Edge Computing and the IoT Practical Course Final Report: Smart Glove Controlling System

Charles Donven and Yucong Ma

July 2021

1 Problem description

An Internet of Things (IoT) system can be abstracted as a group of connected sensors and actuators, and engineers have been exploring different methods to control them. Some controlling methods have already been widely adapted, for example, letting the user to directly enter a command on a central management portal, or making use of sensors to detect environment factors and adjust actuators accordingly. However, there are scenarios where such methods are not adaptive. For example, when a person with limited motion functionality wishes to control IoT devices, the command inputting method might not be so user-friendly.

In this project, we explore a new way for controlling IoT devices. We aim to come up with an intuitive and easy-to-use controlling method using the glove as a wearable device. To fulfill the requirement of this class, the computation would be conducted in an edge computing component. Additionally, we should provide test IoT devices to test our prototype, and for demonstration purposes as well.

2 Design of solution

2.1 Overview

In this overview, we describe how the system should behave during operation with a sequence diagram, and then we explain how we structure the project with an architecture diagram. Here we describe a common use case scenario. At the beginning, the user would be wearing the ”smart glove”. When the user wish to control the IoT devices in certain ways, they could simply make the predefined gestures. Those gestures would be captured by the glove and further be processed at the edge computing unit, where a machine learning algorithm is running to compute and interpret the specific command. In the end, the command would be executed at the corresponding IoT devices. A sequence diagram for this general scenario is placed below as Figure 2.1.

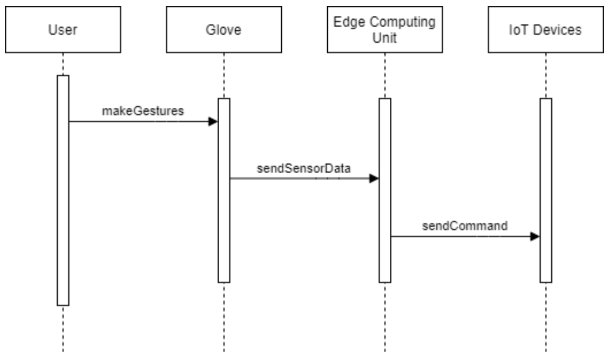


Figure 2.1: Simplified sequence diagram

We use an architecture diagram to give a general idea of how the components are connected in the system, as shown in Figure 2.2. On the glove part, various motion sensors would be connected to a microcontroller, so that the sensor data are sent out from there. The microcontroller is then wirelessly connected to an edge computing component, which is running some messaging protocols to handle communications with the microcontrollers. Finally, the IoT devices are listening to the edge computing component, and are ready to receive commands and change behaviors accordingly.

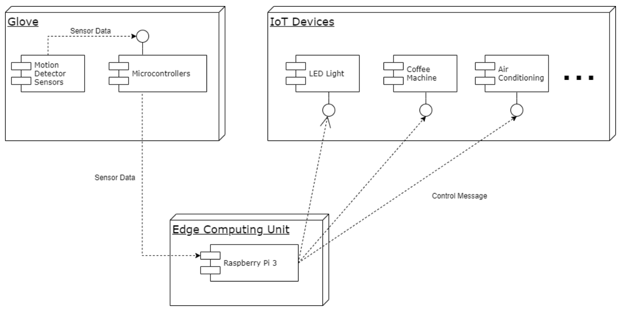


Figure 2.2: Simplified architecture diagram

2.2 Glove

In this system, the user can simply make gestures to send out various commands in order to control IoT devices, and such gestures are captured by a glove equipped with various motion sensors. To capture essential data that compose a gesture, we adopted the IoT concept when building the glove. Motion sensors of various purposes are attached to different areas on the glove, so that the hand movement can be interpreted in a rich description. In general, the glove captures finger angles and hand position movement in the space, which will be combined and be used to determine a gesture. The movement in the 3D space is described in the Yaw-Pitch-Roll system, as shown in Figure 2.3.

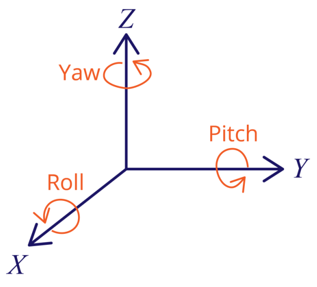


Figure 2.3: Yaw-Pitch-Roll in 3D space [1]

The flex sensors and the IMU sensor, which are responsible for the finger bending angles and the hand movement in a 3D dimension space respectively, are connected to a microcontroller for the data to be further transmitted.

2.3 Microcontrollers

We use microcontrollers to coordinate communications between sensors, actuators, and the edge computing unit. One microcontroller is attached to the arm of the user, and is wired to the motion sensors on the glove to actively collect sensor data. Additionally, it does some simple data preprocessing, and forwards the data to the edge computing unit for further computation.
Some other microcontrollers are used to receive commands from the edge computing unit for the IoT devices, so that they could implement corresponding changes.

2.4 Edge Computing

It is relatively easy to directly use sensor data in the programming logic, if the sensor data doesn't change so frequently. For example, we use static values for the finger position representations, which could be defined as either "bent" or "flat". However, to capture hand movements, we have to make use of continuous sensor data as time goes, and the calculation will be too complex to be done on the local microcontroller. We adopted the concept of edge computing for the computation work. More specifically, we use machine learning to identify the hand movement in a gesture.
For the machine learning model to work, developers will need to collect a large amount of repetitive data for each gesture for model training. This model will then be used to interpret user hand movement in the 3D space. In the main application program, the motion prediction from the machine learning model will be combined with the finger positions to form a final gesture. Given the gesture, a specific command will then be given and be sent out to the IoT devices.

