4051 multiplexer [20], which is also much smaller than another microcontroller. For this solution to work, three of the microcontroller's digital pins must be used as outputs to control the multiplexer. Depending on the status of the three control pins, the multiplexer can map the value of one of eight different channels to a single analog input pin on the microcontroller. By changing the values of the three control pins, eight of the flex sensors can be used as part of a voltage divider to be read. After implementing this, the voltage readings corresponding to the remaining two flex sensors can be directly connected to two other available analog pins. The schematic is shown below.

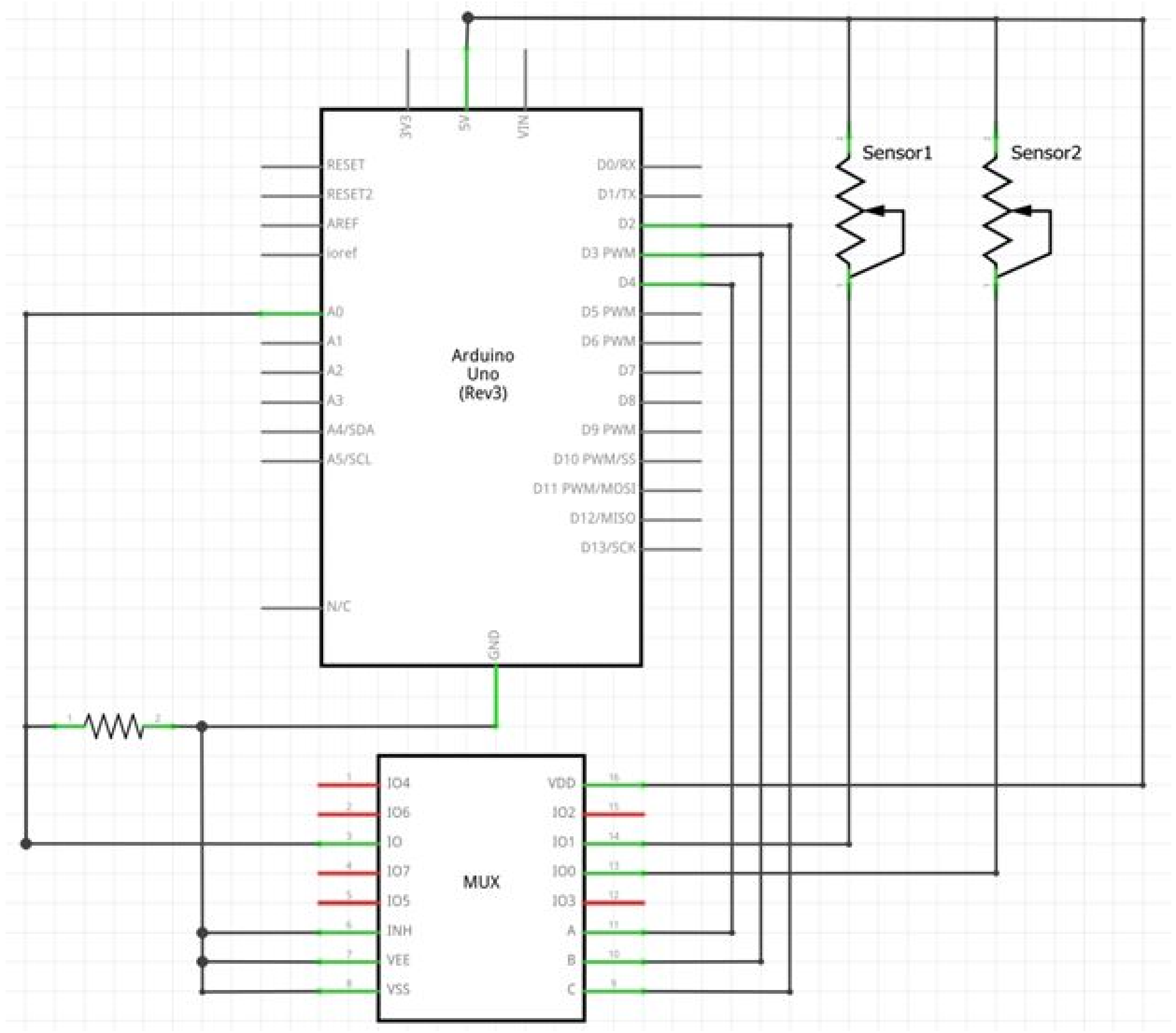


Figure 24: *Multiplexer handling multiple flex sensors. Only two sensors shown.*

When this circuit was first implemented, the values that were read did not make any sense. After several minutes of debugging, it was realized that we were using digital pins 0 and

1. According to official documentation on the Arduino Uno, these pins were reserved for other functions. After shifting the digital pins used, from 0 to 2, to 2 to 4, the circuit functioned as planned.

By the microcontroller's design, the action of reading multiple sensors must be serial. This may lead to problems when frequently polling multiple sensors. If there is too much latency, there may be a discrepancy between the current motion and the data recorded. If the switching speed is too fast, this may corrupt our data; different values from different flex sensors may interfere with each other. Using a multiplexer also adds to the time it takes to read a value. However, with an overall polling rate of 20 Hz, none of these problems have emerged. We have chosen a rate of 20 Hz (50 ms) and there will be 10 flex sensors on the recording glove. Therefore, we can use up to 5 ms at a time to read the value correlating to one flex sensor. If the actual time is more than 5 ms, the microcontroller will fall behind on reading the data. However, according to the documentation, 5 ms is a sufficient amount of time to switch the mux and read the data for one sensor; switching the mux takes nanoseconds, while pin reading/writing takes microseconds.

