CG4002 Embedded System Final Design Project

August 2021 semester

"Dance Dance"

Final Design Report

Group **B01**	Name	Student #	Sub-team	Specific Contribution
Member #1	Yuan Jiayi	A0177893H	HW Sensors	Section 2.3 Section 3.1 Section 6
Member #2	Zhang Yihan	A0177861R	HW FPGA	Section 2.1 Section 3.2 Section 6
Member #3	Andrew Lau Jia Jun	A0182815B	COMMS Internal	Section 2.2 Section 2.4 Section 4.1 Section 6
Member #4	Alex Teo Kang Jie	A0180338E	COMMS External	Section 4.2 Section 6 Section 7
Member #5	Jess Teo Xi Zhi	A0182668N	SW Machine Learning	Section 2.3 Section 5.1 Section 6
Member #6	Amir Azhar Bin Azizan	A0182846U	SW Dashboard	Section 1 Section 2.4 Section 5.2 Section 6

Table of Contents

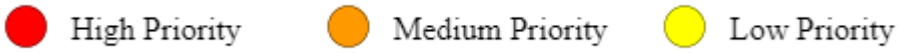
1 System Functionalities

This section of the report details the intended system functionalities of our application. Our team of developers are guided by the principles of the *Agile Manifesto* during our software/product development process. The first and most crucial principle that should be adhered to is to "satisfy the customer" (Beedle, 2001). Our team will be putting the people first. User stories and use cases are great tools that can help us put end users at the center of our conversation. This will aid us in deciding the requirements of our system and the features we will be implementing, as well as in prioritization.

1.1 User Stories

A user story is an informal, general explanation of a software feature written from the perspective of the end user. Its purpose is to articulate how a software feature will provide value to the customer. It is important to note that since our application is catered to both dance coaches and trainees, we have come up with user stories that are categorised in either point of view. The table below shows these user stories that will eventually be the stepping stone for our software development process.

Colour Code:



Trainee

- As a trainee, I want to know if I am doing the right dance moves so that I can easily adjust to it if needed.
- As a trainee, I want to know if I am too fast or too slow so that I can better synchronize with the class.
- As a trainee, I want a wearable that is as light as possible so that I can remain comfortable throughout the dance lesson
- As a trainee, I want a wearable that is small and unnoticeable so that it will not impede any of my movements.

- As a trainee, I want to be able to follow a dance without a camera so that I can use this application wherever and whenever I want.
- As a trainee, I want to be able to see post lesson analytics of my movements so that I can better improve my dancing skills.

- As a trainee, I want to be given advice by an automated bot so that I can easily adjust into a dance routine.

Coach

- As a coach, I want to be able to see if my trainees are doing the correct moves so that I can advise them to make the necessary adjustments.
- As a coach, I want to see if there are any delays so that I can better set the speed of my class to cater to the trainees' needs.

- As a coach I want to be able to see post lesson analytics of my trainees so that I can better tailor my advice for them.
- As a coach, I want to be able to pinpoint the position of a trainee relative to other trainees so that it will be easier for me to give instructions when needed.

- As a coach, I want to be able to see the live figure (a stickfigure) of my trainees' movements so that I can have a visual representation of how they are performing.

1.2 Use Cases

Similar to user stories, use cases also give us developers a better understanding of the user roles, their goals and acceptance criteria. Use cases can provide us with a more in depth look at the steps a user goes through to complete their goal using our software system. This will undoubtedly help us gain a better understanding of how the system should behave and its complexity. Shown below is the use case diagram of our application, followed by the specific use cases demonstrated.

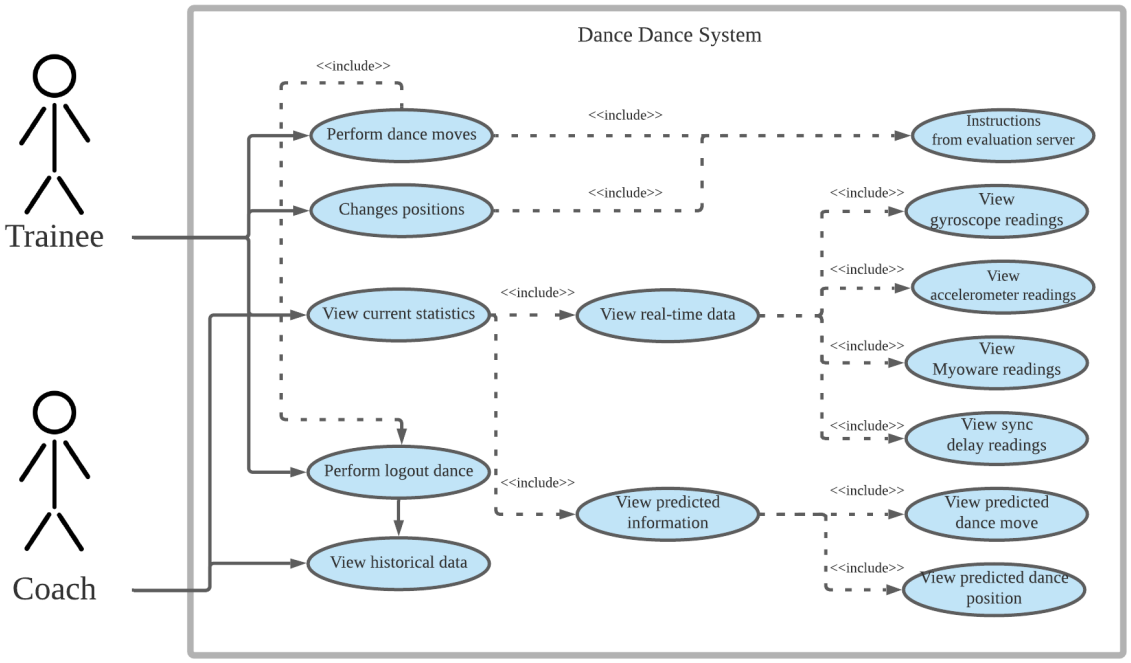


Figure 1.2.1: Use case diagram

UC01 - Accurate Classification of Positions and Dance Moves
Actor: Coach/Trainee
Main Success Scenario (MSS):

1. Actor logs in into the Dance Dance system.
2. Evaluation server outputs the required dance moves and positions of the dancers.
3. Dancers will have to perform the dance moves and/or rearrange themselves as updated on the dashboard.
4. The system will predict the dance moves, relative locations and synchronization delay of the dancers

5. The dashboard will be updated accordingly to show real-time statistics such as raw sensor data, predicted dance moves and delay of the dancers.
6. Steps 2-6 will repeat itself whenever the evaluation server outputs new dance moves and positions.

Use case ends.

UC02 - End Dance Session
Actor: Coach/Trainee
Main Success Scenario (MSS):

1. The evaluation server will inform the actors to end the dance session.
2. Actors will then perform the final dance move to indicate to the system that they are ready to end the session.
3. Upon detection of the final dance move, the dashboard will display that the dance session has come to an end
4. Users will be redirected to the history page to showcase the data received from the session.

Use case ends.

1.3 Feature Lists

Based on the above user stories and use cases, we have managed to get a better idea on the set of features to implement for our application as well as how to prioritise them. As such, the following list showcases the intended features of our application.

