Summer 2023

Incorporating Novel Sensors for Reading Human Health State and Motion Intent into Real-Time Computing Systems

Adam Sawyer
Missouri University of Science and Technology

Department:

INCORPORATING NOVEL SENSORS FOR READING HUMAN HEALTH STATE AND MOTION INTENT INTO REAL-TIME COMPUTING SYSTEMS

by

ADAM RYAN SAWYER

A THESIS

Presented to the Graduate Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

2023

Approved by:

Chenglin Wu, Advisor
Joe Stanley, Co-Advisor
Yun Seong Song

ABSTRACT

Integrating sensors that read states of the human body into everyday life is an increasing desire, especially with the rise of deep learning which requires vast stores of data to make predictions. This work explores integrating these sensors into the human experience through two methods and recording the results. The first of these methods integrates a MXene based field-effect transistor sensor for the 2019-nCov spike protein with a mobile app. This allows the user to read how saturated their breath is with Covid-19. The second method integrates 3D-printed pressure sensors, and a motion capture system, into a glove to read data on the human hand. This glove was then used in a human-robot collaboration project to teach a robot to react to a human collaborator's gestured intent after watching a collection of intentional demonstrations. This work seeks for the sensor application, human data glove, and robot-collaboration framework made in this project to be used in later scientific exploration on integrating sensors into the human experience.

Human-robot collaboration is the key emphasis of this work and was achieved through a combination of human intent prediction and robot policy encoding. Human intent prediction was achieved by a stacked LSTM neural network. This network was trained on demonstrations gathered where an individual wearing the human data glove performed an action, and a robot arm controlled by a human operator was moved through the desired trajectory in response to said action. The robot policy was encoded using a probabilistic movement primitive by learning the actions of the robot during these demonstrations. Once trained, the network could watch the actions of the human wearing the glove and respond with the appropriate robot policy with no human assistance.

ACKNOWLEDGMENTS

I would like to extend my heartfelt gratitude to my girlfriend, Anastasia Reed-Comeaux, for her unwavering support and encouragement during the most challenging part of my academic career. Her belief in me, and constant encouragement were the pillars that kept me going. Without her, completing this degree would have been impossible. She reminded me of my purpose and my worth, and I am grateful for her presence in my life.

I would also like to thank my advisor, Dr. Chenglin Wu, for believing in my abilities, trusting me to have leadership over my projects, and being a friend/mentor, I could rely on for advice. Developing research and leadership skills within the field of engineering is a daunting task but with Dr. Wu's help I have been able to develop skills which I know will follow me through the rest of my career.

In addition to my girlfriend and advisor, I would like to express my appreciation to my friends and family who supported me through this journey. Completing graduate school has been a difficult yet incredibly rewarding experience. Thank you for keeping me determined until the end.

TABLE OF CONTENTS

Page

ABSTRACT iii

ACKNOWLEDGMENTS iv

LIST OF ILLUSTRATIONS viii

LIST OF TABLES x

SECTION

1. INTRODUCTION 1

2. LITERATURE REVIEW 4

2.1. OVERVIEW 4

2.2. APPLICATIONS FOR READING SENSOR DATA OVER BLUETOOTH 4

2.3. HUMAN INTERFACE DEVICES FOR HUMAN-ROBOT COLLABORATION 4

2.4. HUMAN-ROBOT COLLABORATION 6

2.4.1. Learning Complex Robot Trajectories with Movement Primitives. 6

2.4.2. Robot Control Methods. 7

2.4.3. Human Gesture Prediction. 8

3. ANDROID APPLICATION FOR REAL-TIME SENSOR READING 10

3.1. IMPLEMENTATION 12

3.1.1. BLE Functionality. 12

3.1.2. Real-Time Graphing and Saving Data. 13

3.2. RESULTS 15

3.3. SUMMARY 20

4. BLUETOOTH GLOVE PLATFORM FOR CAPTURING HIGH-FREQUENCY SENSOR DATA 22

4.1. IMPLEMENTATION 22

4.1.1. Glove Structure 23

4.1.2. Electronic Components 25

4.1.3. Software 29

4.2. RESULTS 31

4.3. SUMMARY 34

5. HUMAN-ROBOT COLLABORATION ON TASKS LEARNED THROUGH DEMONSTRATION 35

5.1. METHODOLOGY 37

5.1.1. Human Intent Recognition LSTM. 37

5.1.2. Probabilistic Movement Primitives. 44

5.2. IMPLEMENTATION 49

5.2.1. Experimental Setup. 49

5.2.2. Data Collection. 50

5.2.3. Training the LSTM. 55

5.2.4. Training the ProMPs. 57

5.2.5. Real-Time Human-Robot Collaboration. 58

5.3. RESULTS 60

5.4. SUMMARY 62

6. CONCLUSION 63

6.1. SUMMARY OF WORK 63

6.2. NOVELTY OF RESEARCH 63

6.3. PUBLICATION PLAN 64

6.4. FUTURE PLANS 65

6.4.1. Improvements to Human-Robot Collaboration System. 65

6.4.2. Applications of Technology. 65

APPENDIX 67

BIBLIOGRAPHY 91

VITA 94

LIST OF ILLUSTRATIONS

Figure Page

3.1 Diagram of Example Bluetooth Peripheral Transmitting Sensor Data to Sensor Graphing Application 10

3.2 Sensor Application User Process 11

3.3 BLE Connect and Read Process 14

3.4 Graph Layouts for 1, 2, or 3 Graphs 15

3.5 Specificity Verification Experiment 16

3.6 2019-nCov Spike Protein Concentration Study 17

3.7 2019-nCov Spike Protein Sensor Flow Rate Study 18

3.8 2019-nCov Spike Protein Sensor Mask Experiment in a Sealed and Unsealed Mask 19

3.9 Closed Environment Experiment with Single 2019-nCov Spike Protein Sensor 20

4.1 Glove Features Overview 23

4.2 Technical Drawing of Electronic Housing Mount for Glove (dimensions in mm) 25

4.3 Sensor Glove Wiring Diagram – Arduino Nano with HC-05 Chip Variant 27

4.4 Sensor Glove PCB Schematic – Arduino Nano with HC-05 Chip Variant 28

4.5 Sensor Glove Wiring Diagram – Arduino Nano RP2040 Variant 28

4.6 Sensor Glove PCB Design – Arduino Nano RP2040 Variant 29

4.7 Vicon Sensor Glove Subject 31

4.8 Sensor Glove Test Scenario – Sliding Board on Table 32

4.9 Sensor Glove Test Scenario - Drill 33

4.10 Sensor Glove Test Scenario - Ball 33

5.1 Human-Robot Collaboration System Overview 35

5.2 Recurrent Neural Network Structure over Multiple Iterations 38

5.3 LSTM Layer Diagram 40

5.4 Human Intent Recognition LSTM Input 41

5.5 Human Intent Recognition LSTM Output 42

5.6 Human Intent Recognition LSTM Architecture 42

5.7 Batch Gradient Descent vs. Stochastic Gradient Descent on Loss Surface 43

5.8 Trajectory Distribution of Wave on Each Joint of the xArm6 Learned from a set of Demonstrations 45

5.9 Hierarchical Bayesian Model Used by ProMPs 47

5.10 Experimental Setup Design 49

5.11 Data Collection and Training Process 60

5.12 Training Loss vs. Epoch for Human-Intent Recognition LSTM 61

5.13 Human-Robot Collaboration Real-Time Wave Response 61

5.14 Human-Robot Collaboration Real-Time Grab Response 62

LIST OF TABLES

Table Page

4.1 Off-The-Shelf Components and Current Prices 24

4.2 Electronic Components and Current Prices 26

1. INTRODUCTION

As humanity and technology become increasingly intertwined, the importance of embedding sensor systems into everyday life grows. Smartphones and smartwatches are the most common devices that people use to interact with sensors on a daily basis. According to Google sensor types supported by Android devices (such as smartphones and smartwatches) include accelerometers, gyroscopes, heart rate, light, ambient temperature, magnetic field sensors, proximity, pressure, and relative humidity [13]. These sensors are useful for a complete smartphone or smartwatch experience but only the heart rate sensor reads the state of the human body. To design systems that improve the everyday lives of the user it is necessary to incorporate sensors that read the state of the human body. However, simply reading the state of the human body is not the only factor to consider. These sensors must be read at a high enough frequency to satisfy user expectations and the process of using the sensor must be done in an unobtrusive manner. If the sensor system is irritating or uncomfortable to wear, or if it doesn't update fast enough, the user is unlikely to use it for long. Incorporating these sensors into devices familiar to the user can help lower the noticeability and increase acceptance of the system. Ways to achieve this include embedding the sensors into clothing and having an existing device perform all computation and analysis of the sensor data. For example, according to the Pew Research Center as of February 2021 85% of the US population owns a smartphone [14]. Therefore, integrating sensor analysis into a smartphone is unlikely to cause any irritation to the user. The likelihood of irritation becomes even less likely when considering the wide range of highly specific smartphone apps Americans are downloading and using every day. In some cases, it may be necessary to incorporate sensors into their own housings that must be attached to

the user. To justify a user putting on a custom device specific to the task special considerations must be made.

- Does the usefulness the sensor helps achieve outweigh the effort of putting on this custom device?
- Is the device simple to put on and begin using?
- Does the device cause any physical discomfort?

With these considerations taken into account sensor integration into the human experience should cause no grievances for the user.

This project takes on the challenge of integrating novel sensors into two separate environments. The first environment is connecting a smartphone application to a MXene based field-effect transistor sensor for 2019-nCov spike protein and H1N1 virus sensing. The sensor was integrated into a mask that the user can wear while breathing normally. As the user is wearing the mask the sensor readings are sent to an Arduino. This Arduino then transmits the sensor data over Bluetooth Low-Energy to a phone application that graphs the state of the sensor in real-time. The focus of this portion of the project was to find an unobtrusive way to relay this sensor data to a user through the use of a device that the user already has on them. This method is far better from a user experience perspective than current covid detection methods. For example, the BinaxNOW covid-19 antigen self-test requires the user to stick a long swab in each nostril and swirl it for 15 seconds per nostril [15]. On top of this, the sensor reading is not available to the user until a minimum of 15 minutes after the test was taken. Our system provides much more transparency to the user and any data collected can be exported for further analysis.

The second sensor integration involved incorporating 3D-printed pressure sensors and motion capture tracking pearls into a glove. The glove is designed to facilitate Human-Robot collaboration tasks, by allowing the user to transmit the pressure and position of each finger to a workstation controlling a robot arm. The system reacts to the human's intent in real-time, allowing seamless interaction between the user and the robot arm. For example, when the user reaches for a board, the system recognizes the user's intent, and the robot arm grabs the other side of the board then assists in moving it. This system allows for the programming of human-robot collaboration tasks purely through demonstration and can remove the need for any programmer involvement. To use the system all the human user must be concerned with is putting on the glove and switching it on, needing no further technical considerations.

The rest of this thesis covers the implementation of the sensor graphing application and robot collaboration framework. Section 2 is a literature review covering some previous attempts at sensor integration and human-robot collaboration. Section 3 covers the design and implementation of the sensor graphing application. Section 4 covers the design and implementation of the human data collection glove. Section 5 covers the design and implementation of the human-robot collaboration system. The final section covers conclusions on the work and further research.

2. LITERATURE REVIEW

2.1. OVERVIEW

The application areas for sensors reading data about the human body are incredibly broad. Hence, the focus of this literature review is on the specific areas in which we applied sensors to the human body. This section details previous work in graphing real-time sensor data over Bluetooth, glove devices for reading finger-tip pressure and finger position, and efforts to achieve human-robot collaboration.

2.2. APPLICATIONS FOR READING SENSOR DATA OVER BLUETOOTH

The Sensor Graphing application that was created is not the first application to graph sensor data in real-time for example the Sensor Plot Kit for iOS [16] is an example of an API that would allow for the same functionality that was achieved using the Android Sensor Graphing application. Another popular sensor reading API for iOS and Android is Sensing Kit [28] but this is limited to reading the sensors that already exist on your phone and not external sensors. This Sensor graphing application was a steppingstone in transmitting human-centric sensor data over Bluetooth, saving it, and exporting it in a real-time setting. This allowed us to have a higher level of expertise when the sensor glove was designed for the Human-Robot Collaboration Project.

2.3. HUMAN INTERFACE DEVICES FOR HUMAN-ROBOT COLLABORATION

Human interface devices for gesture recognition, a key task in human-robot collaboration work, have two main approaches. The first approach is using a data collection

glove of some sort which prevents the need for the developer to post-process data and often the data is more accurate. The second approach is using a vision-based method to extract key points on the human hand. The second method has gotten much more accurate over time but is still prone to more errors and requires more post-processing.

Glove devices for reading data about the human hand have a wide range of uses because they are involved in the majority of human interaction with the world. For example, Yeo [17] created a wireless sensor glove to read movement and pressure on the thumb. While this glove can tell that the thumb is moving it cannot tell the position of the thumb in space like the human data glove. This is an advantage gained by the use of a motion capture system. In 2021 Zhu [18] created a data glove to read the positions of each of the user's fingers with the use of flex sensors. This allowed the user to teleoperate a robot hand with a similar form to the human hand.

Human interface devices in reference to Human-Robot collaboration are not only limited to gloves. Awais and Henrich used a digital camera to monitor a human hand in a scene and then using image processing extracted an outline of the hand [19]. This approach can be an effective way to guess human intentions from the orientation of the hand, but it does not include pressure data for each fingertip as the human data glove designed for this project does. Wu [20] used a more modern approach than Awais and Henrich through the use of the Leap Motion Controller but this limits the working range of the human operator to 80 cm from the leap motion.

2.4. HUMAN-ROBOT COLLABORATION

Human-Robot collaboration is a field that covers humans and robots working together to complete tasks. The ways in which this can be achieved are numerous. For example, some see human-robot collaboration as introducing methods of commands to robots that are more familiar to humans such as [22], [23] and [24] which all use some form of natural language processing to issue commands to a robot collaborator. While others see the robot as a proxy for a human being such as [4],[11], [20], and [21]. In this approach, the researchers are replacing what would normally be another human in a collaborative task. This project is taking this approach when looking at human-robot collaboration. Like this project, these tasks tend to be performed over a table of some sort because it allows for both the robot and human to have a common place to work while making up for the limited mobility of the robot collaborator.

2.4.1. Learning Complex Robot Trajectories with Movement Primitives. Movement primitives have been a solution to complex motor control in robotics for a while. The first generally used movement primitive was introduced by Stefan Schaal in 2006 and is known as the Dynamic Movement Primitive or DMP [2]. The DMP was then updated in 2013 by Auke Ijspeert [3]. The core idea of the DMP is to describe a trajectory using two parts: a point attractor function and a forcing function. The point attractor function simply pulls the trajectory towards a specific point over time and grows in magnitude. So, at first, the point attractor does not have much effect but at the end of the trajectory, it is the most powerful force. The forcing function is a set of basis functions that diminish in magnitude over time and allow more complex paths to be described. This allows for a complex trajectory to be followed very closely at first but at the end of a DMP you are always

guaranteed to converge on a final waypoint because of the point attractor function. The DMP can also be adjusted in two ways given parameters. The first is it can be spatially scaled meaning it ends its trajectory at a point closer or further away from its original ending waypoint. The second is it can be temporally scaled meaning that it can take a shorter or longer amount of time to complete its complex trajectory. Katharina Mülling used DMPs to describe the trajectories of a robot arm playing table tennis with a human collaborator [4] showing how powerful they can be when paired with a neural network. In 2013 the Probabilistic Movement Primitive or ProMP was introduced by Alexandros Paraschos [5]. ProMPs do not use a point attractor unlike DMPs and instead use a set of Gaussian basis functions to describe a complex trajectory as a probability distribution. Describing a trajectory in this way not only allows us to get a much more accurate representation of a trajectory than DMPs allow for but our ability to adapt ProMPs is also superior. "[ProMPs] can be used to adapt [a] movement at any time point during [a] trajectory's execution" [6] this includes conditioning a trajectory to start/arrive at a specific location or velocity. This means if we use a ProMP to demonstrate the trajectory of a robot arm waving we can condition properties such as the direction it's facing, speed of execution, and stopping point with all joints updating according to these new conditions. All these features of the ProMP can be described with as little as three parameters: initial position, final position, and relative execution time. For these reasons, this project is utilizing the ProMP to learn complex robot trajectories as adaptability is such a large advantage.

2.4.2. Robot Control Methods. In order to learn these trajectories, recordings of demonstrations on the robot arm must be gathered. Matthias Rambow's paper on

Autonomous Manipulation of Deformable Objects based on Teleoperated Demonstrations [7] shows that in order to teleoperate a robot arm with many degrees of freedom a complex control system must be made to give a human finite control of a robot arm. A paper by Gianluca Lentini [8] shows a less expensive approach to teleoperation by using a VR headset with motion trackers but this is software heavy and still relatively expensive. This is a route that is not only time intensive and financially infeasible but also overcomplicates the problem. The best solution to finite robot arm control for complex trajectories is kinesthetic control. Samuel Detzel used Kinesthetic control in order for a surgeon to teach complex trajectories to a robot arm for middle ear surgery [9]. This shows not only how simple kinesthetic teaching can be because someone who is not a robotics expert efficiently operated a robot, but they were also able to get a level of fine control exact enough to use in surgery. Alberto Montebelli's paper on kinesthetic teaching [10] not only showed larger scale tasks like this project is attempting such as wood planing but also showed the types of real-time constraints that were overcome in this project. Their robot arm was able to send its state at a 1 kHz frequency and once a trajectory was recorded it would be repeated at 500 Hz. The robot arm for this project, the UFactory xArm 6 will be operating at 250 Hz.

2.4.3. Human Gesture Prediction. After a set of movement primitives has been learned the next step is intelligently deploying them. Mülling used an RNN gating networking [4] that was trained off demonstrations in an imitation learning stage and then further improved using reinforcement learning when training online. A more recent method from November 2021 saw a team skip the imitation learning step and simulated a robot playing tennis entirely within a simulation then transferred that into reality [11]. This has

the advantage of skipping demonstrations but requires a complex simulation to be built for every task that you want to attempt. This project achieves human gesture prediction through the use of a stacked LSTM (Long Short-Term Memory) Neural Network. Every 20 ms the state of the human operator is sent to the network and a prediction of what gesture the human collaborator is making is produced. Wu [20] used an LSTM in order to predict the gestures of a human in a human-robot collaboration setting but the device for hand recognition, the Leap Motion, limited the gestures to hand movements. Large sweeping movements like waves and grabbing objects that are used in this project would not be able to be recognized by that system.

3. ANDROID APPLICATION FOR REAL-TIME SENSOR READING

The Android Application for Real-Time sensor reading allowed any set of 1, 2, or 3 sensor values to be transmitted over Bluetooth Low Energy and graphed on an Android phone in real-time. A view of the user interface and an example of a Bluetooth Peripheral wired to read sensor data can be seen in Figure 3.1. As the values were being graphed they were saved in memory and could be exported as a csv file of sensor readings with timestamps relative to when the recording started. This system allowed our team to read the state of MXene based field-effect transistor sensors for the 2019-nCov spike protein and the H1N1 virus in real-time while a patient was wearing a mask with the sensor embedded.

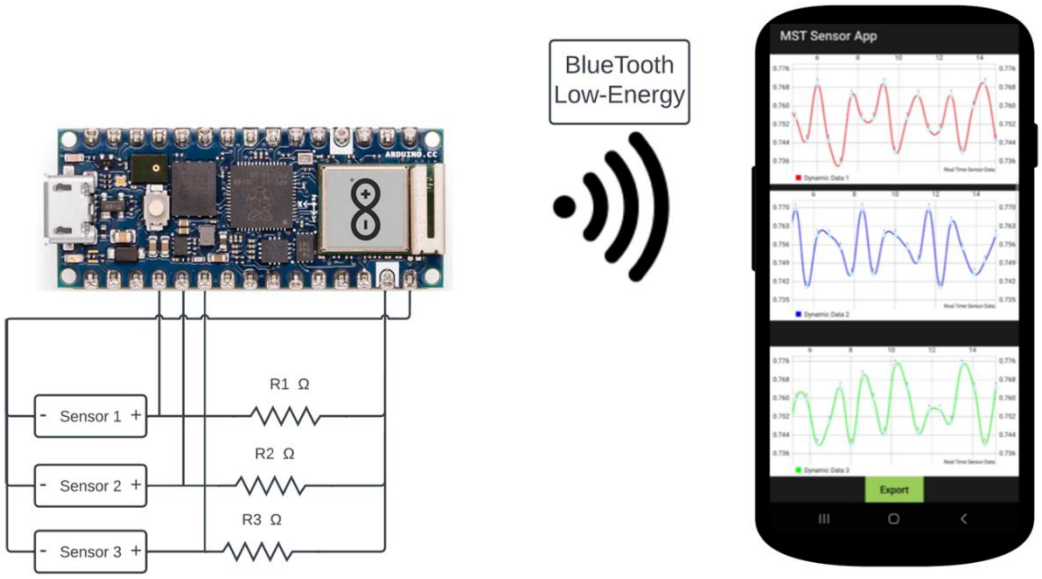

Figure 3.1 Diagram of Example Bluetooth Peripheral Transmitting Sensor Data to Sensor Graphing Application

Another key focus of the design of the application was ease of use. This was because this application was being used to test several patients using the previously mentioned sensors at the hospital. Therefore, efficiently switching between the patients and exporting the collected information needed to be straightforward. To connect, all a user had to do was start the Bluetooth peripheral transmitting the sensor data (in our case this was an Arduino Nano) and open the graphing sensor app. Once in the sensor app the user simply had to follow the steps that are laid out in Figure 3.2.