After reprogramming the microcontroller, testing was done to read data from two different flex sensors. At first we were afraid that there was interference between the two sensors; when one sensor was being bent, the readings corresponding to both sensors were changing. However it was soon realized this was due to the fact that, while bending one sensor, the other sensor was inadvertently moving. After making sure only one sensor was bent at a time (by holding the other sensor in place), it was determined that there was no interference. By using an ohmmeter in conjunction with the Arduino, it was confirmed that the collected readings were accurate. Once again, we were able to confirm that the values varied linearly with the amount of degrees each sensor was bent. Again, we also noticed that the recorded values were offset due to the different resistances of the flex sensors. This was easily corrected once the sensors were mounted in place on the recording glove, where the bending of the flex sensor is repetitive and consistent.

After our initial test with two sensors, we continued to implement functionality for all ten sensors while creating the recording glove. With our final design, there have been no issues with reading values from all ten sensors (code in appendix).

IV. Storing and Smoothing Data

In order to store the output of the microcontroller, a third party program was initially used as a serial port monitor (in the final design, the built-in serial monitor is used, and the built-in serial plotter can automatically display the data graphically for demonstration purposes). In this scenario, the program we used was CoolTerm for Windows. This program was chosen simply because it is free, open source, and easy to use with no setup necessary. The output of the microcontroller is forwarded to the program, and this output can be stored as a text file on the computer. The text file can then be read or edited at a later time, when the microcontroller needs to read it for actuating. The original idea was to use CoolTerm to send the stored data to the Arduino through the serial port. However, for reasons discussed later, this became unnecessary and CoolTerm is no longer used. The downside of our implementation is that the text file must be stored manually. An alternative would be to store the data onboard the microcontroller. However, this would greatly limit the amount of data we can save; our recordings would be severely limited in length since the microcontroller only has 1 KB of EEPROM memory. In addition, it would only be possible to store one recording at a time on the microcontroller. A future recording would have to overwrite the previous recording, and the previous recording would not be saved. In order to save multiple recordings, external storage must be used at some point.

After testing our final prototype of the recording glove, a small problem surfaced with the sensor readings; when bending the flex sensors, the values tended to bounce. When the sensor is bent, the values overshoot to larger numbers. When the sensor is straightened, the values overshoot to smaller numbers. This was especially the case when the user bent his finger very fast, during either flexion or extension of any joint. If the sensor is bent and held in that position, the readings from the sensor will overshoot by up to 10% and then decrease exponentially to a stable, accurate reading. If the user oscillates his finger back and forth very rapidly, the values overshoot but there is no time for the readings to adjust to their correct values. If the user moves his finger slowly however, there is little to no bounce. Therefore, some sort of error correction, smoothing, or filtering is required to convert the raw data to an accurate signal.

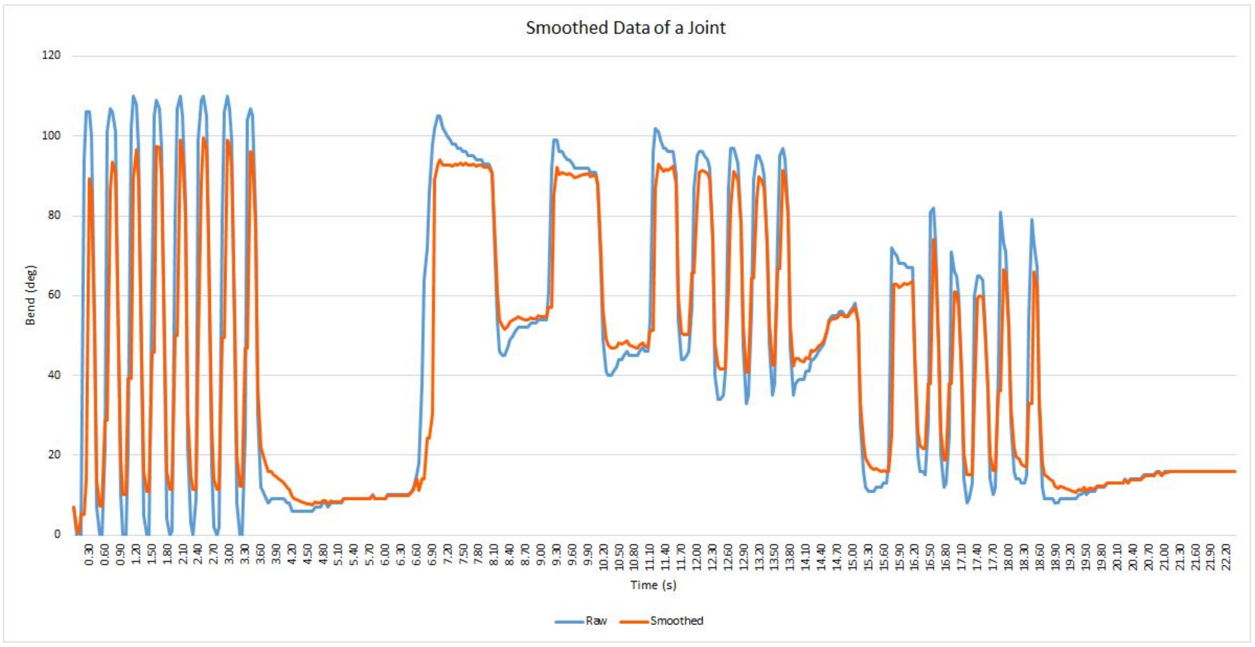


Figure 25: *Raw data compared against processed, smoothed data*

After experimenting with some possible smoothing techniques and low-pass filters, we discovered that a modified version of exponential moving average works best. The results are shown above. This method works well for reducing peaks and troughs, and it also works very well for removing the transient response when the sensor is bent and held. This filter, which was originally implemented in Microsoft Excel, and was later implemented in MATLAB, is now implemented in the Arduino code to simplify the user experience (code in appendix).