- Accurate sensor data received from sensors and displayed on dashboard
- Accurate prediction of dance moves, relative positions and synchrony of dancers
- Fast detection speed of dance moves
- Efficient power consumption of overall wearable system
- Stable and reliable connection between server and system throughout use

- Compact and comfortable wearable device
- Minimal delay (real-time) when updating information on dashboard
- Informative dashboard with excellent user experience (navigation and understanding of data displayed)

- Aesthetically pleasing UI for dashboard
- One-size-fits-all wearable design

2 Overall System Architecture

2.1 System Architecture Design

The architecture design of our system is shown in Figure 2.1.1 below:

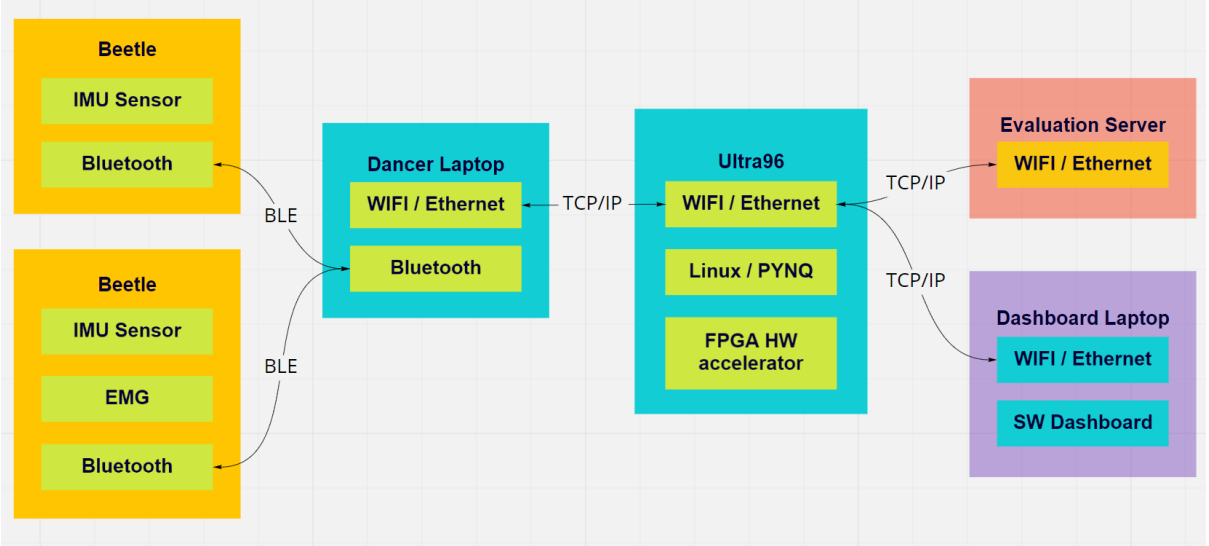


Figure 2.1.1: System architecture design

2.2 Components Communication

2.2.1 Initial Design

The initial hardware components that were involved are as listed below:

- 3V Non-Rechargeable Lithium Coin Cells (CR2032)
- MPU 6050
- MyoWare Muscle Sensor
- Bluno Beetle
- Laptop
- Ultra96-V2

We had initially planned for each dancer to wear 2 Bluno Beetles and 2 MPU 6050, one pair on the chest and the other on the right hand. Additionally, one of the dancers will also wear the MyoWare Muscle Sensor on the right forearm. Figure 2.2.1.1 illustrates the component interactions for a dancer who is not required to wear the MyoWare Muscle Sensor.

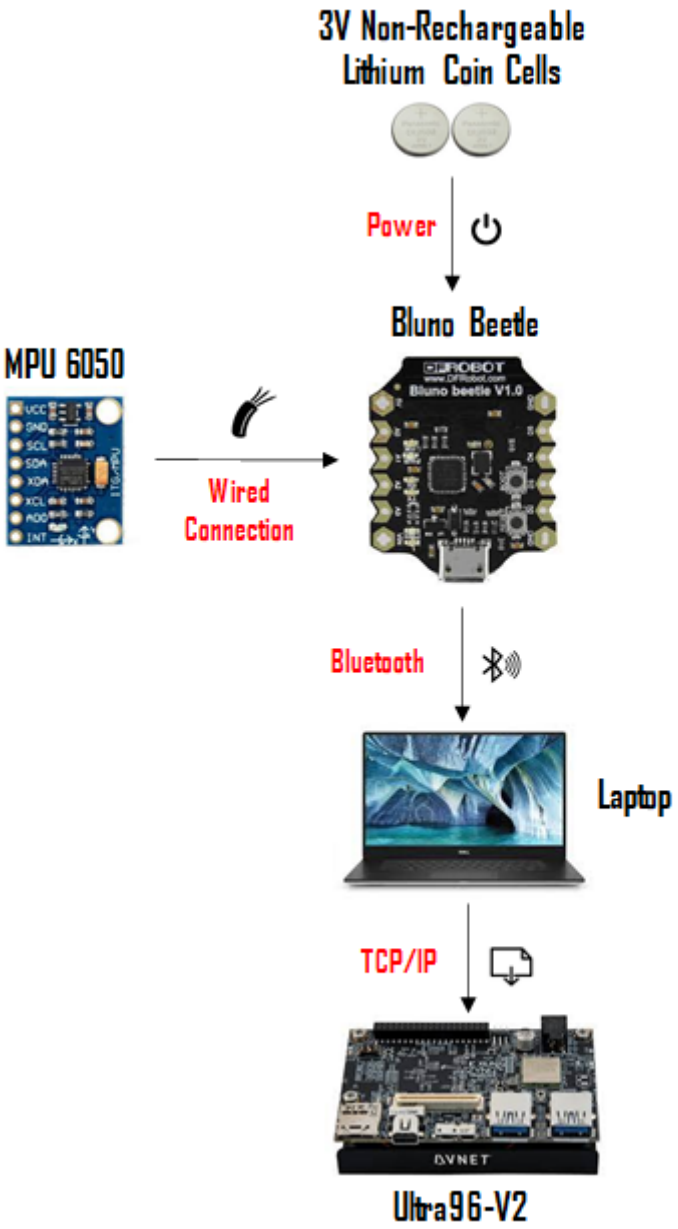


Figure 2.2.1.1: Initial Hardware Components Interaction (without Myoware Muscle Sensor)

The MPU 6050 is the hardware sensor that contains both the accelerometer and gyroscope. It is connected to the Bluno Beetle through a wired connection (Serial) and the sensor readings from the accelerometer and gyroscope are sent to the Bluno Beetle. The Bluno Beetle is powered by two 3V Lithium Coin cells, and its purpose is to receive and process the data transmitted by the MPU 6050 into the necessary packet formats. Thereafter, the Bluno Beetle will transmit the formatted data packets to the laptop through Bluetooth with 3-way handshake protocol implemented.

Once the laptop has received the data packets from the Bluno Beetle, it will proceed to process and parse the data received into the required format before transmitting the formatted data to the Ultra96-V2 through the TCP/IP connection. The Ultra96-V2 receives the formatted data from the laptop and it will communicate with the evaluation server as well as the dashboard separately through TCP/IP.