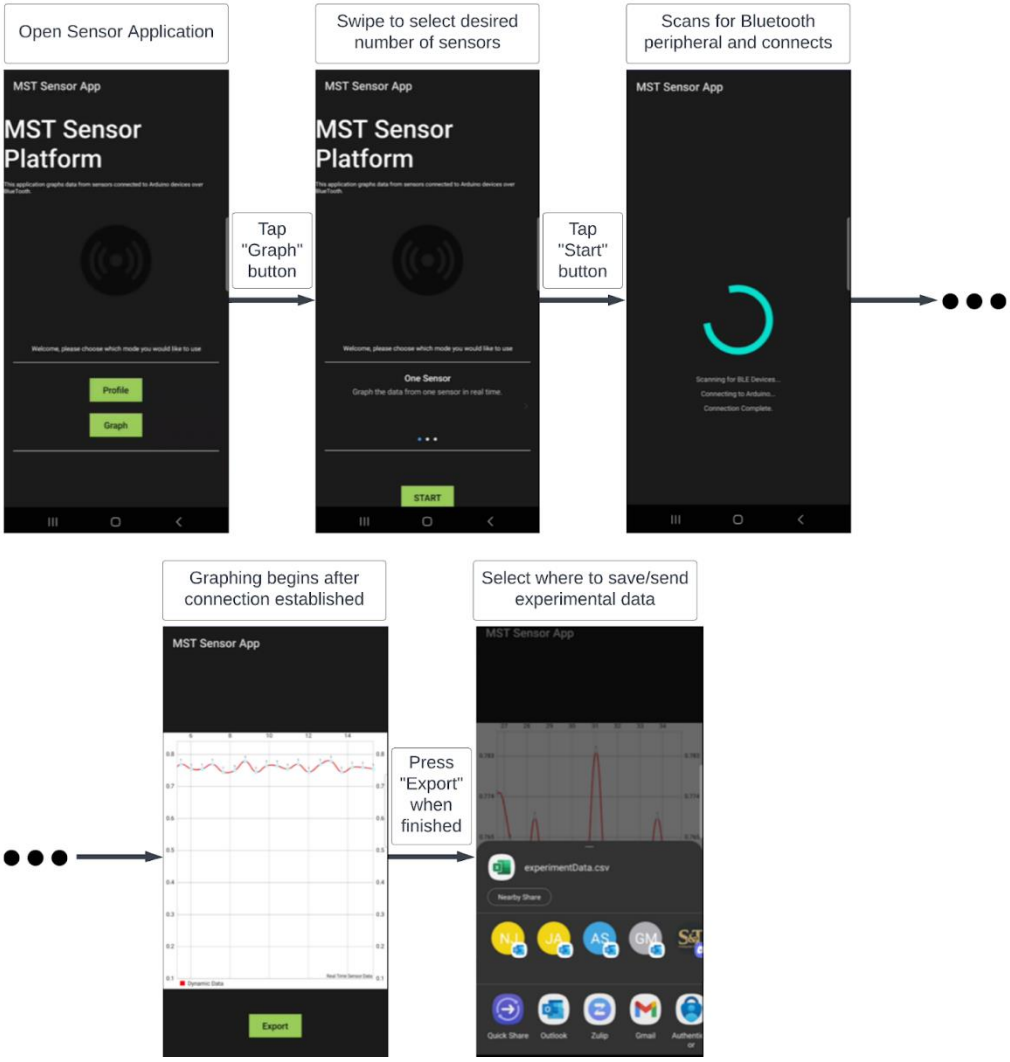

Figure 3.2 Sensor Application User Process

3.1. IMPLEMENTATION

The sensor application was created with the Android Studio IDE using the Java programming language. The system targets Android 11.0 which according to Statcounter means as of March 2023 62.04% [25] of all global Android Devices can run the application thanks to backward compatibility. The application has two key factors to its functionality. The first is scanning for Bluetooth Low Energy (BLE) Devices, connecting to the appropriate device, and reading the device on a regular interval. The second key factor is taking the data read from the BLE device, then saving and graphing it in real-time.

3.1.1. BLE Functionality. Upon pressing the "Start" button the Sensor app will begin scanning for BLE devices to find the appropriate one to connect to. This is implemented through the BLE scan function. BLE uses what are known as UUIDs, Universal Unique Identifiers, these UUIDs are presented to the host device by the peripheral to tell it what services are offered by the device and the data format that will be transmitted. In the case of the sensor application, a UUID associated with both a service and a characteristic is being searched for. A service encapsulates a set of values called characteristics that are transmitted by the BLE peripheral. An example of a service would be a battery service which contains a battery-level characteristic. The host device would in this scenario query the BLE peripheral with the UUIDs associated with the battery characteristic and battery service. In response, the BLE peripheral would return the battery service information. The sensor application has one UUID used to query for the sensor service and sensor characteristics. The sensor service simply encapsulates the characteristic. The sensor characteristic returns a floating point value representing the current sensor reading every time that it is queried. The BLE scan function starts the

Android device's Bluetooth adapter and starts a scanning callback function. The scanning callback function calls a function called "on scan result" every time a new BLE peripheral is discovered. This function reads information about the device to see if it has the expected address. If it has the expected address, a connection is established by calling the connect Bluetooth function. Once the connection is established a callback function is run because the Bluetooth adapter connection state has changed. This function calls the service discovery function which initiates another callback function once services are discovered. This callback function steps through every service and the service that is equal to the expected UUID is selected. Once this is complete every characteristic within the service is stepped through and the characteristic with the expected UUID is selected as well. Finally, the data handling function is called and writes data to the BLE peripheral requesting the state of the sensor characteristic. The BLE peripheral then responds with the state of the sensor. This is repeated in a loop while the graphing and data-saving functionality runs in between BLE peripheral read and write operations. The BLE functionality runs at between 1 and 3 Hz with the number of sensors being scanned affecting the sensor update frequency. This is because it helps with graphing stability and a sensor does not have to read at a rate faster than 1 Hz for the use case of reading the state of a Covid-19 sensor. Figure 3.3 is a diagram showing the flow of functions for the BLE functionality.

3.1.2. Real-Time Graphing and Saving Data. For each of the sensor modes, 1, 2, or 3 sensors, graphing and saving data operates in the same manner just with more graphs on the screen at once. While attempting to make this portion of the app the most difficult part was getting three real-time graphs to run on the phone stably. Originally the project used the GraphView library [26] which advertises the ability to create real-time graphs.

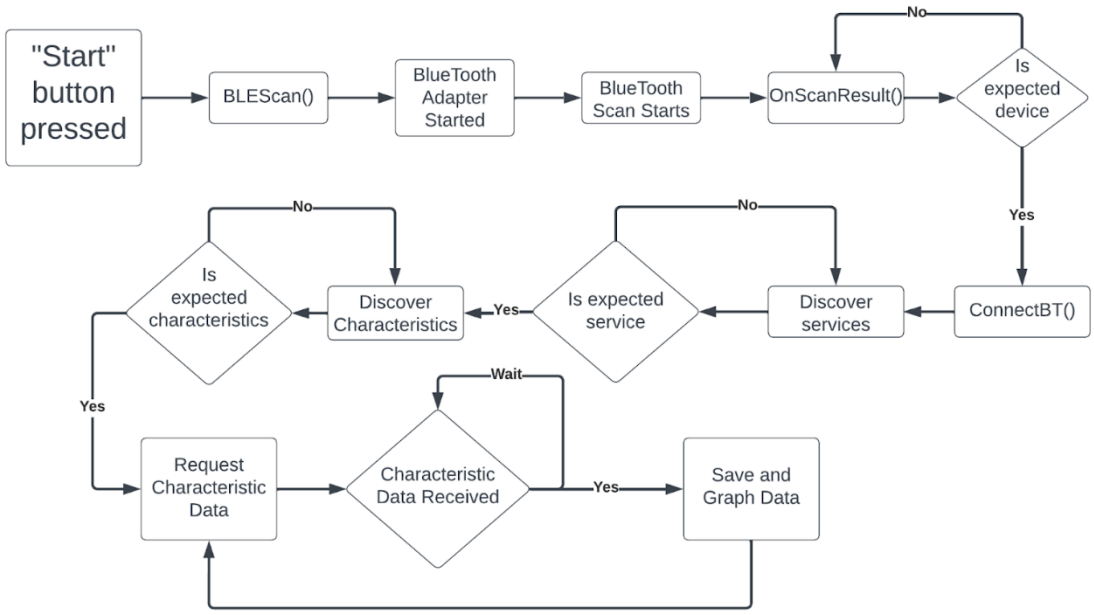

Figure 3.3 BLE Connect and Read Process

While it can produce a functional real-time graph the frequency with which 3 graphs were being updated caused instability and the application would often crash under the stress of frequently updating all of the graphs. This led to the project switching to using the MPAndroidChart library [27] which does not explicitly list real-time graphs as one of its features. Real-time functionality was achieved with this graphing library by limiting the number of points visible on the graph at one time to a maximum of 50. Every time a new data point was received from the BLE peripheral a new data point would be added to the set holding the graph data. The graph would then be notified that the data had changed and the graph window would automatically update to accommodate the new data range. If the length of the graph data exceeded 50 then the first 25 data points would be removed from the set. Figure 3.4 shows the layout of 1 graph, 2 graphs, and 3 graphs. This is where writing to the data to memory becomes a factor. Once the 25 data points were removed they were

written to a file called “experimentData.csv” which was continuously updated throughout the experiment. Once the experiment was complete and the export button was selected this file would then be exported from the program to the application selected by the user.

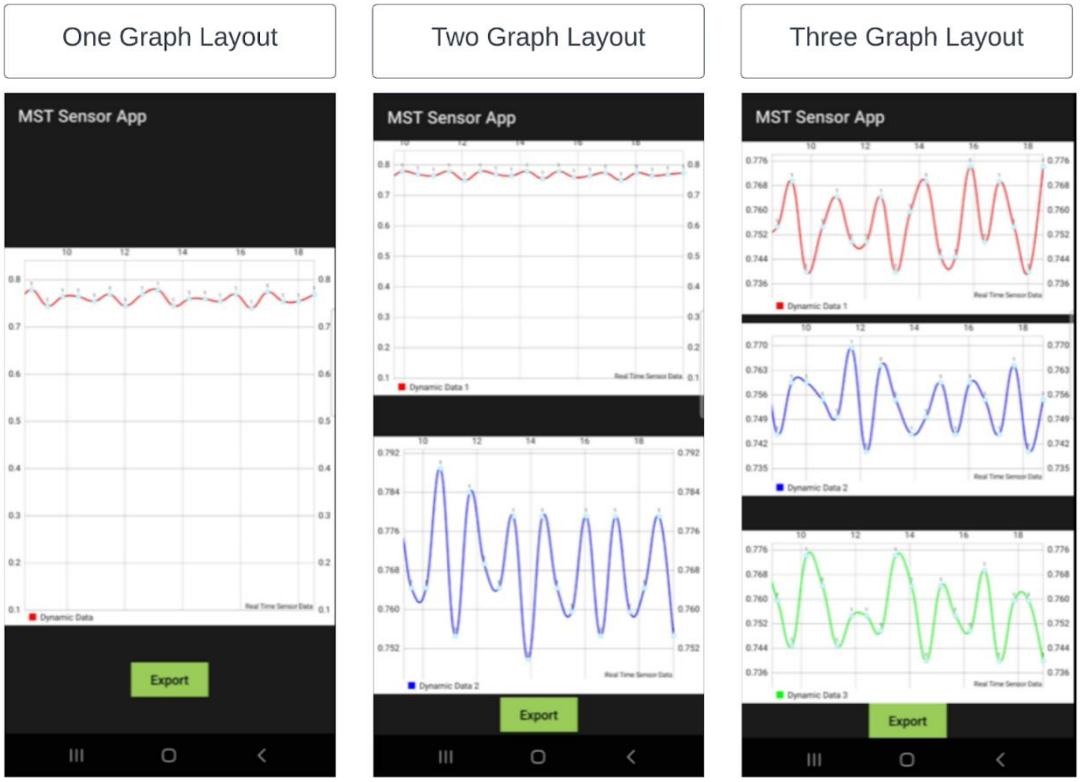

Figure 3.4 Graph Layouts for 1, 2, or 3 Graphs

3.2. RESULTS

After the app was created it was successfully used in a total of 6 experiments in order to collect data on the 2019-nCov spike protein and H1N1 virus sensor. The first of these experiments was a specificity verification experiment. This experiment ensured that the 2019-nCov spike protein sensor did not spike in the presence of the H1N1 virus. While

the sensors were being sprayed in a contained box with the H1N1 virus the sensor app was graphing the results of the experiment in real-time. Figure 3.5 shows a screenshot of the experiment running in real-time. The sensor application was able to clearly show that in

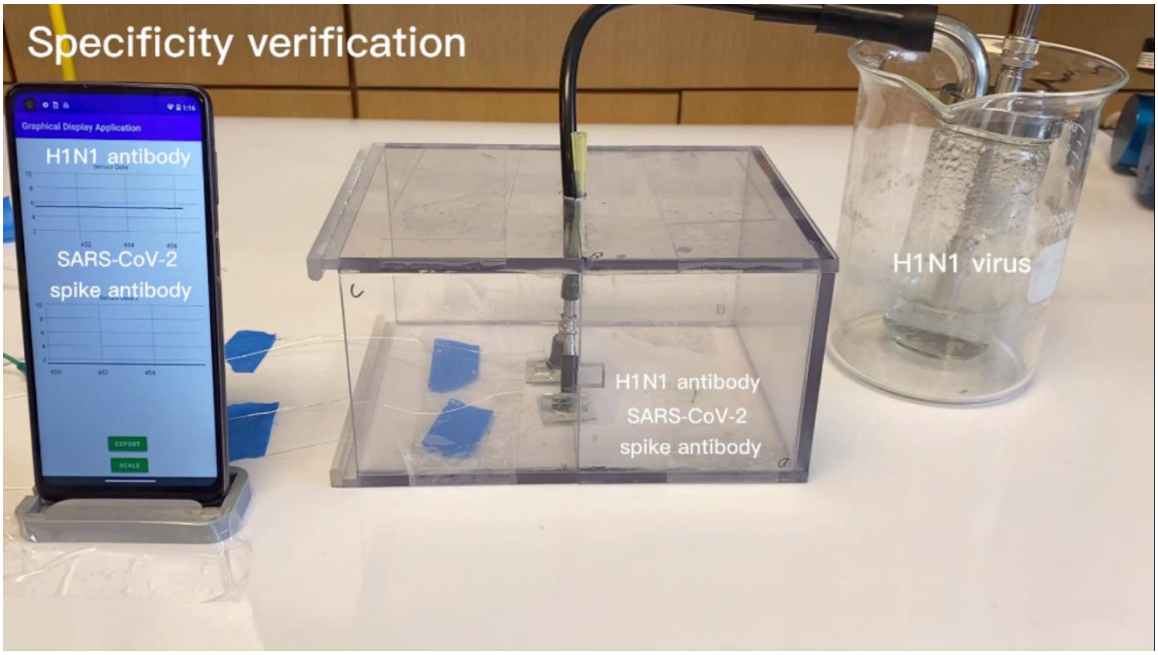

Figure 3.5 Specificity Verification Experiment

the presence of the H1N1 virus, the 2019-nCov spike protein sensor stayed stable while the H1N1 virus sensor spiked. Once the experiment was complete the data was exported and used for further analysis.

The second experiment the sensor app was used in was a concentration study. In this experiment 3 2019-nCov spike protein sensors were used. Sensor 2 was placed directly below the nozzle spraying the 2019-nCov spike protein while sensors 1 and 3 were equally spaced away from the nozzle on the left and right of sensor 2 respectively. This experiment showed the reactivity of the sensors under different concentrations of the spike protein.

The graphing application showed in real-time that sensor 3 had the highest concentration of 2019-nCov spike protein contact because it was closest to the air filtration system which was pulling air into it. While sensors 1 and 2 showed similar reactions to the 2019-nCov spike protein. Figure 3.6 is a screenshot of the experiment where the graphing application is showing these differences. As with the previous experiment once complete the data was exported and used for further analysis.

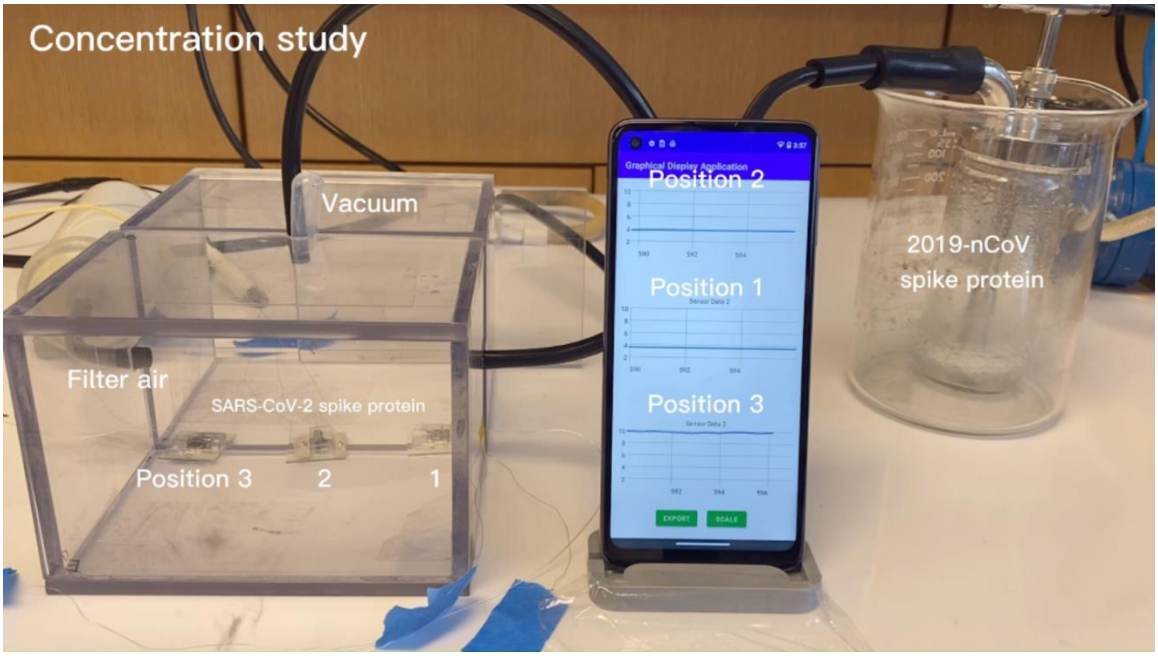

Figure 3.6 2019-nCov Spike Protein Concentration Study

The third experiment was a flow rate study for the 2019-nCov Spike Protein sensor. This was a total of 3 experiments each recording one sensor at a time. Each of the sensors was measured for reactivity to the 2019-nCov Spike Protein in the presence of different flow rates from the nozzle. The three flow rates that were tested are 0.009 fg/s, 0.014 fg/s, and 0.018 fg/s. The sensor application showed the expected results, as the flow rate

increased the number of spike proteins contacting the sensor increased and the reading on the sensor went up. Figure 3.7 is a screenshot of the experiment demonstrating these results. This experimental data was all exported and used for further analysis.

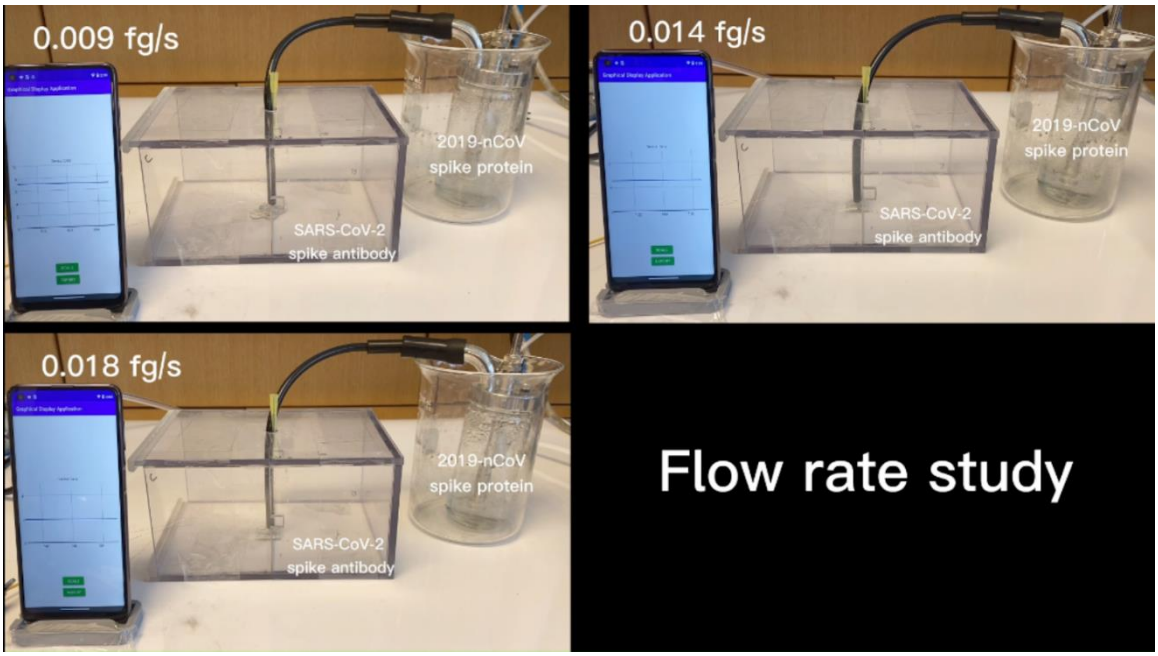

Figure 3.7 2019-nCov Spike Protein Sensor Flow Rate Study

Experiment 4 embedded the 2019-nCov Spike Protein Sensor into an n95 mask and put it on a model of a human head with a nozzle emitting 2019-nCov spike protein through the mouth. The experiment tested the sensor's ability to read the levels of 2019-nCov spike proteins in two scenarios. The first scenario used an unsealed mask that was put on normally allowing for regular airflow through an n95 mask. The second scenario completely sealed the edges of the mask with tape allowing for a higher concentration of the spike protein. While the unsealed mask showed readings above the baseline as expected

the sealed mask had much higher readings. Figure 3.8 shows a screenshot of this discrepancy between the two scenarios.