In addition to implementing the smoothing technique described above, we also needed to adjust the data for safety purposes. After the smoothing algorithm is applied, the final data must be limited to bend angles between 0 and 90 degrees. It is not guaranteed that all clinically normal people can bend their joints beyond these values, so this safety measure is required for IRB approval.

Since a resolution of one degree is more than sufficient for what we wish to accomplish (a change in one degree of any finger joint is unnoticeable), all of the processed values were then converted into integers. This makes processing the data file easier later on and reduces the need for the microcontroller to work with floating point numbers. In addition, the last row of values are truncated when processing the data; this is because it is not guaranteed that the last row will always be complete with ten data values. Truncating this data is acceptable because it only represents the last instant of the recording, which we can assume is irrelevant because the user was likely just trying to stop the recording during this time frame of 50 ms.

Originally, we were satisfied with simply copying and pasting the output of the Arduino into a text file to be saved on a personal computer. Then, we used MATLAB to filter the data and output a new text file. The text file had to be manually transferred from the computer to a microSD card so that the Arduino can read the filtered data via the microSD card shield. However, we wanted a better user experience by automatically creating and saving a file. In addition, we did not want our project to depend on MATLAB or some other third party software. Thus, our final solution implements the following: the recording program interacts directly with the SD card by automatically sending data to it, which is then processed by the Arduino when the user selects the processing mode.

When the user begins recording, the Arduino will automatically create a new text file. While reading and printing the values from the flex sensors, it also fills the open file with the data. Once the user stops recording via a toggle switch, the text file is saved. To process the data, the user must use rotary switch to select the processing mode, then use the toggle switch to start the processing program. The program will prompt the user for the name of the recording file and the name of the output file and then process the data.

When porting the processing from MATLAB to the Arduino, there were three main issues than had to be addressed (more than simply changing the syntax). The first issue is that, while MATLAB makes matrix operations simple to implement, the C++-like Arduino code does not. This issue was rectified by replacing the matrix operations with for loops. The second issue is that reading a text file is not easy on an Arduino; the file must be read one byte at a time. Therefore, in order to parse the file, the Arduino code needs to store the bytes in a character array buffer and delimit the data based on whitespace and newline characters. The third issue is that the Arduino has only 8 KB of SRAM, which is where variables are stored [21]. Since the recording file could be very large, the whole file cannot be loaded in at once. Thus, in order to support large files, only three lines from the recording file are kept in memory at a time (code in appendix).

For reasons discussed in the next subsection, the processing stage also includes converting the filtered bend angles to actual step counts for the actuation motors. Therefore, the values in the output file represent motor steps instead of degrees bent, and less calculations need to be made on the fly during actuation.

V. Actuating One Motor

In order to actuate a motor using the microcontroller, there needed to be a way to transmit the information to the Arduino. Our original solution required CoolTerm; through CoolTerm, it was simple to transmit an entire text file to the microcontroller via the serial port. However, we ran into a major problem when doing this. First, simply transmitting the data resulted in corrupted data. The whole file would not send, and the values would be incorrect. We tried experimenting with Windows or Unix-style newlines, but this was not the issue. In addition, we tried changing the baud rate from the default of 9600, but this did not work either. Finally, we were able to make the transmission work by implementing a transmit delay via CoolTerm's settings. This solution, which allowed the whole file to be sent with no errors, raised another issue; the transmission was too slow. It would take several minutes to transmit a data file representing only 20 seconds of recording. There were two solutions that we came up with to solve the problem. The first solution was to store the text file on an SD or microSD card and use a shield/board to interface it with the microcontroller. The second solution was to copy and paste the contents of the text file into the source code for the actuating program, essentially turning the data into a large static array of integers. Both solutions would avoid using the serial port to communicate. The second solution was easier to implement and did not rely on communication between the microcontroller and any peripherals, however it was also much more inelegant; a user of this device would have to modify the source code of the actuating program every time he or she wanted to actuate a different motion. At the time of making this decision, the first solution added some more hardware to the project, but allowed us to avoid the problem of using CoolTerm. In addition, the user had to transfer the text file from the computer to the SD card manually. Weighing the pros and cons of each solution, we chose to use an SD card. However, the drawback of manually moving files was eliminated when we chose for the recording to be directly stored on the SD card.

When the microcontroller reads from the SD card, it only reads one byte for each SD.read() function call. Therefore, bytes are read from the file and stored in buffers. After detecting a space or newline character (which are used to delimit the values), the contents of the buffer is converted into an integer. This program works with both Windows and Unix-style newlines. All of this transmitting and processing is fast enough to be done on the fly, which solves our original problem.