The data is fed into the software machine learning model that harnesses the processing power of the FPGA to predict the executed dance move, as well as the position of dancer and the synchronisation delay. On the end of the dashboard, the performance of the dancers in terms of correctness of dance moves and dancer position is reflected. The dancers are able to improve on their moves from the feedback provided by the system. On the evaluation server, the dance moves and dancer position that the dancers should adhere to is displayed. The server also receives data from the Ultra96-V2 and evaluates if the dance moves and dancer position are correct.

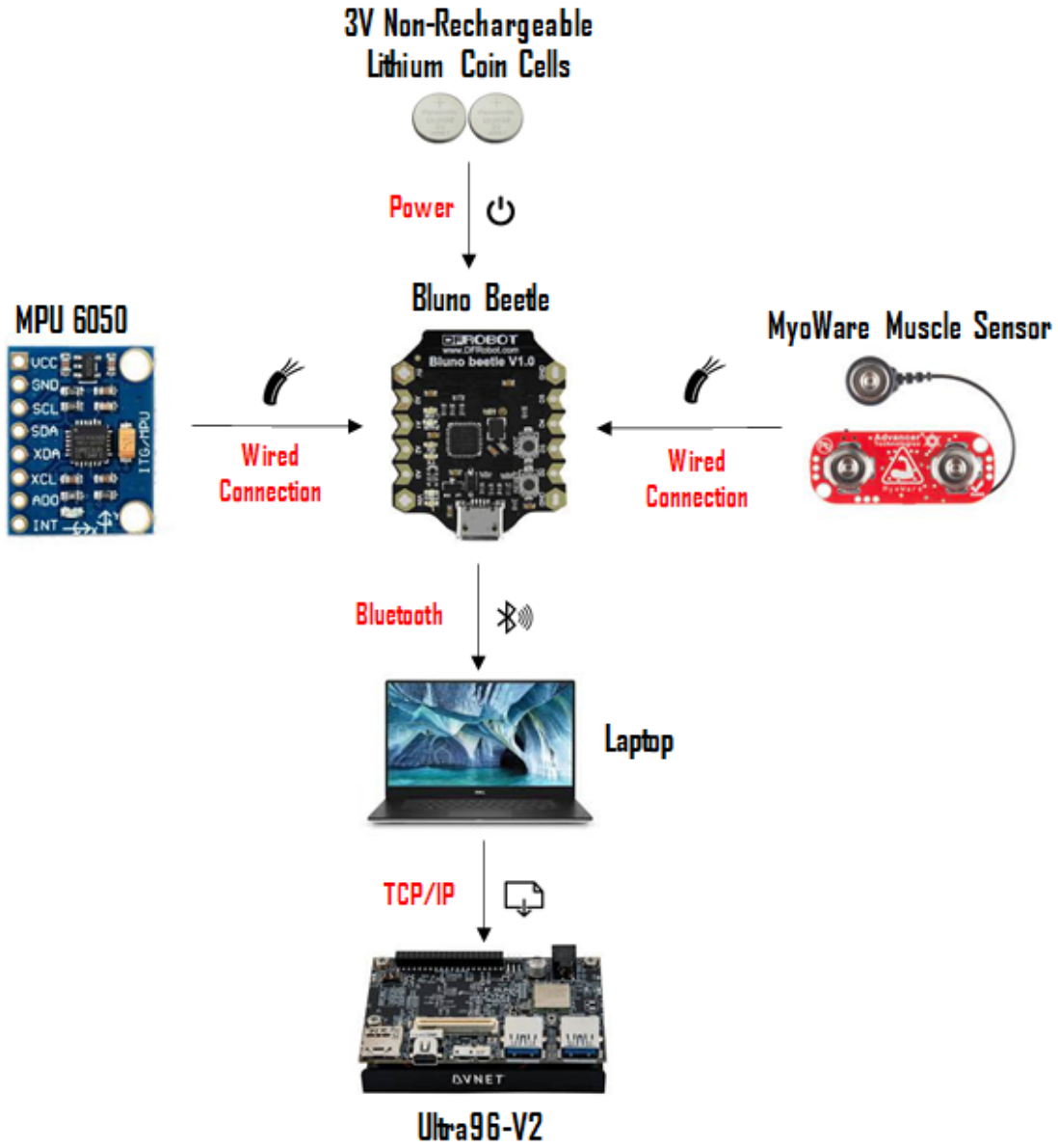


Figure 2.2.1.2: Initial Hardware Components Interaction (with Myoware Muscle Sensor)

Figure 2.2.1.2 illustrates the interaction channels that exist between the hardware components for the only dancer who is wearing the MyoWare Muscle Sensor. The only difference in this setup is the additional communication between the MyoWare Muscle Sensor and the Bluno Beetle through a wired connection. Hence, this dancer will have the extra Myoware Muscle Sensor attached to the same Bluno Beetle.

2.2.2 Final Design

The final hardware components that were involved are as listed below:

- 3.6V Rechargeable Lithium Coin Cells (LIR2450)
- MPU 6050
- MyoWare Muscle Sensor
- Bluno Beetle
- Laptop
- Ultra96-V2

In our final implementation, after rounds of rigorous testing, we decided that the chest beetle was not necessary in determining the dance move or the change in dancer's position. Hence, we decided to remove the chest beetle and each dancer was only required to wear 1 Bluno Beetle and 1 MPU 6050 on the right hand. Similarly, one of the dancers is still required to wear the MyoWare Muscle Sensor on the right forearm. An example of the dancer's physical setup will be illustrated in Section 2.3.

Additionally, we also changed the power source from the initial 2x3V Non-Rechargeable Lithium Coin Cells to 2x3.6V Rechargeable Lithium Coin Cells due to the frequent disconnections that were plaguing the Bluno Beetles. We suspected that the 2x3V Non-Rechargeable Lithium Coin Cells were supplying an insufficient amount of power to the device. It was also costly to replace the Coin Cells and one pair was only able to last for a few rounds of testing.

Hence, we decided to adopt the use of rechargeable batteries that are able to support and sustain the requirements of the system for a longer period of time. Figure 2.2.2.1 illustrates the final component interactions for a dancer who is not required to wear the MyoWare Muscle Sensor with our new choice of power source.