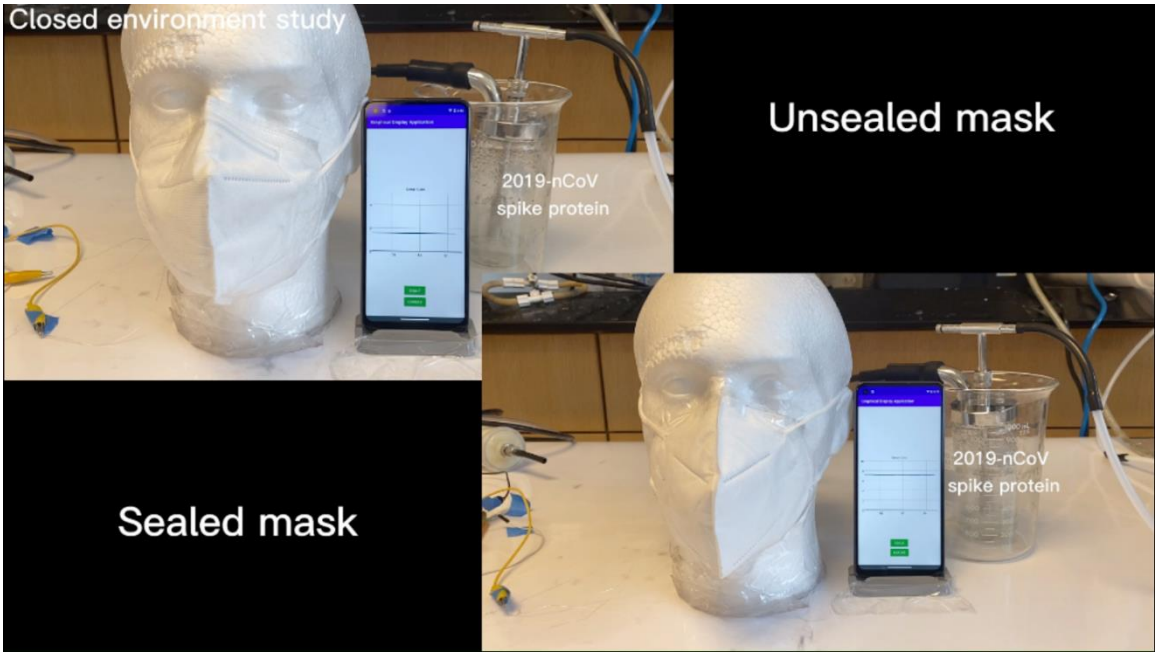

Figure 3.8 2019-nCov Spike Protein Sensor Mask Experiment in a Sealed and Unsealed Mask

The 5th experiment tested a single 2019-nCov Spike Protein Sensor in a closed environment. This was a simple experiment that simply wanted to show the 2019-nCov Spike Protein Sensor changing readings in the presence of 2019-nCov Spike proteins in a sealed-off box. It is conceptually similar to the sealed mask experiment but the volume of the closed space is much larger than the sealed mask. The experiment showed higher readings on the 2019-nCov Spike Protein Sensor and this is reflected on the app. Figure 3.9 shows the experimental setup and the higher readings on the 2019-nCov Spike Protein Sensor.

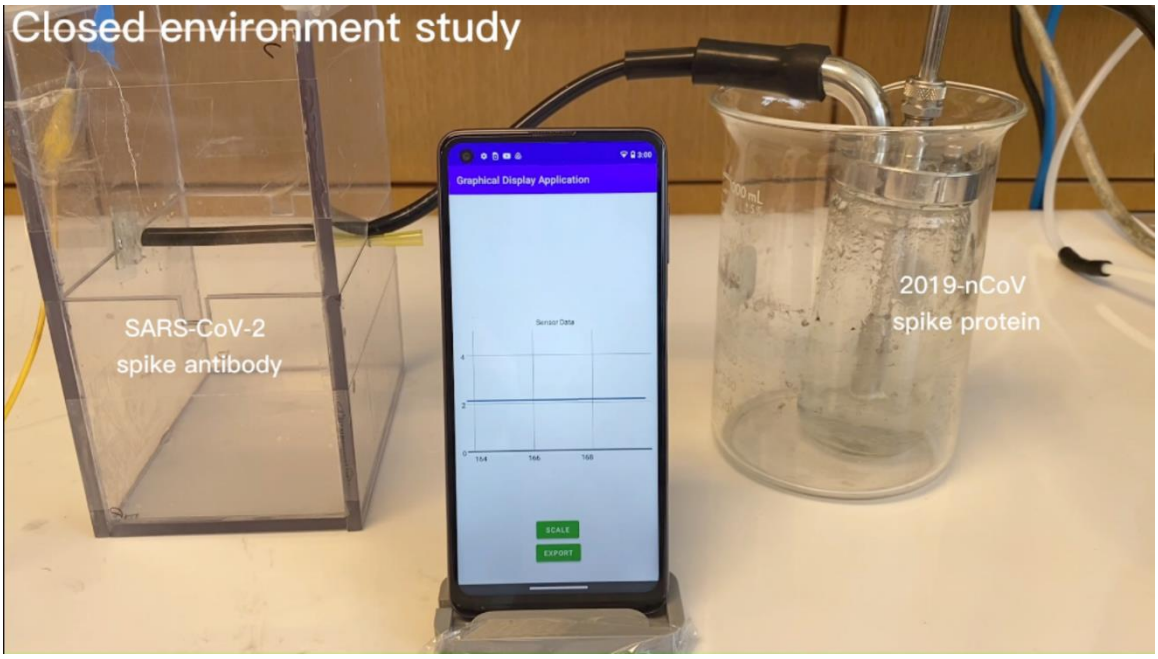

Figure 3.9 Closed Environment Experiment with Single 2019-nCov Spike Protein Sensor

The 6th experiment tested the sensor application in real-life scenarios. A set of masks with embedded 2019-nCov Spike Protein Sensors was brought to Phelps Health and a phone with the sensor graphing application was brought along. The mask was then put on patients while the sensor graphing application collected readings from the sensor. This was done on a total of 85 patients who had previously been tested for covid-19 and confirmed as positive or negative. The phone application allowed for real-time results on the sensor readings with no need for bulky equipment to be present. Due to patient confidentiality, no photos of this experiment are available.

3.3. SUMMARY

This section covered the development of a real-time sensor graphing application. This Android app has the ability to connect to a peripheral over Bluetooth Low Energy,

BLE, and graph the state of up to 3 sensors in real-time. The sensor data is also recorded, timestamped, and saved in a csv file which can easily be exported to app that supports file sharing. This app was developed for the purpose of recording the state of a 2019-nCov Spike Protein Sensor embedded into a n95 mask. This allowed for a convenient hands off method for recording a patients covid-19 status.

4. BLUETOOTH GLOVE PLATFORM FOR CAPTURING HIGH-FREQUENCY SENSOR DATA

The Bluetooth glove platform for capturing high-frequency sensor data allows a user to track the position of their fingers in space as well as track a set of up to six resistance-based sensors. The information on the sensors is transmitted using classic Bluetooth. For this project's use case, the glove was fitted with 5 3D printed pressure sensors, one on each fingertip. The purpose of building this glove was to easily extract the intent of a human being based on the movement and actions of their hand. A construction glove was chosen specifically because the targeted use case of the project's human-robot collaboration system is construction. This glove can easily be switched out for different scenarios since all electronics are stored on a forearm-mounted housing that is secured with Velcro straps. Lastly, the arm has a rechargeable battery that is recharged through a micro-USB port. The hope for this glove is that it cannot only be used as a method to capture human hand data for this human-robot collaboration project but be used in future projects with numerous potential applications. Figure 4.1 provides a general overview of the glove's features and appearance.

4.1. IMPLEMENTATION

This section covers the full breadth of the design of the sensor glove. Aspects of the glove that will be covered include the glove structure, electronic components, and software. Section 4.1.1, Glove Structure, covers all of the non-electronic materials needed to create the glove. Section 4.1.2, Electronic Components, covers every electronic component and how to connect them together. Lastly, Section 4.1.3, Software, covers the

software powering Bluetooth connection and transmission and an overview of the Vicon Nexus SDK. The intention is that this section covers the glove in enough detail that someone seeking to recreate it would have a clear understanding of how to rebuild it.

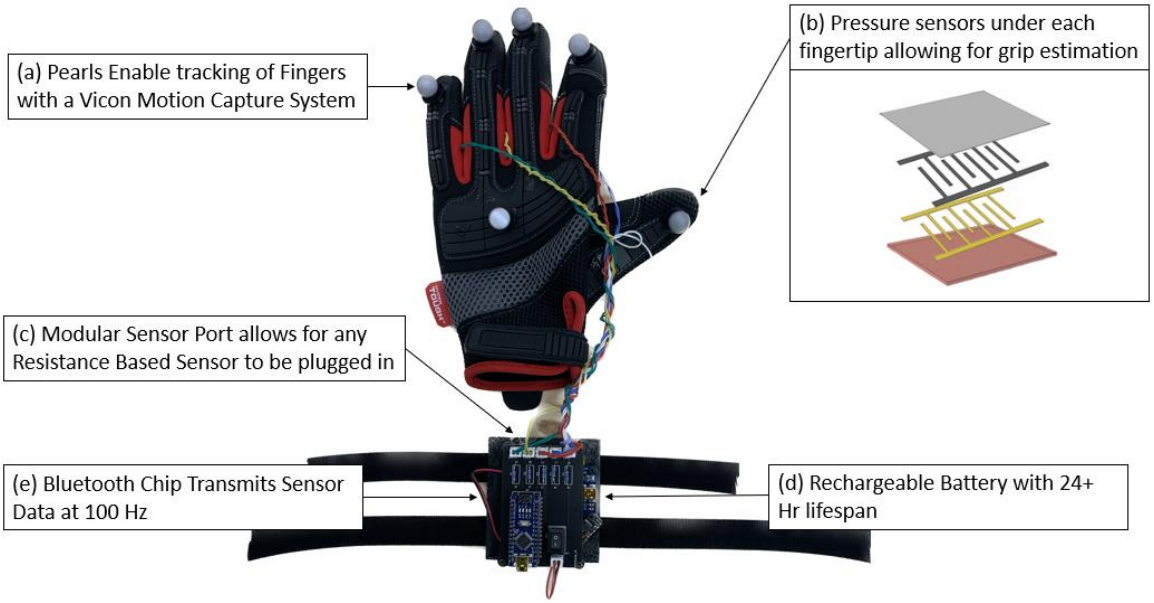

Figure 4.1 Glove Features Overview

4.1.1. Glove Structure. The gloves structure almost entirely consists of a set of off-the-shelf materials that are inexpensive to purchase. Based on current market prices all of these components would cost x dollars. Table 4.1 lists all off-the-shelf materials and their current prices. The only piece of the glove structure that has additional requirements is the custom housing for the electronic components. This component was 3D printed and therefore requires a 3D printer to create. The housing neatly secures all electronic components in place and connecting all electronic components to it only requires 7 screws, 7 nuts, and a small amount of glue. Figure 4.2 shows a technical drawing of the electronics housing mount.

Table 4.1 Off-The-Shelf Components and Current Prices

Component	**Cost**
Hyper Tough High-Performance Black Synthetic Leather Work Gloves	$15.96
VELCRO Brand 15 ft x ¾ in Roll	$7.98
9.5 mm Vicon Tracking Pearl (10 pack)	$70.00
6.4 mm Vicon Tracking Pearl (10 pack)	$70.00
7 x 2mm Machine Screws with 4mm Length	$0.49
7 x 2mm Hex Nuts	$0.49
Total Cost	$164.92

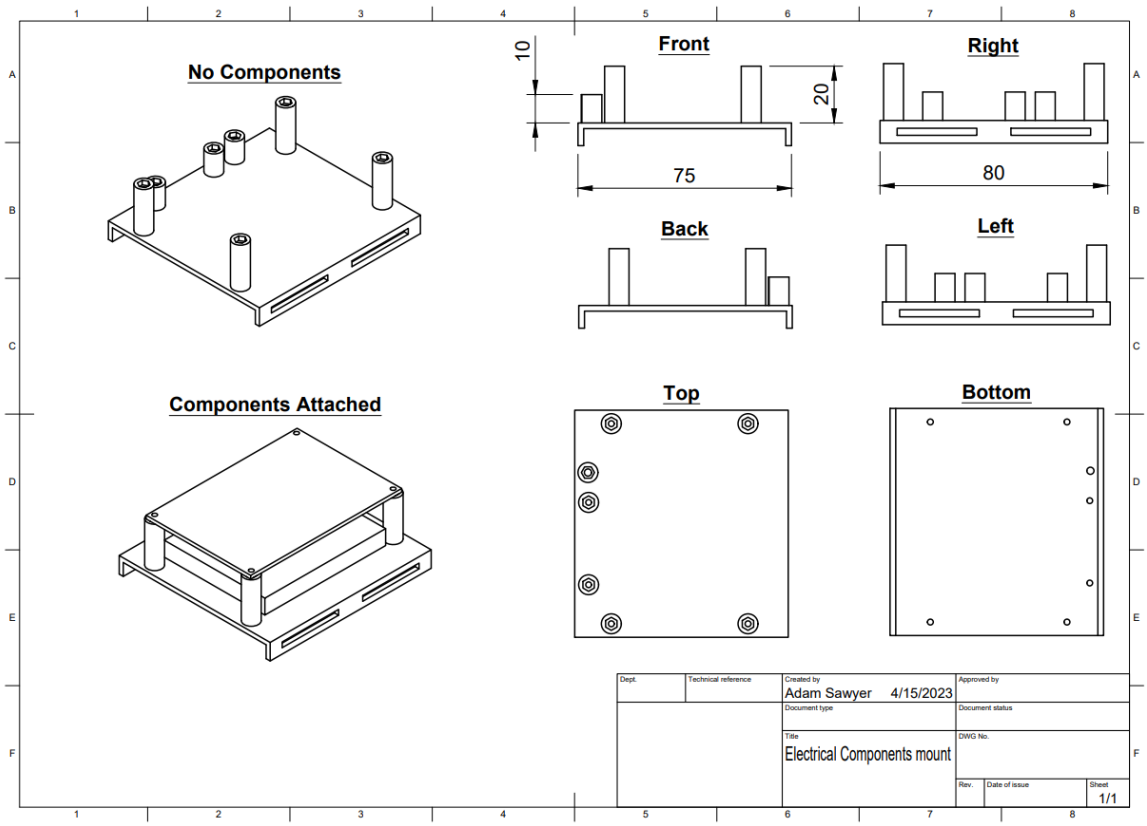

Figure 4.2 Technical Drawing of Electronic Housing Mount for Glove (dimensions in mm)

As can be seen from Figure 4.2 the base plate has two slots. These slots are intended to fit two Velcro straps allowing the user to secure the housing to their arm. In total, the device has 12 Vicon Motion tracking pearls. Only five of these pearls are used by our system for data, the pearls at the tips of the fingers. The other 7 pearls are used to reinforce the proper labeling of the fingers by the Vicon motion tracking system.

4.1.2. Electronic Components The electronic components consist mainly of parts that can be sourced from suppliers online with only one custom-made component. These off-the-shelf components are listed in Table 4.2 with their current market prices. All of the electronics are connected together using a custom-designed PCB. This PCB holds all of

the components except for the charging circuitry and DC-DC power booster. All computing on the device is done from an Arduino Nano. The Arduino Nano does not have built-in Bluetooth connectivity, so this is enabled by wiring it into an HC-05 Bluetooth chip.

Table 4.2 Electronic Components and Current Prices

Component	**Cost**
Adafruit PowerBoost 500C Charger	$18.50
Legion SS01-BBIWA-RA20-R (Power Switch)	$1.20
LiPo Battery - 3.7v 2500 mAh	$14.95
Arduino Nano	$24.90
Adafruit MiniBoost 5v @ 1A	$3.95
Total Cost	$63.50

The HC-05 can deliver sensor data every 10 ms giving the device a frequency of 100 Hz. Figure 4.3 and Figure 4.4 show the wiring diagram and PCB schematic for the Arduino Nano variant of the glove design respectively. An alternative PCB has been designed and tested with the Arduino Nano RP 2040. This chip is much more powerful than the Arduino Nano having an onboard IMU as well built-in BLE and Wi-Fi support. The limiting factor of this device is that it only has 4 usable analog ports when Wi-Fi or BLE are enabled. If a limited number of Analog ports is not an issue this is the

recommended chip to use for a more versatile glove. In the case of this project, the Arduino Nano was used because reading pressure sensor values from all 5 fingers was critical to the use-case. Figures 4.5 and 4.6 show the wiring diagram and PCB schematic for the Arduino Nano RP2040 variant of the glove design. Powering both of the chips is a 3.7V LiPo 2500 mAh battery. This battery has been tested and runs continuously for 24+ hours before requiring a recharge. This is advantageous to our project because the human-robot collaboration portion of this project is targeting the construction field where workers labor for long hours often with no outlets nearby.

