Tele-operation of a Robotic Arm with Attached Palpating End Effector

Honors Instrumentation Project Manual

Michael Ainsworth, Nicholas Maritato,
Dante Navarro, Arik Slepyan

Johns Hopkins University

Table of Contents

1. Introduction

1.1 Overview of Project

The work presented in this document represents the collective work of ***Michael Ainsworth***, ***Nicholas Maritato***, ***Dante Navarro***, and ***Arik Slepyan*** in the creation of a teleoperated robotic device for the aimed usage of remote palpation in healthcare. While the end goal of this system is in creating a quantifiable remote palpation system, the goal of this document is to outline the steps and processes required to build a proof-of-concept system from start to finish. The aim is thus to provide an educational how-to guide to allow others to build off of this work in the hopes of creating a more commercially viable system.

In the development of our system, we teleoperated the Universal Robots UR5 robot arm. This arm was chosen as it is an industrially viable robot arm that is available at Hopkins via the Wyman Park robotics labs. In this document, we lay forth the steps we took to create a wireless arm tracker that the user can wear to have complete control over moving the UR5 in a very intuitive way. With this, the user can move their arm to control the movement in the UR5, allowing for completely wireless, remote palpation control.

The other half of the system is the development of the palpation sensing handpieces that attach to the end of the UR5 robot arm. In this work, two separate handpieces were developed, the *Palpating Puck*, and the *Helping Hand*. The *Palpating Puck* is a simple puck of soft silicone that has a grid of pressure sensitive fabric and temperature sensors on the end of it, allowing for quantifiable patch-region palpation feedback. The *Helping Hand* is a 3D printed actuatable hand that has silicone fingertips with a similar pressure sensitive fabric built in. In this way, the user can wear a glove in addition to the arm tracker that has integrated flex sensors that allow for wireless actuation of the hand. The glove also has integrated piezo motors for haptic feedback. Together, the overall system allows for robust and human-like palpation.

1.2 Github Repository

All the code for wireless communication and control of the UR5 and the STL files for the 3D printed palpation handpieces is made publically available via our Github:

https://github.com/Nick7244/Honors-Instrumentation.git

NOTE: The following files on our Github were provided via the Robot Devices, Kinematics, Dynamics, and Controls course at Hopkins (EN.530.646): ***DH.m***, ***ROTX.m***, ***ROTY.m***, ***ROTZ.m***, ***SKEW3.m***, ***twistExp.m***, ***ur5_interface.m***, ***ur5FwdKin.m***, ***ur5InvKin.m***, and ***ur5InvKinControl.m***.

1.3 Acknowledgements

We would like to acknowledge all those that helped us in the development of this system. Specifically we'd like to thank ***Dr. Nitish Thakor*** for his continued guidance, inspiration, and motivation throughout development, ***Dr. Simon Leonard*** for his assistance in getting access and connection control to the UR5 robots, and ***Dr. Jin Seob Kim*** for his assistance in understanding the basics of our arm tracker math, especially in the usage of quaternions and transformation matrices. We'd also like to thank ***Sriramana Sankar*** for his help in the development of flexible, fabric-based force sensors.

2. Robot Arm Teleoperation

Teleoperation is an up and coming field within medicine that can allow for many new advances within the healthcare space. In this portion of our work, we break down the development of a teleoperated robot arm that can be controlled by an intuitive and unobtrusive wireless arm tracker. This can be potentially applied to various medical applications, including remote palpation. For our purposes, we used the Universal Robots UR5 robot arm in the development of this teleoperation system.

2.1 Setting up a Linux Environment

Connection to and control of the UR5 (either via the actual robot or via the robot simulation software) requires a computer with a Linux based operating system to run the Robot Operating System (ROS). For purposes of compatibility, Ubuntu is recommended. For this project, we used Ubuntu 16.04, Xenial release. This version of Ubuntu was chosen because the specific ROS version we needed only works with Xenial and older (see below, ***Installing ROS and connecting to the UR5 Simulation***).

Since we needed to install Ubuntu on one of our own personal computers, this required setting up a dual boot on an existing Windows or Mac computer. The dual boot setup process is in our experience less complicated for Windows PCs so this is recommended over Mac if you are attempting to set up a dual boot. There is minimal documentation on setting up a dual boot on a Mac so proceed at your own risk.

The process of setting up a dual boot involves the following steps:
- Partitioning your hard drive to create dedicated partitions for Ubuntu
- Downloading the Ubuntu OS installer onto a bootable USB drive
- Booting your PC from the USB drive
- Assigning the proper hard drive partitions for the installer to install on

This process can be quite risky and if you are not careful, you may risk losing some or all the data stored on your hard drive. Because of this, it is highly recommended that you ***backup your entire hard drive*** before beginning this process. Detailed instructions for the process of setting up this dual boot for Ubuntu 16.04 onto a Windows 10 based computer can be found at the following link:

https://itsfoss.com/install-ubuntu-1404-dual-boot-mode-windows-8-81-uefi/

Following this Linux environment setup, you can now install and operate ROS through the Linux boot.

2.2 Installing ROS and Connecting to the UR5 Simulation

Once you have a Linux environment setup, you must next download and install ROS. In order to connect with the robots in the Wyman Park building, it is recommended that you use the ROS Kinetic release. Instructions for installing this release on your Linux system can be found at the following link:

http://wiki.ros.org/kinetic/Installation/Ubuntu

You must then download the repository for the UR5 specific libraries to be able to run the UR5 simulation software. This repository and instructions for its installation come from the Robot Devices, Kinematics, Dynamics, and Control course taught at Johns Hopkins (EN.530.646). In order to install these libraries and set up the simulation software, use the following set of instructions (again these are provided by documentation from EN.530.646):

1. Open a terminal in Linux
2. Install git with "sudo apt-get install git"
3. Get source code with "git clone https://git.lcsr.jhu.edu/mxu29/Labs.git"
4. Go into the folder with "cd Labs/lab0"
5. "sed -i -e 's/nr$//' robot dkdc.sh"
6. "sudo chmod 755 robot dkdc.sh"
7. "./robot dkdc.sh"

Now, whenever you wish to start the UR5 simulation, go into the Labs/lab0 folder from the terminal ("cd Labs/lab0"), and once there run the following commands:

1. source devel/setup.bash
2. roslaunch robot_dkdc ur_simulation.launch

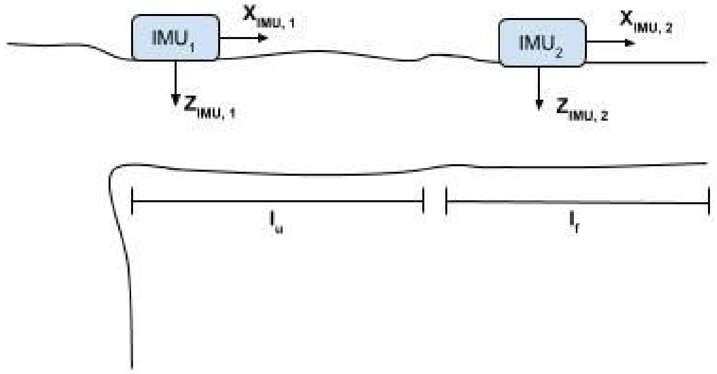
This should start the Gazebo service to launch the UR5 simulation software. Using this simulation software is incredibly important, as it allows the user to simulate the UR5 control without fear of damaging the machine. Many safety precautions are needed in using the real system (such as collision detection and obstacle avoidance), so it is always recommended to test and demonstrate proper usage on the simulation first before using the real system.

2.3 User Arm Tracking Development and Math

Now that we've defined how to set up your computer in preparation for UR5 control, we will switch gears and demonstrate the development of the arm tracker system. In developing the teleoperation control of the UR5, we developed a wireless arm tracker that is able to continuously estimate the user's wrist position and send that to the computer controlling the UR5 to move the robot via an inverse kinematics MATLAB script (more on this later). To design a true teleoperation device, we must have the UR5 mimic the movements of the user by knowing the spatial coordinates of the user's arm. This requires an arrangement of sensors along the user's arm to determine absolute position and orientation. In completing this task, we used the following materials:

- Arduino Mega (Arduino Uno or Nano may also suffice)
- 2 Adafruit BNO055 absolute orientation sensors
- 2 ELEGOO mini breadboards
- Electrical wire
- Velcro straps

The overall goal of this arm tracker device is to calculate the elbow and wrist position of the user relative to their shoulder based on the readings from a set of Inertial Measurement Units (IMUs). In our case, the IMUs we used are the Adafruit BNO055. We based our arm tracker on the work presented in [1]. In the referenced work, a 4 degree of freedom (DOF) arm tracker is built using a single IMU and a potentiometer. We built upon this work to create a 6-DOF arm tracker using two IMUs, one placed on the upper arm and the other on the upper forearm. The diagram below demonstrates the setup for our tracker.



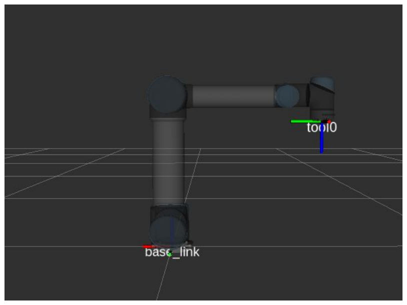


Figure 1: (a) Sample schematic of arm tracker placement and all important variables for calculation. Arm is extended in the "T-pose" position, which acts as the initialization position for the robot. (b) Gazebo simulation of UR5 robot mimicking user arm position, this is the equivalent "T-pose" for the robot.