[1]https://randomnerdtutorials.com/esp32-mpu-6050-accelerometer-gyroscope-arduino/

2.5 IoT Devices and Command

We use LED lights as well as a web application as the active IoT devices that the user could control in the current phase of this project. The web application simulates an LED light, which is used to show the scalability of the project, and for a better demonstration in an online presentation. The user could choose which device should be turned on, what color it shows, and in what degree of brightness.

The devices and the colors are managed in a row, such that when the user selects the device and the color, the commands would be in the format of ”Next device/color” or ”Previous device/color”. Figure 2.4 and Figure 2.5 are used to show this scheme. For example, when the program starts, the device #0 is chosen by default. By executing the ”Previous device” command, the device #4 is now chosen; if the command is ”Next device” instead, then device #1 will be turned on. Similarly, the color is arranged such that the white color, color #7, is shown by default, and can be changed in the same way. The brightness is adjusted by ”lower brightness” or ”higher brightness”. Initially set as 100%, the brightness could be lowered or raised as the wish of the user.

Off	🟦	🟩	🟦	🟥	🟪	🟨	⬜
Color #0	Color #1	Color #2	Color #3	Color #4	Color #5	Color #6	Color #7

Figure 2.4: Colors with indexes

3 Implementation of solution

3.1 Glove and Sensors

The glove is equipped with four flex sensors and an IMU sensor, and is connected to an ESP32 microcontroller[2]. We decide to use resistive flex sensors[3], which are placed on the index finger, the middle finger, the ring finger, and the little finger, with the

[2]https://www.espressif.com/sites/default/files/documentation/esp32-wrover-e_esp32-wrover-ie_datasheet_en.pdf

[3]https://www.spectrasymbol.com/product/flex-sensors/

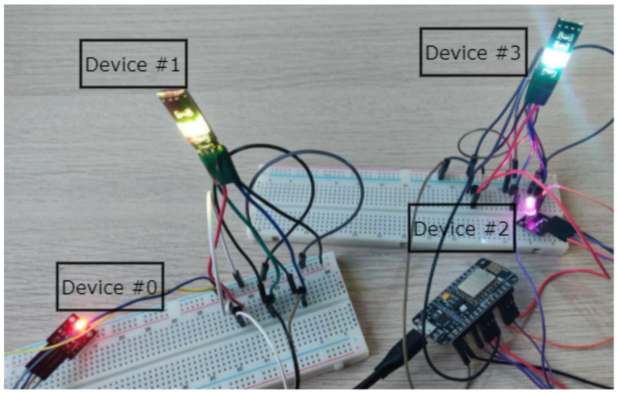


Figure 2.5: IoT devices with indexes (note: Device #4 is the web application and not included in the image)

tip of the sensor close to the middle joint on the finger and wraps around the base joint in the middle of the flex sensor. The IMU sensor is sewed inside the glove close to the base joint of the index finger. The finished glove is shown below as Figure 3.1, and the circuit connection is shown in Figure 3.2. When the flex sensor bends in different angles, the resistance of the sensor changes accordingly, which can be detected in the wired circuit. The IMU sensor can detect a change of the hand position in all of the three axes: Yaw, Pitch, and Roll. These motion sensors are adequate for capturing a gesture as defined in this project.

Here we give some more details into how the sensors work. When the shape of a flex sensor changes, its resistance will also change, thus changing its voltage value in the circuit. The flex sensors are connected to the ADC ports on the ESP32 so that the voltage values can be read. We calculate the resistance value of the sensor, and map this resistance value to an angle. Predefined STRAIGHT_RESISTANCE, which is 30.000 Ω, will be mapped to 0 degree, and BEND_RESISTANCE, 50.000 Ω, is 90 degrees. Any resistance value in between will be linearly converted to the bending degree of the sensor.

The IMU sensor that we use in this project, bno08x[4],

[4]https://learn.adafruit.com/adafruit-9-dof-orientation-imu-fusion-breakout-bno085/arduino