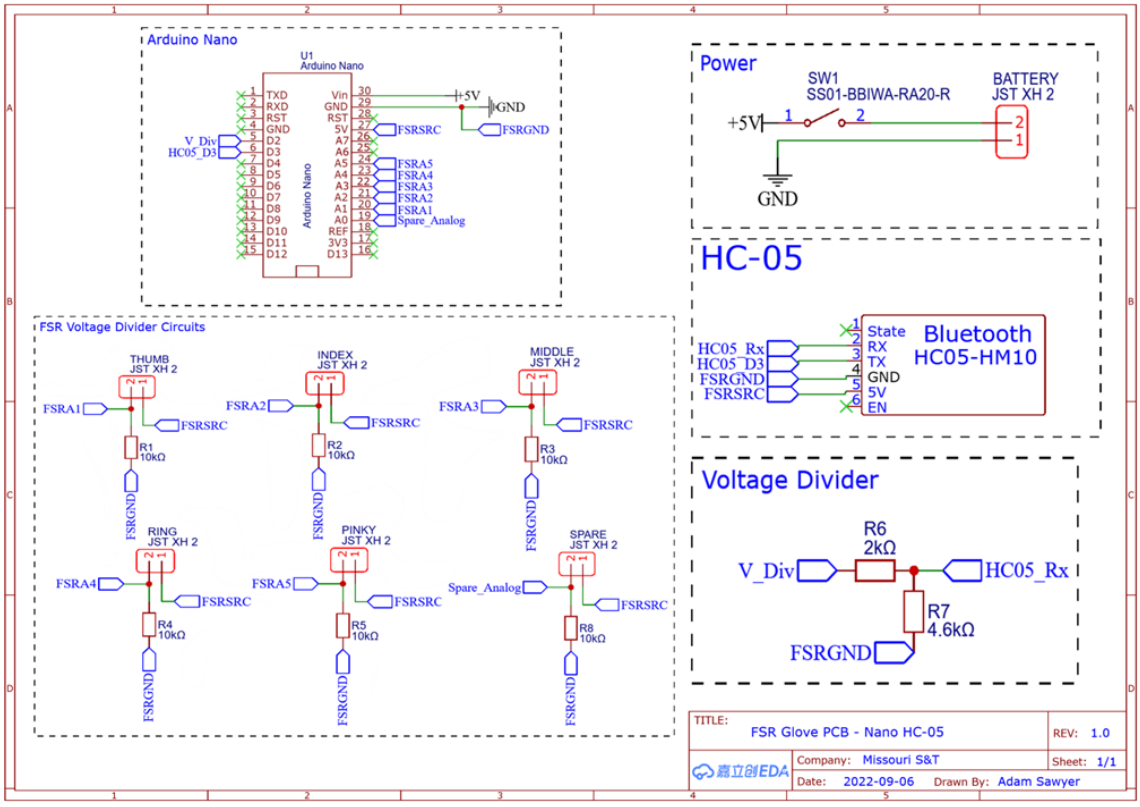

Figure 4.3 Sensor Glove Wiring Diagram – Arduino Nano with HC-05 Chip Variant

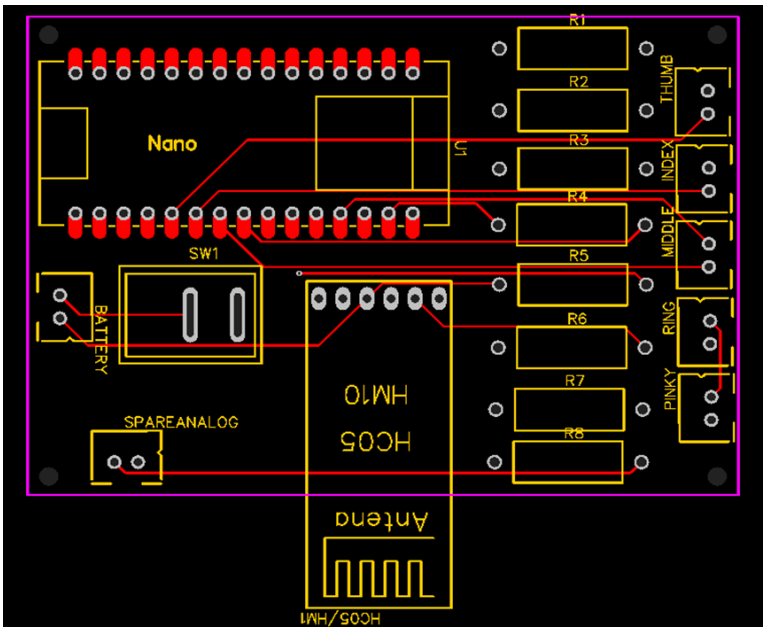

Figure 4.4 Sensor Glove PCB Schematic – Arduino Nano with HC-05 Chip Variant

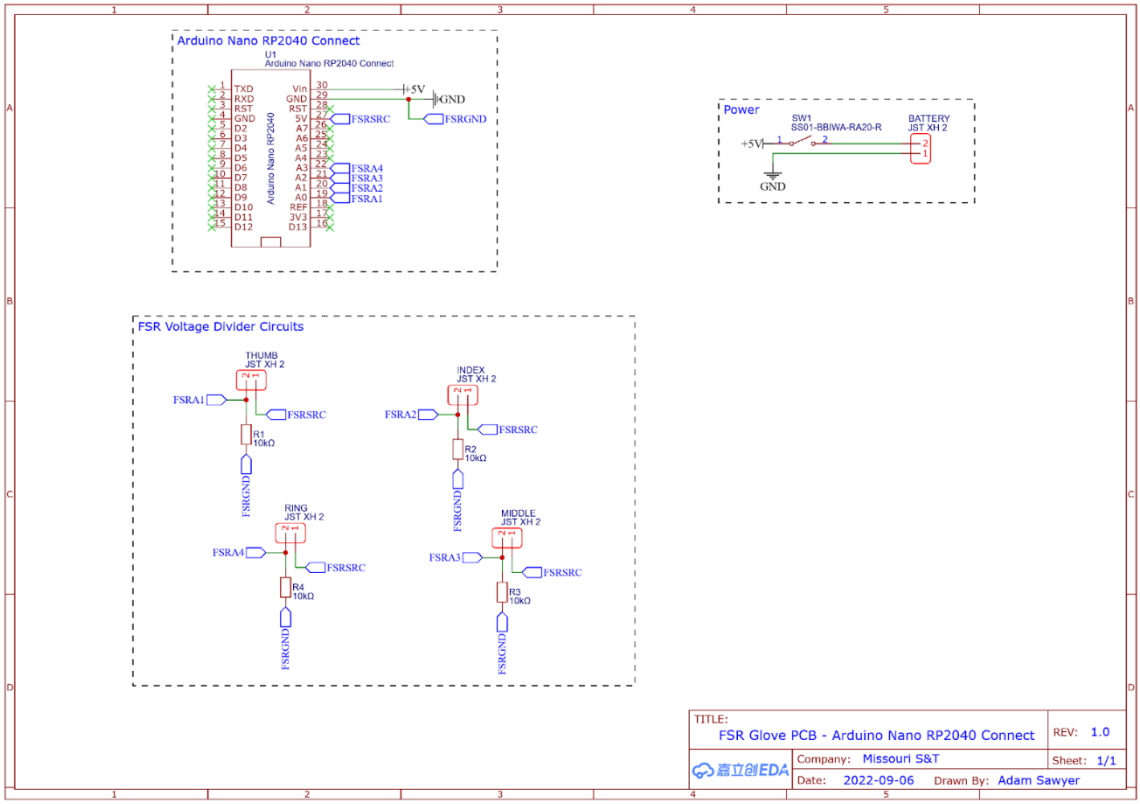

Figure 4.5 Sensor Glove Wiring Diagram – Arduino Nano RP2040 Variant

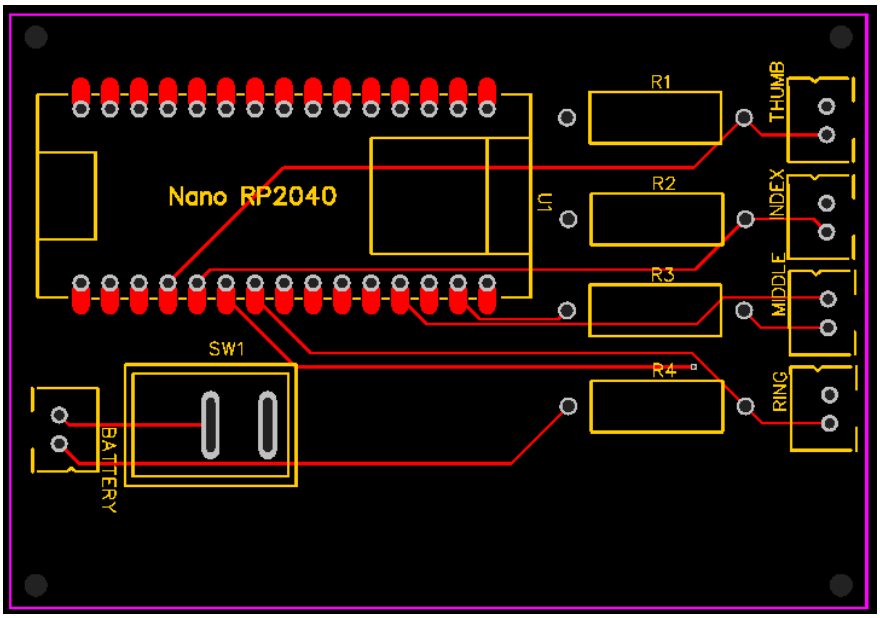

Figure 4.6 Sensor Glove PCB Design – Arduino Nano RP2040 Variant

4.1.3. Software. The software for the sensor glove has three components. The first of these components is the code that is powering the Arduino. This software controls the collection of sensor data and the following transmission of that data over Bluetooth. The second component is the Vicon System tracking the position of the glove in space. Lastly, the third component is the host device receiving the data from the glove. This section focuses on the first two components. This is due to the fact that the third component is very task specific. The function of the host computer for the human-robot collaboration project is covered in Section 5.2.2.

The Arduino Bluetooth pseudocode can be found in Algorithm 4.1 with the code listed in the appendix. As can be seen in the pseudo-code below the Arduino waits for a host device to establish a connection and send a start signal. This is due to an issue of alignment between the host device and the Arduino. If the host device were to read in the

```
Input  : StartSignal from HumanDataWorkstation
Output: FingerPressure sent to HumanDataWorkstation
1 BTSerialConn ← CreateBTSerialConn(ReceivePin, TransmissionPin);
  // Conection runs at a Baud Rate of 115200
2 Wait for StartSignal from HumanDataWorkstation;
3 while True do
4 |   FingerPressure ← read state of each finger sensor;
5 |   FingerPressure ← Concatenate(FingerPressure, FingerPressure);
  |   // Transmitting twice guarantees a full message is
  |      received
6 |   BTSerialConn ← SendData(FingerPressure);
7 |   Wait for 10 ms to elapse;
  |   // Writing a message faster than every 10 ms will crash
  |      the HC-05 chip
8 end
```

Algorithm 4.1 Bluetooth Glove Operation

number of bytes that make up the entire message, 6 bytes, it is very common for the host pc to receive the end of a previous message and the start of a new one. This can be overcome by sending the same message twice. Therefore, when the host pc reads double the message length, 12 bytes, it is guaranteed to receive at least one whole message instead of an incomplete combination of two messages. An alternative solution is to only transmit messages from the Arduino when a request message is received from the host device. While this guarantees that the next 6 bytes received from the Arduino will be a complete message it introduces a new more detrimental problem to proper device operation, latency. When the HC-05 chip switches from writing to reading operations a delay of ~70 ms is introduced into the system. This takes the glove from a 100 Hz system down to 14 Hz, an 86% decrease in frequency. The small overhead of writing the same message twice is far smaller than the overhead of context switching between reading and writing on the HC-05.

The Vicon system is also critical to the functioning of the glove. It tracks each of the markers on the glove using a set of 10 cameras. Each of these markers is labeled automatically by the Vicon system through the creation of what is known as a subject. The subject learns what the name of each marker is based on its relative position to a set of other expected markers. Figure 4.7 shows what the subject looks like from within the Vicon GUI. In Section 5 the Vicon SDK is covered in more detail. This creates a Python interface into Vicon allowing the marker data to be extracted from the system in real-time, without this the human-robot collaboration project could not have functioned.

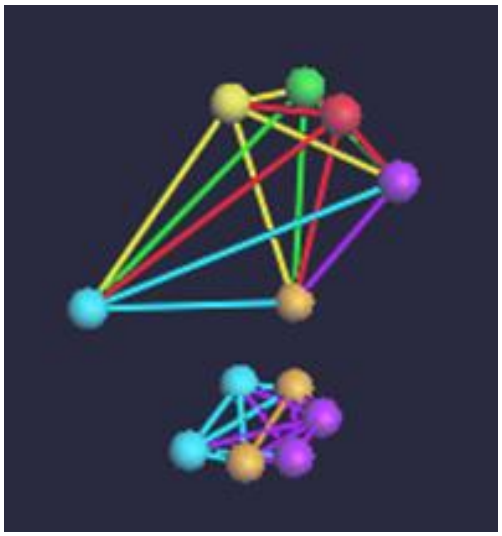

Figure 4.7 Vicon Sensor Glove Subject

4.2. RESULTS

The glove was tested in four different scenarios, sliding a board on a table, operating a drill, and manipulating a ball. These tests each show the accuracy and response time of the glove in several scenarios that demonstrate vastly different methods of

manipulation with the hand. Figure 4.8 shows a screenshot of the first scenario, sliding a board on a table.

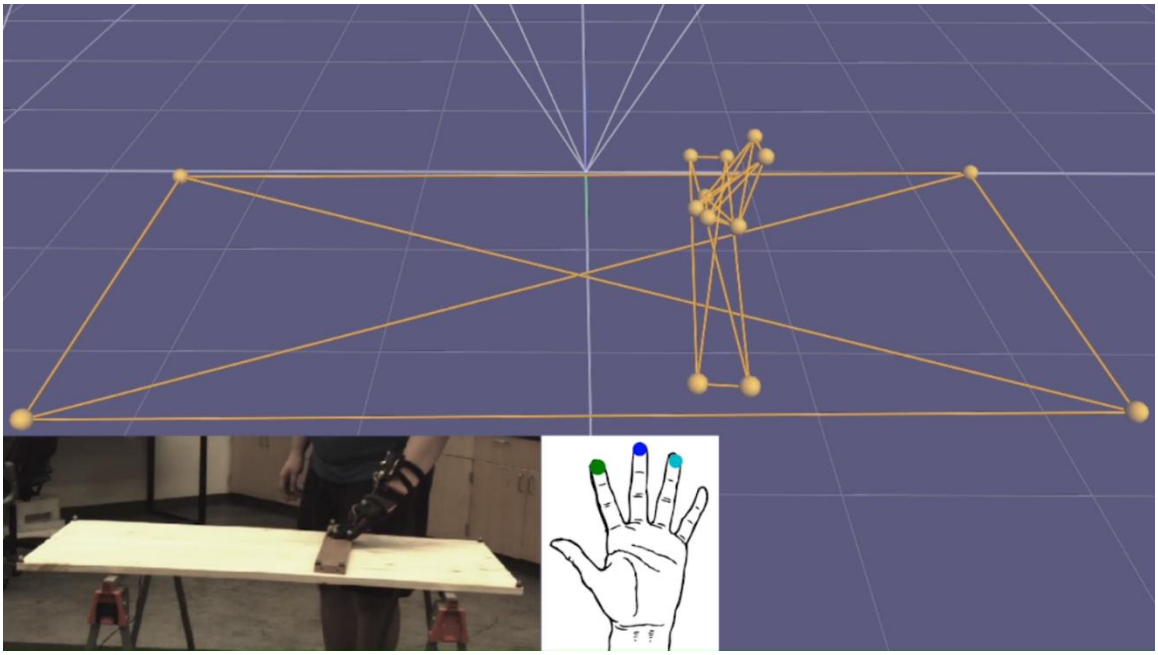

Figure 4.8 Sensor Glove Test Scenario – Sliding Board on Table

As Figure 4.8 shows these test scenarios show a real video of the test in the bottom left, the visualization from the Vicon system in the background, and lastly a visualization of the pressure on each finger represented by circles that grow and shrink on each fingertip. This demonstration shows the glove's ability to accurately and quickly digitize the state of the human hand. The next two scenarios shown in Figures 4.9 and 4.10 demonstrate the glove's ability to accurately mark each finger in scenarios that may cause occlusion or the Vicon system to mistake one finger for another.

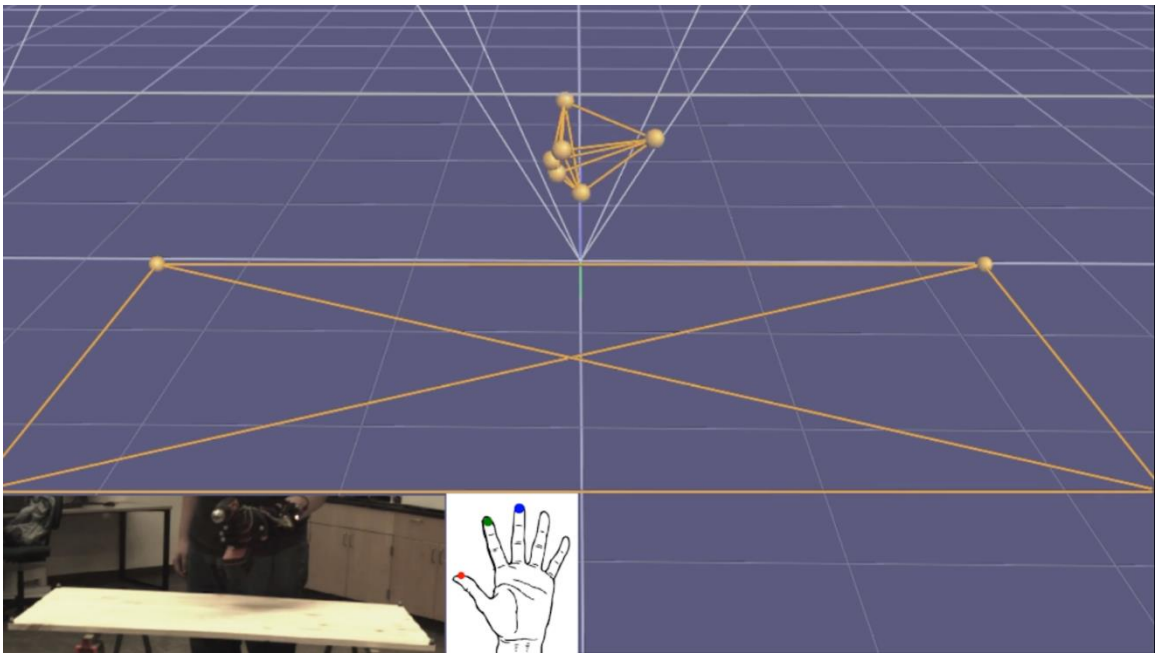

Figure 4.9 Sensor Glove Test Scenario – Drill

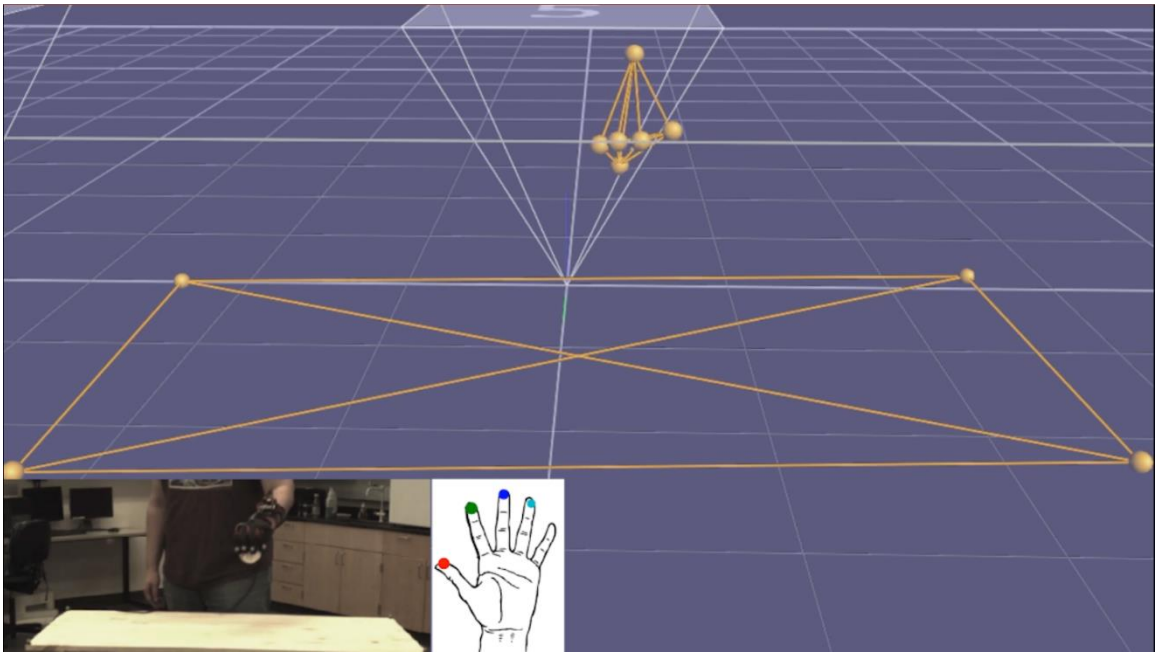

Figure 4.10 Sensor Glove Test Scenario - Ball

These scenarios demonstrate that the sensor glove is a robust system able to easily track the state of the human hand even in scenarios that are difficult for a motion capture system to track. This system is later used in Section 5 to determine human intent in the human-robot collaboration project.

4.3. SUMMARY

This section covered the development of a Bluetooth enabled sensor glove. This sensor glove can transmit the state of up to 6 resistance-based sensors at 100 Hz over Bluetooth while using a Vicon Motion Capture System to track the location of the fingers in space. The sensor glove was developed with the intent of being used in the human-robot collaboration system covered in Section 5 but it is believed that it could be used in future research where recording data about the human hand is required.

5. HUMAN-ROBOT COLLABORATION ON TASKS LEARNED THROUGH DEMONSTRATION

This system attempts to view a human collaborator working on a translation task over a desk with a kinesthetically controlled robot arm over several demonstrations and using imitation learning, learns to control the robot arm to complete the same translation task with the human in real-time. Figure 5.1 Provides a visual overview of the whole system.

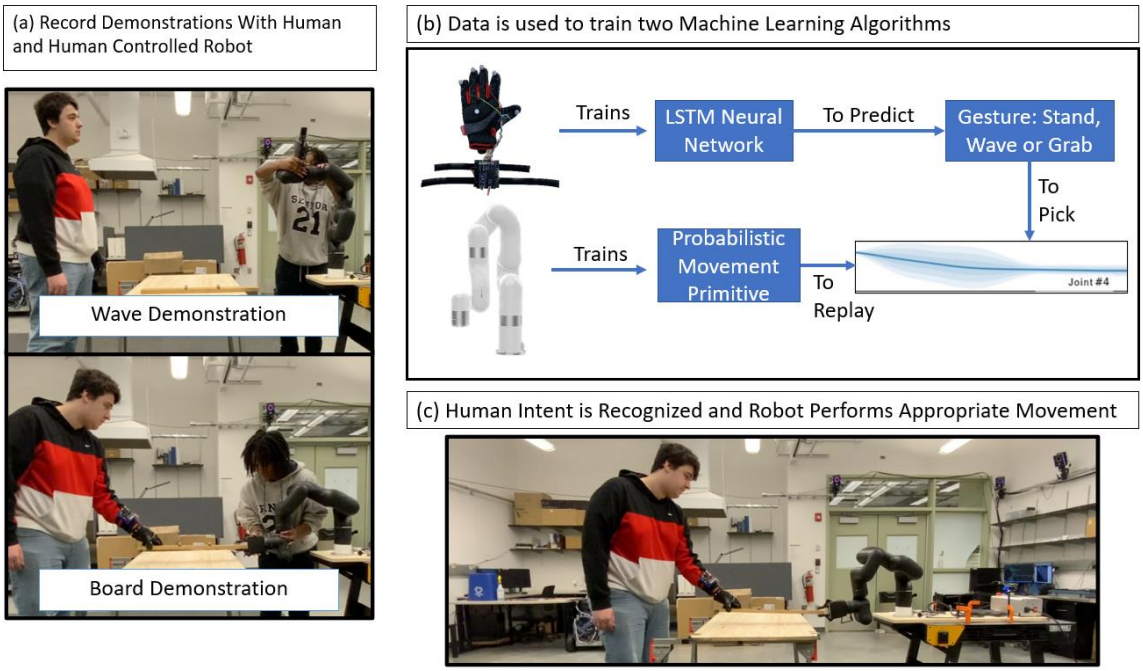

Figure 5.1 Human-Robot Collaboration System Overview

To track the human collaborator's movements the Bluetooth Sensor glove, from Section 4, is used. While running the system the sensor data from the glove as well as the position of each finger is transmitted over an ethernet cable, using the UDP protocol, to a workstation, referred to as the robot workstation from now on, running the xArm 6 control

policy. The system has two modes of operation that determine how the data being sent to the robot workstation is used. The first mode is demonstration recording mode. This mode collects demonstrations in order to train the system. When data from the sensor glove arrives at the robot workstation it is time-stamped and saved to a csv, comma-separated value, file. At the same time, a second independent script is recording the state of the robot arm and saves that to a csv. The robot and human data are kept separate because they are used to train two learning systems. The system uses the human data demonstrations to train a form of RNN, recurrent neural network, known as an LSTM, Long Short-Term Memory. The LSTM learns what movement the human is attempting to make by watching several demonstrations of each movement. For example, the LSTM can learn to recognize when a human is attempting to wave. Once the LSTM has recognized that a gesture is being performed it then tells the robot arm to perform the corresponding trajectory that was learned from demonstration. In the case of a human waving the LSTM would tell the robot arm to wave back in response. The trajectory that the robot has learned from several demonstrations is known as a movement primitive. A movement primitive allows for complex trajectories to be learned from demonstrations as well as reduces the number of commands required to describe a complex trajectory. For example, instead of having every joint position saved that describes a robot's wave there is instead a learned equation that can be used to describe the entire trajectory. The specific type of movement primitive used for this project is known as a ProMP, Probabilistic Movement Primitive, and is covered in further detail later in this section. Currently, the LSTM can decide from three total options no ProMP should be used at this moment, the wave ProMP should be used, or the pick and place board ProMP should be used. Once the LSTM's confidence that a particular gesture

is being performed exceeds 90% the corresponding gesture is performed in response on the robot arm. While this system is currently only limited to two response actions from the robot it is easily scalable to many more actions by recording a larger amount of training data.