In the above diagram, $\boldsymbol{l_u}$ and $\boldsymbol{l_f}$ represent the length of the upper arm and forearm respectively, $\boldsymbol{X_{IMU,1}}$, $\boldsymbol{X_{IMU,2}}$, $\boldsymbol{Z_{IMU,1}}$, and $\boldsymbol{Z_{IMU,2}}$ represent the x and z axis of the two IMUs respectively. As can be seen, the first IMU is placed on the upper arm of the user and the second is placed on the lower forearm of the user. The arm, as shown, is extended in the "T-pose", which for us we designated as the starting position of the system and the calibration pose of the robot (as seen in the right figure). In the MATLAB code, this pose is achieved with a joint space vector of [0 -π/2 -π/2 -π/2 π/2 0] (see below, ***Connecting to and Controlling the UR5 from MATLAB***, for how to achieve this pose with the robot). For calculating wrist position, it is important to introduce the ***quaternion***, which is a four-element vector useful for encoding position and orientation in 3D space. The quaternion is composed of one real element and three imaginary elements, representing an axis of rotation, and the angle about the axis that is rotated. In the development of this arm tracker, we monitor and calculate wrist position by defining a quaternion for the upper arm, $\boldsymbol{q_u}$, and forearm, $\boldsymbol{q_f}$, of the user. In this initialization/calibration phase, these quaternions are initialized to $\boldsymbol{q_u} = \boldsymbol{q_f} = [1\ 0\ 0\ 0]$ as the arm is resting in the "T-pose" position seen above. From there, the quaternion is updated directly from the IMU using the following command in the Arduino sketch:

```
imu::Quaternion quat = <BNO055_sensor_object>.getQuat();
```

See the code in ***Full_Arm_Tracking_UserEnd.ino*** on the Github for more information and details on quaternion extraction and usage. Once the quaternions $\boldsymbol{q_u}$ and $\boldsymbol{q_f}$ are calibrated and set up for continuous reading from the IMU, we can use the readings to calculate the elbow position relative to the should ($\boldsymbol{p_{e,s}}$) and wrist position relative to the elbow ($\boldsymbol{p_{w,e}}$) as follows:

$$p_{e,s} = q_u \otimes [l_u\ 0\ 0] \otimes q_u^*$$

$$p_{w,e} = q_f \otimes [l_f\ 0\ 0] \otimes q_f^*$$

Here, $\boldsymbol{q}_{<>}^{*}$ represents the conjugate of the quaternion. This means that the three imaginary portions of the quaternion are negative (i.e. $\boldsymbol{q}^{*} = [\boldsymbol{q_0}\ \boldsymbol{-q_1}\ \boldsymbol{-q_2}\ \boldsymbol{-q_3}]$). In addition, the cross product symbol represents quaternion multiplication. We define quaternion multiplication as follows:

$$q_1 = w_1 + x_1 i + y_1 j + z_1 k$$

$$q_2 = w_2 + x_2 i + y_2 j + z_2 k$$

$$\begin{aligned} q_1 \otimes q_2 \ = \ & (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) \\ & +(w_1 w_2 + x_1 x_2 + y_1 y_2 - z_1 z_2) i \\ & +(w_1 w_2 - x_1 x_2 + y_1 y_2 + z_1 z_2) j \\ & +(w_1 w_2 + x_1 x_2 - y_1 y_2 + z_1 z_2) k \end{aligned}$$

The equations above demonstrate how to calculate the elbow position relative to the shoulder and the wrist position relative to the elbow, but we ultimately wish to obtain the wrist position relative to the shoulder to send to the UR5. This is done by simply adding the respective components of $\boldsymbol{p}_{e,s}$ and $\boldsymbol{p}_{w,e}$. There is, however, one more step that must be completed to properly send coordinates to control the robot from the T-pose; they must be properly scaled/transformed to reflect the orientation and position of the "T-pose" of the robot. We determined this transformation experimentally to be $\boldsymbol{p}_{w,s\ robot} = [-\boldsymbol{p}_{w,s\,(1)},\ -\boldsymbol{p}_{w,s\,(2)},\ \boldsymbol{p}_{w,s\,(3)} + \boldsymbol{0.5}]$. The following overall position transform thus achieves the proper coordinates to send to the robot in controlling the UR5 directly from the arm tracker:

$$p_{w,s} = -p_{e,s(1)} - p_{w,e(1)},\ \ -p_{e,s(2)} - p_{w,e(2)},\ \ p_{e,s(3)} + p_{w,e(3)} + 0.50$$

2.4 User Arm Tracking Sensor Setup

After defining the math of the arm tracking, we construct the arm tracker as displayed below:

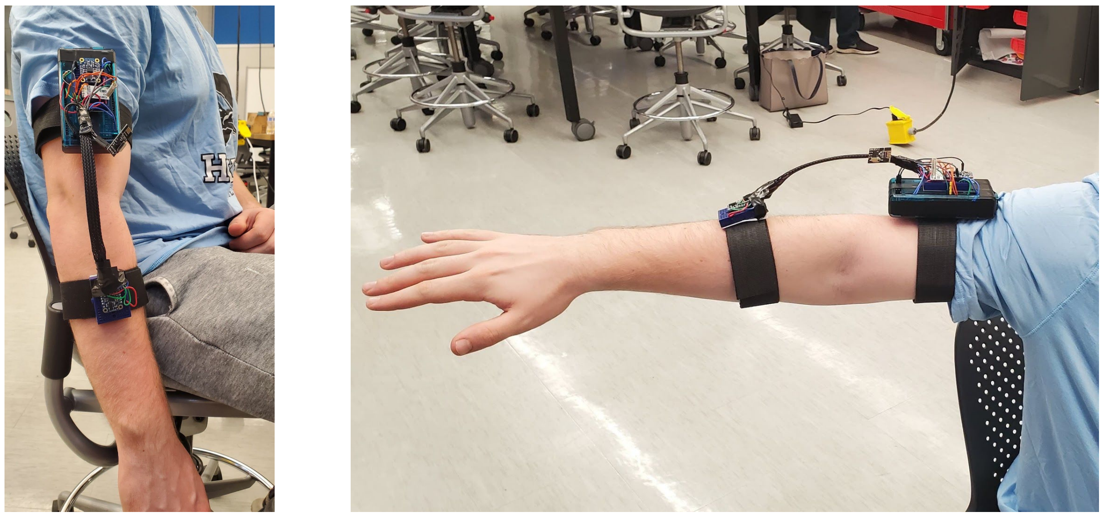


Figure 2: (a) Side view demonstrating arm tracking hardware. This device contains both BNO055 chips for encoding 3D spatial position of the elbow and wrist. (b) Front view of the arm tracker in the "T-pose" position.

As mentioned earlier, the sole sensor used in the arm tracker is the Adafruit BNO055 absolute orientation sensor. The BNO055 is an inertial measurement unit (IMU) with 9-DOF. This IMU provides sensor data from an accelerometer, gyroscope, and magnetometer in addition to many other helpful calculations. For determining arm tracking position, the most important measurement is obtaining the four-point quaternion vectors, $\boldsymbol{q_u}$ and $\boldsymbol{q_f}$, as defined above. These quaternions could be calculated manually from the raw accelerometer, gyroscope, and magnetometer data, however we found these calculations to be inconsistent and computationally heavy on the Arduino. Part of the reason we chose the BNO055 is that it has built-in quaternion calculation which reduces the complexity and error that comes with manual calculation, since all the computation is completed internally on the sensor's internal processor. We therefore only need to compute the wrist position using these quaternion output values as outlined in the previous section.

To begin assembly of the arm tracker, we will start with the BNO055. The pins for this sensor must be physically soldered to the chip. The pinout of this chip is outlined in the following figure:

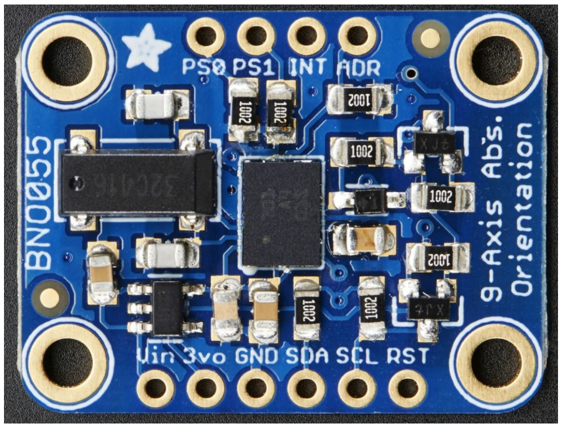


Figure 3: BNO055 pinout from Adafruit. Relevant pins for our project include Vin, GND, SDA, SCL, and ADR. [2]

The BNO055 is an I2C communication device, so wiring this sensor will involve using the SDA (I2C data) and SCL (I2C clock) pins. The GND pin will go to ground on the Arduino, the V_{in} pin will go to the 5V pin on the Arduino, and the SDA and SCL pins will go to the SDA and SCL pins on the Arduino. Once wired, in order to connect the sensor to the Arduino sketch, we must download and include the BNO055 library, which allows for the various built in calculations and library functions for the sensor. Once downloaded, this can be accessed in the Arduino IDE using the line:

```
#include <Adafruit_BNO055.h>
```

We can start reading from a single IMU by initializing a variable for the device. This is accomplished with the line:

```
Adafruit_BNO055 bno = Adafruit_BNO055(-1);
```

This sets an address for the device. By default, the address for a single BNO055 is 0x28. In our case, we used two BNO055s, so we will have to physically change the address of the two IMUs. This is accomplished using the ADR pin on the BNO055. By attaching the ADR pin on one IMU to ground, the address remains 0x28. By attaching the ADR pin on the second IMU to 3.3V on the Arduino, the address changes to 0x29. These can then be initialized in the Arduino sketch with the lines:

```
Adafruit_BNO055 bnoA = Adafruit_BNO055(-1, 0x29);
Adafruit_BNO055 bnoB = Adafruit_BNO055(-1, 0x28);
```

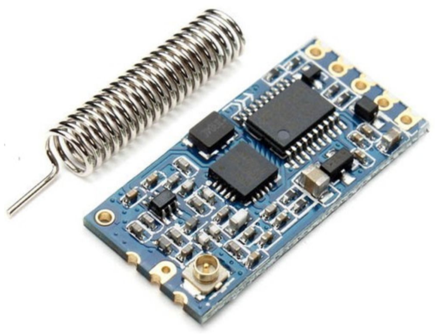
In this case, the bnoA and bnoB variables can be accessed separately throughout the code. The Arduino Mega has multiple SDA and SCL pins, so we do not need to change anything with wiring for including a second IMU. At this point, we have full connection to both the sensors within the sketch and can begin reading the quaternion values. As a reminder, the following command is used to read the quaternion values from the sensor:

```
imu::Quaternion quat = <BNO055_sensor_object>.getQuat();
```

For further details on code usage and sensor readings, refer to the ***Full_Arm_Tracking_UserEnd.ino*** file in the Github.