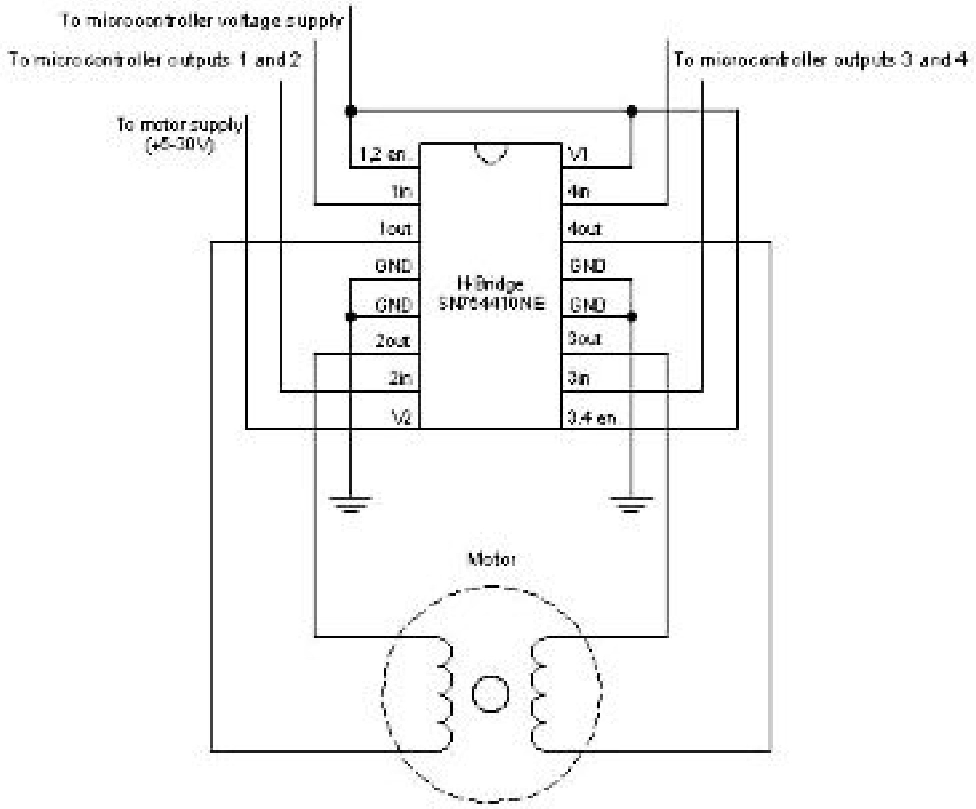


Figure 26: *H-bridge controlling one stepper motor [22]*

The motors we use are bipolar stepper motors [18]. Driving one motor requires four pins on the microcontroller as well as one H-bridge IC (SN754410NE). The motor is rated for 12 V, therefore a power supply external to the Arduino is necessary. We considered using batteries, however in our final design we realized we were drawing up to 5 A during actuation. This operation is not practical with batteries, so we use the power supply in the lab instead.

Using the raw data read from the microSD card was not acceptable; the text file stored bend angles, not motor rotation commands. We first tried to convert bend angles to step counts on the fly. First, we had to calculate the difference in bend angles between the current state and the previous state (a positive difference implied clockwise rotation, while a negative difference implied counter-clockwise rotation). After calculating this, the number of steps to move the motor was calculated as follows: Using a spool with a radius of 1 cm, and the equation $S = r\theta$ to find arc length, the length of wire that is retracted or extended equates to $\theta\pi/180$ cm, where θ is in degrees. θ can be converted to steps, based on the properties of the motor (our motors are 200 steps/revolution, or 1.8 degrees/step). To determine the relationship between length of cable pulled and bend degrees, this had to be manually calibrated for each joint by observing how many steps it takes to move the user's joint 45 degrees. By observation, this relationship was mostly linear, especially at bend angles less than 80 degrees. The values for this relationship can be different for different users, so the code must be modified if a different user were to use the actuating exoskeleton.

After implementing this function in the software, we found that it worked very well. However, this led us to realize another problem; converting the text file data into step commands required varying amounts of time. Converting 0 degrees to 0 steps took almost no time, and these commands completed too quickly. Converting a non-zero amount of degrees to a non-zero amount of steps using floating-point operations such as division required too much time to process. In order to solve this problem, two actions had to be taken. The first action was to move this conversion process to the main processing stage. The second action was to add a delay of up to 5 ms for each motor that required no movement. After implementing this, the timing of the motor movement matched the timing of the recorded motion.

VI. Actuating Multiple Motors

After successfully actuating one motor, the next step was to upscale and actuate multiple motors. Since we require two motors per finger, we chose to implement one fully-functional exoskeleton finger first. With simple modifications to the code, the Arduino was able to actuate both motors. However, four issues were identified at this stage.

The first issue was that, since each motor requires four pins for control, there are not enough pins on the Arduino Uno. This issue was easily rectified by using an Arduino Mega, which is larger and has more pins.

The second issue was as follows: our device assumes that, when actuation begins, the user's fingers are fully extended and the state of all the motors are such that the cables are taut. If the motors do not start in the correct position, the actuated movements will not match the recorded movements. In order to rectify this, the motors must return to their original positions after actuating. This solution is implemented in software; for each motor, an offset value keeps track of how many steps the motor is away from its start position. After actuation finishes, the offset value is used to step the motors back to their original positions. Now, as long as the motors start in their correct positions, they will end in their correct positions after each actuation session.

The third issue was that, even after the device finished actuating the motors, the motors were still left on; their coils were still energized, and the holding torque kept the motors from moving. This consumed a lot of power and made the motors hot. To fix this, the output pins of the Arduino that control the motors are all set low.

The fourth issue, which was most concerning, related to the operation of the motors. According to the Arduino documentation, only one motor can be driven at a time [23]. We proposed three solutions to this problem. The first solution was to buy more microcontrollers, potentially as much as one microcontroller per stepper motor. The second solution was to purchase several stepper motor AutoDrivers [24]. The third solution was to decompose each step command into smaller commands (one step clockwise/counterclockwise), and loop through these commands quickly while increasing the speed of the motors, essentially interleaving them. After reviewing the specifications for the AutoDrivers, we determined that they had many features that were not useful to us, and these features complicated their design and use. Using one AutoDriver for one motor also seemed very inelegant. In addition, they were more expensive than Arduinos. When considering the second proposal of using more Arduinos, similar concerns were raised; synchronizing the operations of the Arduinos would add complexity to the project, and using each Arduino to operate one or two motors seemed very inelegant. The third proposed solution was most desirable; it would add no hardware and only requires modification to the actuation software. Therefore, our best option was to try interleaving the control of the motors using one Arduino. However, we were not sure if this solution would work.

After implementing the interleaving solution, we were satisfied with what we observed. When one motor is set to 60 rpm and commanded to move a quarter of a turn, it takes 245 ms on average. When two motors are set to 60 rpm and are commanded to move a quarter of a turn sequentially, the entire process takes 490 ms on average; this is twice the time it takes for one motor to move, which is expected. When interleaving the motors, alternating commands to make them move one step at a time until they each make a quarter turn, the total amount of time the operation takes is 245 ms. This is surprising, because it seems like the Arduino is able to move both motors at the same time, despite the documentation. With this pleasant surprise, we were able to interleave all of the motors on the actuating exoskeleton without any more issues.