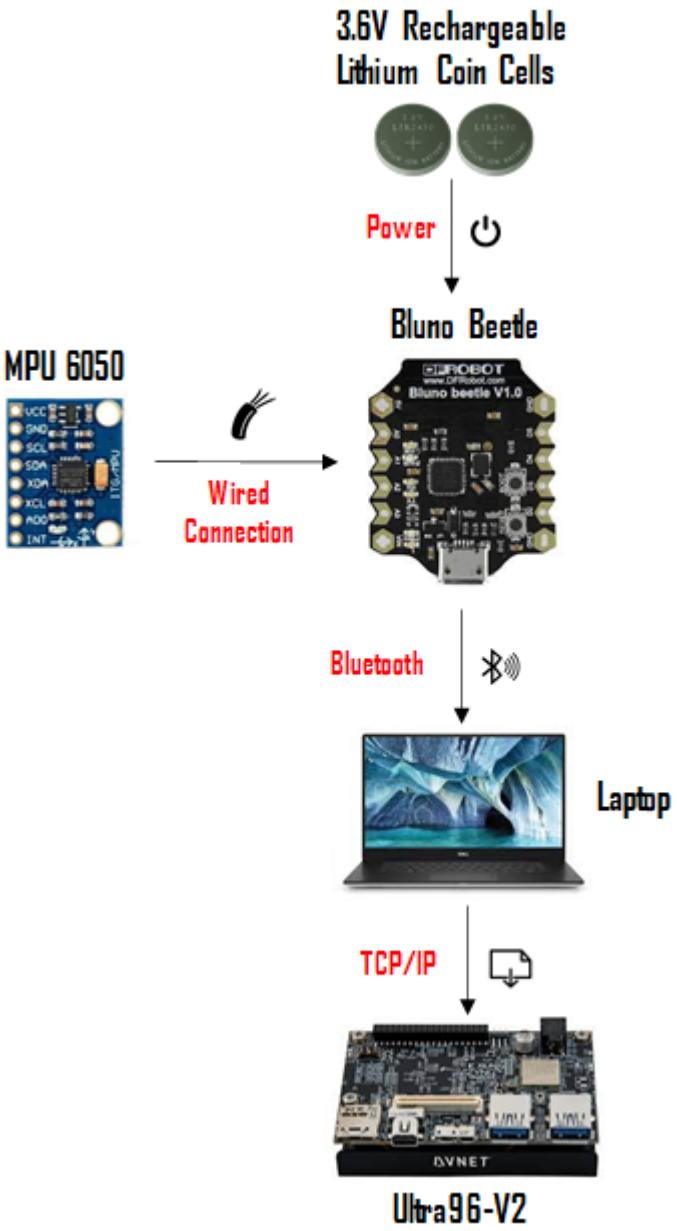


Figure 2.2.2.1: Final Hardware Components Interaction (without Myoware Muscle Sensor)

Figure 2.2.2.2 illustrates the final hardware interaction channels between the hardware components for the dancer who is wearing the MyoWare Muscle Sensor. Similarly, the initial 3V Non-Rechargeable Lithium Coin Cells have been replaced with the 3.6V Rechargeable Lithium Coin Cells.

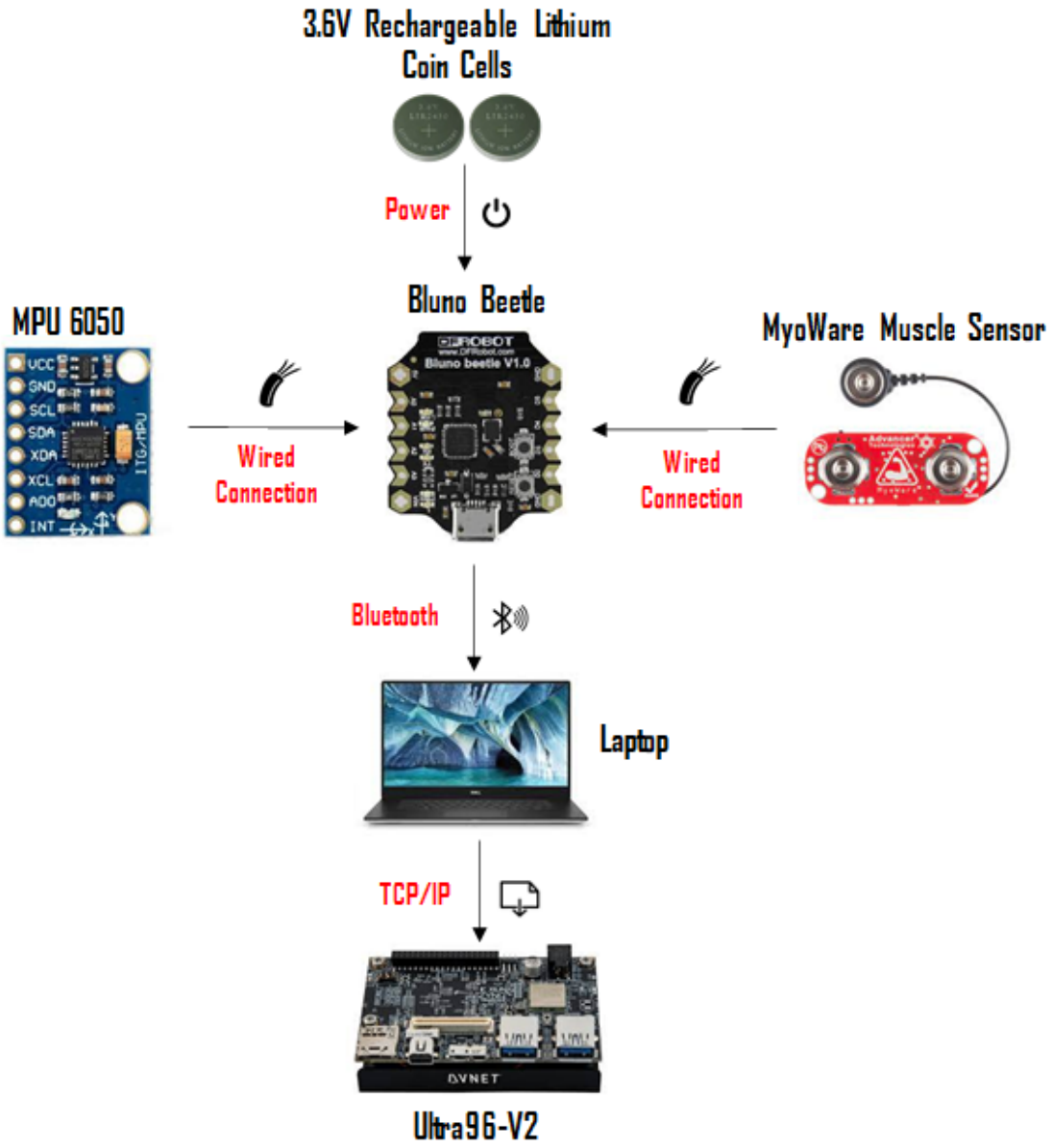


Figure 2.2.2.2: Final Hardware Components Interaction (with Myoware Muscle Sensor)

In summary, the communication among the different components in our project largely remained the same as the set up that we had in mind during our initial planning. However, we have decided to adopt the use of rechargeable 3.6V Lithium Coin Cells in our final implementation instead as elucidated and illustrated in the diagrams above.

2.3 Initial and Final Form of System

In this section, the initial and final form of the system will be shown with photos. The reason for the changing of the design will be provided as well.

2.3.1 Initial Design

All the components and the placement of them for one dancer in the initial design are shown as below.