2.5 Arduino to Arduino Wireless Communication with the HC-12

Sample code for HC-12 wireless communication is located in the Github under the filename "HC_12_Setup_1.ino" and "HC_12_Setup_2.ino". These files should be run on seperate Serial ports on separate Arduinos to establish communication. ***These files are only sample code for implementing wireless communication. These commands are already integrated into Full_Arm_Tracking_UserEnd.ino and Full_Arm_Tracking_UR5End.ino to connect the user arm tracker Arduino with the robot control PC Arduino.***



Now that we have developed the arm tracker and implemented the math for computing wrist position, we need to set up wireless communication for remote operation of the UR5. For this project, we needed to communicate between the user's arm tracker and the computer controlling the UR5 simulation. This allows us to wirelessly transmit the wrist coordinates to control the UR5. When considering wireless communication options, we wanted a chip that can perform small to medium range communication without being overly costly. An ideal sensor for this use case is the HC-12 Wireless Communication Module. The HC-12 has a transmitting power from -1dBm to 20dbm, a receiving sensitivity of -117dBm to -100dBm, and a working frequency band from 433.4MHz to 472.0MHz [3]. To set up a two way wireless connection, all you need is two HC-12 chips and a seperate Arduino for each chip. For our implementation, one HC-12 was located on the arm tracker with the Arduino Mega, and the second HC-12 was located with the laptop which controls the UR5, connected to an Arduino Uno.

The first step in using the HC-12 is to solder on the necessary pieces. The pins and the wire coil must both be soldered to the HC-12 chip on opposite sides. Once soldered, the HC-12 can be attached to an Arduino with the following pinout connection:

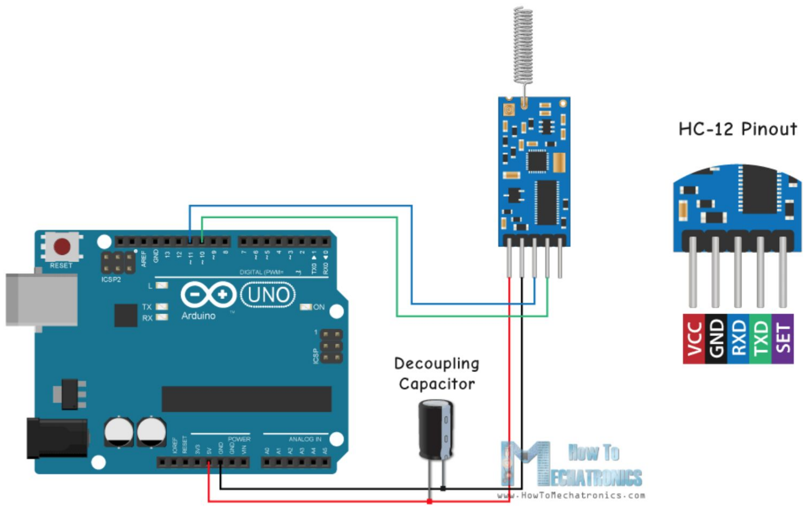


Figure 4: Wiring diagram for using the HC-12(left). Full pinout of the HC-12 (right). [4]

The V_{CC} pin attaches to 5V on the Arduino, the GND pin connects to ground on the Arduino, and the RXD and TXD pins attach to digital output pulse width modulation (PWM) pins on the Arduino. To provide stability to the power supply of the module, we attach a decoupling capacitor between the V_{CC} and GND pins. Any capacitor value between 0.1μF and 10μF should be sufficient. In our case, we used a 1μF capacitor.

When two functioning HC-12s are connected wirelessly, a few simple commands can be used to send data to and from the two Arduinos. This provides an easy method for transmitting simple data between Arduinos such as positional coordinates calculated from the IMU sensor readings. For our project, we use the HC-12 to communicate the wrist position coordinates from the arm tracker Arduino to the Arduino connected to the UR5 control PC. If you are attempting to communicate with both Arduinos connected to the same computer, it is important to open up the Arduino IDE twice. In each window, specify the Serial Port for each Adruino. This method allows you to open up two Serial Monitors and communicate between each, whereas otherwise it would be impossible.

Once connection between HC-12s is established, the code can be integrated into the two arduinos. The following commands can be used to send and receive data to and from the two HC-12s:

Sending:

```
HC12.print(data);                   // print the data to the HC-12

while (HC12.available()) {          // If HC-12 has data
        Serial.write(HC12.read());  // Send the data to Serial monitor
}
while (Serial.available()) {        // If Serial monitor has data
        HC12.write(Serial.read());  // Send that data to HC-12
}
```

Receiving:

```
HC12.listen();

while (HC12.available()) {              // If HC-12 has data
        data = HC12.read();             // Read data

        // do stuff with data
}
```

For further details of implementation of HC-12 in the code, see the ***Full_Arm_Tracking_UserEnd.ino*** and ***Full_Arm_Tracking_UR5End.ino*** files on the Github.

2.6 Arduino to MATLAB Communication

Sample code for communicating via the Serial line is located in the Github under the filename "MATLAB_communication.m". ***This file is only sample code for implementing Arduino to MATLAB communication. These commands are already integrated into the main MATLAB communication file, UR5_Control.m.***

Up to this point, we have demonstrated how to extract and calculate the user's wrist position and send it wirelessly from one Arduino to another. However, as will be demonstrated below, we control the UR5 simulation from a MATLAB script, ***UR5_Control.m*** (see ***Connecting to and Controlling the UR5 from MATLAB***). Therefore, we must have a method to send the calculated wrist position coordinates from the Arduino receiving the HC-12 data to the MATLAB script that connects to the robot simulation. This can be done by initializing Serial Communication through the base MATLAB file.

In connecting an Arduino to MATLAB, you first need to know the Serial Port connection name for the connected Arduino. USB Serial Port connection names vary between computers, so a useful tool is the "instrfind" command in MATLAB. This command will display all available and in-use ports for the given computer. Once you identify the port that is being used by the Arduino, you can establish a connection to MATLAB. For us this looked like the following:

```
x=serial('/dev/ttyACM0','BAUD', 9600);
fopen(x);
```

In this case, our port name was '/dev/ttyACM0', a common port name for Linux systems.

<u>NOTE:</u> If you are ever having trouble connecting to your Arduino via usb, first find this port name and then run the following command in the terminal (replacing "/dev/ttyACM0" with your own port name):

```
chmod a+rw /dev/ttyACM0
```

Once you open the connection in MATLAB, it is important to make sure you close and delete the port variable at the end of your script:

```
fclose(x)
delete(x)
```

If the port is not closed in this fashion, this can become an issue when continually reading in values from the Arduino. If the port is constantly open and the user hard-stops the code, the port will still be open. This will make the code unusable until that variable is closed and deleted.

Once data is being sent through the opened port, it can be read with the fscanf command:

```
fscanf(x);              // Read in data from the current Serial port variable
```

In the case of our robot coordinates, we will send final end-effector (wrist position) coordinates as comma-separated values that can be easily identified using this above command. Coordinates will come in the format:

```
x.xx,y.yy,z.zz!         // Example:  0.34,-0.45,0.24!
```

Here, each value is a spatial position on the coordinate axis (x, y, and z respectively). The user can then differentiate between the coordinates by iterating through the string and identifying commas. The exclamation point at the end is just a useful addition to know when you are at the end of a coordinate send command. See ***UR5_Control.m*** for how to extract the individual values from this coordinate string, and see ***Full_Arm_Tracking_UserEnd.ino*** and ***Full_Arm_Tracking_UserEnd.ino*** for how to form this coordinate string and print it to the Serial line (all of these files are in the Github).

2.7 Connecting to and Controlling the UR5 from MATLAB

Our code for controlling the UR5 is integrated into the main MATLAB file, UR5_Control.m.

Now that we've established connection between MATLAB and the Arduino receiving the wrist coordinates from the HC-12, we need to start using these coordinates to actually control the robot simulation. The base code for connecting to the simulation from MATLAB is in the ***ur5_interface.m*** file on Github. This is a class definition that defines the interaction between MATLAB and the Gazebo interface with the simulation. Connection is thus established by declaring an object of this class within your own script:

```
my_ur5 = ur5_interface();
```

There are a few key functions with this class that become useful in using the robot. Namely, the following functions we found useful in our prototyping and development:

```
my_ur5.move_joints(<6x1 joints vector>, <time of movement in seconds (i.e. related to speed)>);
```

- This method moves the robot to the desired joint state. Here, the 6x1 joints vector represents the desired states of the 6 individual joints of the robot, each value mapping between 0 and 2π radians. These six joints are, in order, ***shoulder_pan_joint***, ***shoulder_lift_joint***, ***elbow_joint***, ***wrist_1_joint***, ***wrist_2_joint***, and ***wrist_3_joint***. Alternatively, there are a few preset joint locations that can be used. For instance, if you want to command the robot into the "home" position, you can replace the joints vector input argument with the predefined ***my_ur5.home*** class object (which holds the joint vector for the home position of the robot). The other predefined position is ***my_ur5.DH_base***. Both of these joint states are defined in the ***ur5_interface.m*** file. As mentioned earlier, the joint space vector for the "T-pose" is [0 -π/2 -π/2 -π/2 π/2 0].

my_ur5.get_current_joints();

- This method returns, as a 6x1 vector, the current state of each of the 6 aforementioned joints, in radians.

my_ur5.get_current_transformation();

- This method returns the 4x4 transformation matrix of the current state of the robot, as mapped from the base of the robot to the end-effector of the robot.