5.1. METHODOLOGY

This section covers the two key learning algorithms fundamental to this project's success. The first learning algorithm is the human intent recognition LSTM, allowing for gestures to be identified from multiple sequences of human hand data. The second learning algorithm allows robot trajectories to be learned from a sequence of robot joint positions.

5.1.1. Human Intent Recognition LSTM. As stated earlier the human intent recognition LSTM is a form of RNN, recurrent neural network. From a high level, this means that previous inputs, as well as the current input data, affect the LSTM's prediction. The RNN structure is ideal for time series data like a sequence of time-stamped human hand data. Figure 5.2 shows the structure of an RNN over multiple predictions. As can be seen from the figure a recurrent neural network takes a Hidden input, H, from the previous iteration of the RNN and an input X representing the current state of the system. It uses both of these values in order to make a prediction which is the output Y. The simple RNN has an inherent disadvantage though. It treats all previous inputs as equal. This means that it quickly forgets events that happened in the past. This can be seen in equation 5.1 which describes the forward propagation function of the hidden layer of a simple RNN.

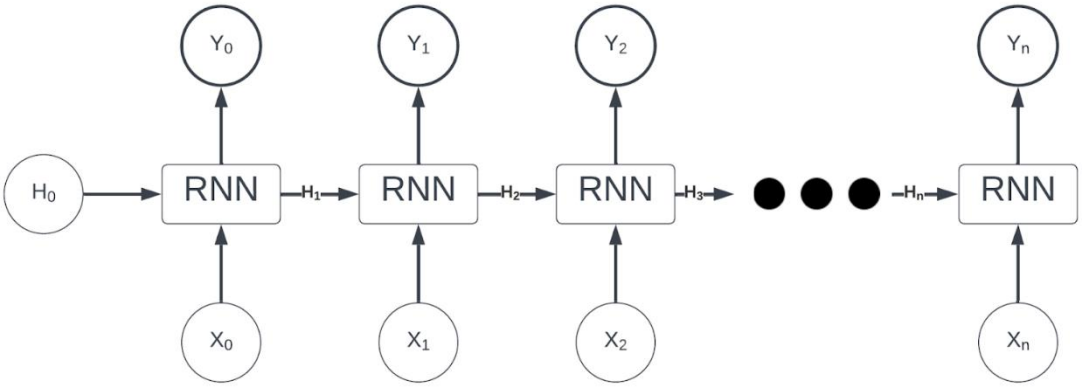

Figure 5.2 Recurrent Neural Network Structure Over Multiple Iterations

$$H^t = a(W_{hh}H^{t-1} + W_{hx}X^{t-1} + b_h) \quad (5.1)$$[1]

In equation 5.1 it shows that to determine the current hidden state H^t, an activation function is used on a set of three inputs. The previous hidden state H^{t-1} multiplied by its corresponding weights W_{hh}, the current input X^{t-1} multiplied by its corresponding weights, and the bias b_h. The flaw with the recurrent neural network can be seen within the forward propagation step for the previous hidden state $W_{hh}H^{t-1}$, this function does not have the ability to distinguish between important events that occurred previously in the time sequence. In effect, it is averaging all previous events and multiplying them by a set of learned weights. Therefore, the effect of an event that happened 200 timesteps ago has a significantly smaller impact on H^{t-1} than an event that just occurred 1 timestep ago. When recognizing human gestures this means that a simple RNN would quickly forget what had recently happened and therefore have a difficult time recognizing long gestures. The LSTM

[1] Equations on LSTM architecture adapted from [27]

makes up for this disadvantage through the use of its attention mechanism. The attention mechanism allows an LSTM to learn what are the most important data points that occur in a sequence and therefore give them a higher level of influence on the prediction of the network. This is done through a set of 6 equations describing the forward propagation of an LSTM. The first equation is Equation 5.2 which describes the memory cell.

$$c^t = \Gamma_u * \dot{c}^t + \Gamma_f * c^{t-1} \tag{5.2}$$

The memory cell, c^t, is a new value passed between iterations. This gives the LSTM the ability to choose what information to remember and what to forget. This is done by multiplying a new candidate memory cell, $\dot{c}^t$, with an update gate, Γ_u, and multiplying what was previously saved in memory, c^{t-1}, with a forget gate, Γ_f. Therefore, during every call to an LSTM, a decision is made as to whether the memory cell, c^t, should keep the same value or be replaced by the new candidate memory cell, $\dot{c}^t$. Equation 5.3 shows the computation for, $\dot{c}^t$.

$$\dot{c}^t = tanh(W_c[H^{t-1}, X^t] + b_c) \tag{5.3}$$

The candidate memory cell is computed using the previous hidden state and the current input which are multiplied by a set of learned weights W_c and have a bias b_c added to them before going through the tanh activation function. The update and forget gates are described in equations 5.4 and 5.5 respectively.

$$\Gamma_u = \sigma(W_u[H^{t-1}, X^t] + b_u) \tag{5.4}$$

$$\Gamma_f = \sigma(W_f[H^{t-1}, X^t] + b_f) \tag{5.5}$$

These gates have a very similar form to the candidate memory cell, but they have their own learned weight matrices, W_u and W_f, as well as their own learned biases, b_u and b_f. They

also pass through the sigmoid activation function, represented by σ, instead of tanh. After c^t the hidden layer output H^t is determined. Equation 5.6 shows how H^t is calculated.

$$H^t = \Gamma_o * tanh(c^t) \tag{5.6}$$

The hidden layer output is computed by the memory cell, c^t, which is multiplied by an output gate, Γ_o. Equation 5.7 describes the output gate.

$$\Gamma_o = \sigma(W_o[H^{t-1}, X^t] + b_o) \tag{5.7}$$

The output gate is structured in the same fashion as the previous gates in equations 5.4 and 5.5 with its own learned weight matrix, W_o and bias, b_o. Finally, once the H^t is determined it can be passed through any activation function with its own weights and bias to generate the output Y^t. Figure 5.3 is a diagram describing the function of an LSTM in a graphical manner as opposed to the equations just covered.

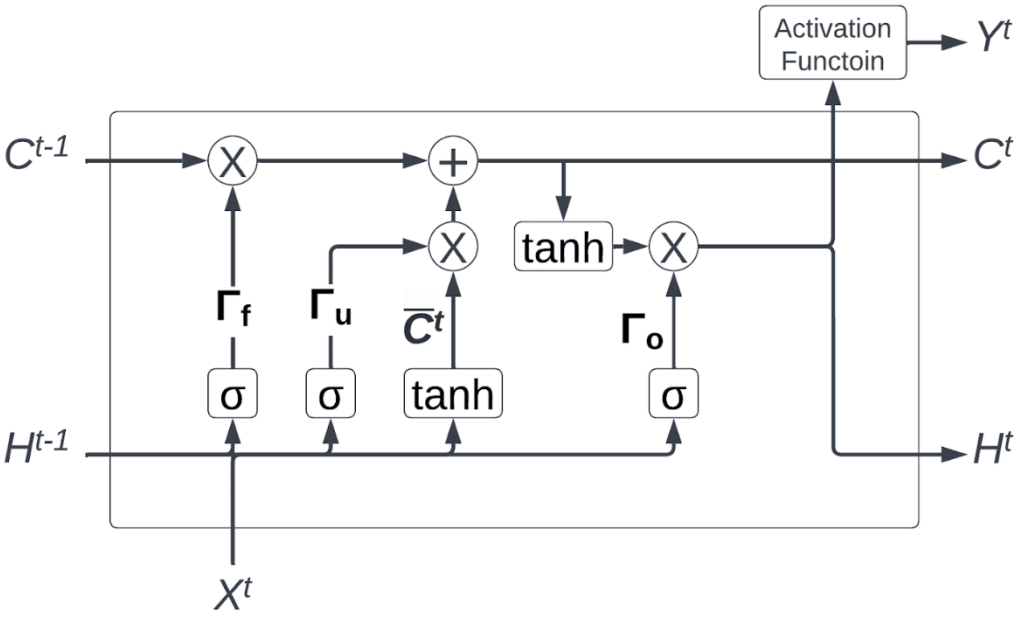

Figure 5.3 LSTM Layer Diagram

The form of LSTM made for this project is many-to-many which means that for every timestep the LSTM produces a prediction. This allows the gesture prediction to be updated rapidly. The input into the network is a set of 21 input variables describing the state of the human user's hand. This includes 5 (X, Y, Z) coordinates describing the position of the fingers in space, 5 pressure readings, and one timestamp (describing the time elapsed since the value was received). Figure 5.4 is a diagram showing all of the input variables and where they come from on the glove. The output of the network is a set of 3 percentage point values that add up to 100. These describe the confidence level that the human gesture being performed is either no gesture, a wave, or a grab. Figure 5.5 is a diagram showing the output from the network.

Figure 5.4 Human Intent Recognition LSTM Input

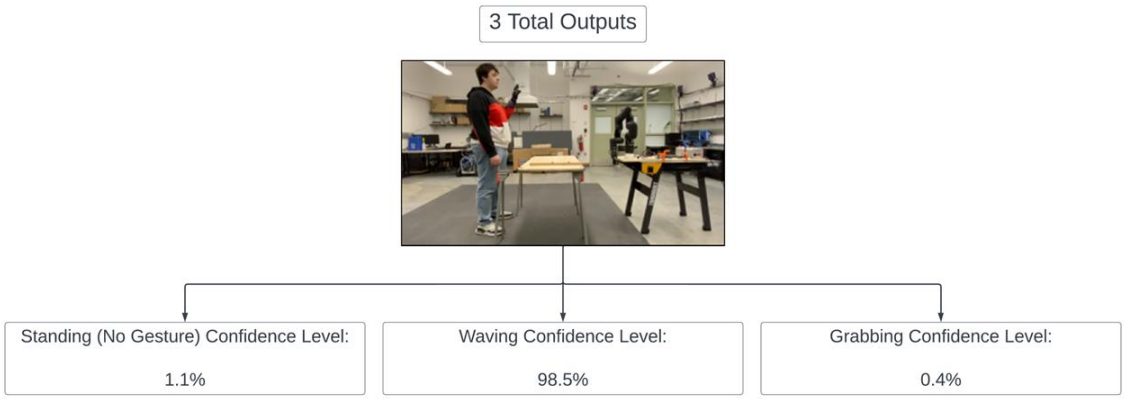

Figure 5.5 Human Intent Recognition LSTM Output

The architecture of the network is a 6-layer stacked LSTM with a dense output layer and SoftMax activation function. In total, the network has 183,155 trainable parameters. Figure 5.6 is a diagram of the network architecture used for the project. The size and depth of this network was large enough to recognize the long-range dependencies required to recognize a gesture while not being so large that it overfits to the training set. In order to further prevent overfitting, the first LSTM layer has a 20% dropout rate while the next 5 LSTM layers have a 30% dropout rate. Dropout randomly turns the weight of a specified percentage of neurons to 0. This prevents a neural network from learning the training set too specifically which causes a network to have a hard time generalizing to new inputs.

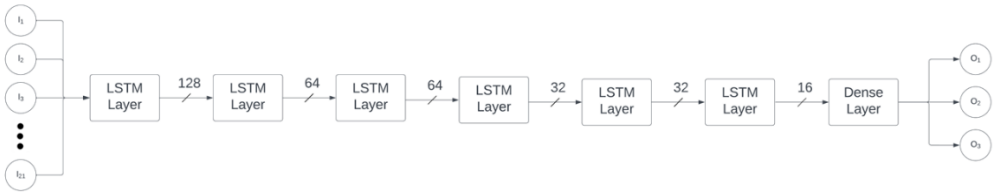

Figure 5.6 Human Intent Recognition LSTM Architecture

The human intent recognition LSTM was trained using the Categorical Cross entropy loss function and the Adam optimizer with stochastic gradient descent. Categorical cross-entropy is a loss function used when you have a set of one-hot encoded categories to choose from. Equation 5.8 shows the math describing the loss function.

$$CE = -\sum_{i=0}^{n} groundTruth_i * log(prediction_i) \quad (5.8)$$

The Adam optimizer combines rmsPROP and momentum optimizers to create an optimizer that can quickly move down shallow portions of the loss surface but can also slow down the learning rate to ensure that a minimum is not overshot. Lastly, Stochastic gradient descent is batch gradient descent with a batch size of 1. This causes the movement across the loss surface to be erratic and it can overshoot the global minimum. This is because when calculating the gradient only one sample is used which does not accurately represent the loss surface. Figure 5.7 shows the difference between batch and stochastic gradient descent. This is used because the sequences the neural network is training on are variable in length. Padding the sequences to be the same length can be used to overcome this but it does change the statistical distribution of the data, so it was not chosen as the solution.

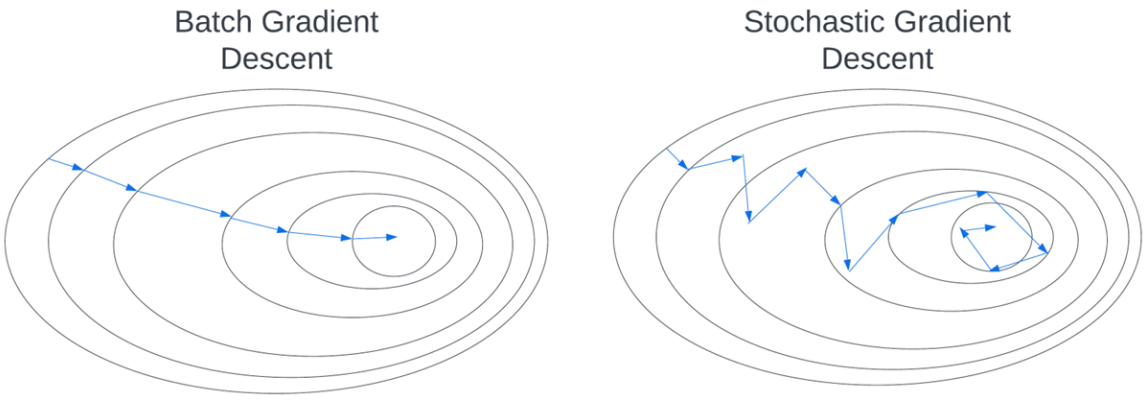

Figure 5.7 Batch Gradient Descent vs. Stochastic Gradient Descent on Loss Surface

5.1.2. Probabilistic Movement Primitives. Probabilistic Movement Primitives, ProMPs, enable the robot policy to be encoded by watching a series of demonstrations. The robot policy is the strategy the robot uses in pursuit of its goals. In this case, that means the trajectory the robot uses in response to a certain gesture performed by the human. ProMPs represent the trajectory as a distribution over the set of all demonstrated trajectories. Figure 5.8 shows a graph of the trajectory distribution learned on each joint of the xArm 6 after learning from a series of wave demonstrations. ProMPs are used in this project because they achieve 3 functions: reduction of the action space for the LSTM, issuing high-frequency commands to the xArm 6, and the ability to modulate movements to the specific scenario.

The first and most important feature of ProMPs is the ability to reduce the action space of the human-intent recognition LSTM. The action space is the n-dimensional space describing the output of a neural network, where n is the number of output values. Currently, the LSTM has a finite action space of 3. The 3 actions are stand, wave, and grab. This is only possible because 2 ProMPs describe all of the joint movements that make up the wave and grab robot trajectories. If a ProMP was not used the LSTM would have to describe the exact position all 7 degrees of freedom (6 joints and 1 end effector) must move to as the output of every call to the LSTM. This makes the action space of the LSTM, 7 highly sensitive floating-point values. They are sensitive because sending a value outside of the expected range could cause the xArm 6 to perform in dangerous ways, potentially damaging itself, or knocking over items in the environment. Another factor is that the current position would be coupled to the previous position since the LSTM saves the state of previous predictions. Therefore, errors in the LSTM prediction could compound over

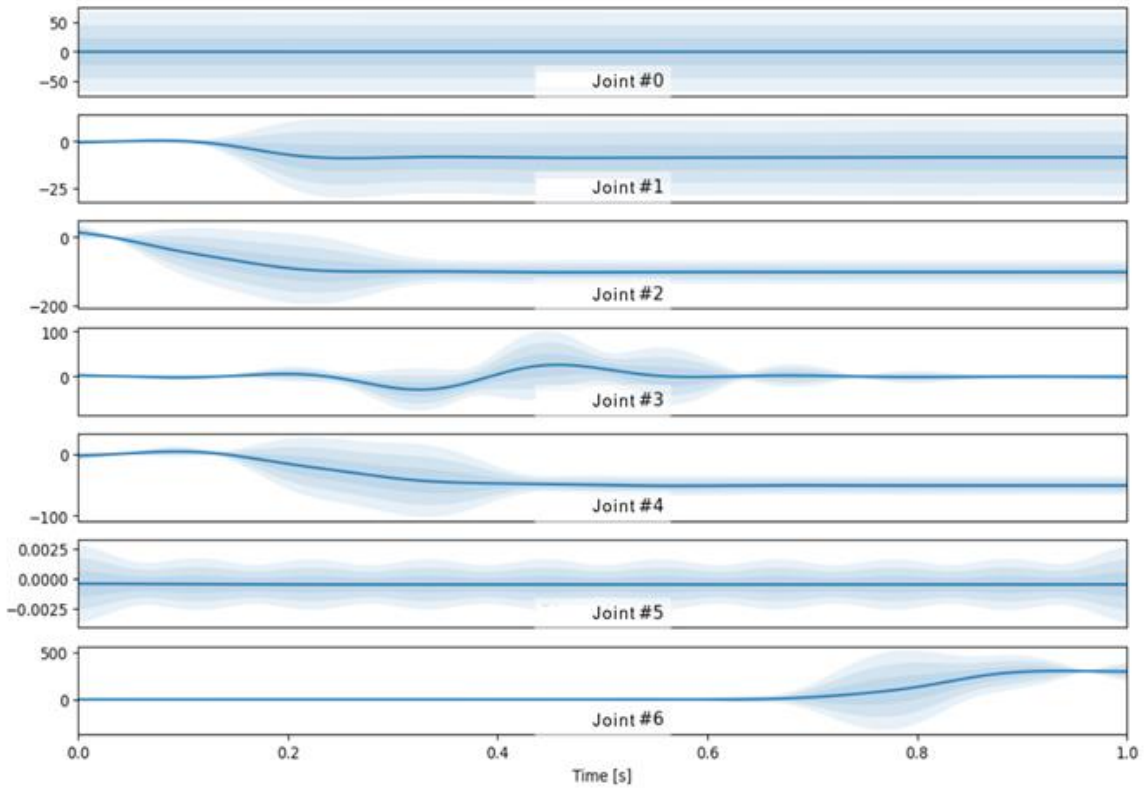

Figure 5.8 Trajectory Distribution of Wave on Each Joint of the xArm6 Learned from a set of Demonstrations

time causing increasing error. This means that the LSTM must be precise when it is guessing a response movement to the human collaborator with no room for error. The ProMP overcomes these issues by describing the trajectory as a function that can be queried for a precise position to move each joint, keeping all responses in a safe working area with no chance of compound errors.

The second justification for using ProMPs is related to a frequency constraint caused by the collection of the Human data. After optimization, the state of the human data can only be gathered every 20 ms. This is due to the fact that receiving a Bluetooth transmission on the state of the pressure sensors takes 10 ms and receiving the state of the

markers takes 10 ms as well. Both of these actions are blocking and therefore require 20 ms for both to be completed. If both actions could be computed in parallel, it may be possible to get the response time down to 10 but this is still not fast enough. The xArm 6 takes new joint position commands at a rate of 250 Hz, which means a response from the LSTM must arrive in 4 ms. With current data collection restraints, this is not possible to overcome. Achieving a 4 ms response time for the LSTM may also not be ideal for future scalability. If the network were scaled to recognize a large library of gestures at some point it is feasible that the processing time for the LSTM exceeds 4 ms. ProMPs can respond instantaneously because they generate all of the points the x Arm 6 will move along when they are queried initially. This means that no processing time is required between each x Arm 6 joint command which makes implementation easier and more scalable. A robot response that would take 10,000 timesteps has the same inter-command computational overhead as a response that takes 100,000 timesteps.

Lastly, ProMPs allow for the trajectory to be modulated in two key ways. These include temporal modulation and waypoint conditioning. Temporal modulation allows the total execution time of a trajectory to be sped up or slowed down by a coefficient known as the phase variable. Waypoint conditioning allows for a trajectory to arrive at a specific point within the learned distribution. For example, if the xArm 6 was trained to perform a wave it could be conditioned to perform that same wave facing 45° to the left. Currently, ProMP conditioning is not part of the robot-collaboration framework because far more demonstrations must be collected for this to work effectively. The functionality can be added to the network with modifications to the final layer of the network but without an

adequate amount of demonstration data, a model cannot produce an accurate enough waypoint or temporal modulation coefficient for human-robot collaboration.

ProMPs are implemented using a HBM, Hierarchical Bayesian Model, which allows for a distribution to be extracted from a set of trajectories. Figure 5.9 shows the HBM.