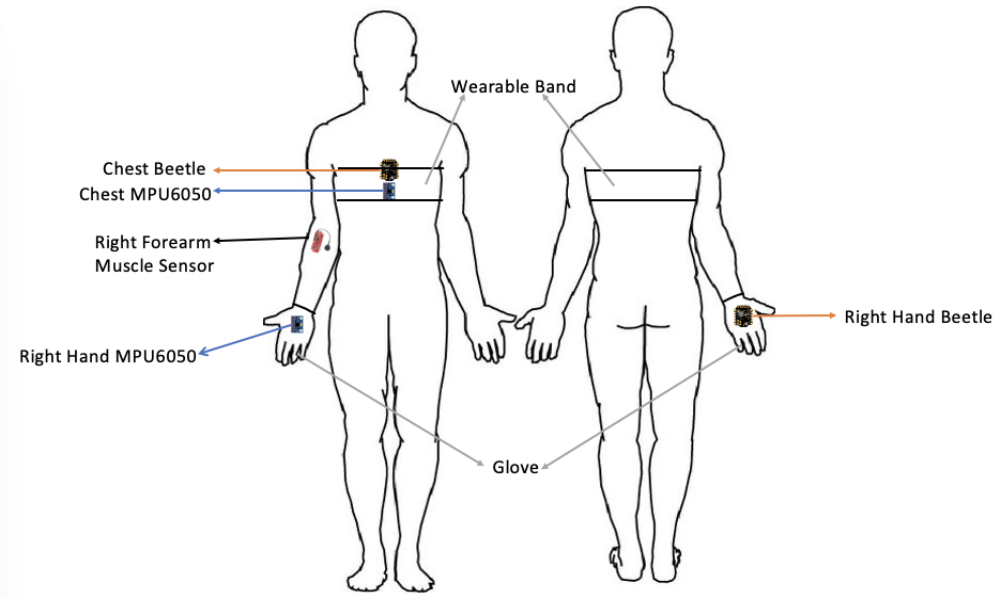


Figure 2.3.1.1: Initial Design of the System

Figure 2.3.1.2: Photo of Initial Design

Right hand

- 1 glove
- 1 Beetle
- 1 MPU6050

Right forearm

- 1 MyoWare Muscle sensor (connect to the right hand Beetle)

Chest

- 1 wearable band
- 1 Beetle
- 1 MPU6050

2.3.2 Final Design

All the components and the placement of them for one dancer in the final design are shown as below.

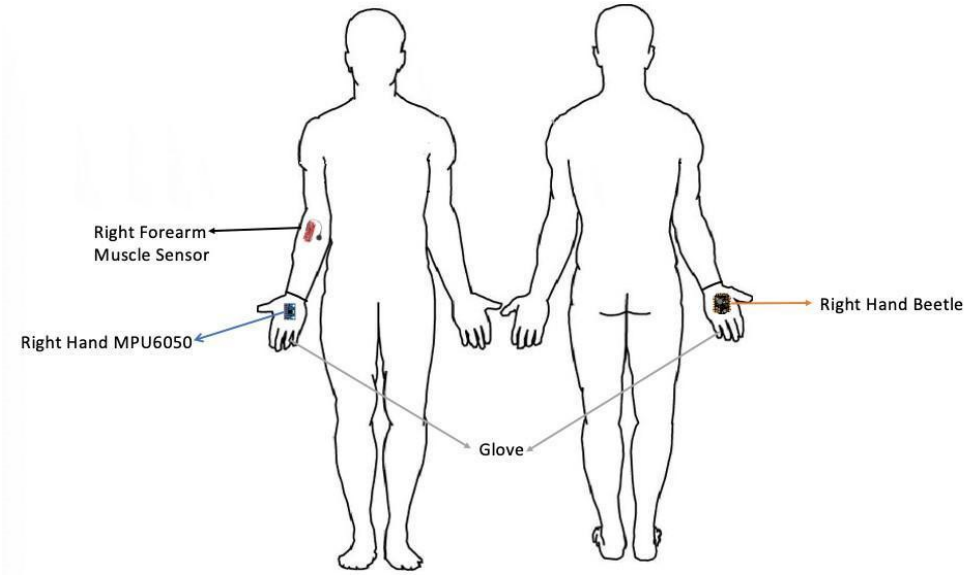


Figure 2.3.2.1: Intended Final Design of the System

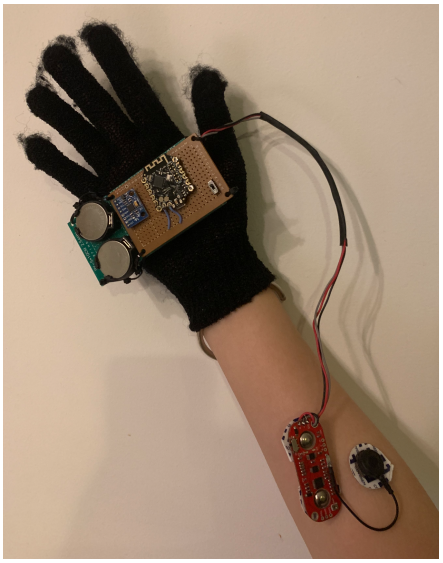


Figure 2.3.2.2: Photo of Final Design (with EMG Sensor)

Right hand

- 1 glove
- 1 Beetle
- 1 MPU6050

Right forearm

- 1 MyoWare Muscle sensor (connected to the right hand Beetle)

2.3.3 Design Changes

In the initial design, we used 2 beetles and 2 MPUs for 1 dancer, and we had set the intended final design to 1 beetle and 1 MPU per dancer if the system is able to work well. The reason for this is to ensure that we can have a backup set for each dancer, as well as a smoother data transmission with less possibility of bluetooth interference between the multiple beetles.

After the first run without the chest set in week 8, we realized that the right hand set is adequate in ensuring that the dance movements are accurately predicted. We decided to retain the design with only the right hand set and tested several times for the detection of position in week 10. Likewise, the accuracy for both position and dance move detection falls in an acceptable interval. Hence, we decided to adopt the use of only 1 beetle and 1 MPU for each dancer in the final implementation. The final set up can be seen in Figures 2.3.2.1 and 2.3.2.2.

2.4 Algorithm For Activity Detection Problem

2.4.1 Initial Algorithm

This section will outline the initial high-level algorithm used in our application to perform dance move classification, dancer position prediction and muscle fatigue measurement. Our initial implementation included the use of 2 Bluno Beetles, 1 on the chest and the other on the right hand. The plan for the detection of position change was initially to be executed by the ML model.

The major steps of the algorithm are listed below. The steps are also succinctly summarized in a sequence diagram in Figure 2.4.1.1 below.