While the ***move_joints*** method was useful for us in moving the robot, it is limited as it required knowledge of where we want the 6 joints states to be in controlling the robot. In our project, we extract 3D world coordinates of the user's wrist (see above, ***User Arm Tracking Development and Math)***, so we wanted to use that as input to control the robot. This involved the use of ***inverse kinematics***. This is the process of converting world coordinates to joint angles. It computes the joint angles necessary to achieve the desired 3D coordinates, based on the current state of the robot. In our codebase, this is achieved via the ***ur5InvKinControl.m*** and ***ur5InvKin.m*** files in the Github. The steps for moving the robot via inverse kinematics are as follows:

1. Read in the desired 3D world coordinates as received by the HC-12 from the arm tracker

2. Convert the 3D coordinates into a 4x4 transformation matrix
    a. The upper left 3x3 is a 3D rotation matrix mapping to the orientation of the handpiece
    b. The upper right 3x1 column is the translation vector associated with the (x, y, z) coordinates of the user's wrist position
    c. The lower 1x4 row is simply [0, 0, 0, 1]
    d. See ***UR5_Control.m*** for more details on this

3. Send the transformation matrix to ***ur5InvKinControl***
    a. ur5InvKinControl(<transformation matrix>, my_ur5, <time of movement in seconds>);

4. Within ***ur5InvKinControl***, the ***ur5InvKin*** script is called upon to convert the transformation matrix to a set of optimal joint states. It then finds the optimal joint state for the given wrist position coordinates that requires the smallest change in joint state (minimum distance to current position) and sends those joint states to the ***move_joints()*** method of the ur5 class object to move the robot to the desired coordinate position.

2.8 Telecommunication Data Flow Summary

The following diagram summarizes the overall flow of data from extracting the sensor readings from the IMUs to moving the UR5 simulation robot (Fig. 5).

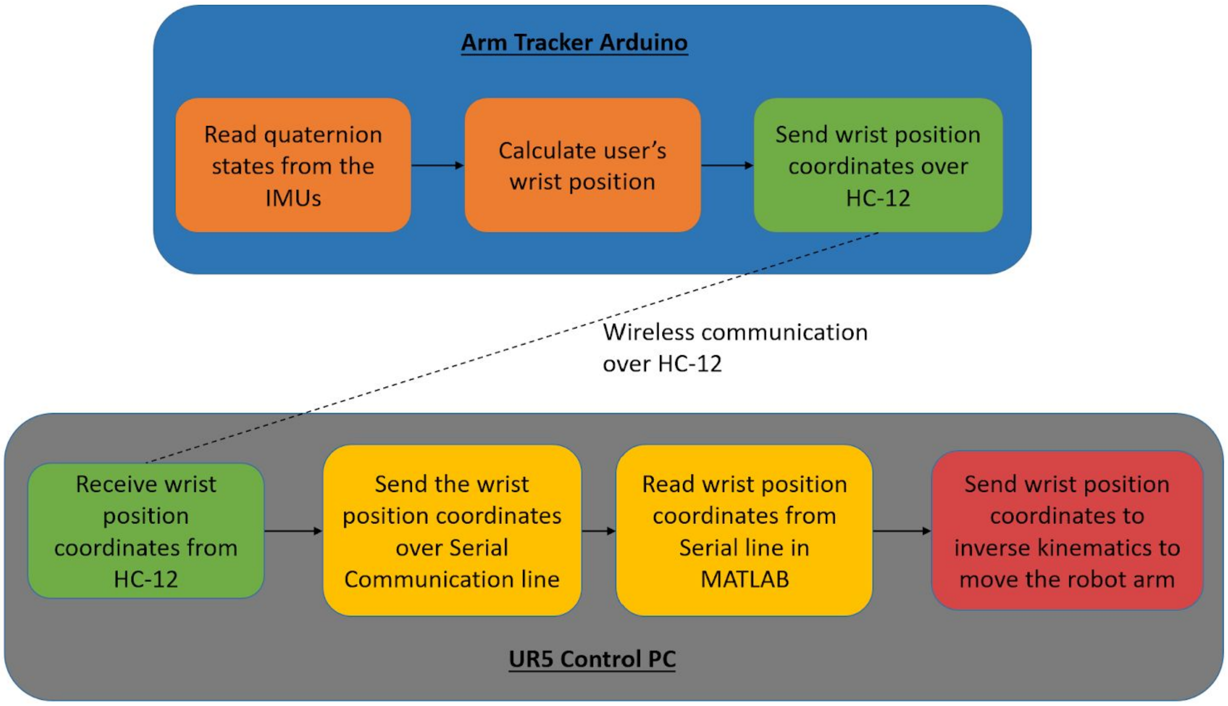

Arm Tracker Arduino
Read quaternion states from the IMUs
Calculate user's wrist position
Send wrist position coordinates over HC-12
Wireless communication over HC-12
Receive wrist position coordinates from HC-12
Send the wrist position coordinates over Serial Communication line
Read wrist position coordinates from Serial line in MATLAB
Send wrist position coordinates to inverse kinematics to move the robot arm
UR5 Control PC

Figure 5: Flow of Data from the sensor readings

3. Development of Palpating UR5 End-Effectors

The UR5 robotic arm is highly modular, and many different end effectors can be attached to the robot. In this section of the work, two end effectors are designed and developed to perform simple palpation tasks. Remote, teleoperated, palpation is highly desirable for the treatment and monitoring of many conditions such as ebola and other highly infectious diseases. The first end effector developed we call the *Palpating Puck*. The puck represents the most simple implementation of a palpating end effector, with no actuation and a straight forward design. The second end effector developed was the *Helping Hand*. This hand is the primary end effector that was developed, and is shaped like a human hand. Moreover, the hand can be actuated (opened/closed) and has soft silicone finger tips with integrated tactile sensors.

The stl files for 3-D printed parts can all be found in the attached Github link below. Code for controlling the *Helping Hand*/*Palpating Puck* as well as the graphical user interface (GUI) for visualization of the sensor data can also be found in the attached Github repository in the master directory.

https://github.com/Nick7244/Honors-Instrumentation/tree/master/3D_Models

3.1 Palpating puck

The *Palpating Puck* is the first end-effector for the UR5 robot that was developed in this work (Fig. 6). This puck is designed to obtain pressure and temperature readings upon direct contact with a wound, which have been shown to be reliable metrics for quantifying wound severity [5]. The UR5 robot has 6-DOF control, so the position of the *Palpating Puck* can be manipulated through the arm tracking outlined section 2, ***Robot Arm Teleoperation***. The puck is made of soft silicone, and has a 9 taxel, 3x3 fabric based tactile pressure sensor; as well as two thermistors to measure temperature. The puck can be easily attached to the UR5 through a mounting piece and features a hexagonal recess to ensure an easy fit. A microcontroller (Arduino UNO) was programmed to interface with/record measurements from the tactile sensor and the thermistors. A GUI was also developed using python to visualize the temperature and pressure data across the puck, in real time.

The *Palpating Puck* is composed of four distinct layers:

1. UR5 mounting base piece
2. Silicone body
3. Flexible, fabric-based tactile sensing grid
4. Silicone cap with embedded thermistors

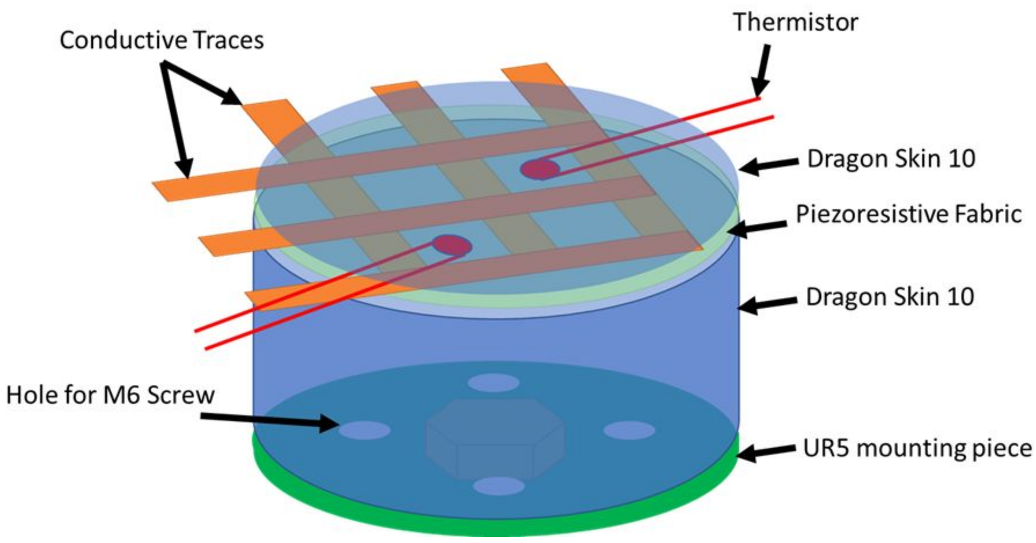


Figure 6: Diagram showing different layers and components of the *Palpating Puck.*

To begin the construction of this device, we started with the UR5 connecting base piece. To build this, a CAD file representing the UR5 end piece was first downloaded from:

https://automationdistribution.com/universal-robots-ur3-ur5-ur10-arm-cad-drawings/.