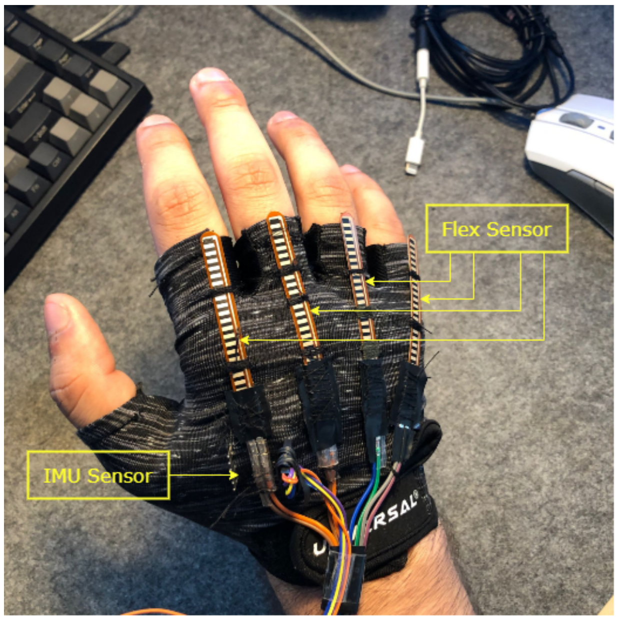


Figure 3.1: A fully connected glove

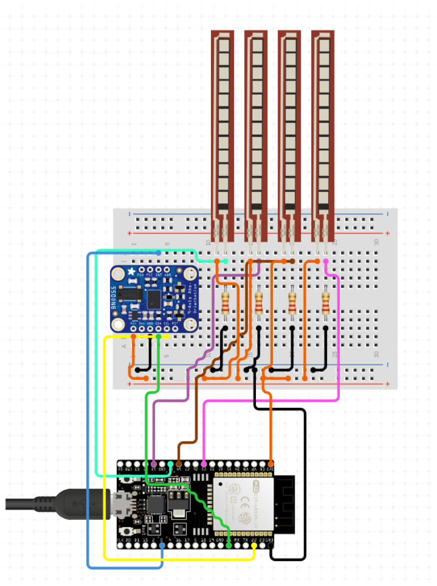


Figure 3.2: Glove and ESP32 connection

passes position values in unit quaternions to the microcontroller. We convert them to the Euler angles which are human readable and consistent with the Yaw-Pitch-Roll scheme. By reading and analyzing the change in each axis, the machine learning model is able to predict the type of the space motion that the user is making. The code for getting the finger angles and IMU Euler angles can be found in Appendix B.1.
The microcontroller that the glove is connected to, in every 50 milliseconds, publishes a message containing current flex sensor angles and the Euler angles of the IMU sensor. By reading the flex sensor values, we know how the user's fingers are bent. Then interpret the IMU sensor values, we can later calculate the motion of the hand on the edge computing unit.

3.2 Messaging Protocol

We choose MQTT[5] as the wireless messaging protocol between microcontrollers and the edge computing unit. Compared to UDP based protocols, MQTT runs on top of TCP/IP which guarantees reliability [2], which is important because using incorrect sensor data for a gesture interpretation might result in sending out wrong control messages. MQTT also provides easier connection setup as compared to the pairing process in Bluetooth. MQTT is based on the publisher/subscriber pattern, where the publisher or the subscriber does not need to know the identity of each other, which also provides convenience for our development to some extent.
We use Mosquitto[6] as the concrete software that implements the MQTT protocol on the edge computing device. Mosquitto will function as a broker in the messaging flow. The MQTT server IP is predefined on the microcontroller to initiate a connection. To ensure security, we configure MQTT server username and password on the broker, and microcontrollers also need to provide the correct credentials to be connected. The microcontroller on the glove side works as a publisher, which constantly publishes sensor data to the "/esp32/glove" topic. On the edge device, a MQTT client is running in the messaging program (see Appendix B.2 communication.py). It

[5]https://mqtt.org/

[6]https://mosquitto.org/

subscribes to this topic to receive sensor data in an infinite loop. Once a message is interpreted as command, the MQTT client on the edge computing unit publishes it to the topic which is being listened to by the desired IoT device, and the device can change behavior accordingly. On the IoT device side, two LED lights are connected to microcontrollers, which listen to topics defined for each of the light. The web application also subscribes to a unique topic.

3.3 Gesture

The definition of a gesture in this project is a combination of a finger indication and a space motion, with a duration of approximately 1,5 seconds since we use 30 MQTT messages to define a gesture.
A finger indication means that, the user bends some of the fingers and keeps the other fingers flat, and the positions of the fingers will be made at the beginning of a gesture and will not change throughout this gesture. After some live testing of the glove, we discovered that the flex sensors do not provide predictable voltage values due to the sewing position on the glove. At different times when the finger is bent to the same angle, the voltage values we read are very likely to slightly differ. We then decided to read the result value of the flex sensors in ranges. In the program, if the bending angle of the flex sensors is greater than 50 degrees, we simply define that this finger is bent. By combining the finger position results from all four fingers, we get a finger indication pattern which is ready to be used to decide the final gesture.
The fingers are also used to indicate when a gesture starts. At the beginning, all of the four fingers, which are covered by the flex sensors as described in the glove subsection, need to be stretched out flat. Then, either the index finger only, or the index finger and the middle finger at the same time, need to stay stretched out flat, and the rest of the fingers would be bent by the user. The finger indication is not only used as the beginning of a gesture, but also it should not change during the gesture. Figure 3.3 shows the difference between the two types of finger indications. We define that, when only the index finger is flat, the user is choosing a different device. And when both of the index finger and the middle finger are flat, the user intends to choose a different color for LED lights.

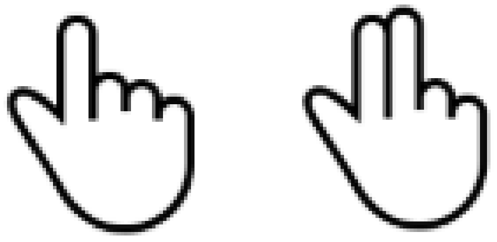
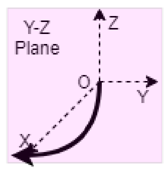

(a) Only index finger stays flat (b) Index finger and middle finger both flat

Figure 3.3: Two types of finger indications[7]

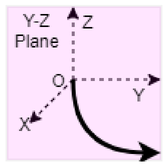
The space motion refers to a change of the hand position in different 3D dimensions. By moving the arm in different ways, the IMU sensor position could be changed in any of the 3D dimensions (according to Figure 2.3 naming scheme, these dimensions are X-Roll, Y-Pitch, and Z-Yaw). In this project, we start with testing five space motion patterns, as we think this would be adequate for the demonstration purpose of this project. As a side note, motions operated on the X-Y plane are not considered when we define gestures, because a movement on the X-Y plane would be difficult for the audience to see during a virtual presentation.
The five space motions are "Left swipe", "Right swipe", "Clockwise circle", "Counter-clockwise circle", and "Pull", as we show in Figure 3.4. The "Left swipe" is constrained on the Y-Z plane. Starting from the origin point O, the hand draws a 90-degree arc in the clockwise direction around the X axis. Similarly, the "Right swipe" draws a 90-degree arc in the counter-clockwise direction. The "Clockwise circle" starts in the same pattern as the "Left swipe", but it moves faster and will finish a whole circle using the same amount of time. The "Counter-clockwise circle" is very similar to the "Clockwise circle", the only difference is the opposite direction of the movement. Finally, the "Pull" is operated on the X-Z

[7]Icons taken from https://app.diagrams.net/

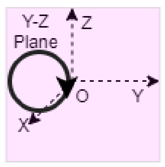
plane, unlike the other motions. The user pulls the forearm towards himself/herself, which also draws a 90-degree arc on this plane; the user then put it back along the same arc, and the hand will be back at the starting point.