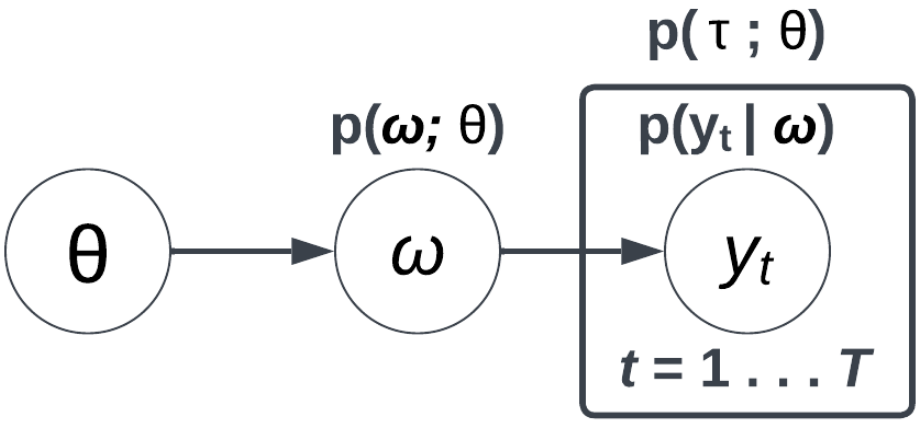

Figure 5.9 Hierarchical Bayesian Model Used by ProMPs[2]

A trajectory is described as a set of joint angles over time, $\tau=\{q_t\}_{t=0...T}$. A weight vector, ω, is multiplied with a basis function vector $\boldsymbol{\Phi}_t = [\varphi_t, \dot{\varphi}_t]$ to produce a trajectory, y_t, shown in equation 5.9.

$$y_t = [q_t, \dot{q}_t] = \boldsymbol{\Phi}_t \, \omega \tag{5.9}$$

The variables q_t, $\dot{q}_t$ represent the angle and angular velocity at time t respectively. A similar relationship holds for variables φ_t, $\dot{\varphi}_t$ which represent the time dependent basis

[2] Figures and Equations on ProMP implementation are adapted from [6]

function for the angle and its derivative representing angular velocity respectively. This equation, y_t, is used with ω, and the variance of y, Σ_y, to create the probability of observing a trajectory τ. This is shown in equation 5.10.

$$p(\tau|\omega) = \prod_t \mathcal{N}(y_t|\boldsymbol{\Phi_t}\ \omega, \Sigma_y) \tag{5.10}$$

A distribution $p(\omega;\ \theta)$ is defined to show the variance of the trajectories where $\theta = \{\mu_\omega, \Sigma_\omega\}$ which are the mean and variance of ω respectively. Finally using the previously defined equations the probability distribution over the trajectories $\boldsymbol{\tau}$ can be defined in equation 5.11.

$$p(\tau;\ \theta)\ =\ \int p(\tau|\omega)p(\omega;\ \theta)d\omega \tag{5.11}$$

Using learning methods, the weight vector, ω, can be fit so the probability distribution better matches the set of trajectories.

The basis function vector, $\boldsymbol{\Phi}_t$, described earlier can represent any basis function that the user desires. The implementation used by the human-robot collaboration framework is a Gaussian radial basis function. This basis function has the form shown in equation 5.12.

$$\varphi_t = e^{-\varepsilon\|t-t_i\|^2} \tag{5.12}$$

Where ε is a parameter to scale the input of the radial basis function and t_i is a value a fixed distance from the current timestep t. Radial basis functions are effective at approximating complex functions because they are capable of computing infinite interactions between the input variables.

5.2. IMPLEMENTATION

The Human-Robot collaboration framework was implemented using a combination of Python and C++. The human data workstation is a Windows 10 PC and all scripts used on the device were written in Python. The robot data workstation is a Linux PC. A majority of the scripts written on this device were written in Python but scripts with a tight time constraint, specifically the robot trajectory recording script were written in C++. All tasks took place over a desk with the human operator on one side and xArm 6 on the other. For the remainder of this section, all references to code will be accompanied by pseudo-code. The actual code used to implement this project can all be found in the appendix.

5.2.1. Experimental Setup. The experimental setup takes place over a desk with the human user on one side and the robot collaborator on the other. The experimental workspace is surrounded by a set of 10 Vicon motion tracking cameras. Figure 5.10 shows a diagram of the experimental setup. The cameras allow for total coverage of the workspace but when a marker gets occluded software solutions were programmed to make up for these problems. This is covered in Section 5.2.2 Data Collection.

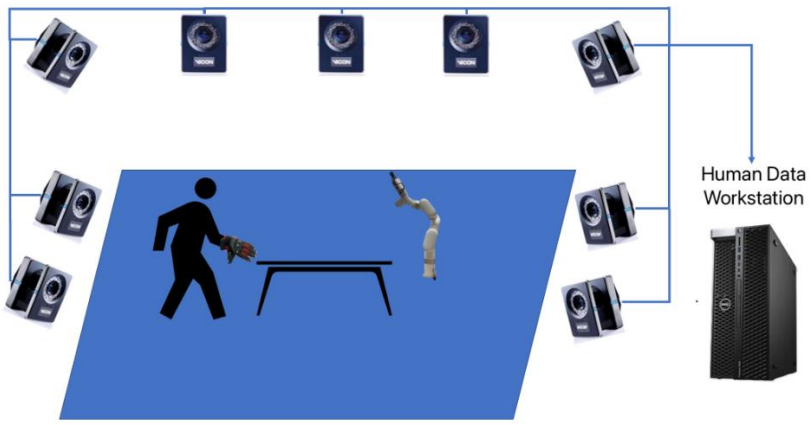

Figure 5.10 Experimental Setup Design

5.2.2. Data Collection. In order for the human-robot collaboration system to function, sensor glove data must be collected on the human data workstation and transmitted over an ethernet cable to the robot data workstation where it is saved alongside the robot data as a demonstration to train the human-intent recognition LSTM as well as a ProMP for the movement being performed on the robot arm. The collection of data and transmission to the robot workstation is an identical operation regardless of if the system is saving demonstrations to memory or the system is performing real-time human-robot collaboration. This section focuses on the collection of data on the human-data workstation, transmission to the robot workstation, and lastly the saving of those demonstrations to memory. Information on how the robot workstation operates when performing real-time human-robot collaboration can be found in Section 5.2.5.

The human-data workstation collects data from two sources: the Vicon system, and pressure sensors on the glove. Algorithm 5.1 shows pseudocode explaining the initialization, data collection, and transmission process for the human-data workstation.

```
Output: HumanData sent to RobotWorkstationSocket
ViconClient ← InitVicon();
BlueToothSocket ← InitBlueTooth();
RobotWorkstationSocket ← Connect(RobotWorkstationIP);
while ViconClient is connected do
    GloveMarkerState ← ReadVicon(ViconClient);
    BlueToothMessage ← ReadBlueTooth(BlueToothSocket);
    Data ← Concatenate(GloveMarkerState, BlueToothMessage);
    RobotWorkstationSocket ← SendData(Data);
end
```

Algorithm 5.1 Human Data Workstation Client

As can be seen in the pseudocode for algorithm 5.1 the human-data workstation initializes the Vicon system, Bluetooth connection, and robot workstation connection in lines 1-3. After this, the human data workstation collects data on the glove markers from the Vicon system and the state of the sensors from the Bluetooth connection. It then concatenates that data together and transmits it to the robot workstation. Algorithm 5.2 shows the initialization of the Vicon system from line 1 of Algorithm 5.1.

```
   Output: ViconClient and CoordOrigin
1  ViconClient ← Connect(ViconAddress);
2  TableMarkers ← GetMarkers(ViconClient, Table);
3  CoordOrigin [x] = (TableMarkers [1,x] + TableMarkers [2,x]) / 2;
4  CoordOrigin [y] = (TableMarkers [3,y] + TableMarkers [4,y]) / 2;
5  CoordOrigin [z] = (TableMarkers [1,z] + TableMarkers [2,z] +
     TableMarkers [3,z] + TableMarkers [4,z]) / 4;
6  Return ViconClient, CoordOrigin
```

Algorithm 5.2 Initialize Vicon

In Algorithm 5.2 the Vicon client is connected to the workstation, before the markers representing the four corners of the table the human and robot collaborators are working over are gathered. The four markers are then used to determine an origin point in the center of the table in lines 3-5. This is a pre-processing step to produce consistent coordinates between uses of the human-robot collaboration framework. This is done by shifting the coordinates of the markers on the sensor glove to be relative to the origin point at the center of the table. By performing this step, it becomes easier for the human-intent LSTM to recognize gestures.

As discussed in Section 4.1.3 the sensor glove has onboard software for transmitting the sensor data on the device when connected to the host pc. In the case of the human-robot collaboration system, the host device is the human-data workstation. Algorithm 5.3 shows how this connection to the sensor glove is initialized (this initialization occurs on line 2 of Algorithm 5.1).

```
   Output: BlueToothSocket
1  Devices ← DiscoverDevices();
2  for device in Devices do
3  |  if device name == Pressure Glove then
4  |  |  BlueToothSocket ← Connect(device address);
5  end
6  SendData('1', BlueToothSocket) // Sending a 1 to the BlueTooth
       Glove initializes data transmission
7  ;
8  Return BlueToothSocket
```

Algorithm 5.3 Initialize Bluetooth

The last initialization step for the human data workstation is to establish a connection to the robot workstation. This is done by simply listening for an initialization message from the IP associated with the robot workstation. Once this message is received the data begins being read and transmitted to the robot workstation.

Data is read and transmitted from the human-data workstation every time a request is received from the robot workstation. Algorithm 5.4 Shows how data is read from the Vicon system.

```
Input  : ViconClient, CoordOrigin
Output: GloveMarkerState
1 Subjects ← GetSubjects(ViconClient);
2 GloveMarkerState ← Get(Glove Subject from Subjects)
3 for marker in GloveMarkerState do
4 |   if marker is occluded then
5 |   |   marker = previous marker position + (moving average of marker
  |   |    velocity)x(TimeSinceLastRead);
6 end
7 Return GloveMarkerState;
```

Algorithm 5.4 Read Vicon

The Vicon reading algorithm reads the state of the markers on the glove. If it is the case that a marker is occluded, the previous position of the occluded marker is updated with the moving average of the marker velocity multiplied by the time since the last data read was performed. This prevents demonstration data from having missing values while providing a reasonable estimate of the marker's position. After the Vicon system is read the sensor glove is read over Bluetooth. This is covered in Algorithm 5.5.

```
Input  : BlueToothSocket, TimeSinceLastRead
Output: BlueToothMessage
// BlueTooth messages of MessageLength Bytes are transmitted
   every 10 ms, therefore to get the most recent message you
   must read (TimeSinceLastRead/ 10 ms) + 1 messages out of
   the buffer
1 Buffer ← BlueToothRead(MessageLength x ((TimeSinceLastRead/ 10
   ms) + 1 ), BlueToothSocket);
2 BlueToothMessage ← GrabLastMessage(Buffer);
3 Return BlueToothMessage;
```

Algorithm 5.5 Read Bluetooth

Messages from the sensor glove are transmitted every 10 ms which fills the buffer on the human-data workstation with messages. To get the most recent message this means that a whole message must be read and discarded for every 10 ms that have passed since the last message was read from the buffer. This is implemented on lines 1 and 2 in Algorithm 5.5.

Once data is transmitted to the robot workstation one of two scripts will be run. This section covers the Data collection server which collects data from the xArm 6, and the human-data workstation before saving them as a demonstration for training the LSTM and ProMP. Algorithm 5.6 shows how the Data Collection Server functions.

```
   Output: HumanDataFile and RobotDataFile
1  Connect(xArmIP);
2  Connect(HumanDataWorkstationIP);
3  Wait until user presses enter;
   // Records robot trajectory to RobotDataFile
4  OpenSubprocess(RecordTrajectory());
5  while User has not pressed q do
6  |  HumanDataFile ← WriteToFile(RequestHumanData());
7  end
```

Algorithm 5.6 Data Collection Server

After connecting to the xArm 6 and human-data workstation the data collection server waits for the user to press enter before starting. Once this has occurred a subprocess known as record trajectory is opened. This is a program written in C++ that reads and records the state of the xArm 6 at 250 Hz. After this subprocess is started data is requested from the human-data workstation before being written to a file. Once the user presses ‘q’

the data collection for the xArm 6 and human data end. Record trajectory had to be written as a subprocess because requesting human data is blocking and takes on average 20 ms to complete, 5 times slower than the 4 ms response time that the x Arm 6 recording needs to achieve. Algorithm 5.7 shows how the record trajectory script functions.

```
Input  : xArmIP
Output: xArm6 Trajectory File
Connect(xArmIP);
while User has not pressed q do
    TrajFile ← WriteToFile(ReadJointPositions());
    Wait until 4 ms elapsed;
end
```

Algorithm 5.7 Record Trajectory

Once a set of demonstrations have been collected the human data needs to be labeled before it is ready to train the LSTM. This is done through a demonstration labeling script where the user pans through a visualization of the human demonstration data and marks what gesture was performed as well as on what frames said gesture began and ended. Algorithm 5.8 shows how the demonstration labeling script functions.

5.2.3. Training the LSTM. The human-intent recognition LSTM was implemented using TensorFlow 2.0 and the sequential Keras API. Algorithm 5.9 covers the process of creating and training the LSTM. After data has been loaded an initial model is created with the Keras API known as the stateless model. This is a model that does not save the internal state of the LSTM between calls to the LSTM.

```
Input  : Unlabeled Human Demonstration
Output: Human Demonstration Labels
Gesture ← ReadUserInput(Stand, Wave, Grab);
StartFrame ← ReadUserInput(0 – FrameCount);
EndFrame ← ReadUserInput(StartFrame – FrameCount);
for i ← FrameCount do
    if 0 ≤ i < StartFrame then
        LabelsFile ← WriteToFile(Stand);
    else if StartFrame ≤ i < EndFrame then
        LabelsFile ← WriteToFile(Gesture);
    else
        LabelsFile ← WriteToFile(Stand);
end
```

Algorithm 5.8 Demonstration Labeling

```
Input  : Dataset of all human data demonstrations
Output: LSTM able to predict human intent
TrainingData ← LoadDataset();
// Create stateless model with same architecture as LSTM
   Figure
StatelessModel ← CreateModel(Using CategoricalCE loss and ADAM
   optimizer);
StatelessModel ← TrainModel(Using TrainingData for 10 epochs with a
   batch size of 1);
GraphLoss(StatelessModel);
// Create stateful model with same architecture as LSTM
   Figure
StatefulModel ← CreateModel(Using CategoricalCE loss and ADAM
   optimizer);
StatefulModel ← ApplyWeights(StatelessModel);
LiteModel ← TFLiteConversion(StatelessModel);
SaveModel(LiteModel);
```

Algorithm 5.9 LSTM Model Creation and Training

Therefore, in order to use this model an entire sequence must be provided to it. A model of this form is only useful for training because in real-time operation the model will get a single sample every 20 ms. After training for 10 epochs and graphing the training loss (shown in Section 5.3) a new model known as the stateful model is created with the Keras sequential API. This model has an identical architecture to the stateful model, but its state is saved between calls to the model therefore you can provide one sample at a time. The weights from the trained stateless model are then transferred to the stateful model. Lastly, the stateful model is converted to a TensorFlow Lite model before being saved. TensorFlow Lite models have optimizations for running models on weaker devices such as cell phones. The reason that it is used in this scenario is that it causes the model to produce predictions in under 1 ms on the robot workstation making the overhead insignificant. Lastly, the model is saved to memory for later use in real-time operation.

5.2.4. Training the ProMPs. ProMPs were trained with the publicly available Movement Primitives library [12] for Python. This library allows for ProMPs to be trained from a set of trajectories, conditioned on positions, and have their execution time modulated. Algorithm 5.10 covers the ProMP training script created for the human-robot collaboration system. For the two possible responses from the xArm 6 a ProMP was trained. These were a wave ProMP and a grab ProMP. From a computing resources perspective, this is the most demanding portion of this project as training these movement primitives took upwards of two hours and 70 gigabytes of RAM. Once the ProMPs are trained the resources needed to utilize them do become computationally insignificant.

```
Input  : Directory of Robot Trajectory Samples
Output: ProMP Trained on Samples
Samples ← LoadDirectory(SampleDirectory);
for Sample in Samples do
    if length of Sample < maxLengthSample then
        pad Sample to length of maxLengthSample;
end
Times ← array from 0.000 to 0.004*maxLengthSample;
ProMP ← trainProMP(Times, WeightsPerDimension, Samples);
PlotProMP(ProMP);
SaveFile(ProMP);
```

Algorithm 5.10 ProMP Trainer

5.2.5. Real-Time Human-Robot Collaboration. As stated in section 5.2.2, real-time human-robot collaboration and data collection operate identically in regard to operation on the human-data workstation. When running in real-time mode the robot workstation is running a real-time server script. The Pseudo code for this script can be found in algorithm 5.11.

```
Output: Robot movement in response to human collaborator
Connect(xArmIP);
Connect(HumanDataWorkstationIP);
Wait until user presses enter;
while User has not pressed q do
    HumanState ← RequestHumanData();
    RobotPrediction ← LSTMPredict(HumanState);
    if CurProcess ended or No CurProcess then
        if RobotPrediction == Wave then
            CurProcess ← OpenSubprocess(ReplayProMP(Wave));
        else if RobotPrediction == Grab then
            CurProcess ← OpenSubprocess(ReplayProMP(Grab));
end
```

Algorithm 5.11 Real-Time Server

Similar to algorithm 5.6 the real-time server script establishes a connection to the xArm 6 and human data workstation and then waits for the user to press enter before beginning operation. Once enter has been pressed data is requested from the human data workstation. The human intent recognition LSTM saved to memory in algorithm 5.9 is then used to make a prediction using the data received from the human data workstation. If the prediction confidence that a wave or grab is being performed exceeds 90% and no gesture is currently being performed, then it performs the associated robot response by opening the replay ProMP script as a subprocess. This loops until the user ends the program by pressing ‘q’. The reason that the replay ProMP script must be a subprocess is related to achieving a 250 Hz frequency for controlling the xArm 6 because requesting data from the human-data workstation blocks for 20 ms. Algorithm 5.12 covers how the replay ProMP subprocess functions.

```
Input  : ProMP File and Optional Final Waypoint
Output: ProMP Trajectory Replayed on xArm 6
Connect(xArmIP);
ProMP ← OpenFile(ProMPFile);
if FinalWaypoint provided then
    ProMP ← Condition(ProMP, FinalWaypoint);
MoveXArm(InitialPos);
while ProMP trajectory not complete do
    MoveXArm(GetNextPos(ProMP()));
    Wait untl 4 ms have passed;
end
```

Algorithm 5.12 Replay ProMP

The replay ProMP subprocess is straightforward. The script connects to the xArm 6 and the ProMP that was requested as an input parameter is loaded. If the user has supplied a final waypoint to condition the ProMP to arrive at then it is conditioned on that position. Once this is complete the xArm 6 is moved to the initial position of the trajectory before a command is sent every 4 ms until the trajectory completes.

5.3. RESULTS

For each human-robot collaboration task 20 demonstrations were recorded in order to train the LSTM and ProMPs. Figure 5.11 shows an overview of this process. After training the human-intent LSTM was able to achieve a training loss of 0.0334. This is shown graphically in Figure 5.12.

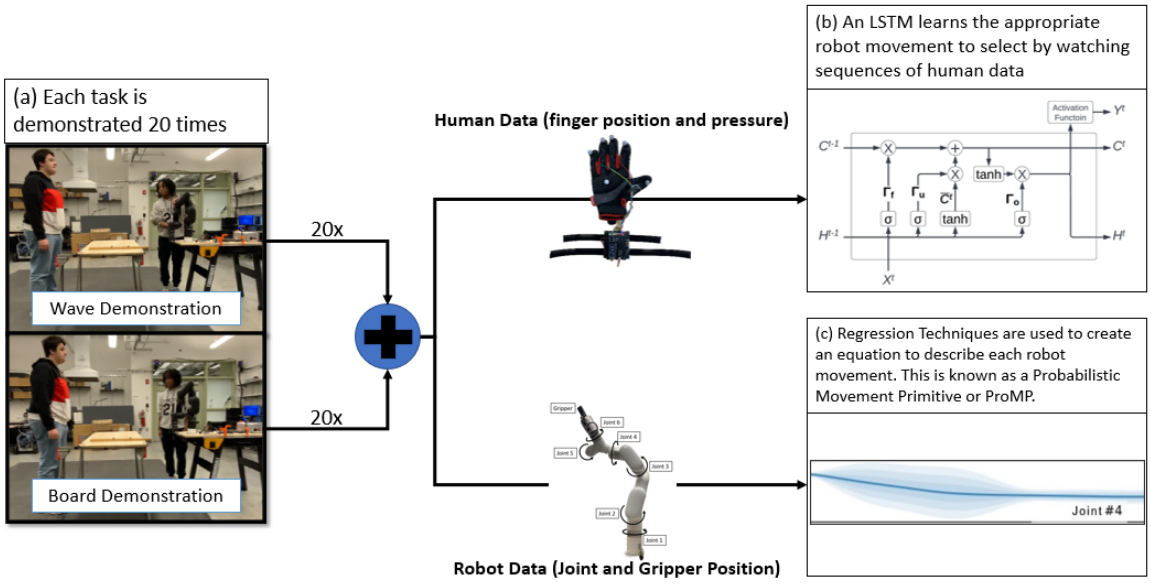

Figure 5.11 Data Collection and Training Process

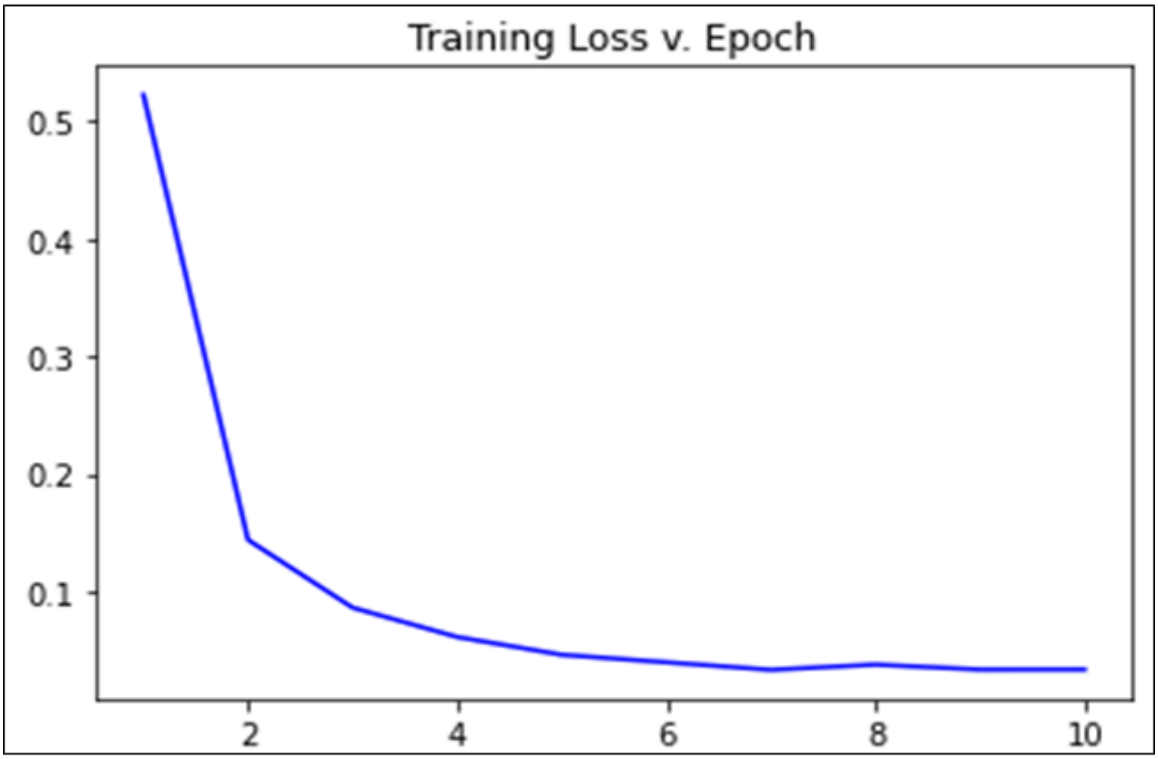

Figure 5.12 Training Loss vs. Epoch for Human-Intent Recognition LSTM

This translated to accurate real-time results. Standing, waving, or grabbing was detected with high accuracy when a gesture began, and this prediction accuracy was sustained throughout the entire movement. Figures 5.13 and 5.14 show the process for detecting and responding to a wave and grab respectively.