4. TRIALS

We were only able to test the actuating exoskeleton on Zach, because we did not realize our design would require such a tight fit to prevent the actuating exoskeleton from sliding around. If we redesigned the exoskeleton to have adjustable finger thickness, we could fit it to most hand sizes. The only other issue is that the exoskeleton would need to be calibrated for each person. We had Zach practice for 30 minutes with the glove learning the song “Ode to Joy” and 30 minutes without the glove learning “Hallelujah”. The songs are similar in difficulty and in length. Zach had 60% fewer mistakes over an average of 3 trial runs, which is similar to the Georgia Tech results. However, any gains made in correct style or positioning would be difficult to measure. Future tests could record the participants after they finished practicing to compare it with the original recorded sample.

5. CONCLUSION

We have successfully demonstrated the feasibility of Force-Feedback Passive Learning. We had hoped to run the trial with more participants to make the results more meaningful, but we did not have the time to redesign the actuating exoskeleton. The trial run showed similar results to the vibrotactile learning research, but that research was different in that they had the participants learn the entire song naturally prior to practicing. We believe force-feedback is more versatile and can be applied to many skills that require motor learning.

6. APPENDIX

A. Code

The code below was written for an Arduino Mega. The Arduino IDE or Arduino Web Editor is required to run it. In addition, the Serial Monitor must be open in order to interact with the user.

```
/*
  This program is a combination of the recording program and the actuating
program. Selection between the two modes is made using external switches on the
breadboard.
*/
#include <SPI.h>
#include <SD.h>
#include <Stepper.h>

// For Mode Selection
#define RECORD_PIN A6
#define ACTUATE_PIN A7
#define START_STOP A8
enum mode {RECORD, PROCESS, ACTUATE};
mode operatingMode;

// For Recording
#define muxA A3 // LSB
#define muxB A4
#define muxC A5 // MSB

// For Processing
#define STEPS_PER_REV 200 // steps per revolution for the motor, also used in
actuating
#define NUM_SENSORS 10 // also used in actuating
#define SPOOL_RADIUS 1 // in cm
float maxLengthChange[NUM_SENSORS] = {1, 1, 3.3, 3.3, 3, 3.7, 3, 3, 2.5, 2.2};
// in cm, for each joint
```

```
float fraction[NUM_SENSORS] = {1, 1, 0.4375, 0.4375, 0.5, 0.375, 0.5, 0.375,
0.375, 0.375};

// For Actuating
/*
  SD Card Pins, For Reference:
  GND GND black
  VCC 5V white
  MISO 50 gray
  MOSI 51 purple
  SCK 52 blue
*/
#define CS 53 // SD card pin for CS, green
#define MAX_MOTOR_PIN 49 // change these to match the bounds on the pins used
for the motor
#define MIN_MOTOR_PIN 18 // motor pins should be used sequentially
int SPEED_MULTIPLIER = 12000 / STEPS_PER_REV;
Stepper *motor[NUM_SENSORS] = {
  new Stepper(STEPS_PER_REV, 0, 0, 0, 0), // f11
  new Stepper(STEPS_PER_REV, 0, 0, 0, 0), // f12
  new Stepper(STEPS_PER_REV, 49, 48, 47, 46), // f21
  new Stepper(STEPS_PER_REV, 45, 44, 43, 42), // f22
  new Stepper(STEPS_PER_REV, 41, 40, 39, 38), // f31
  new Stepper(STEPS_PER_REV, 37, 36, 35, 34), // f32
  new Stepper(STEPS_PER_REV, 33, 32, 31, 30), // f41
  new Stepper(STEPS_PER_REV, 29, 28, 27, 26), // f42
  new Stepper(STEPS_PER_REV, 25, 24, 23, 22), // f51
  new Stepper(STEPS_PER_REV, 21, 20, 19, 18)  // f52
};

void setup() {
  Serial.begin(9600);

  while (!Serial) {} // wait for serial port to connect. Needed for native USB
port only
  if (!SD.begin(CS)) {
    Serial.println("SD card failed, or not present");
    while (1); // stop the program
  }
}

void loop() {
  Serial.println("Please make sure the START_STOP switch is off.");
  while(digitalRead(START_STOP) == HIGH){} // wait for user to toggle off
  Serial.println("Ready to start...");
  while(digitalRead(START_STOP) == LOW) {
    delay(100); // effectively debouncing
    if(digitalRead(RECORD_PIN) == HIGH) {
      if(operatingMode != RECORD) {Serial.println("Mode switched to RECORD.");}
      operatingMode = RECORD;
    }
    else if(digitalRead(ACTUATE_PIN) == HIGH) {
      if(operatingMode != ACTUATE) {Serial.println("Mode switched to
ACTUATE.");}
```

```
      operatingMode = ACTUATE;
    }
    else {
      if(operatingMode != PROCESS) {Serial.println("Mode switched to 
PROCESS.");}
      operatingMode = PROCESS;
    }
  }

  while(Serial.available()) { // clear the input buffer
    Serial.read();
  }

  switch (operatingMode) {
    case RECORD:
      doRecording();
      break;
    case PROCESS:
      doProcessing();
      break;
    case ACTUATE:
      doActuating();
      break;
  }
}

/*
  This program records data from all the sensors.
  Each value in a time frame is space-delimited, with different time frames 
newline-delimited.
  Each time frame is separated by 50 ms.
*/
void doRecording() {
  unsigned long time = millis();

  pinMode(muxA, OUTPUT); // LSB
  pinMode(muxB, OUTPUT);
  pinMode(muxC, OUTPUT); // MSB

  while(SD.exists(String(time % 100000000) + ".txt")) { // make a new recording 
file
    time = millis();
  }
  File recording = SD.open(String(time % 100000000) + ".TXT", FILE_WRITE);
  if(!recording) {
    Serial.print("Error opening file: ");
    Serial.println(String(time));
    return;
  }

  Serial.println("Recording started!");
  Serial.println("Recording to " + String(time) + ".TXT");
  while(digitalRead(START_STOP) == HIGH) {
    digitalWrite(muxC, LOW);
digitalWrite(muxB, LOW);