A mounting piece was then designed using AutoCAD 2019, and 3D printed using a MakerBot® Replicator™ 2 (Fig. 7). The goal of these parts is to provide a secure and safe method for attaching the puck to the UR5 that remains reliably attached, even after intense UR5 movement.

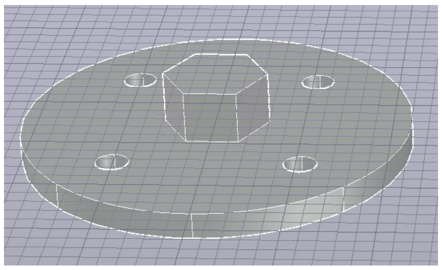


Figure 7: 3D Rendering of UR5 Mounting Base Plate for *Palpating Puck.*

To fabricate the body of the puck, a silicone mold was designed and 3D printed (Fig. 8). Silicone is the primary material used for soft robotics research, as it provides the required texture needed for medical practice. In our case, Dragon Skin™ 10 Medium was used, and cast inside the mold with the following procedure:

1. Mix equal parts of Dragon Skin™ 10 Medium A and B in a cup for 3 minutes by hand.
2. Pour the mixture into the mold and leave overnight to set.

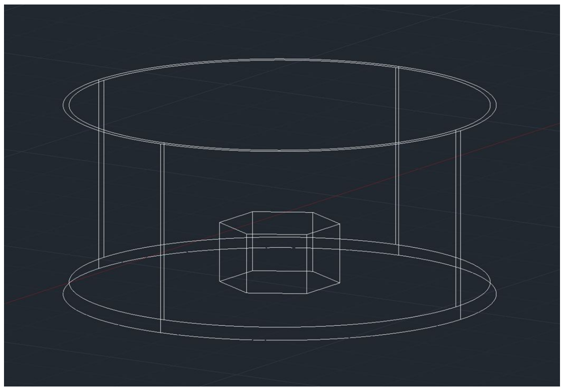


Figure 8: 3D Rendering of the Mold for the *Palpating Puck.*

The next step in the design process is to fabricate the tactile sensor. A procedure was followed as described by Osborn *et al* [6], and outlined at the website below:

http://jhuneurolab.wiki-site.com/index.php/Touch_Sensors

The tactile sensor being built is composed of flexible conductive fabric, fusible interfacing, and a layer of piezoresistive material. The sensor is cut and composed in a grid shape that is demonstrated in the above tutorial. Ideally, this method will provide a grid interpretation of force inputs from the end effector. This method can be used to fabricate sensory arrays in a variety of sizes and grid arrays. In our case, we designed a 3x3 sensor grid spanning the width of the base mold.

Next, to fabricate the top silicone cap, an additional silicone mold was 3D printed (Fig. 9), and more Dragon Skin™ silicone was cast in the mold. This top layer of silicone has a special recess incorporated into the design to allow for the integration of thermistors into the puck.

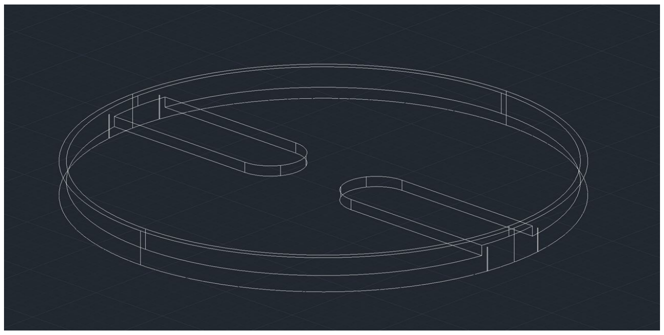


Figure 9: 3D Rendering of the Mold for the Silicone Cap with Space for Thermistors.

To interface with the fabricated tactile sensor and obtain readings based on pressure, a series of voltage divider circuits are built to interact with each piezoresistive grid element (Fig. 10). Each "cross" intersects with a force sensor. These intersection elements will be denoted as "taxels".

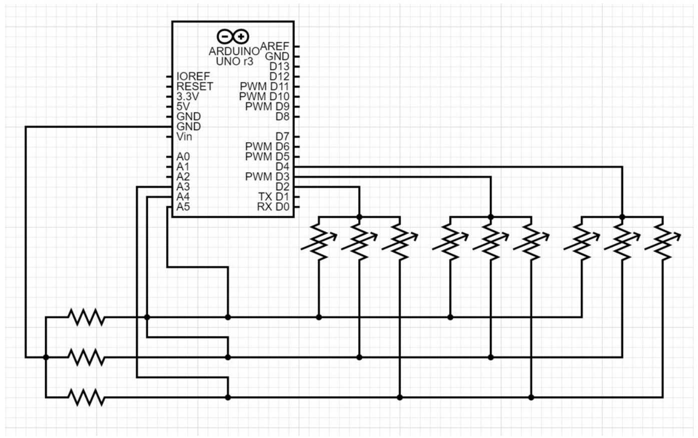


Figure 10: Drawing showing voltage divider circuits spanning a 3x3 fabric force sensor.

In order to obtain readings from these sensors, we must incrementally provide power to each voltage divider. The algorithm used to evaluate the 9 taxels went as follows:

```
For i in range(2,5):
        Set Digital pin (i) = HIGH (5V)
        Read voltage at A3, A4, and A5.
        Set Digital pin (i) = LOW(0V)
```

The completed *Palpating Puck*, and its different layers can be seen below (Fig. 11).

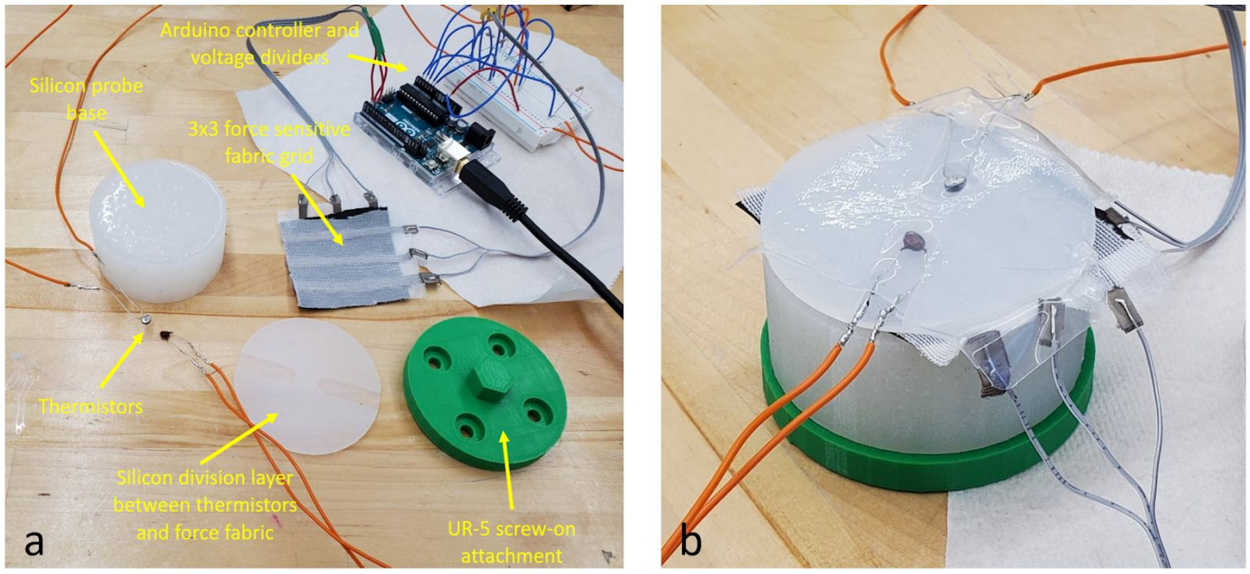

Arduino controller and voltage dividers
Silicon probe base
3x3 force sensitive fabric grid
Thermistors
Silicon division layer between thermistors and force fabric
UR-5 screw-on attachment
a
b

Figure 11: Overall design and development of *Palpating Puck*. (a) Disassembled and labeled soft-robotic probe with force and temperature sensors. (b) Assembled soft-robotic probe with force and temperature sensors.

3.2 *Palpating Puck GUI Design*

Lastly, to visualize and interpret the pressure and temperature data across the puck, a Graphic User Interface (GUI) was developed in Python (Fig. 12). The connected Arduino sends the voltage readings through the serial port and the Python script reads these values from the serial line and updates the GUI. The GUI was built using PySimpleGUI, a straight-forward library geared towards GUI development.

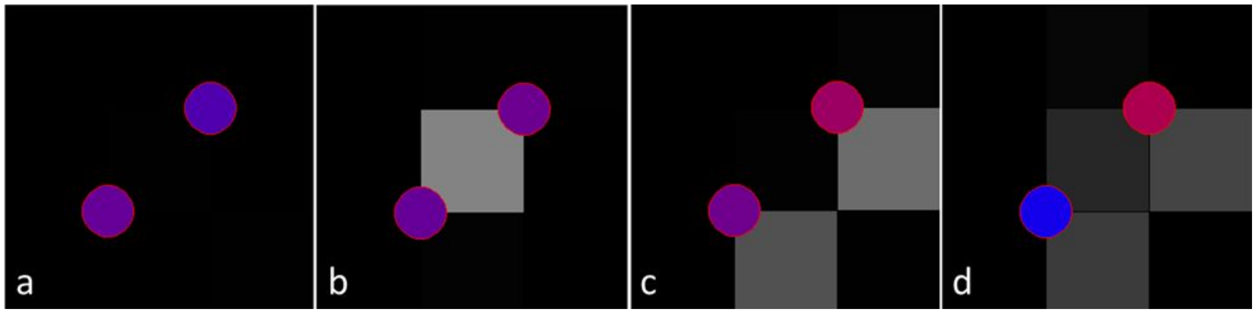


Figure 12: GUI showing forces and temperature across the area of fabric-based sensors. (a) Baseline sensor readings showing no pressure, and ambient temperature. (b) Sharp force in the center of probe. (c) Two forces on the force sensors with a slight temperature increase. (d) Dispersive force with different temperature values across the probe.