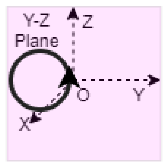
(a) Left swipe



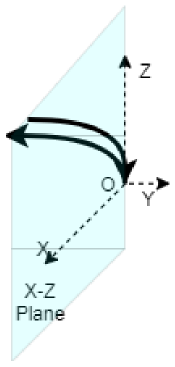
(b) Right swipe



(c) Clockwise circle

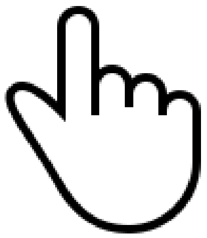


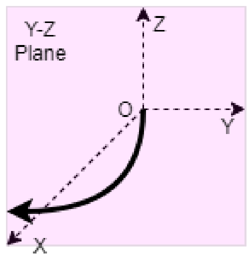
(d) Counter-clockwise circle



(e) Pull

Figure 3.4: Five types of space motions


(a) Finger indication [8]



(b) Space motion

Figure 3.5: Gesture for command "Next device"

Figure 3.5 is used to show the decomposition of a gesture, using an example of the command "Next device". The user needs to keep the index finger flat, while bending the other three fingers from the initial flat position, as shown in figure 3.5a, to indicate that the gesture now starts. As stated before, this finger position will not change during the gesture. Then the hand moves in the "Left swipe" motion as we have illustrated before. Given the gesture, the program is supposed to interpret the command "Next device" and publish it to the chosen IoT device.

3.4 Machine Learning

In this project, we would need to predict predefined gesture based on sensor data, which means that we are solving a classification problem. In Figure 3.6, we show an example of how a generic classification task can be visually defined.

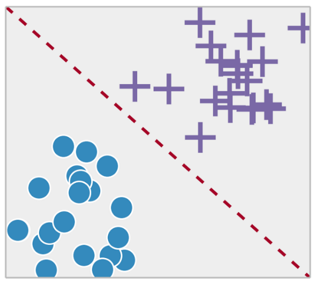


Figure 3.6: Classification model labels data points and classifies them[9]

For an early testing phase, we collecte 10 sets of movement data for each of the five space motions, and use python Seaborn[10] package to plot them. We use csv files to store the IMU sensor data, and put each set of data for every gesture in separated files. Each file includes 30 rows from different time instances during the gesture, and each row contains the yaw, pitch, and row values at that instance. To get an idea how the data points distribute, we randomly pick 2 yaws, 3 pitches, and 2 rolls as data features for the plotting purpose. The result of this analysis can be found in the Appendix in Figure A.1. Each row of plots use the same feature on the Y-axis, and each column

[8]Icons taken from https://app.diagrams.net/

[9]https://towardsdatascience.com/supervised-vs-unsupervised-learning-14f68e32ea8d

[10]https://seaborn.pydata.org/

of plots have the same data feature on the X-axis. The diagonal plots are histogram plots of the data feature on the row. We can intuitively see that the machine learning approach should work well for our project. For example, give a closer look at the plot at row pitch_0 and column pitch_20, as in Figure 3.7. We can see that the pull gesture can be distinctively separated from the rest, based on the pitch feature among the whole data values. In the end, the machine learning model automatically conducts feature extraction, and labels the data points using all data features we provide to it.

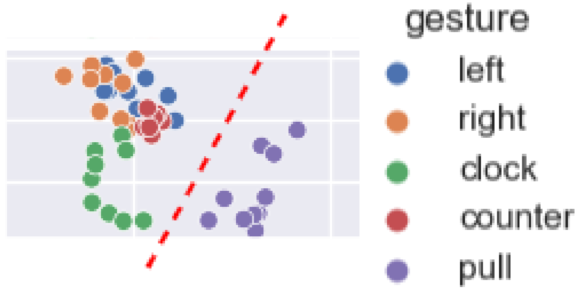


Figure 3.7: Pull gestures separated from the others on the Pitch feature

We then proceed to collect 100 sets of gesture data for each of the five motions. To see how well the prediction of different motions performs, we perform some experiment and compare the result. We use the Decision Tree Classifier (DTC) model for this task. For each round of experiment, we set one motion as the targeting motion, change the labels for the 100 targeting motion data points to "1", and change the labels for the 400 others to "0". As shown in Figure 3.6, the model tries to figure out a "line" to separate the target data points ("1"s) from the rest ("0"s) based on data features. For a test data point, it then assigns a label to this data point and compare it with the given result. With a limited working time, we skip the advanced feature extraction and simply give all of the 90 features that belong to one gesture to the model (30 Yaws, 30 Pitches, 30 Rolls). All gesture data are stored in separate csv files and will be read by the testing program. The final results of the prediction accuracy are recorded as in Table 3.1.

Space motion	Left swipe	Right swipe	Clockwise circle	Counter-clockwise circle	Pull
Accuracy	99,33%	95,33%	94,67%	95,33%	99,33%

Table 3.1: Machine learning prediction accuracy for space motions

As we can see in Table 3.1, the machine learning model works pretty well so far on predicting the space motions. In the next step, we collect 150 sets of sensor data for each type of the motions for the purpose of model training. We then develop the complete application using python3 for the project, and conduct further live testing on different gestures.