digitalWrite(muxA, LOW);
int value11 = analogRead(A0);
int f11 = map(value11, 0, 1023, 0, 80);
digitalWrite(muxA, HIGH);
int value12 = analogRead(A0);
int f12 = map(value12, 0, 1023, 0, 80);
digitalWrite(muxB, HIGH);
digitalWrite(muxA, LOW);
int value21 = analogRead(A0);
int f21 = map(value21, 460, 270, 0, 80);
digitalWrite(muxA, HIGH);
int value22 = analogRead(A0);
int f22 = map(value22, 460, 250, 0, 80);
digitalWrite(muxC, HIGH);
digitalWrite(muxB, LOW);
digitalWrite(muxA, LOW);
int value31 = analogRead(A0);
int f31 = map(value31, 465, 275, 0, 80);
digitalWrite(muxA, HIGH);
int value32 = analogRead(A0);
int f32 = map(value32, 470, 230, 0, 80);
digitalWrite(muxB, HIGH);
digitalWrite(muxA, LOW);
int value41 = analogRead(A0);
int f41 = map(value41, 430, 275, 0, 80);
digitalWrite(muxA, HIGH);
int value42 = analogRead(A0);
int f42 = map(value42, 490, 230, 0, 80);
int value51 = analogRead(A1);
int f51 = map(value51, 430, 230, 0, 80);
int value52 = analogRead(A2);
int f52 = map(value52, 480, 240, 0, 80);
/* // For calibration
Serial.print(String(value11) + " ");
Serial.print(String(value12) + " ");
Serial.print(String(value21) + " ");
Serial.print(String(value22) + " ");
Serial.print(String(value31) + " ");
Serial.print(String(value32) + " ");
Serial.print(String(value41) + " ");
Serial.print(String(value42) + " ");
Serial.print(String(value51) + " ");
Serial.println(String(value52));
*/
Serial.print(String(f11) + " ");
Serial.print(String(f12) + " ");
Serial.print(String(f21) + " ");
Serial.print(String(f22) + " ");
Serial.print(String(f31) + " ");
Serial.print(String(f32) + " ");
Serial.print(String(f41) + " ");
Serial.print(String(f42) + " ");
Serial.print(String(f51) + " ");
```

```
    Serial.println(String(f52));

    recording.print(String(f11) + " ");
    recording.print(String(f12) + " ");
    recording.print(String(f21) + " ");
    recording.print(String(f22) + " ");
    recording.print(String(f31) + " ");
    recording.print(String(f32) + " ");
    recording.print(String(f41) + " ");
    recording.print(String(f42) + " ");
    recording.print(String(f51) + " ");
    recording.println(String(f52));
    delay(50);
  }
  recording.close();
  Serial.println("Done recording!");
  Serial.println("Saved to " + String(time) + ".TXT");
}

/*
  This program takes two file names as an input.
  The first file should be a recording file, while the second file is the
resulting filtered data.
  The filtered bend angles are printed, and the final data is converted from
degrees to motor steps.
*/
void doProcessing() {
  String buf[3][NUM_SENSORS], inName = "", outName = "";
  float alpha1 = 0.1, alpha2 = 0.8, ema[3][NUM_SENSORS], temp[3][NUM_SENSORS],
filtered[3][NUM_SENSORS];
  int steps[3][NUM_SENSORS];

  Serial.print("Enter the name of the recording file: ");
  while(!Serial.available()); // wait for input
  while(Serial.available()) {
    inName = Serial.readStringUntil('.'); // assume .TXT
  }
  inName += ".TXT"; // append file extension
  inName.toUpperCase();
  Serial.println(inName);

  File inFile = SD.open(inName);
  if(!inFile) {
    Serial.print("Error opening file: ");
    Serial.println(inName);
    return;
  }

  Serial.print("Enter the name of the output file: ");
  delay(100); // stall
  while(Serial.available()) { // clear the input buffer
    Serial.read();
  }
  while(!Serial.available()); // wait for input
  while(Serial.available()) {
    outName = Serial.readStringUntil('.'); // assume .TXT
  }
  outName += ".TXT"; // append file extension
  outName.toUpperCase();
  Serial.println(outName);

  SD.remove(outName); // overwrite existing files
  File outFile = SD.open(outName, FILE_WRITE);
  if(!outFile) {
    Serial.print("Error opening file: ");
    Serial.println(outName);
    return;
  }

  Serial.println("Processing started!");
  int i = 0, numSamples = 0;
  while(inFile.available()) { // find number of samples
    char character = inFile.read();
    if(character == '\n') {
      ++numSamples;
    }
  }
  inFile.close();
  inFile = SD.open(inName); // restart reading
  while(inFile.available()) {
    // Parse the file
    char character = inFile.read();
    if(character > 47 && character < 58 || character == 45) { // 0 to 9 and -,
since the value can be any int
      buf[2][i] += character;
    }
    else if(character == '\r' || character == ' ') {
      ++i;
    }
    else { // newline character
      // Buffer is filled with 10 new values
      if(buf[1][0] == "") { // first line of file
        for(int k = 0; k < NUM_SENSORS; ++k) { // this assumes your hand starts
completely straight
          ema[2][k] = buf[2][k].toInt(); // auto-cast to float
          temp[2][k] = ema[2][k];
          filtered[2][k] = ema[2][k];
        }
      }
      else if(buf[0][0] == "") { // second line of file, special case
        for(int k = 0; k < NUM_SENSORS; ++k) {
          // limit filtered from 0 to 90
          if(filtered[1][k] > 90) {
            filtered[1][k] = 90;
          }
          else if(filtered[1][k] < 0) {
            filtered[1][k] = 0;
          }
          // plot filtered
          if(k != NUM_SENSORS - 1) {
            Serial.print(String(filtered[1][k]) + " "); // prints 2 decimal
places by default
          }
          else {
            Serial.println(String(filtered[1][k]));
          }

          // convert angles to step commands
          steps[1][k] = filtered[1][k] * maxLengthChange[k] * STEPS_PER_REV /
(180 * PI * SPOOL_RADIUS); // auto-cast to int
        }
      }
      else { // bulk of file: filter
        for(int k = 0; k < NUM_SENSORS; ++k) {