The GUI modulates color to represent pressure and temperature values read in by the various sensing elements. The pressure sensing utilizes individual squares to represent the taxels which are color coded from black (no force) to white (max force). The thermistor uses color coded circles blue (minimum temperature) to red (max temperature). Each of these color codes were calibrated specifically based on minimum and maximum values. The code for this can be found on the Github in ***PalpatingPuckGUI.py***.

3.3 *Helping hand*

The *Helping Hand* is the main UR5 end-effector developed in this work. The hand is based on a design developed by HendoJ [7]. This hand was chosen because of its skeleton shape, and its simple actuation mechanism. The skeleton shape is important as it allows for the development of silicone 'finger caps' that emulate human fingers (soft exterior skin + hard inner bone). The *Helping Hand*'s actuation is similar to popular e-NABLE prosthetics, with a series of strings and elastic (fishing line and rubber bands). To reproducibly actuate the hand, a DC motor is utilized. An Arduino is programmed to control the motor, and a custom-made spool is attached to the motor to tighten/loosen hand strings and actuate the hand.

Overall, the *Helping Hand* is composed of five main components:

1. A UR5 connecting base piece
2. 3D printed hand skeleton
3. A DC motor and custom-made spool
4. Silicone finger-caps
5. Flexible, fabric-based tactile sensing grids

Additionally, a glove is developed to tele-actuate the *Helping Hand*, and to provide haptic feedback to the user (see below, ***Glove with Flex Sensor and Haptic Feedback***).

To build the UR5 connecting piece for the *Helping Hand*, the connecting piece for the *Palpating Puck* was modified to remove the hexagonal connector, and was then 3D printed (Fig. 13).

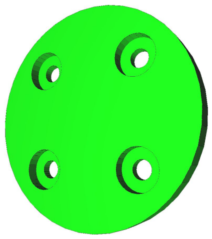


Figure 13: 3D Rendering of UR5 Mounting Base Plate for *Helping Hand*.

To fabricate the 3D printed hand, the downloaded CAD file was converted to STL and 3D printed using a Makerbot Replicator 2. To assemble the 22 pieces, the following tutorial was closely followed:

https://www.instructables.com/id/3D-Printed-Bionic-Hand-Skeleton/

This model can be seen in Fig. 14a. To actuate the hand, instead of using bungee cord and nylon string (as recommended by the tutorial), fishing line and rubber bands were used. The actuation mechanism relies on an elastic force provided by the rubberband that consistently pulls the fingers into the closed position. This means the resting hand position is closed. A DC motor is used to pull nylon strings around a custom designed spool to pull back the finger against the elastic force from the rubberband, thus opening the hand.

To mount the DC motor to the base piece, a mounting bracket was designed and 3D printed (Fig. 14b). An h-bridge chip is utilized to switch the rotation direction of the DC motor on demand, thus allowing for control over opening and closing of the hand. An Arduino Mega is used due to the need for more digital pins in controlling the motor and H-bridge as well as the taxel voltage dividers. We also switched from a solderless breadboard to a protoboard to allow for hand movement without risk of loose connections.

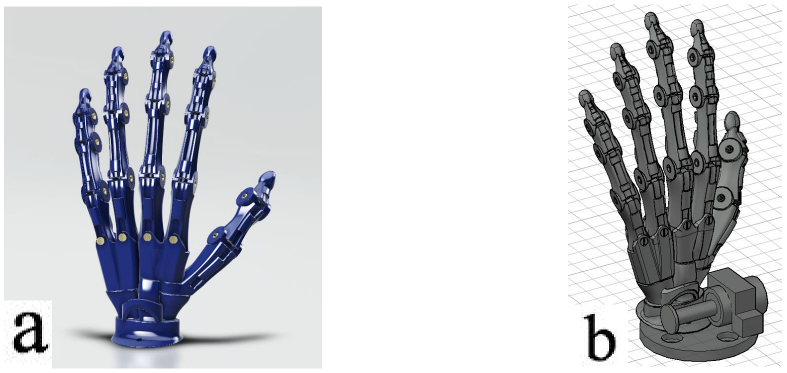


Figure 14: (a) 3D Rendering of 3D Printed Hand. (b) 3D Rendering of the *Helping Hand* system. DC motor, mounting bracket, and custom-made spool can be seen mounted to the base piece.

To fabricate the silicone finger-tip caps, three molds were 3D printed (Fig. 15). The fingertips of the 3D printed hand were then dipped into Dragon Skin 10, and left to set overnight in the mold, allowing the silicone to mold around the 'bone'.

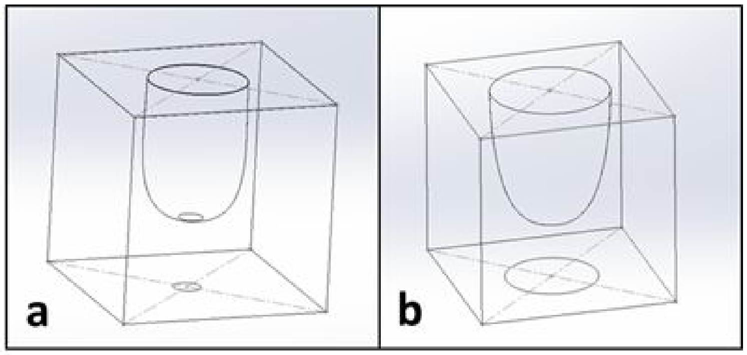


Figure 15: (a) Silicone finger mold. (b) Silicone thumb mold.

The flexible, fabric-based tactile sensing grids for the fingertips were developed using the same procedure as the *Palpating Puck* but smaller and with thinner conductive traces (Fig. 16). Special consideration was made to develop a 3x2 design so that the fingertip had one central taxel for direct palpation and two peripheral taxels to read in the indirect palpation.

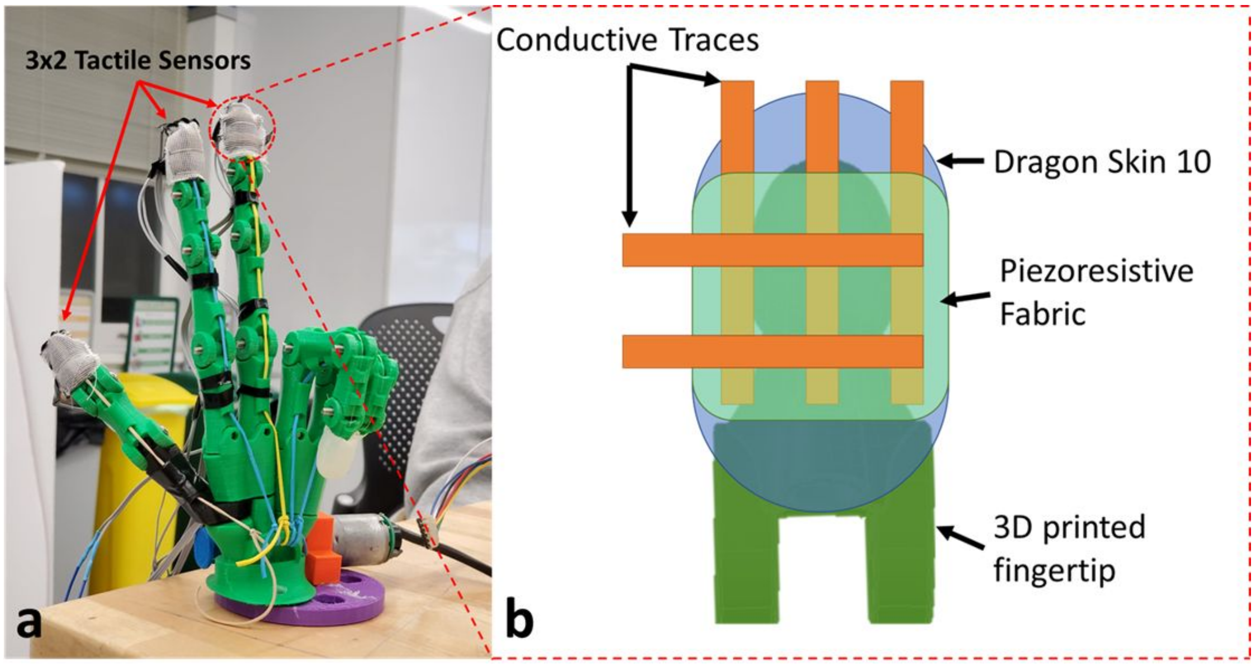


Figure 16: (a) Picture of the *Helping Hand* with attached tactile sensors on soft fingertips. (b) Zoomed in schematic drawing of fingertip, showing the conductive traces, piezoresistive fabric, silicone finger cap, and the 3D printed 'bone'.

3.4 *Helping Hand GUI Design*

The GUI design is integrated with the output of the Arduino from the Serial port (Fig. 17). As described previously, the Arduino code outputs the voltage from the flexible touch sensor. This GUI differs from the one presented earlier as the tiled visual interface is overlaid with a picture of the hand for intuitive visual representation of the force experienced on the fingertips. The outputs from the Arduino (from the sensor) are read into a python program and utilized to create a color scale. We chose the color scale black to white indicating no force, to a maximum force. The values are scaled according to inputted maximum and minimum from preliminary testing but at times, require recalibration.