3.5 Gesture Interpretation

On the edge computing unit, which is placed on the Raspberry Pi 3B[11], we run a messaging program to receive sensor data from the glove, and use it in two kinds of ways. We either use this unit to collect the data to train our model, or we use it to predict the performed gesture in a live approach. In the subsection 3.5.1, we explain how the message flow is realized in an live running scenario. In subsection 3.5.2, we evaluate how well the workflow performs, and come up with ideas to improve our final result.

3.5.1 Workflow Design

When the Raspberry Pi receives new sensor data from the glove, the system processes it and applies the machine learning model on it, and publishes the command to the IoT devices. Figure A.2 in the Appendix shows a detailed architecture diagram. The labels in the diagram indicate the information passed between components, and the italics labels show action relationships.
The messaging program, shown as the Communication Unit on the architecture diagram, runs a MQTT

[11]https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

client which subscribes to the topic that the glove publishes sensor data to. Each MQTT message received by the MQTT client contains four flex sensor values, IMU sensor values, and a timestamp. Sensor data included in the message will later be used to interpret gestures by the machine learning model, as can be seen in Appendix B.4. We also use a state variable to switch between ready, gesture, and after_gesture states. Initially, the user stretches out all four fingers, and it will be set to the ready state. Given each message, we extract the flex sensor values and calculate whether each of the four fingers are bent or flat, and get a combined finger indication. If it is a valid, predefined finger indication (either only the index finger is flat, or the index finger and the middle finger are both flat), the program switches to the gesture state. Exactly starting from this message, we record the IMU sensor values from the following 30 messages. Once the counter reaches the end, we switch to after_gesture state, and pass a matrix of IMU sensor values to the machine learning model. To get back to the ready state, the user simply needs to stretch out all four fingers again. In Appendix B.3 there is a simplified code snipped of the actual implementation of this state switching and the gesture recording procedure.
In addition of having now a detected command, we use index variables to keep track of which device is currently being selected, so that the command will be published in the correct topic (since each IoT device subscribes to their unique topic).

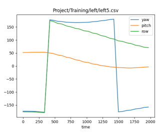
3.5.2 Gesture Testing and Adjustment

To see how well the workflow and the machine learning model perform on the fly, we conduct some more testings. This is necessary before we finalize the mapping of the gesture to the command. First, we run live experiments to verify the performance of space motion predictions, instead of reading static csv files from the folder. In the live testing, the microcontroller, which is connected to the glove, is constantly publishing sensor data as messages every 50 milliseconds. The messaging program on the Raspberry Pi receives and examines all of the messages, and determines whether the user starts a gesture through the finger indication. The developer wears the glove and makes gestures in a predefined order, and compares it to the gesture prediction given by the program. 12 repetitions are made for each gesture. The results are recorded in the following Figure 3.2.

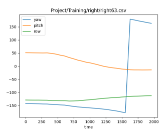
Expected motion	Result Counting					Accuracy
	Left swipe	Right swipe	Clockwise circle	Counter-clockwise circle	Pull	
Left swipe	12	0	0	0	0	100,00%
Right swipe	1	11	0	0	0	91,67%
Clockwise circle	0	0	12	0	0	100,00%
Counter-clockwise circle	0	2	6	4	0	33,33%
Pull	0	0	0	0	12	100,00%

Table 3.2: Live test of five motions

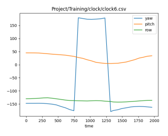
We can see that during the live testing, all of the other motions can be well predicted, except for the counter-clockwise circle. To find out the reason why counter-clockwise circle motion has relatively much lower prediction accuracy, we plot the continuous values of the Yaw, Pitch, and Roll angles during each gesture, and we get the result as shown in Figure 3.8.



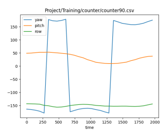
(a) Left swipe



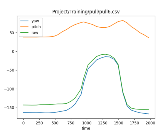
(b) Right swipe



(c) Clockwise circle



(d) Counter-clockwise circle



(e) Pull

Figure 3.8: Plotting of five types of space motions

From the plots, we can see that the "Counter-clockwise circle" plot, which is operated on the Y-Z plane, shares resemblances compared to the other gesture plots that are also on the same plane ("Left swipe", "Right swipe", and "Clockwise circle"). This might be the reason why it did not have a good prediction accuracy during the live experiment, and we decide to discard this motion for our gesture-to-command mapping. After the adjustment, we finalize the gesture-to-command mapping, as shown in the following Table 3.3.

Finger indication	Space motion	Command
Index finger stays flat	Left swipe	Next device
	Right swipe	Previous device
Index finger and middle finger stay flat	Left swipe	Next color
	Right swipe	Previous color
	Clockwise circle	Lower brightness
	Pull	Higher brightness

Table 3.3: Gesture-to-command mapping

3.6 IoT Devices

Theoretically, any IoT device that is able to connect to a MQTT server can be used here in the project. Restricted to the actual devices we currently have, we are exclusively using LED lights for the demonstration purposes. To show the scalability of the project, we also develop a web application, which is a MQTT subscriber and shows the correct LED color pattern on the front page.
We use one ESP8266[12] microcontroller to connect two LED lights. Each microcontroller subscribes to two topics, "/esp8266/⋆.1", "/esp8266/⋆.2", and each of them corresponds to a different light. The ⋆ represents a specific id of a microcontroller, since there can be more than one in the system. We differentiate between an analogue (.1) and a digital(.2) LED. The reason for the differentiation is, on the ESP8266, there are not enough Pulse Width Modulation (PWM) pins to control more than one dimmable LED which. Since we want to implement more LEDs on the given microcontrollers, we also configure a digital LED using the other non PWM pins.
The message received on the IoT device contains the instruction on a change of the color scheme, and we use an if-else block in the code to act accordingly.