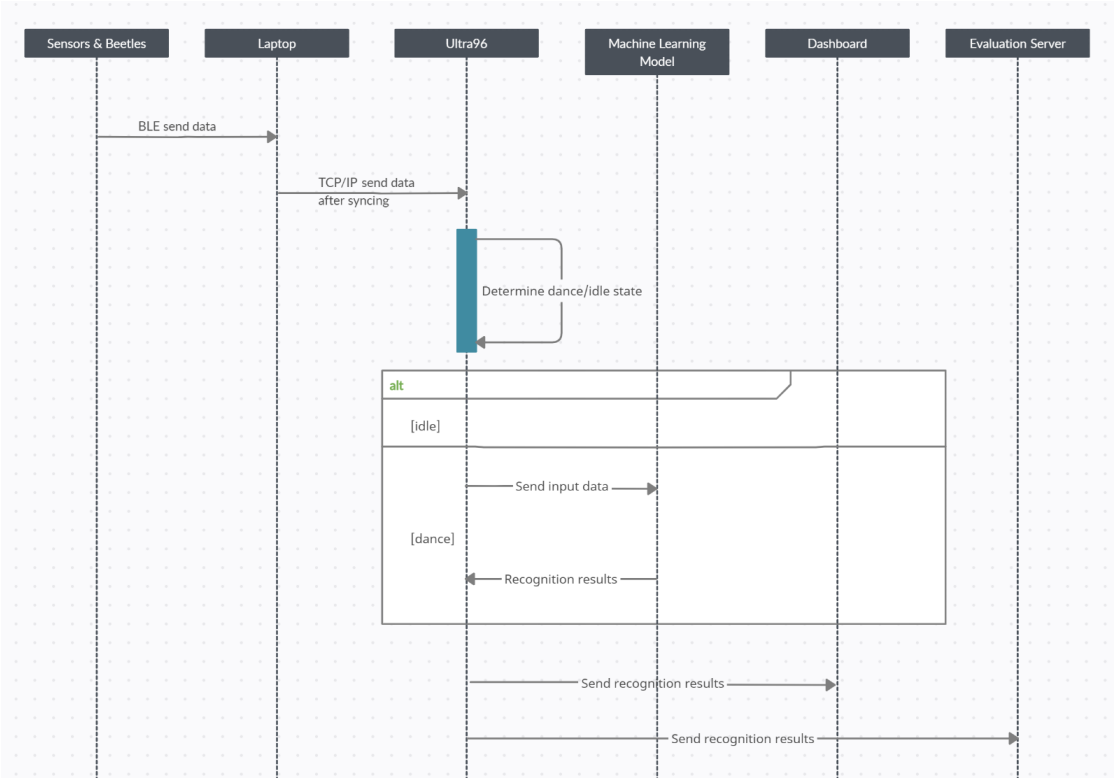


Figure 2.4.1.1: Sequence diagram for initial detection algorithm

Dance Move and Dancer Position Classification

1. Inertial Measurement Units (IMU), which are placed on the right hand and on the chest area, will output data when a dancer performs a move.
    a. IMUs will output 3-axial accelerometer and gyroscope readings

2. Readings will be collected by the Bluno Beetles, which are placed on the right hand and on the chest area.
    a. Single Bluno Beetle placed in each area to receive data from the respective IMUs
    b. Each IMU and Bluno Beetle pair have a wired connection

3. Bluno Beetle forwards IMU readings to the laptop
    a. Communication via Bluetooth (BLE) with 3-way handshake initiated

4. Laptop forwards readings to Ultra96
    a. Communication via TCP/IP

5. Ultra96 starts the dance move classification
    a. Feature processing and engineering performed on the raw data - including the calculation of yaw, pitch and roll
    b. Features will be passed into a Machine Learning model, ideally a Convolutional Neural Network (CNN)

6. Concurrent with the previous step, the Ultra96 also identifies dancer's relative positions
    a. Using the IMU readings, the Ultra96 will calculate if a dancer has remained in the same position or has switched positions

7. Ultra96 sends the results of the dance move classification and positions to the dashboard
    a. Communication via TCP/IP

<u>Muscle Fatigue Measurement</u>

1. Myoware muscle module, which is placed on the right upper arm will output data when a dancer performs a move.
    a. Module will output EMG signals

2. Bluno Beetle forwards IMU data to the laptop
    a. Communication via Bluetooth (BLE)

3. Laptop forwards data to Ultra96
    a. Communication via TCP/IP

4. Ultra96 forwards IMU data to the dashboard
    a. Communication via TCP/IP

2.4.2 Final Algorithm

As explained in the earlier sections, the final implementation of our system involves the use of only 1 Bluno Beetle on the right hand of the dancer. We have also decided to implement the detection of position changes on the Bluno Beetle instead using the Gyroscope Y values. This is because initial testing revealed that this mode of detection is rather effective and accurate, hence we decided to stick to this implementation. Hence, the ML model is no longer required to process and predict the position changes. Figure 2.4.2.1 reflects the changes that we have made to the system in the form of a sequence diagram.

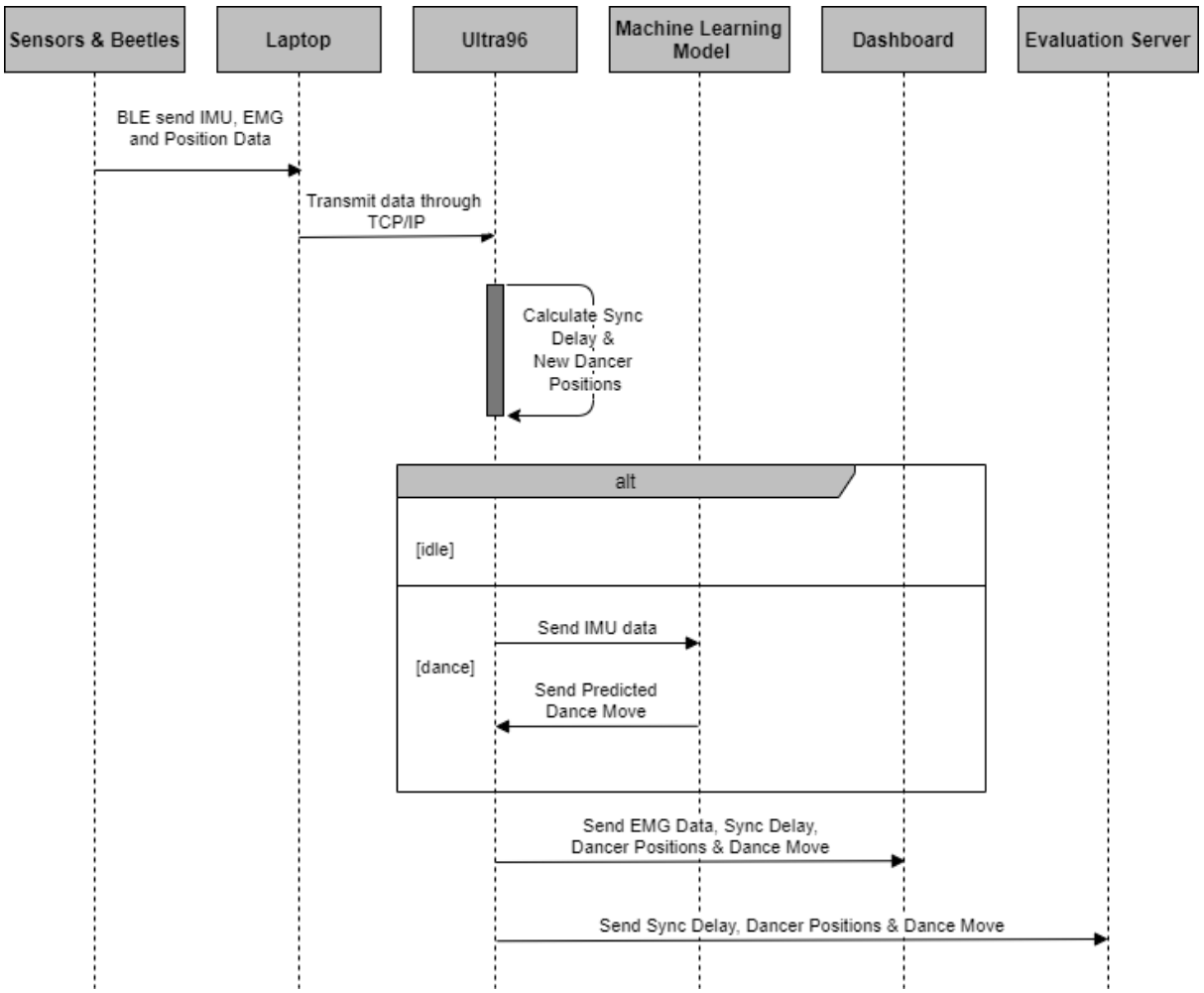


Figure 2.4.2.1: Sequence diagram for final detection algorithm

Dance Move and Dancer Position Classification

1. Inertial Measurement Unit (IMU), which is placed on the right hand, will output data when a dancer performs a move.
    a. IMUs will output 3-axial accelerometer and gyroscope readings

2. Readings will be collected by the Bluno Beetle, which are also placed on the right hand.
    a. The Bluno Beetle receives data from the respective IMUs
    b. Each IMU and Bluno Beetle pair has a wired connection
    c. Bluno Beetle processes and packs the data into the stipulated data format
    d. Bluno Beetle also determines the state of movement of dancers based on the IMU readings received.