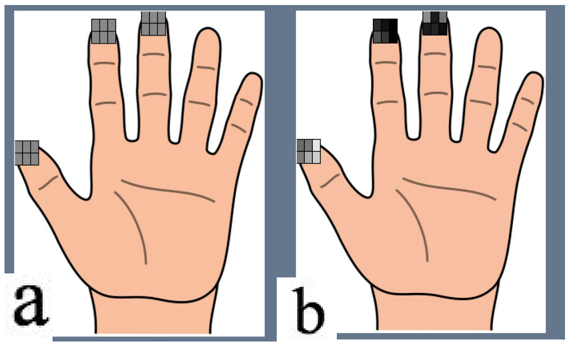


Figure 17: Developed GUI showing 6 taxels (3x2 configuration) on each fingertip. Taxels change color from black to white as pressure and force increases. (a) Shows medium pressure on all fingers. (b) Shows high pressure on the thumb., and small pressures on the pointer and middle fingers.

PySimpleGUI was a useful resource to create this GUI. Each finger has a 6 taxel, 3x2 tactile sensor. These taxels are represented by grey boxes on fingertips, with each box representing one taxel. The voltage values are read into the python program in the same manner as the *Palpating Puck*, through the serial port.

3.5 *Glove with Flex Sensor and Haptic Feedback*

To actuate the *Helping Hand*, we must design a device on the user end to input hand closed/open. In order to do this, we decided to develop a "glove" that fits over the user's hand and uses flex sensors to determine finger movement (open/close). In addition, to relay haptic feedback from the *Helping Hand* back to the user, we included vibration motors on the fingertips of this glove. The development of this glove consists of three specific components:

1. Flex Sensor
2. Vibrating Motor Disks
3. 2 Way RF-Connection

The hand tracker glove utilizes two components to relay information to the hand. The ***flex sensor*** provides information to the actuated hand on if the glove hand is opened or closed. The flex sensor is a simple variable resistor that increases in resistance as the sensor is bent. The flex sensor used in this case was the Spectra Symbol - 4.5" Flex Sensor (Fig. 18).

Figure 18: Picture of Spectra Symbol - 4.5" Flex Sensor [8].

The circuit for the flex sensor utilizes a voltage divider circuit (Fig. 19). The circuit diagram and Arduino connection created by SparkFun are included in this figure. The code for this system is included in the Github in the ***Full_Arm_Tracking_UserEnd.ino*** file. This sensor can then be attached to the glove to bend with its associated finger. By scaling the changing resistance of the sensor, we can obtain the finger angle of the user which will be relayed back to the *Helping Hand* for actuation control via the DC motor.

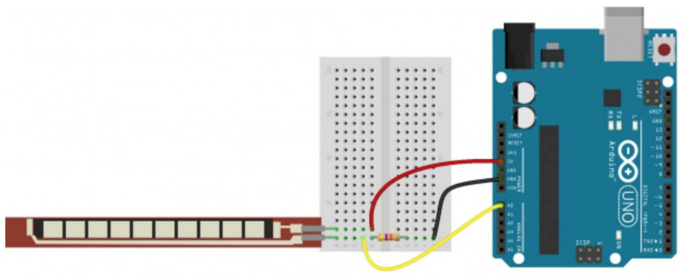


Figure 19: Flex Sensor Arduino connection (Red wire to the 5V pin and the yellow wire to the A0 pin) [9].

The next component utilized for the glove were the ***vibration sensors*** that provide haptic feedback from the force sensors on the actuated hand (Fig. 20). For this, we utilized the force sensor output to provide one of three levels: soft, medium and hard forces. To determine which category the force was placed in, we designated a threshold from the average force for each of the fingers. These values were determined upon inspection of the force values outputted on the serial monitor. The vibration intensity of these motors is implemented via the PWM value that is inputted based on the threshold case of the force sensor output. The Arduino connection to the motor is exemplified below [10], while the code for this system is in the ***Full_Arm_Tracking_UserEnd.ino*** and ***Helping_Hand.m*** files.

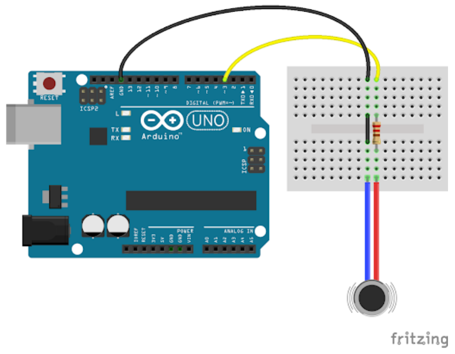
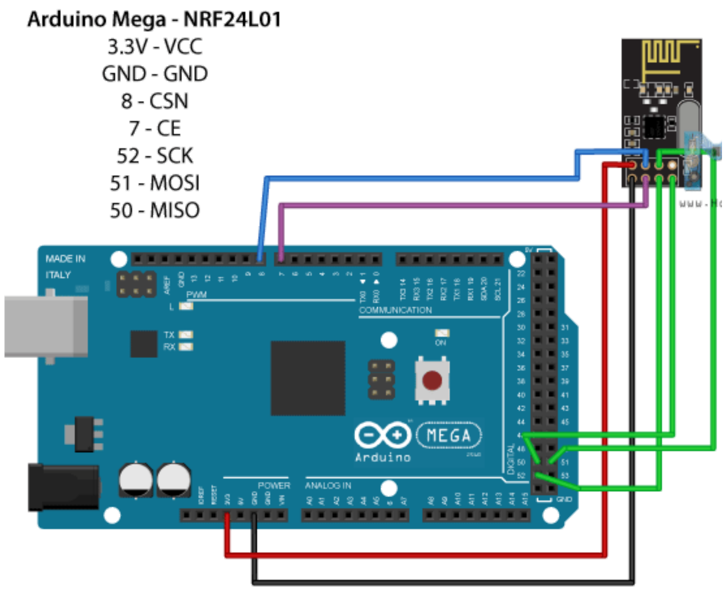


Figure 20: Vibrating Disk Arduino connection [10].

The 2 way communication between the user's glove and the *Helping Hand* is handled by ***two single chip radio-frequency (RF) transceivers***, specifically the nRF24L01 (Fig. 21). There is a transceiver connected to the *Helping Hand* as well as the user's arm tracker which includes the haptic glove. These transceivers open a communication pipe that can both write and receive simple messages, but not at the same time. This creates an issue since we need to both send actuation information from the glove to the hand while also sending force feedback information from the hand to the glove. Because of this, in the Arduino code we alternate between two states, sending and receiving, every 50msec. We implement this from the glove side by sending an 'o' or 'c' character to our actuated hand in order to open or close, based on the state of the flex sensor. This communication also receives data from the silicone force sensors on the fingertips in the form of "o", "s", "m", and "h" which indicate what threshold value the force sensor has surpassed and what strength to vibrate the motors on the glove. The code for this system is included in the Github in the ***Full_Arm_Tracking_UserEnd.ino*** and ***Helping_Hand.m*** files. The connection of the RF transceivers to the Arduino is as follows:



Arduino Mega	NRF24L01
3.3V	VCC
GND	GND
8	CSN
7	CE
52	SCK
51	MOSI
50	MISO

Figure 21: RF Arduino Mega connection [11].

4. Final Results and Summary

The following diagrams demonstrate the completed system working from arm tracking to robot motion, including the usage of the *Helping Hand*.

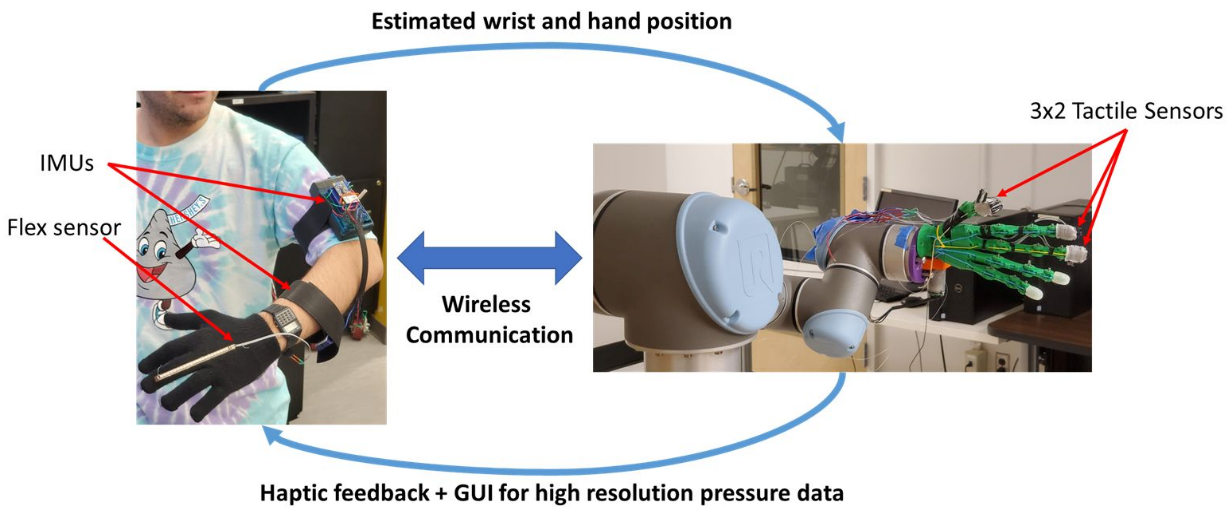

Estimated wrist and hand position
IMUs
Flex sensor
3x2 Tactile Sensors
Wireless Communication
Haptic feedback + GUI for high resolution pressure data

Figure 22: Integration of the entire system.