[12]https://www.mikrocontroller-elektronik.de/nodemcu-esp8266-tutorial-wlan-board-arduino-ide/

We use numbers from 0 to 7 for the color code, and it translates to a RGB color. For example, if the color is set to "5", which is "101" in binary, it then translates the color to the RGB value "FF00FF" which is the color Magenta. If the command comes as "Next color" and when the current color is 4, then the color code 5 will be chosen and the light will be set to Magenta. The brightness ranges between 0% and 100%. Upon a command "Lower brightness" or "Higher brightness", we decrease/increase the brightness value by 10% accordingly for the analog LEDs. In Appendix B.5 the code displayed that sets up the LED when a change command is recognized.
The web application is a JavaScript program, which also runs the MQTT protocol and subscribes to its own topic. Given a command, we decode it in the same way as we did for the LED lights, and simply change the css setting for the LED simulation component to show the color scheme change.

4 Evaluation of solution

4.1 Performance

We measure the performance of the final project during an intensive live testing. A total number of 494 gestures have been attempted, and the results, together with the time spent for the gesture interpretation, are logged. The result is shown in the Figure 4.1. After discarding the counter-clockwise circle motion, the overall gesture recognition works good in general. In addition, during the past model training session, we notice that previous training gestures were only collected from one developer, which caused live gestures made by the other developer cannot be recognized well. So we adapt to the fact that different people have their own way of making hand motions, and have added training data from the other developer to increase flexibility. Other than the accuracy, the average time needed by machine learning model prediction is lower than the theoretical reaction time of 361 ms for humans given 1 bit of information [1], which means that the commands are sent out to the IoT devices before the user could notice.

Expected Command	Result Counting		Accuracy	Average time (s)
	Correct	Wrong		
Next device	67	2	97,10%	0,0560
Previous device	50	5	90,91%	0,0574
Next color	66	6	91,67%	0,0583
Previous color	62	2	96,88%	0,0605
Lower brightness	119	5	95,97%	0,0607
Higher brightness	104	6	94,55%	0,0603

Table 4.1: Final test result

4.2 Strengths

We think the project made a good attempt to solve the initial problem, and there were some solid progress towards the goal, which is to provide users with an easier and more intuitive way to control IoT devices. The "smart glove" is easy to use, even for people with limited motion functionalities, since predefined gestures are intuitive and short. The gestures can be well interpreted by our program, and the machine learning model is extendable to new gestures. And theoretically, any MQTT compatible IoT device can be implemented into the program. We see that the potential of the use cases of the smart glove controlling system is infinite.

4.3 Weaknesses and Improvements

In the final presentation, a question brought up by the audience is very worth considering. Situations might happen where the connection gets lost in the middle of a gesture. We need a protection mechanism, such that if the system detects that the sensors are offline, then it should not attempt to predict this gesture at all. This way we could avoid generating and sending wrong commands that are not intended from the user.
During testing, we notice a weakness on the hardware configuration of the glove that needs to be improved. So far we use extended cords to connect sensors and the microcontroller. However, with the glove being put on and off, and the frequent hand movement, some cords might be off at some point and cut off the sensor connection. There has also been minor position shifts of the flex sensor, which is why we have been reading flex sensor values in a rather static way. To have a more stable build of the glove, we would firstly solder the components, and moreover, explore a replacement for the flex sensors. Another weakness we notice is that, the gesture training somehow depends on the personal movements from which the data are collected, in other words, who trained the model. When the model is only trained by one person and will be used by this same person, then the overall accuracy will be close to 100%, as we noticed during testing. However, if another person attempts to use the same glove, the result will differ as he/she might move the arm in a different way. If this product is released to the market, there would potentially be numerous customers and each of them might move the arm in their own ways. In this scenario, the performance of the product would not be ensured.

4.4 Future Work

As mentioned before, we are using all of the 90 features of a gesture (30 Yaws, 30 Pitches, 30 Rolls) to run our machine learning model. To improve the result of this, we would need to spend more time on data analysis and data augmentation.
To make sure that we achieve the goal to implement an "intuitive" system, we would have to conduct an empirical survey to ensure that this requirement is fulfilled. Another aspect to continue working on is that there is no frontend portal in the project. Any change in the configuration needs to be done directly in the backend code. If the user wants to add a new IoT device, or disable an existing one in the system, it is currently impossible to easily do so in a frontend interface. It is also the case for a change in the predefined gestures. If we would have more time, we would develop a frontend page to allow new gesture training, old gesture discarding, as well as adding and deleting commands and IoT devices from the system easily in a visual way. This could also fix the weakness of the project being adaptive to more people, as they could train their own glove. However, this might take a tremendous amount of time and make the overall workflow much more complex. This is certainly a trade-off that we need to consider.

References

[1] R Hyman. Stimulus information as a determinant of reaction time. *Journal of Experimental Psychology*, 45 (3):188–96, 1953.

[2] OASIS Standard. Mqtt version 5.0. *Retrieved June*, 22:2020, 2019.