3. Bluno Beetle processes the Gyroscope Y value from the IMU and determines the direction of position change.
    a. The detection of position change is carried out only when a dancer transits from the state of moving to not moving.

4. Bluno Beetle forwards the formatted IMU readings and position change data to the laptop
    a. Communication via Bluetooth (BLE) with 3-way handshake initiated

5. Laptop receives readings and position change data from the Bluno Beetle
    a. Laptop verifies that the received data from the Bluno Beetle are complete and performs data fragmentation handling if needed
    b. Laptop further verifies that the received data is accurately transmitted from the Bluno Beetle by calculating and comparing the checksum values
    c. Laptop extracts the relevant data and format the data into the stipulated format required by the external communications code that is running on the Ultra96

6. Laptop forwards formatted readings to Ultra96
    a. Communication via TCP/IP

7. Ultra96 performs the dance move classification
    a. Feature processing and engineering performed on the raw data - Gyroscope X,Y,Z and Accelerometer X,Y,Z values
    b. Features will be passed into a Machine Learning model, ideally a Convolutional Neural Network (CNN)
    c. FPGA performs hardware acceleration to expedite the processing of dance move data

8. Ultra96 receives the results of the dance move classification
    a. Ultra96 also calculates the dancers' new positions based on the position change data provided
    b. Ultra96 also calculates the sync delay among the dancers based on the timestamp data provided

9. Ultra96 sends the results of the predicted dance move, dancers' new positions and overall sync delay to the evaluation server and the dashboard
    a. Communication via TCP/IP

<u>Muscle Fatigue Measurement</u>

1. Myoware muscle module, which is placed on the right upper arm will output data when a dancer performs a move.
    a. Module will output EMG signals

2. Bluno Beetle calculates the Mean Amplitude from the EMG signals received
    a. Bluno Beetle assembles the Mean Amplitude data alongside the IMU data into a DATA packet and send this packet to the laptop
    b. Communication via Bluetooth (BLE) with 3-way handshake initiated

3. Laptop receives readings from the Bluno Beetle
    a. Laptop verifies that the received data from the Bluno Beetle are complete and performs data fragmentation handling if needed

b. Laptop further verifies that the received data is accurately transmitted from the Bluno Beetle by calculating and comparing the checksum values
c. Laptop extracts the relevant data and format the data into the stipulated format required by the external communications code that is running on the Ultra96

4. Laptop forwards formatted readings to Ultra96
    a. Communication via TCP/IP

5. Ultra96 forwards EMG data to the dashboard
    a. Communication via TCP/IP

3 Hardware Details

3.1: Hardware sensors

The hardware sensors, including accelerometers, gyroscopes, and a microcontroller with bluetooth, are responsible for the raw data collection of the wearable device and transmits the data to the laptop. This section will further explain the components, pin layouts, schematics, power design, and libraries used.

3.1.1 Hardware Components

MPU 6050

MPU 6050 is an Inertial Measurement Unit (IMU) which has a 3 axis accelerometer and a 3 axis gyroscope. The accelerometer measures the acceleration using the 3 dimensional axis. The gyroscope measures the rotational velocity or rate of the change of angular position along the X, Y, Z axis. Therefore, we can get the sensor orientation information by combining the accelerometer and gyroscope data. This is based on this provided datasheet (MPU 6050 datasheet, n.d.).

Bluno Beetle

The Bluno Beetle is a board based on Arduino Uno with Bluetooth 4.0 (BLE). It uses Arduino IDE to upload the code, and the bluetooth will be used for communication between the Ultra96 and the Bluno Beetle. This is based on this provided datasheet (BlunoBeetle datasheet, n.d.).

MyoWare Muscle Sensor

The MyoWare Muscle Sensor measures the muscle activations using Electromyography (EMG). It measures the filtered and rectified electrical activity of a muscle, then outputs voltages depending on the amount of activity. This is based on this provided datasheet (Myoware User Manual, n.d.).

Utra96-V2

The Ultra96-V2 is an Arm-based, Xilinx Zynq UltraScale+ MPSoC development board. Bluetooth will be used to connect with the Beetle and gain the data for the FPGA. This is based on this provided datasheet (Ultra96-V2, n.d.).

3.1.2 Pin Table

This section will include the pinout diagram of the hardware components and the pin table of connections. Each dancer will have 1 device settled as shown in Figure 2.3.3 and 2.3.4. The beetle and mpu sensor are soldered and attached to the glove on the right hand using velcro strips.

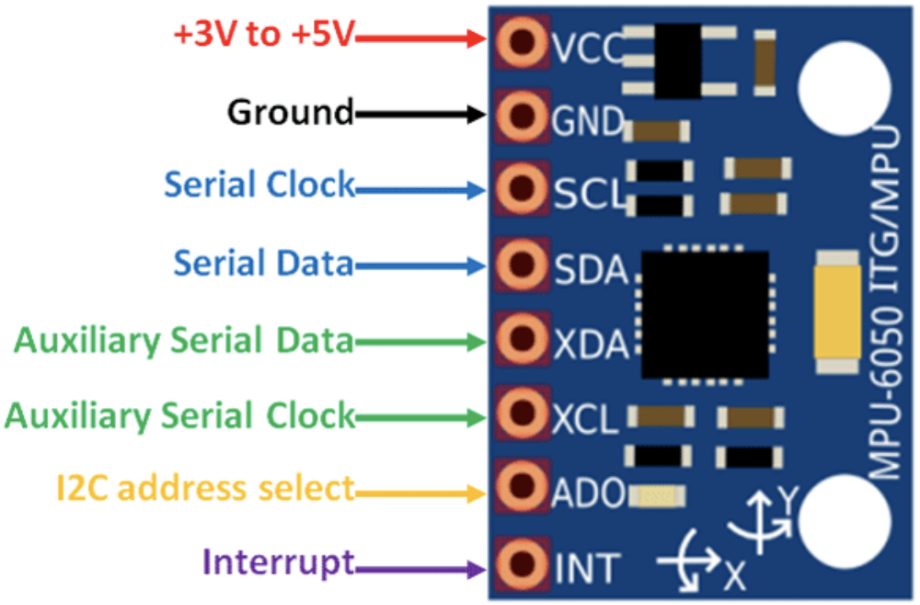


Figure 3.1.2.1: MPU6050 Pinout diagram (Adapted from (Naik, 2019))