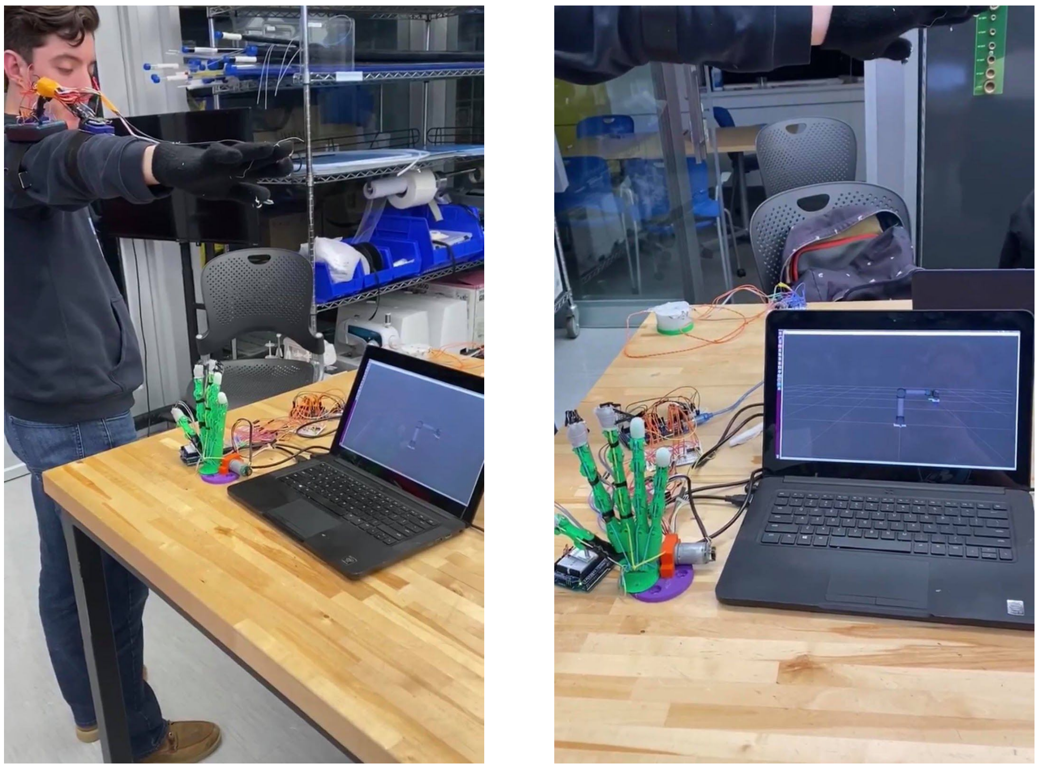


Figure 23: Arm and robot in T-pose, hand and glove open.

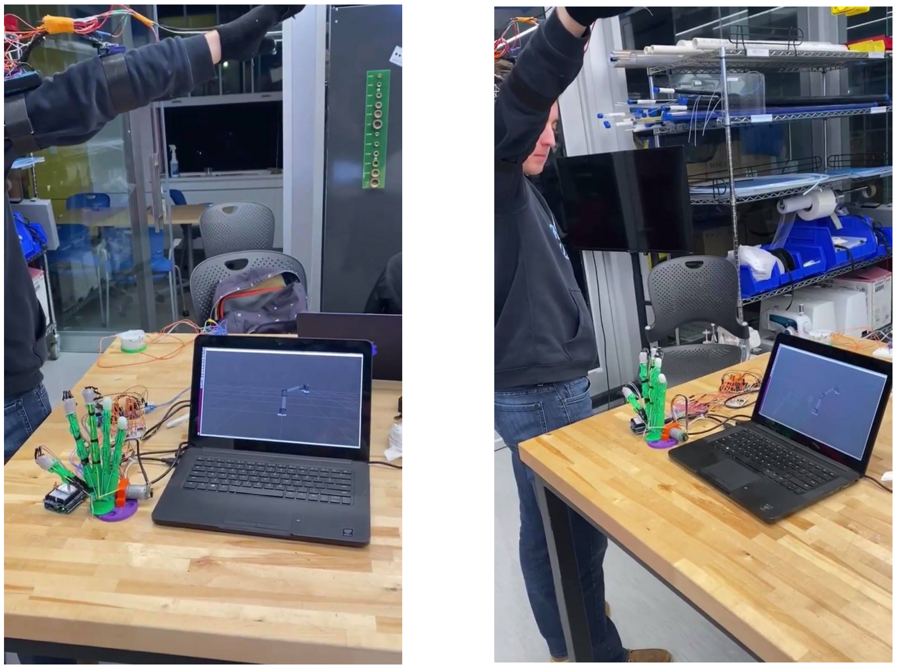


Figure 24: Arm and robot in upward position, hand and glove open.

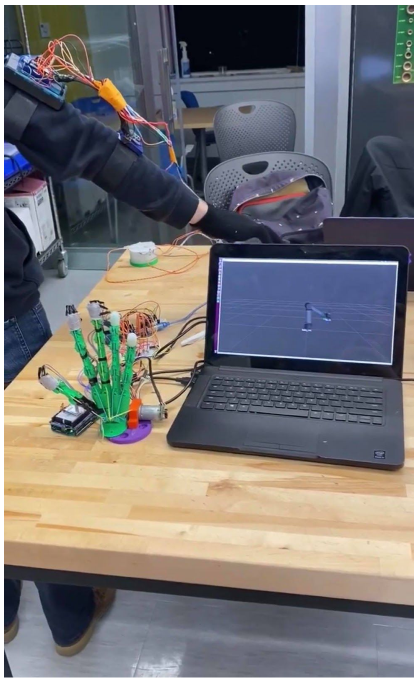


Figure 25: Arm and robot in downward position, hand and glove open.

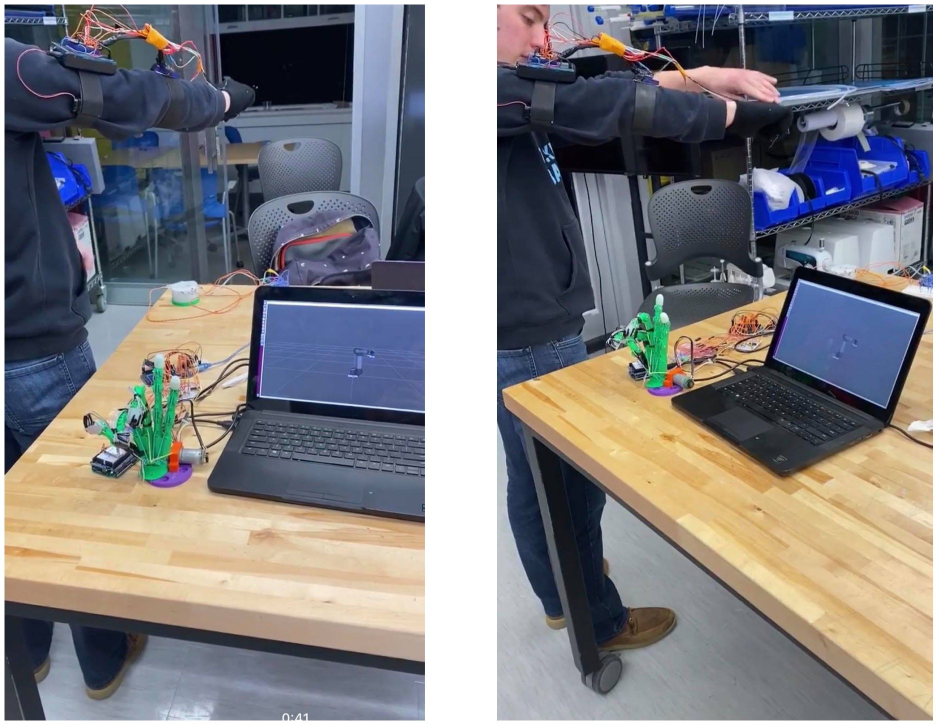


Figure 26: Arm and robot in forward position, hand and glove closed.

This concludes our work on this project. We hope this serves as a useful educational guide in building and developing a similar system that can be tailored towards actual medical applications. If you have any questions, feel free to reach out to us at the contacts below:

Nicholas Maritato - nmaritato14@gmail.com
Michael Ainsworth - michael.j.ainsworth@gmail.com
Dante Navarro - dantenavarro1234@gmail.com
Arik Slepyan - slepyana@gmail.com

5. References

[1] T. Taunyazov, B. Omarali and A. Shintemirov, "A novel low-cost 4-DOF wireless human arm motion tracker," 2016 6th IEEE International Conference on Biomedical Robotics and Biomechatronics (BioRob), Singapore, 2016, pp. 157-162. doi: 10.1109/BIOROB.2016.7523615

[2] Townsend, K. (n.d.). Adafruit BNO055 Absolute Orientation Sensor. Retrieved from https://learn.adafruit.com/adafruit-bno055-absolute-orientation-sensor/pinouts

[3] Dejan, Zeeshan, Nedelkovski, D., Nedelkovski, D., Gill, Dejan, … Antoine. (2019, December 15). Arduino and HC-12 Long Range Wireless Communication Module. Retrieved from https://howtomechatronics.com/tutorials/arduino/arduino-and-hc-12-long-range-wireless-communication-module/

[4] Electrical Engineering & Electronics Community. (n.d.). Retrieved from https://www.allaboutcircuits.com/

[5] Bellare, Anuj, et al. "Development of Tensile Strength Methodology for Murine Skin Wound Healing." MethodsX, vol. 5, 2018, pp. 337–344.

[6] Osborn, L. E. (2014). Reflex: A Closed-Loop Tactile Feedback System for Use in Upper Limb Prosthesis Grip Control. *MSE Graduate Thesis*. Retrieved from https://jscholarship.library.jhu.edu/handle/1774.2/38045

[7] HendoJ. (n.d.). Retrieved from https://www.instructables.com/member/HendoJ/

[8] Flex Sensor 4.5". (n.d.). Retrieved from https://www.sparkfun.com/products/8606

[9] Flex Sensor Hookup Guide. (n.d.). Retrieved from https://learn.sparkfun.com/tutorials/flex-sensor-hookup-guide/all

[10] Vibrating Disk Hookup Guide (n.d.). Retrieved from http://www.limulo.net/website/coding/physical-computing/vibration-motor-test.html

[11] Arduino Wireless Communication - NRF24L01 Tutorial. (2020, February 24). Retrieved from https://howtomechatronics.com/tutorials/arduino/arduino-wireless-communication-nrf24l01-tutorial/