          ema[1][k] = alpha1 * buf[1][k].toInt() + (1 - alpha1) * ema[0][k]; //
floating-point multiplication

          if(numSamples == 1 || buf[2][k].toInt() - buf[1][k].toInt() <= 10) {
// near end of file, or big change
            temp[1][k] = alpha2 * buf[1][k].toInt() + (1 - alpha2) * ema[0][k];
          }
          else {
            temp[1][k] = alpha1 * buf[1][k].toInt() + (1 - alpha1) * ema[0][k];
          }

          if(temp[1][k] - temp[0][k] > 5) {
            filtered[1][k] = temp[0][k];
          }
          else {
            filtered[1][k] = temp[1][k];
          }

          // limit filtered from 0 to 90
          if(filtered[1][k] > 90) {
            filtered[1][k] = 90;
          }
          else if(filtered[1][k] < 0) {
            filtered[1][k] = 0;
          }

          // plot filtered
          if(k != NUM_SENSORS - 1) {
            Serial.print(String(filtered[1][k]) + " "); // prints 2 decimal
places by default
          }
          else {
            Serial.println(String(filtered[1][k]));
          }

          // convert angles to step commands
          //steps[1][k] = round((filtered[1][k] - filtered[0][k]) * 
maxLengthChange[k] * STEPS_PER_REV / (180 * PI * SPOOL_RADIUS)); // auto-cast 
to int
          steps[1][k] = round((filtered[1][k] - filtered[0][k]) * 2 * PI * 
SPOOL_RADIUS * fraction[k] * 2 * STEPS_PER_REV / (180 * PI * SPOOL_RADIUS));
        }
      }

      for(int k = 0; k < NUM_SENSORS; ++k) {
        buf[0][k] = buf[1][k];
        buf[1][k] = buf[2][k];
        buf[2][k] = ""; // reset buf

        ema[0][k] = ema[1][k];
        ema[1][k] = ema[2][k];

        temp[0][k] = temp[1][k];
        temp[1][k] = temp[2][k];

        filtered[0][k] = filtered[1][k];
        filtered[1][k] = filtered[2][k];

        steps[0][k] = steps[1][k];
        steps[1][k] = steps[2][k];

        if(buf[0][0] != "") { // skip first loop, still loading in data
          if(k != NUM_SENSORS - 1) {
            outFile.print(String(steps[0][k]) + " ");
          }
          else {
            outFile.println(String(steps[0][k]));
          }
        }
      }

      i = 0; // reset the buffer counts
      --numSamples;
      if(numSamples == 0) { // done processing
        break;
      }
    }
  }
  inFile.close();
  outFile.close();
  Serial.println("Done processing!");
  Serial.println("Saved to " + outName);
}

/*
  This program reads the data sent to it, and actuates the motors.
  Data is read from a text file on the SD card, byte-wise.
  Data is converted from bytes to integers, then used to drive the motors.
*/
void doActuating() {
  String buf[NUM_SENSORS], fileName = "";
  int steps[NUM_SENSORS];
  int offset[NUM_SENSORS] = {0};

  Serial.print("Enter the name of the actuating file: ");
  while(!Serial.available()); // wait for input
  while(Serial.available()) {
    fileName = Serial.readStringUntil('.'); // assume .TXT
  }
  fileName += ".TXT"; // append file extension
  fileName.toUpperCase();
  Serial.println(fileName);

  File dataFile = SD.open(fileName);
  if(!dataFile) {
    Serial.print("Error opening file: ");
    Serial.println(fileName);
    return;
  }

  Serial.println("Actuating started!");
  int i = 0;
  while(dataFile.available()) {
    // Parse the filtered recording file
    char character = dataFile.read();
    if(character > 47 && character < 58 || character == 45) { // 0 to 9 and -,
since the value must be an int from -90 to 90
      buf[i] += character;
    }
    else if(character == '\r' || character == ' ') {
      ++i;
    }
    else { // newline character
      // Buffer is filled with 10 values, actuate motors
      int zeroCount = 0; // number of motors that don't move

      for(int j = 0; j < NUM_SENSORS; ++j){
        steps[j] = buf[j].toInt();
        offset[j] += steps[j]; // distance from starting position
        if(steps[j] == 0) {
          ++zeroCount;
        }
        motor[j]->setSpeed(90); // set speed for all motors
      }

      int motorsDone = 0;
      bool firstLoop = true;
      while(motorsDone < NUM_SENSORS) {
        motorsDone = 0;
        for(int j = 0; j < NUM_SENSORS; ++j) {
          if(zeroCount > 0) {
            delay(5);
            --zeroCount;
          }
          if(steps[j] > 0) { // pull cable
            //if(firstLoop) {motor[j]->setSpeed(steps[j] * SPEED_MULTIPLIER);}
// set speed on first loop
            motor[j]->step(-1);
            --steps[j];
          }
          else if(steps[j] < 0) { // release cable
            //if(firstLoop) {motor[j]->setSpeed(steps[j] * SPEED_MULTIPLIER *
-1);}
            motor[j]->step(1);
            ++steps[j];
          }
          else { // no more steps for this motor
            ++motorsDone;
          }
        }
        firstLoop = false;
      }

      for(int j = 0; j < NUM_SENSORS; ++j) { // reset buf
        buf[j] = "";
      }
      i = 0; // reset the buffer counts
    }
  }
  dataFile.close();
  Serial.println("Done actuating!");

  // Return motors to starting positions
  int motorsDone = 0;
  for(int i = 0; i < NUM_SENSORS; ++i) {
    motor[i]->setSpeed(60);
  }
  while(motorsDone < NUM_SENSORS) {
    motorsDone = 0;
    for(int j = 0; j < NUM_SENSORS; ++j) {
      if(offset[j] > 0) { // release cable
        motor[j]->step(1);
        --offset[j];
      }
      else { // steps <= 0 for this motor, no action
        ++motorsDone;
      }
    }
  }

  // reset all pins to LOW, saves power/heat
  for(int i = MIN_MOTOR_PIN; i < MAX_MOTOR_PIN + 1; ++i) {
    digitalWrite(i, LOW);
  }
}
```