A Appendix: Additional Figures

A.1 Data Feature table

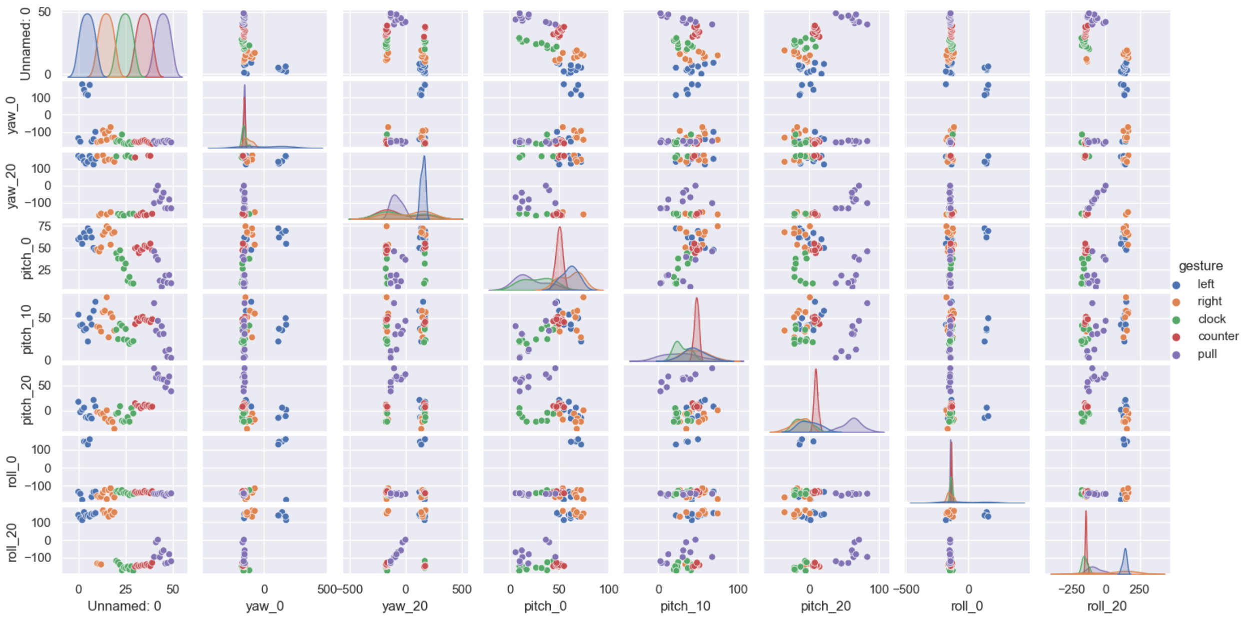
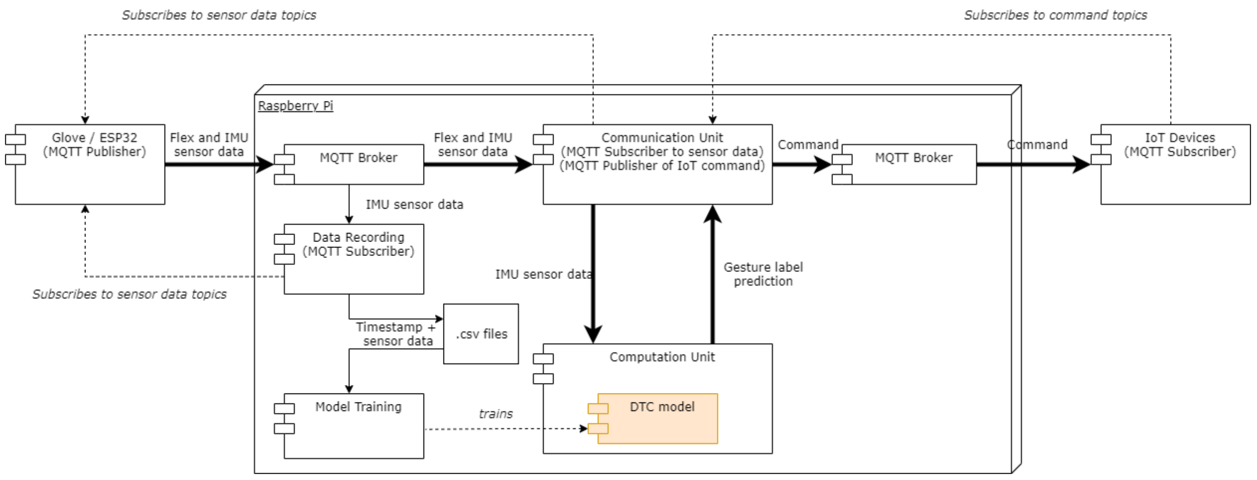


Figure A.1: Evaluation of plots on 7 data features

A.2 Architecture Diagram



Subscribes to sensor data topics
Subscribes to command topics
Raspberry Pi
Glove / ESP32 (MQTT Publisher)
Flex and IMU sensor data
MQTT Broker
Flex and IMU sensor data
Communication Unit (MQTT Subscriber to sensor data) (MQTT Publisher of IoT command)
Command
MQTT Broker
Command
IoT Devices (MQTT Subscriber)
IMU sensor data
Data Recording (MQTT Subscriber)
Subscribes to sensor data topics
IMU sensor data
Gesture label prediction
Timestamp + sensor data
.csv files
Computation Unit
Model Training
trains
DTC model

Figure A.2: Architecture diagram

B Appendix: Code Snippets

B.1 Get finger angles and convert IMU quaternion to Euler angles

Listing 1: gloveESP32.ino

```
    //calculate the flex sensor resistance and map them to angles in 0 to 90 degrees
    float getFingerAngle(ESP32AnalogRead adc){
        float resistorV = adc.readVoltage();
        float flexV = FLEX_VCC-resistorV;
        float flexR = R_DIV * (flexV/resistorV);
        float angle = map(flexR, STRAIGHT_RESISTANCE, BEND_RESISTANCE, 0, 90.0);
        return angle;
    }

    //Euler angle struct
    struct euler_t {
        float yaw;
        float pitch;
        float roll;
    } ypr;

    //convert quaternion representations to Euler angles
    void quaternionToEuler(float qr, float qi, float qj, float qk, euler_t* ypr,
        bool degrees = false) {
        float sqr = sq(qr);
        float sqi = sq(qi);
        float sqj = sq(qj);
        float sqk = sq(qk);

        ypr->yaw = atan2(2.0 * (qi * qj + qk * qr), (sqi - sqj - sqk + sqr));
        ypr->pitch = asin(-2.0 * (qi * qk - qj * qr) / (sqi + sqj + sqk + sqr));
        ypr->roll = atan2(2.0 * (qj * qk + qi * qr), (-sqi - sqj + sqk + sqr));

        if (degrees) {
            ypr->yaw *= RAD_TO_DEG;
            ypr->pitch *= RAD_TO_DEG;
            ypr->roll *= RAD_TO_DEG;
        }
    }
}
```