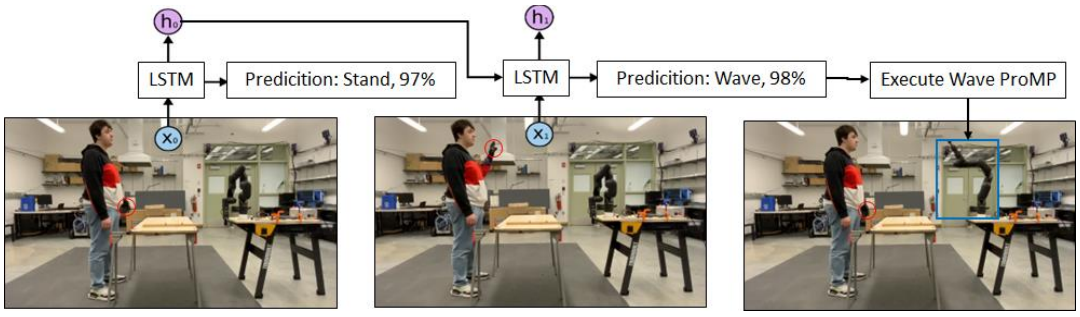

Figure 5.13 Human-Robot Collaboration Real-Time Wave Response

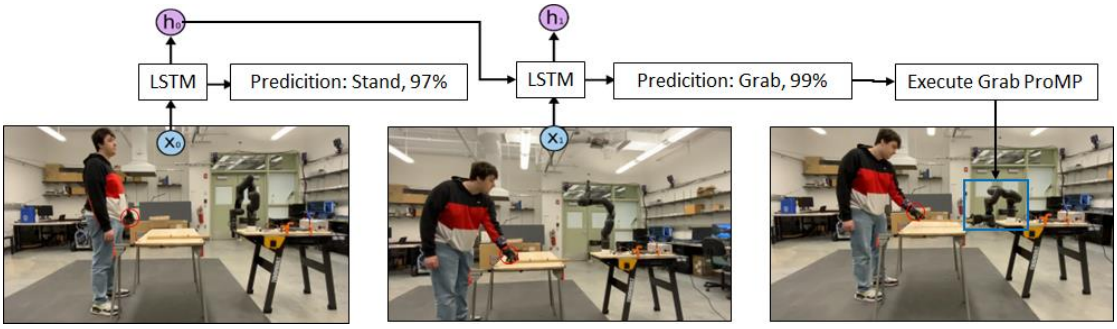

Figure 5.14 Human-Robot Collaboration Real-Time Grab Response

5.4. SUMMARY

This section showed a robust system for training a robot arm to function in a human-robot collaboration setting. By recording as little as 20 example demonstrations an LSTM can be trained through imitation learning to recognize the intent of a human collaborator. Once the human collaborators intent is recognized a probabilistic movement primitive is then used to respond with the appropriate trajectory on the robot arm. This allows a robot arm to assist a human in tasks such as picking up and placing a board down and waving in response to the collaborator.

6. CONCLUSION

6.1. SUMMARY OF WORK

This work incorporated novel sensors for reading the state of human health and motion intent into two real-time computing settings. The first of these settings was a sensor graphing application that could record and graph the state of up to 3 sensors in real-time. This was used to read the Covid-19 status of 85 patients in a hospital wearing a n95 mask with a sensor embedded inside. The second setting saw a novel pressure sensor incorporated into a glove to read the pressure on each finger. This glove was also able to read the position of the fingers in space with the use of the Vicon Motion Capture System. The glove was used to collect data to infer the gesture a human was performing with an LSTM, Long Short-Term Memory, neural network in a human-robot collaboration system. Once a gesture was inferred a Probabilistic Movement Primitive, ProMP, was used to perform a trajectory in response to the human collaborator.

6.2. NOVELTY OF RESEARCH

This first system in this work combined an n95 facemask with an embedded Covid-19 sensor allowing for the state of the wearer to be viewed in real-time and saved on a smartphone. This sensor application was also generalizable and could be used with any type of sensor.

The human-robot collaboration framework utilized a stacked LSTM to recognize human motion intent in real-time and select a Probabilistic Movement Primitive, ProMP, to use for the robot response. The human-intent recognition LSTM and ProMP were both

trained using 20 demonstrations per task. The combination of these two algorithms creates a system that can be scaled to many more tasks. This system was also made possible by a custom cost-effective glove for reading the state of the human hand with Bluetooth. This gloves ability to read pressure on fingertips provided a more detailed description of human gestures than would normally be seen with gloves used for the purpose of human-robot collaboration.

6.3. PUBLICATION PLAN

Currently two publications are actively being developed. Associated with the work in Section 3 on the sensor application the paper,

1. *Nucleic Acid Probes Capture COVID-19 Virus in Low-Cost Rapid Testing MXene-Graphene Field Effect Transistors*. The authors of this paper are Jiaoli Li, Yuwei Zhang, Yanxiao Li, Congjie Wei, Adam Sawyer, ZheKun Peng, DongHyun Kim, Risheng Wang and Chenglin Wu.

The second paper associated with Sensor Glove in Section 4 is titled,

2. *Reach to Grasp: 3D Printed Dual Mode Contact Sensor*. The authors of this paper are Jiaoli Li, Yu Li, Adam Sawyer, Mingyuan Sun, Bo Li, Chenglin Wu.

The work in Section 5 is intended to be made into a publication with the title,

3. *Human Robot Collaboration in Construction Using LSTM and ProMP*. The authors will be Adam Sawyer, Anastasia Reed-Comeaux, Jiaoli Li, Yun Seong Song, Joe Stanley and Chenglin Wu.

6.4. FUTURE PLANS

6.4.1. Improvements to Human-Robot Collaboration System. Improvements that can make this system more useful in the future have been recognized throughout its development. The first improvement would be tracking the human hand through a key point detection algorithm which is fed image data through a single HD camera. While the Vicon system is effective it must be noted that it reduces the portability and increases the cost of deploying a system such as this in any practical system. Containing the camera system to a single generic HD camera would reduce the cost significantly and make the device easily portable. Another improvement that should be made in the future is adding reinforcement learning to the systems real-time operation so the systems response can improve as it is collaborated with. Currently the system only relies on imitation learning therefore it only improves initially when training off pre-recorded demonstrations. Lastly recording many more demonstrations and a larger breadth of gestures would improve the usability and accuracy of the system. The combination of reinforcement learning, and greater data collection would also enable ProMP waypoint and time modulation to be a feature of the systems real-time operation. Therefore, one learned ProMP could be used to generate many movements.

6.4.2. Applications of Technology. The sensor application developed in Section 3 offers utility to the health sector and academic sensor research. The health sectors use case was proven by the deployment of the sensor embedded n95 mask with 85 patients at the hospital. Researchers looking for a convenient and pre-programmed way to test the outputs of their novel sensors can use the sensor application view and export experimental data.

The sensor glove in Section 4 shows promises in being used in future academic endeavors that need to read the state of the human hand efficiently. The glove's modularity in regard to sensor attachments allows for many different use cases in academia. The specific design of the glove for this project also equips it to be used in construction because it is a work glove with a long battery life. The long battery life would prevent workers from having to stop their work to recharge the device. Lastly, the human-robot collaboration system in Section 5 shows promises for collaboration in work bench-bound translation tasks. This includes assisting in moving items and providing components to the worker.

APPENDIX

SOURCE CODE

This section contains the source code associated with all of the Pseudo Code from within the main sections. It contains a mix of Python, C, and C++.

```
#include <SoftwareSerial.h>

#define TxD 2
#define RxD 3
SoftwareSerial mySerial(RxD, TxD);

const int THUMB = A1;
const int INDEX = A2;
const int MIDDLE = A3;
const int RING = A4;
const int PINKY = A5;

bool started = false;

void setup() {
  mySerial.begin(115200);
}

void loop() {
  if (mySerial.read() == '1' or started){
    if (started == false){
      started = true;
    }
    byte message[6] = {(byte)255,(byte)(analogRead(THUMB) >> 2),
                       (byte)(analogRead(INDEX) >> 2),
                       (byte)(analogRead(MIDDLE) >> 2),
                       (byte)(analogRead(RING) >> 2),
                       (byte)(analogRead(PINKY) >> 2)};
    byte message_double[12] = {message[0], message[1], message[2],
                               message[3], message[4], message[5],
                               message[0], message[1], message[2],
                               message[3], message[4], message[5]};
    mySerial.write(message_double, 12);
    delay(10);
  }
}
```

Source Code 1 Bluetooth Glove Operation – Written in C[3]

[3] Covers Pseudocode in Algorithm 4.1

```
# Collects data from Vicon and BlueTooth Glove, combines it and transmits over
ethernet
from vicon_dssdk import ViconDataStream
import bluetooth
import socket
import csv
import datetime as dt
import numpy as np
import time
import win32api
import win32process
import struct


grab = {'PWF': False, 'PWB': False, 'TWF': False, 'TWB': False, 'MWF': False,
'MWB': False,
            'palm': False, 'thumb': True, 'index': True, 'middle': True, 'ring':
True, 'pinky': True}

record = {'bar1': [[],[],[]], 'bar2': [[],[],[]], 'bar3': [[],[],[]],
          'bar4': [[],[],[]], 'pinky': [[],[],[]], 'ring': [[],[],[]],
          'middle': [[],[],[]], 'index': [[],[],[]], 'thumb': [[],[],[]]}

def init_vicon():
    #setting up connection to the Vicon Nexus
    client = ViconDataStream.Client()
    client.Connect('localhost:801')
    client.SetBufferSize(10)
    client.EnableMarkerData()
    client.SetStreamMode(ViconDataStream.Client.StreamMode.EServerPush)

    #establishing the origin with reference to the table
    client.GetFrame()
    P1 = list(client.GetMarkerGlobalTranslation("table", "table1"))[0]
    P2 = list(client.GetMarkerGlobalTranslation("table", "table2"))[0]
    P3 = list(client.GetMarkerGlobalTranslation("table", "table3"))[0]
    P4 = list(client.GetMarkerGlobalTranslation("table", "table4"))[0]
    origin = []
    for num1, num2 in zip(P1, P4):
        temp = (num1+num2)/2
        origin.append(temp)
    #current process of establishing the Z coordinate by averaging the Z of each
table marker
    origin[2] = (P1[2]+P2[2]+P3[2]+P4[2])/4
```

```
    return client, origin

# Returns the data from vicon system as comma separated values
def read_vicon(client, origin):
    # vals = ""
    vals = bytearray()
    client.GetFrame()
    subjectNames = ["bar", "glove2"]
    for subjectName in subjectNames:
        markerNames = client.GetMarkerNames( subjectName )
        for markerName, _ in markerNames:
            position = list(client.GetMarkerGlobalTranslation( subjectName,
markerName ))
            if (subjectName == 'glove2' and grab[markerName] == False):
                continue
            #This handles situations where the marker is occluded
            elif position[1] == True:
                if not len(record[markerName]):
                    continue
                #calculates the average velocity and adds that to the most recent
marker position to generate positions for missing markers
                else:
                    if len(record[markerName][0]) == 0:
                        vals.extend(struct.pack(f"<3f",0, 0, 0))
                        position = [0, 0, 0]
                    else:
                        avg_vel_temp =
(sum(np.diff(np.array(record[markerName][0])))/len(record[markerName][0]),
sum(np.diff(np.array(record[markerName][1])))/len(record[markerName][0]),
sum(np.diff(np.array(record[markerName][2])))/len(record[markerName][0]))
                        avg_vel = [avg_vel_temp[0] if avg_vel_temp[0] >= 1 else
0, avg_vel_temp[1] if avg_vel_temp[1] >= 1 else 0, avg_vel_temp[2] if
avg_vel_temp[2] >= 1 else 0]
                        position = [record[markerName][0][-1] + avg_vel[0],
record[markerName][1][-1] + avg_vel[1], record[markerName][2][-1] + avg_vel[2]]
                        vals.extend(struct.pack(f"<3f", position[0], position[1],
position[2]))
            #if not occluded, transform the marker positions so they're relative
to the origin and then add to vals
            else:
                position = [a[0] - a[1] for a in list(zip(position[0], origin))]
                vals.extend(struct.pack(f"<3f", position[0], position[1],
position[2]))
```

```
            #at the start of adding things to the record, this duplicates the
initial position so there will always be two positions to calculate velocity from
            if len(record[markerName][0]) == 0:
                record[markerName][0].append(position[0])
                record[markerName][1].append(position[1])
                record[markerName][2].append(position[2])

                record[markerName][0].append(position[0])
                record[markerName][1].append(position[1])
                record[markerName][2].append(position[2])

            #this section maintains the record length at 200 by popping before
adding a new position
            elif len(record[markerName][0]) == 15:
                record[markerName][0].pop(0)
                record[markerName][0].pop(1)
                record[markerName][0].pop(2)

                record[markerName][0].append(position[0])
                record[markerName][1].append(position[1])
                record[markerName][2].append(position[2])
            else:
                record[markerName][0].append(position[0])
                record[markerName][1].append(position[1])
                record[markerName][2].append(position[2])
    return vals

last_read = None

def init_BT():
    global last_read

    nearby_devices = bluetooth.discover_devices(lookup_names=True)
    print("Found {} devices.".format(len(nearby_devices)))

    sock = bluetooth.BluetoothSocket()
    for addr, name in nearby_devices:
        if name == 'PressureGlove':
            sock.connect((addr, 1))

    sock.send('1'.encode("utf-8"))
    last_read = time.perf_counter()


    print("BT Connected")
```

```
    return sock

def read_BT(bt_socket):
    global last_read
    buffer = bytearray()
    read = max(int((time.perf_counter() - last_read)/0.010) + 1, 1)
    read = int(12*read)
    while len(buffer) < read:
        buffer.extend(bt_socket.recv(read - len(buffer)))
    last_read = time.perf_counter()

    idx = buffer[:buffer.rfind(255)].rfind(255)
    return buffer[idx + 1:idx + 6]

# Connects to the linux workstation and waits for start command
def connection_init():
    client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    client.bind(('', 4444))

    print("Connected")

    message = client.recv(1).decode("utf-8")
    while message != "0":
        print(message)
        message = client.recv(1).decode("utf-8")

    return client

def send_data(client, data):
    client.sendto(data, ('192.168.2.15', 4444))

def main():
    win32process.SetPriorityClass(win32api.GetCurrentProcess(),
win32process.REALTIME_PRIORITY_CLASS)

    vicon_client, origin = init_vicon()
    bt_socket = init_BT()
    client = connection_init()

    while vicon_client.IsConnected():
        try:
            if client.recv(1).decode('utf-8') == '0':
                strt = time.perf_counter()
                d_1 = read_vicon(vicon_client, origin)
```

```
                vic_tim = time.perf_counter() - strt
                 strt = time.perf_counter()
                 d_2 = read_BT(bt_socket)
                 bt_tim = time.perf_counter() - strt
                 data = d_1 + d_2
                 strt = time.perf_counter()
                 print(f"Message Bytes: {len(data)}")
                 send_data(client, data)
                 send_tim = time.perf_counter() - strt

                 print(f"Elapsed time:\n\tVicon Time: {vic_tim}\n\tBT Time:
{bt_tim}\n\tSend Time: {send_tim}")

        except KeyboardInterrupt:
            vicon_client.Disconnect()

if __name__ == "__main__":
    main()
```

Source Code 2 Human Data Workstation Client – Written in Python[4]

[4] Covers Pseudocode in Algorithms 5.1, 5.2, 5.3, 5.4, and 5.5

```
import socket
import keyboard
import time
import struct
import subprocess
import os
from datetime import datetime
from xarm.wrapper import XArmAPI


def handle_err_warn_changed(item):
    print('ErrorCode: {}, WarnCode: {}'.format(item['error_code'],
item['warn_code']))

def main():
    ip = "192.168.1.224"

    arm = XArmAPI(ip, do_not_open=True)
    arm.register_error_warn_changed_callback(handle_err_warn_changed)
    arm.connect()

    server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server.bind(('', 4444))

    input("Press enter key to start experiment")
    server.sendto('0'.encode("utf-8"), ('192.168.2.10', 4444))

    loop_sum, loop_cnt = 0, 0
    max_loop = 0

    first_recv = True

    rec_demonstration = ""

    while not keyboard.is_pressed('q'):
        server.sendto('0'.encode("utf-8"), ('192.168.2.10', 4444))
        buffer = bytearray()
        while len(buffer) < 113:
            print(len(buffer))
            server.settimeout(10.0)
            buffer.extend(server.recv(113 - len(buffer)))
            server.settimeout(0.0)

        if first_recv == True:
```

```
            process = subprocess.Popen(["./custom_scripts/record_trajectory",
"192.168.1.224"])
            first_recv = False

        buffer = struct.unpack("<27f5B", buffer)

        arm_pos = arm.get_position()[1]
        rec_demonstration += ','.join([str(b) for b in buffer]) +
f",{arm_pos[0]},{arm_pos[1]},{arm_pos[2]},{arm_pos[3]},{arm_pos[4]},{arm_pos[5]},
{time.time()}\n"

    process.terminate()

    demonstration = open(os.path.join(os.path.dirname(__file__), '..',
'Demonstrations', 'demo_stand', 'demo_stand_' + (datetime.now()).strftime('%d-%m-
%Y_%H:%M:%S')) + '.csv', "w+")
    demonstration.write(rec_demonstration)

if __name__ == "__main__":
    main()
```

Source Code 3 Data Collection Server – Written in Python[5]

[5] Covers Pseudocode in Algorithm 5.6

```cpp
#include <chrono>
#include <thread>
#include <fstream>
#include <ctime>
#include <xarm/wrapper/xarm_api.h>

void record_traj(XArmAPI *arm);

// Get current date/time, format is YYYY-MM-DD_HH:mm:ss
const std::string currentDateTime() {
    time_t     now = time(0);
    struct tm  tstruct;
    char       buf[80];
    tstruct = *localtime(&now);

    strftime(buf, sizeof(buf), "%Y-%m-%d_%X", &tstruct);

    return buf;
}

int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Please enter IP address\n");
        return 0;
    }

    std::string port(argv[1]);
    std::cout << port << std::endl;
    XArmAPI *arm = new XArmAPI(port);

    std::cout << "To stop recording press q" << std::endl;

    record_traj(arm);

    return 0;
}

void record_traj(XArmAPI *arm){
    using clock = std::chrono::system_clock;
    using sec = std::chrono::duration<double>;
    auto start = clock::now();

    std::ofstream trajFile;
    trajFile.open("rec_trajectories/demo_stand/rec_traj_stand_" +
currentDateTime() + ".csv", std::ios::out);
```

```
    trajFile << "Jp_1, Jp_2, Jp_3, Jp_4, Jp_5, Jp_6, Gp, Jv_1, Jv_2, Jv_3,
                 Jv_4, Jv_5, Jv_6, Je_1, Je_2, Je_3, Je_4, Je_5, Je_6, time
(ms)\n";

    fp32 position[7], gripper_pos[2], velocity[7], effort[7];
    while (true)
    {
        start = clock::now();

        arm->get_joint_states(position, velocity, effort);
        arm->get_gripper_position(gripper_pos);

        auto cur_time = std::chrono::system_clock::now();
        // We are using the xArm 6 therefore we do not record index 6 for
position, velocity
        // or effort because it refers to a nonexistent 7th joint
        trajFile << position[0] << "," << position[1] << "," << position[2] <<
","
                 << position[3] << "," << position[4] << "," << position[5] <<
","
                 << gripper_pos[0] << "," << velocity[0] << "," << velocity[1] <<
","
                 << velocity[2] << "," << velocity[3] << "," << velocity[4] <<
"," << velocity[5] << ","
                 << effort[0] << "," << effort[1] << "," << effort[2] << ","
                 << effort[3] << "," << effort[4] << "," << effort[5] << ","
                 <<
std::chrono::duration_cast<std::chrono::milliseconds>(cur_time.time_since_epoch()
).count() << '\n';

        sec duration = clock::now() - start;
        std::this_thread::sleep_for(std::chrono::milliseconds(4) - duration);

    }
    trajFile.close();
}
```

Source Code 4 Record Trajectory – Written in C++[6]