7. REFERENCES

1. Huang, K., Do, E. L., and Starner, T. PianoTouch: A wearable haptic piano instruction system for passive learning of piano skills. In Proceedings the International Symposium on Wearable Computers, IEEE, 2008, pp. 41-44

2. Seim, Caitlyn and Reynolds-Haertle, Saul and Srinivas, Sarthak and Starner, Thad.
Tactile Taps Teach Rhythmic Text Entry: Passive Haptic Learning of Morse Code,
Proceedings of the 2016 ACM International Symposium on Wearable Computers,IEEE,2016.

3. Cynthia C. Norkin, D Joyce White. Measurement of Joint Motion: A Guide to Goniometry,Fourth Edition.

4. Andreas Wege and Günter Hommel. Development and Control of a Hand Exoskeleton for Rehabilitation of Hand Injuries. Technische Universität Berlin Berlin, Germany.
Intelligent Robots and Systems, 2005.

5. Stergiopoulos, P.; Fuchs, P. and, Laurgeau, C.: Design of a 2-Finger Hand Exoskeleton for VR Grasping Simulation, EuroHaptics 2003

6. Shinichi Furuya, Martha Flanders, and John F. Soechting. Hand kinematics of piano playing, Journal of Neuropsychology 2011.

7. Sparkfun Flex Sensor Tutorial
https://learn.sparkfun.com/tutorials/flex-sensor-hookup-guide

8. Favetto, Ambrosio, Appendino, et al. Embedding an Exoskeleton Hand in the Astronaut’s EVA Glove: Feasibility and Ideas. International Journal of Aerospace Sciences, 2012
http://article.sapub.org/10.5923.j.aerospace.20120104.03.html

9. SpectraSymbol Thinpot Datasheet

http://www.spectrasymbol.com/wp-content/uploads/2016/12/THINPOT-DATA-SHEET-Rev-B.pdf

10. SpectraSymbol Flex Sensor Datasheet

https://www.sparkfun.com/datasheets/Sensors/Flex/flex22.pdf

11. Zwieten, K. J. & P Schmidt, Klaus & Bex, Geert & L Lippens, Peter & Duyvendak, Wim. An Analytical Expression for the D.I.P. - P.I.P. Flexion Interdependence in Human Fingers. Acta of bioengineering and biomechanics / Wroclaw University of Technology, 2015

https://www.researchgate.net/publication/275659149_An_Analytical_Expression_for_the_DIP_-_PIP_Flexion_Interdependence_in_Human_Fingers

11. Majeau, Lucas & Borduas, Jonathan & Loranger, et al. Dataglove for consumer applications. 2011

https://www.researchgate.net/publication/263892786_Dataglove_for_consumer_applications

12. Lu, Pengfei & Huenerfauth, Matt. Accessible Motion-Capture Glove Calibration Protocol for Recording Sign Language Data from Deaf Subjects.

http://www.qc.cuny.edu/Academics/Degrees/DMNS/Faculty%20Documents/Heunerfath2.pdf

13. Thingiverse 3D Printed Exoskeleton Hands

https://www.thingiverse.com/thing:892654

14. Thingiverse Exo Glove

https://www.thingiverse.com/thing:2394197

15. MedicineNet.com Medical Illustrations: Finger Anatomy Picture

https://www.medicinenet.com/image-collection/finger_anatomy_picture/picture.htm

16. Hackster.io Complete Motor Guide for Robotics

https://www.hackster.io/taifur/complete-motor-guide-for-robotics-05d998

17. Adafruit All About Stepper Motors

https://learn.adafruit.com/all-about-stepper-motors/types-of-steppers

18. Adafruit Stepper Motor - NEMA-17 size

https://www.adafruit.com/product/324

19. Sparkfun Small Stepper Motor

https://www.sparkfun.com/products/10551

20. Texas Instruments CMOS Single 8-Channel Analog Multiplexer/Demultiplexer With Logic-Level Conversion

http://www.ti.com/lit/ds/symlink/cd4051b.pdf

21. Arduino Memory

https://www.arduino.cc/en/Tutorial/Memory

22. Code, Circuits, & Construction: Stepper Motors

http://www.tigoe.com/pcomp/code/circuits/motors/stepper-motors/

23. Arduino Stepper: step(steps)

https://www.arduino.cc/en/Reference/StepperStep

24. Sparkfun AutoDriver - Stepper Motor Driver(v13)

https://www.sparkfun.com/products/13752

25. Pinshape: 3D Printed Galacta Hand

https://pinshape.com/items/30234-3d-printed-galacta-hand