B.2 Messaging program

Listing 2: communication.py

```
    #subscribe to the Glove_Topic as soon as it is connected to the Glove publisher
```

```
def on_connect(client, userdata, flags, rc):
    client.subscribe(GLOVE_TOPIC)

#receives the incoming message of the subscribed topic
def on_message(client, userdata, msg):
    #the Queuestores incomming message
    #which are later processed one after the other
    messageQueue.pushNewMessage(str(msg.payload), client)
    #function call that starts on processing the received message
    controlSystem.handle_queue(client)

#setup of the mqtt client
def main():
    mqtt_client = mqtt.Client()
    mqtt_client.username_pw_set(MQTT_USER, MQTT_PASSWORD)
    mqtt_client.on_connect = on_connect
    mqtt_client.on_message = on_message

    mqtt_client.connect(MQTT_ADDRESS, 1883)
    mqtt_client.loop_forever()
```

B.3 State switching and gesture interpreting

Listing 3: app.py/get_command

```
# given a global gesture mode and message index.
def get_command(self, tokens):
    #gesture_mode 1 is "ready"
    if gesture_mode == 1:
        hand = get_finger_positions()
        if hand == 1 or hand == 3:
            gesture_mode = 2
    #gesture_mode 3 is "after_gesture"
    if gesture_mode == 3:
        hand = get_finger_positions(tokens[:4])
        if hand == 15 or hand == 7:
            gesture_mode = 1
    #gesture_mode 2 is "gesture"
    if gesture_mode == 2:
        #start of gesture recording
        if message_index == 0:
            hand = get_finger_positions()
            sensor_data = []
        #during gesture recording
        if message_index < 30:
            #write IMU data to matrix
            write_to_matrix(tokens[5:8])
            message_index += 1
        #end of gesture recording
        if message_index == 30:

            #get_gesture_prediction() accesses the written matrix
            gesture = get_gesture_prediction()

            #interpret the data result
            command = ''
            if hand == 1:
                if gesture == 1:
                    command = 'next_device'
                elif gesture == 2:
                    command = 'previous_device'
                else:
                    command = 'gesture_unrecognized'
            elif hand == 3:
                if gesture == 1:
                    command = 'next_color'
                elif gesture == 2:
                    command = 'previous_color'
                elif gesture == 3:
                    command = 'lower_brightness'
                elif gesture == 4:
                    command = 'higher_brightness'
                else:
                    command = 'gesture_unrecognized'
            else:
                command = 'gesture_unrecognized'

            message_index = 0
            gesture_mode = 3

            return command
    return 'incomplete_command'
```

B.4 Using machine learning model for gesture prediction

Listing 4: app.py

```
#matrix storing sensor data for one gesture
sensor_data = []
#load trained machine learning model
model = pickle.load(open('dtc.sav', 'rb'))

#write sensor data to matrix
def write_to_matrix(tokens):
    current_time =
        (datetime.datetime.now() - start_time).total_seconds() * 1000
    row = [current_time]
    row += tokens
    sensor_data.append(row)

#process format of matrix
def matrix_transpose_and_flatten():
    #store previously collected sensor data
    sensor_data_matrix = np.concatenate(np.array(sensor_data)
        .transpose())[30:]
    sensor_data = []

#get gesture prediction
def get_gesture_prediction():
    matrix_transpose_and_flatten()
    prediction = get_machine_learning_prediction()
    return evaluated_prediction(prediction)

#get prediction from machine learning model
def get_machine_learning_prediction():
    user_sensor_data_matrix = []
    user_sensor_data_matrix.append(sensor_data_matrix)
    prediction = model.predict(user_sensor_data_matrix)
    return prediction[0]
```

B.5 Color and brightness adjustments on the LED

Listing 5: LED_Device.ino

```
//This function sets the LED based on the brightness and the color,
//which was determined in a function before.
//The brightness is a given percentage, the color a number from 0 to 7.
void setLED(int color, int brightness, char* topic){
    //A color is interpreted as 3Bit number which is decoded as
    //RGBb(binary) color e.g. 001b is green, 100b is red.
```

```
        int blue = color % 2;
        color = color >> 1;
        int green = color % 2;
        color = color >> 1;
        int red = color % 2;

        //This checks to decide which of the two mounted light should be changed.
        if(!strcmp(topic, analog_led_topic)){
            set_analog_led( red, green, blue, brightness);
        } else {
            set_digital_led(red, green, blue);
        }
    }

    //This shows how a analog LED is set up. For a digital it is similar
    //except there is no brightness involved.
    void set_analog_led(int red, int green, int blue, int ratio) {
        //This calculation makes the natural logarithmic
        //brightness changes appear linearly.
        int pwmIntervals = 100;
        int stepSize = (pwmIntervals * log10(2)) / (log10(255));
        int brightness = pow (2, (ratio / stepSize)) - 1;

        //Set the different pins of the RGB LED,
        //setting the same brightness for all selected LEDs.
        analogWrite(RED_ANALOG_PIN, red * brightness);
        analogWrite(GREEN_ANALOG_PIN, green * brightness);
        analogWrite(BLUE_ANALOG_PIN, blue * brightness);

}
```