[6] Covers Pseudocode in Algorithm 5.7

```
import sys

def main():
    file = open(sys.argv[1],'r')
    length = len(file.readlines())
    file.close()

    gesture = input("What is the gesture?\n\t0: Stand\n\t1: Wave\n\t2: Grab\n")

    if gesture != '0':
        start = input("On what frame does the gesture start?\n")
        stop = input("On what frame does the gesture end?\n")

    dir = (sys.argv[1]).split('/')
    print(dir)

    folder = dir[-2]

    labels_dir = dir[-1][:-4] + "_labels"
    labels = open("./Demonstrations/" + folder + "/labels/" + labels_dir, "w+")

    if gesture != '0':
        for i in range(0,int(start)):
            labels.write("0\n")
        for i in range(int(start),int(stop) + 1):
            labels.write(gesture + '\n')
        for i in range(int(stop) + 1, length):
            labels.write("0\n")

    else:
        for i in range(length):
            labels.write("0\n")

    labels.close()


if __name__ == "__main__":
    main()
```

Source Code 5 Demonstration Labeling[7]

[7] Covers Pseudocode in Algorithm 5.8

```
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.preprocessing import sequence
from keras.layers import Dropout
from keras.layers import Dense, GRU, LSTM
from keras.utils import to_categorical

from os import listdir

print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))

# Viewing Data
sample = pd.read_csv('./Demonstrations/demo_grab/demo_grab_27-03-
2023_20:53:56.csv', header=None)

# We drop the markers for the board and set the time column to be the difference
from the previous row
sample = sample.drop(list(range(15,27)) + list(range(32,38)), axis = 1)
sample.iloc[:,-1] = sample.iloc[:,-1].diff()
sample.iloc[0,-1] = 0.0
sample.head()

input_shape = (1,21)

# Loading Dataset
data = []

keep, cnt = 4, 0

# Stand Data
for d in listdir("./Demonstrations/demo_stand/"):
    if d == "labels":
        continue

    df = pd.read_csv("./Demonstrations/demo_stand/" + d, header=None)
    df = df.drop(list(range(15,27)) + list(range(32,38)), axis = 1)
    df.iloc[:,-1] = df.iloc[:,-1].diff()
    df.iloc[0,-1] = 0.0

    labels = open("./Demonstrations/demo_stand/labels/" + d[:-4] + "_labels")
    df['39'] = np.array([float(x) for x in labels.read().splitlines()],
dtype="float32")
    labels.close()
```

```
    if cnt % 4 == 0:
        data.append(df.to_numpy())
    cnt += 1

# Wave Data
for d in listdir("./Demonstrations/demo_wave/"):
    if d == "labels":
        continue

    df = pd.read_csv("./Demonstrations/demo_wave/" + d, header=None)
    df = df.drop(list(range(15,27)) + list(range(32,38)), axis = 1)
    df.iloc[:,-1] = df.iloc[:,-1].diff()
    df.iloc[0,-1] = 0.0

    labels = open("./Demonstrations/demo_wave/labels/" + d[:-4] + "_labels")
    df['39'] = np.array([float(x) for x in labels.read().splitlines()],
dtype="float32")
    labels.close()

    data.append(df.to_numpy())

# Grab Data
for d in listdir("./Demonstrations/demo_grab/"):
    if d == "labels":
        continue

    df = pd.read_csv("./Demonstrations/demo_grab/" + d, header=None)
    df = df.drop(list(range(15,27)) + list(range(32,38)), axis = 1)
    df.iloc[:,-1] = df.iloc[:,-1].diff()
    df.iloc[0,-1] = 0.0

    labels = open("./Demonstrations/demo_grab/labels/" + d[:-4] + "_labels")
    df['39'] = np.array([float(x) for x in labels.read().splitlines()],
dtype="float32")
    labels.close()

    data.append(df.to_numpy())

for d in data:
    print(d.shape)
len(data)

# Model Creation
model = Sequential()
```

```
model.add(LSTM(128,return_sequences=True,
dropout=0.2,input_shape=(None,input_shape[1])))
model.add(LSTM(64,dropout=0.3,return_sequences=True))
model.add(LSTM(64,dropout=0.3,return_sequences=True))
model.add(LSTM(32,dropout=0.3,return_sequences=True))
model.add(LSTM(32,dropout=0.3,return_sequences=True))
model.add(LSTM(16,dropout=0.3,return_sequences=True))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')

model.summary()

def TrainGenerator(data):
    cnt = 0
    while True:
        idx = cnt % len(data)
        if idx == 0:
            train_data_perm = np.random.permutation(len(data))

        cnt += 1
        x_train = (data[train_data_perm[idx]])[:,:-1]
        y_train = (data[train_data_perm[idx]])[:,-1]
        y_train = to_categorical(y_train,3)
        y_train = y_train.reshape((y_train.shape[0], 3))
        y_train = y_train[None,...,None]
        x_train = x_train[None,...]
        print(x_train.shape)
        print(y_train.shape)
        yield x_train, y_train

history = model.fit(TrainGenerator(data), epochs=10, steps_per_epoch=len(data),
shuffle=True)

loss = history.history["loss"]

import matplotlib.pyplot as plt

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'b')
plt.title("Training Loss v. Epoch")

plt.show()

x = (data[0])[:,:-1]
```

```
print(model.predict(x[None,...]))

# Making Stateless LSTM Model to Stateful
stateful_model = Sequential()
stateful_model.add(LSTM(128,stateful=True,return_sequences=True, dropout=0.2,
                        batch_input_shape=((1,input_shape[0],input_shape[1])),
                        input_shape=input_shape))
stateful_model.add(LSTM(64,stateful=True,dropout=0.3,return_sequences=True))
stateful_model.add(LSTM(64,stateful=True,dropout=0.3,return_sequences=True))
stateful_model.add(LSTM(32,stateful=True,dropout=0.3,return_sequences=True))
stateful_model.add(LSTM(32,stateful=True,dropout=0.3,return_sequences=True))
stateful_model.add(LSTM(16,stateful=True,dropout=0.3,return_sequences=True))
stateful_model.add(Dense(3, activation='softmax'))
stateful_model.compile(loss='categorical_crossentropy', optimizer='adam')

stateful_model.set_weights(model.get_weights())

stateful_model.summary()

x = (data[25])[:,:-1]
y = (data[25])[:,-1]
print(y[0])

stateful_model.save("stateful_LSTM_NoRobotPos")
converter = tf.lite.TFLiteConverter.from_saved_model("stateful_LSTM_NoRobotPos")
tflite_model = converter.convert()
with open("stateful_LSTM_NoRobotPos.tflite", "wb") as f:
    f.write(tflite_model)

from time import perf_counter

def predict(tflite, tensor):
    tflite.set_tensor((tflite.get_input_details())[0]["index"], tensor)
    tflite.invoke()
    output = tflite.get_tensor(tflite.get_output_details()[0]["index"])
    probabilities = np.array(output[0])
    return probabilities

x = (data[44])[:,:-1]
y = (data[44])[:,-1]

tflite_model = tf.lite.Interpreter("stateful_LSTM_NoRobotPos.tflite")
tflite_model.allocate_tensors()
```

```
tflite_model.reset_all_variables()
for x_i in x:
    print(f"Input Tensor: {x_i}")
    start = perf_counter()
    prediction = predict(tflite_model,(np.array(x_i.reshape(input_shape),
dtype="float32")[None,...]))
    print(f"Prediction: {prediction}", end="\n")

x = (data[20])[:,:-1]
y = (data[20])[:,-1]

tflite_model = tf.lite.Interpreter("stateful_LSTM_NoRobotPos.tflite")
tflite_model.allocate_tensors()


tflite_model.reset_all_variables()
for x_i in x:
    start = perf_counter()
    print(predict(tflite_model,(np.array(x_i.reshape(input_shape),
dtype="float32")[None,...])), end=" ")
    print(f"time: {(perf_counter() - start) * 1000} ms")
```

Source Code 6 LSTM Model Creation and Training[8]

[8] Covers Pseudocode in Algorithm 5.9

```
from movement_primitives.plot import *
from movement_primitives.promp import ProMP
from movement_primitives.io import read_pickle, write_pickle
import matplotlib.pyplot as plt
import numpy as np
from os import listdir
def main():
    demos = np.array([np.loadtxt('./replayable_trajectories/demo_stand/' +
dir_content, delimiter=',', dtype=float) for dir_content in
listdir('./replayable_trajectories/demo_stand')])
    print(demos.shape)
    for i in range(len(demos)):
        print(demos[i].shape)

    print("---------")
    max_rows = max([d.shape[0] for d in demos])

    for d_idx in range(len(demos)):
        if demos[d_idx].shape[0] < max_rows:
            demos[d_idx] = np.append(demos[d_idx], [demos[d_idx][-1,:] for i in
range(max_rows - demos[d_idx].shape[0])], axis=0)

    for i in range(len(demos)):
        print(demos[i].shape)

    print("---------")

    demos = np.array([d for d in demos])
    print(demos.shape)
    print("---------")

    times = np.array([ np.arange(0, 0.004*max_rows, 0.004, dtype=float) for i in
range(len(demos))])
    print(times.shape)

    print("---------")

    print(times[0,0:10])

    print("---------")

    traj_promp = ProMP(demos.shape[2], n_weights_per_dim=15)
    print('done')

    traj_promp.imitate(times, demos, min_delta=0, verbose=1)
```

```
 plot_distribution_in_rows(traj_promp.mean_trajectory(times[0,:]),
np.sqrt(traj_promp.var_trajectory(times[0,:])), times[0,:])
    plt.show()

    write_pickle("./demo_stand_promp", traj_promp)

    traj_promp = read_pickle("./demo_stand_promp")

    plot_distribution_in_rows(traj_promp.mean_trajectory(times[0,:]),
np.sqrt(traj_promp.var_trajectory(times[0,:])), times[0,:])
    plt.show()

    return 0

if __name__ == "__main__":
    main()
```

Source Code 7 ProMP Trainer[9]

[9] Covers Pseudocode in Algorithm 5.10

```
import socket
import keyboard
import time
import struct
import subprocess
import os

import tensorflow as tf

from datetime import datetime
import numpy as np
from xarm.wrapper import XArmAPI

def handle_err_warn_changed(item):
    print('ErrorCode: {}, WarnCode: {}'.format(item['error_code'],
item['warn_code']))

def predict(tflite, tensor):
    tflite.set_tensor((tflite.get_input_details())[0]["index"], tensor)
    tflite.invoke()
    output = tflite.get_tensor(tflite.get_output_details()[0]["index"])
    probabilities = np.array(output[0])
    return probabilities

def main():
    robot_controller =
tf.lite.Interpreter("/home/arsrbt/Documents/ML/stateful_LSTM_NoRobotPos.tflite")
    robot_controller.allocate_tensors()


    server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server.bind(('', 4444))

    input("Press enter key to start system")
    time.sleep(10)
    server.sendto('0'.encode("utf-8"), ('192.168.2.10', 4444))

    first_recv = True
    prev_time = None

    p = None

    strt_1 = time.perf_counter()

    rec_demonstration = ""
```

```
    while not keyboard.is_pressed('q'):
        server.sendto('0'.encode("utf-8"), ('192.168.2.10', 4444))
        buffer = bytearray()
        while len(buffer) < 113:
            server.settimeout(10.0)
            buffer.extend(server.recv(113 - len(buffer)))
            server.settimeout(0.0)

        buffer = list(struct.unpack("<27f5B", buffer))
        buffer = buffer[0:15] + buffer[27:]
        buffer += [time.time()]

        data = (np.array(buffer, dtype="float32")).reshape((1,21))

        if first_recv == True:
            prev_time = data[0,-1]
            data[0,-1] = np.float32(0.0)
            first_recv = False
        else:
            temp = data[0,-1]
            data[0,-1] = np.float32(temp - prev_time)
            prev_time = temp

        # Make prediction based on current state
        prediction = predict(robot_controller, data[None,...])[0,:]
        movement = np.argmax(prediction)

        if movement == 0:
            print(f"STAND with {prediction[movement]:.2%} confidence")
        elif movement == 1:
            print(f"Wave with {prediction[movement]:.2%} confidence")
        elif movement == 2:
            print(f"Grab with {prediction[movement]:.2%} confidence")

        if movement != 0 and float(f"{prediction[movement]:.2%}".strip('%')) >
0.9:
            if p == None:
                if movement == 1:
                    p =
subprocess.Popen(["python3","./custom_scripts/load_replay_Promp.py",
"./ProMPs/demo_wave/demo_wave_promp"])
                if movement == 2:
                    p =
subprocess.Popen(["python3","./custom_scripts/load_replay_Promp.py",
"./ProMPs/demo_grab/demo_grab_promp"])
```

```
        if p != None:
            poll = p.poll()
            if poll is not None:
                robot_controller.reset_all_variables()
                p = None

if __name__ == "__main__":
    main()
```

Source Code 8 Real-Time Server[10]

[10] Covers Pseudocode in Algorithm 5.11

```python
from xarm.wrapper import XArmAPI
from movement_primitives.io import read_pickle
from movement_primitives.promp import ProMP
import numpy as np
import sys
import time

def replay_trajectory(arm: XArmAPI, lines):


    # Set arm and gripper to initial position
    arm.set_servo_angle(angle=lines[0][0:-1], speed=80, mvacc=100, wait=True,
radius=None)
    arm.set_gripper_position(pos=lines[0][-1], wait=True)

    while arm.get_is_moving():
        time.sleep(0.1)

    arm.set_mode(1)
    arm.set_state(state=0)
    time.sleep(0.1)

    for line in lines[1:]:
        strt = time.time()
        # Subtract elapsed time from max wait time
        arm.set_servo_angle_j(angles=line[0:-1])
        arm.set_gripper_position(pos=line[-1], wait=False)
        wait_time = 0.004 - (time.time() - strt)
        if wait_time > 0:
            time.sleep(wait_time) # Refresh at 250 hz

def handle_err_warn_changed(item):
    print('ErrorCode: {}, WarnCode: {}'.format(item['error_code'],
item['warn_code']))

def main():
    ip = '192.168.1.224'

    arm = XArmAPI(ip, do_not_open=True)
    arm.register_cmdnum_changed_callback(handle_err_warn_changed)
    arm.connect()
    arm.motion_enable(enable=True)
    arm.set_mode(0)
    arm.set_state(0)
    arm.set_gripper_enable(True)
```

```
arm.set_gripper_mode(0)
    arm.set_gripper_speed(5000)

    traj_ProMP = read_pickle(sys.argv[1])
    file = open(sys.argv[1][:sys.argv[1].rfind('/')] + "/max_steps.txt", 'r')
    max_steps = int(file.readline())/2
    file.close()
    times = np.arange(0, 0.004*max_steps, 0.004, dtype=float)

    if len(sys.argv) > 2:
        replay_trajectory(arm, (traj_ProMP.condition_position((np.array([float(x)
for x in sys.argv[2].split(',')])), None, t=1.0)).mean_trajectory(times))
    else:
        replay_trajectory(arm, traj_ProMP.mean_trajectory(times))

if __name__ == "__main__":
    main()
```

Source Code 9 Replay ProMP[11]

[11] Covers Pseudocode in Algorithm 5.12

BIBLIOGRAPHY

[1] J. Peters, D. D. Lee, J. Kober, D. Nguyen-Tuong, J. A.Bagnell, and S. Schaal, "Robot Learning," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Cham: Springer International Publishing, 2016, pp. 357–398. doi: 10.1007/978-3-319-32552-1_15.

[2] S. Schaal, "Dynamic Movement Primitives -A Framework for Motor Control in Humans and Humanoid Robotics," in *Adaptive Motion of Animals and Machines*, H. Kimura, K. Tsuchiya, A. Ishiguro, and H. Witte, Eds. Tokyo: Springer, 2006, pp. 261–280. doi: 10.1007/4-431-31381-8_23.

[3] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal, "Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors," *Neural Computation*, vol. 25, no. 2, pp. 328–373, Feb. 2013, doi: 10.1162/NECO_a_00393.

[4] "Learning to select and generalize striking movements in robot table tennis." https://journals.sagepub.com/doi/epdf/10.1177/0278364912472380 (accessed Mar. 10,2023).

[5] A. Paraschos, C. Daniel, J. R. Peters, and G. Neumann, "Probabilistic Movement Primitives".

[6] A. Paraschos, C. Daniel, J. Peters, and G. Neumann, "Using probabilistic movement primitives in robotics," *Auton Robot*, vol. 42, no. 3, pp. 529–551, Mar. 2018, doi: 10.1007/s10514-017-9648-7.

[7] M. Rambow, T. Schauß, M. Buss, and S. Hirche, "Autonomous manipulation of deformable objects based on teleoperated demonstrations," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2012, pp. 2809–2814. doi: 10.1109/IROS.2012.6386002.

[8] G. Lentini, G. Grioli, M. G. Catalano, and A. Bicchi, "Robot Programming without Coding," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, May 2020, pp. 7576–7582. doi: 10.1109/ICRA40945.2020.9196904.

[9] S. Detzel, M. Steinberger, and T. C. Lueth, "A Kinesthetic Teaching System for a Robotic Arm for Middle Ear Surgery," in *2018 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, Dec. 2018, pp. 1870–1875. doi: 10.1109/ROBIO.2018.8665216.

[10] A. Montebelli, F. Steinmetz, and V. Kyrki, “On handing down our tools to robots: Single-phase kinesthetic teaching for dynamic in-contact tasks,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 5628–5634. doi: 10.1109/ICRA.2015.7139987.

[11] Y. Gao, J. Tebbe, and A. Zell, “Optimal Stroke Learning with Policy Gradient Approach for Robotic Table Tennis,” *Appl Intell*, Oct. 2022, doi: 10.1007/s10489-022-04131-w.

[12] “Movement Primitives.” DFKI GmbH, Robotics Innovation Center, Mar. 08, 2023. Accessed: Mar. 10, 2023. [Online]. Available: https://github.com/dfki-ric/movement_primitives

[13] “Sensor types,” *Android Open Source Project*. https://source.android.com/docs/core/interaction/sensors/sensor-types (accessed Apr. 13, 2023).

[14] 1615 L. St NW, S. 800 Washington, and D. 20036 U.-419-4300 | M.-857-8562 | F.-419-4372 | M. Inquiries, “Mobile Fact Sheet,” *Pew Research Center: Internet, Science & Tech*. https://www.pewresearch.org/internet/fact-sheet/mobile/ (accessed Apr. 13, 2023).

[15] “BinaxNOW™ COVID-19 Antigen Self-Test.” https://www.globalpointofcare.abbott/en/product-details/binaxnow-covid-19-antigen-self-test-us.html (accessed Apr. 13, 2023).

[16] D. A. Lo, “Sensor Plot Kit: An iOS Framework for Real-time plotting of Wireless Sensors”.

[17] J. C. Yeo, C. Lee, Z. Wang, and C. T. Lim, “Tactile sensorized glove for force and motion sensing,” in *2016 IEEE SENSORS*, Oct. 2016, pp. 1–3. doi: 10.1109/ICSENS.2016.7808596.

[18] S. Zhu, A. Stuttaford-Fowler, A. Fahmy, C. Li, and J. Sienz, “Development of a Low-cost Data Glove using Flex Sensors for the Robot Hand Teleoperation,” in *2021 3rd International Symposium on Robotics & Intelligent Manufacturing Technology (ISRIMT)*, Sep. 2021, pp. 47–51. doi: 10.1109/ISRIMT53730.2021.9596972.

[19] M. Awais and D. Henrich, “Human-robot collaboration by intention recognition using probabilistic state machines,” in *19th International Workshop on Robotics in Alpe-Adria-Danube Region (RAAD 2010)*, Jun. 2010, pp. 75–80. doi: 10.1109/RAAD.2010.5524605.

[20] B. Wu, J. Zhong, and C. Yang, “A Visual-Based Gesture Prediction Framework Applied in Social Robots,” *IEEE/CAA Journal of Automatica Sinica*, vol. 9, no. 3, pp. 510–519, Mar. 2022, doi: 10.1109/JAS.2021.1004243.

[21] M. Ewerton, G. Neumann, R. Lioutikov, H. Ben Amor, J. Peters, and G. Maeda, “Learning multiple collaborative tasks with a mixture of Interaction Primitives,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 1535–1542. doi: 10.1109/ICRA.2015.7139393.

[22] B. Akan, B. Cürüklü, G. Spampinato, and L. Asplund, “Towards robust human robot collaboration in industrial environments,” in *2010 5th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, Mar. 2010, pp. 71–72. doi: 10.1109/HRI.2010.5453264.

[23] R. Adamini *et al.*, “User-friendly human-robot interaction based on voice commands and visual systems,” in *2021 24th International Conference on Mechatronics Technology (ICMT)*, Dec. 2021, pp. 1–5. doi: 10.1109/ICMT53429.2021.9687192.

[24] J. Brawer, O. Mangin, A. Roncone, S. Widder, and B. Scassellati, “Situated Human–Robot Collaboration: predicting intent from grounded natural language,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2018, pp. 827–833. doi: 10.1109/IROS.2018.8593942.

[25] “Mobile Android Version Market Share Worldwide,” *StatCounter Global Stats*. https://gs.statcounter.com/android-version-market-share/mobile/worldwide/ (accessed Apr. 14, 2023).

[26] J. Gehring, “Chart and Graph Library for Android.” Apr. 12, 2023. Accessed: Apr. 14, 2023. [Online]. Available: https://github.com/jjoe64/GraphView

[27] S. Hochreiter and J. Schmidhuber, “Long Short-term Memory,” *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997, doi: 10.1162/neco.1997.9.8.1735.

[28] “SensingKit mobile sensing framework,” *SensingKit*. https://www.sensingkit.org/ (accessed Apr. 26, 2023).

VITA

Adam Ryan Sawyer received a bachelor's degree in computer engineering with M minors in mathematics and computer science from The Missouri University of Science and Technology in the Fall of 2021. The final year of his undergraduate career saw his involvement in research projects related to Covid-19 spike protein sensors with Dr. Chenglin Wu and his team. Adam began his master's degree in Spring of 2021 where his research focus became machine learning and its applications in the field of robotics. In July of 2023, he received his Master of Science in Computer Engineering from Missouri University of Science and Technology. In June 2023 he became a Software Engineer at the Boeing Corporation.