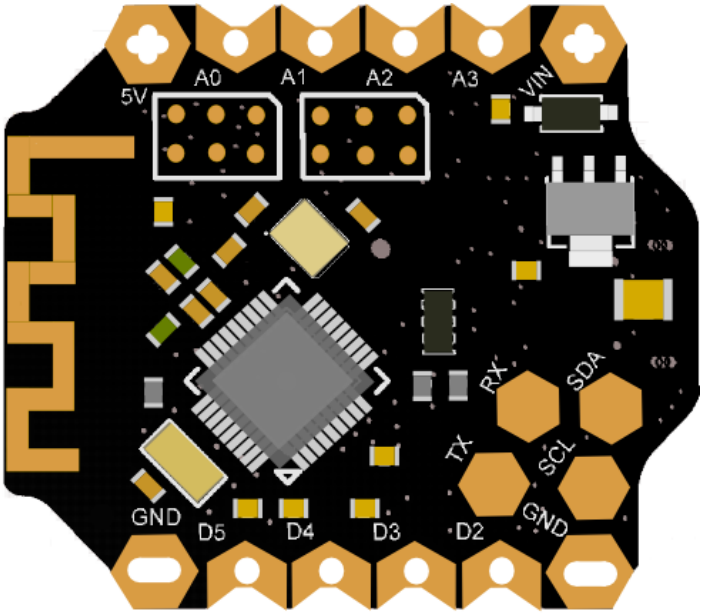


Figure 3.1.2.2: Bluno Beetle Pinout Diagram (Adapt from (Bluno_beetle_sku_dfr0339, n.d.))

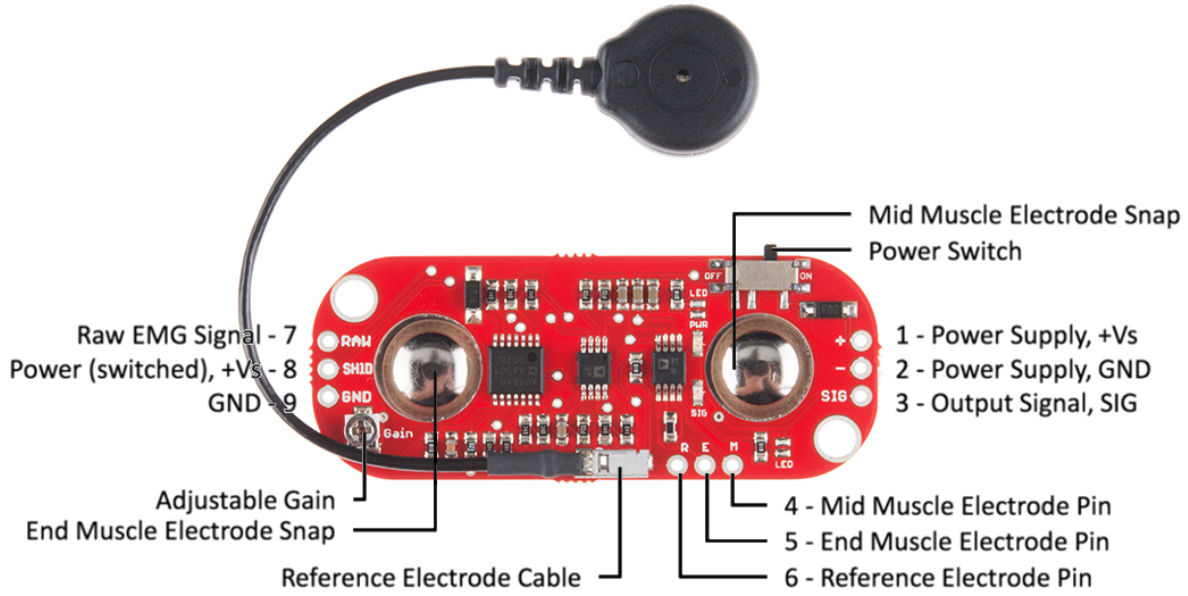


Figure 3.1.2.3: MyoWare Muscle Sensor Pinout Diagram (Adapt from (*MyoWare muscle Sensor Kit, n.d.*))

Bluno Beetle	**MPU6050**
5V	VCC
GND	GND
D2	INT
SCL	SCL
SDA	SDA

Table 3.1.2.1: Pin table for the connections from the Bluno Beetle to the MPU6050

Bluno Beetle	**MyoWare Muscle Sensor**
5V	1 - Power Supply, +Vs
GND	2 - Power Supply, GND
A0	3 - Output Signal, SIG

Table 3.1.2.2: Pin table for the connections between the Bluno Beetle and the MyoWare Muscle Sensor

Power Supply	**Bluno Beetle**
+	VIN
-	GND

Table 3.1.2.3: Pin table for the connections between the power supply and the bluno beetle

3.1.3 Schematics

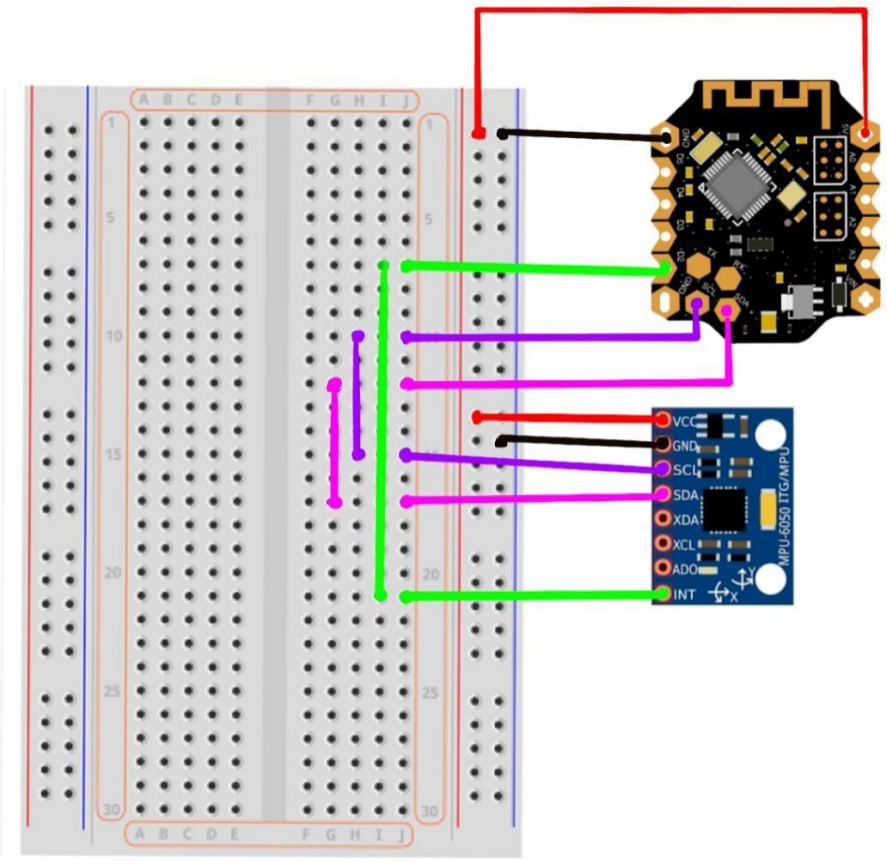


Figure 3.1.3.1: Schematics without the MyoWare Muscle Sensor

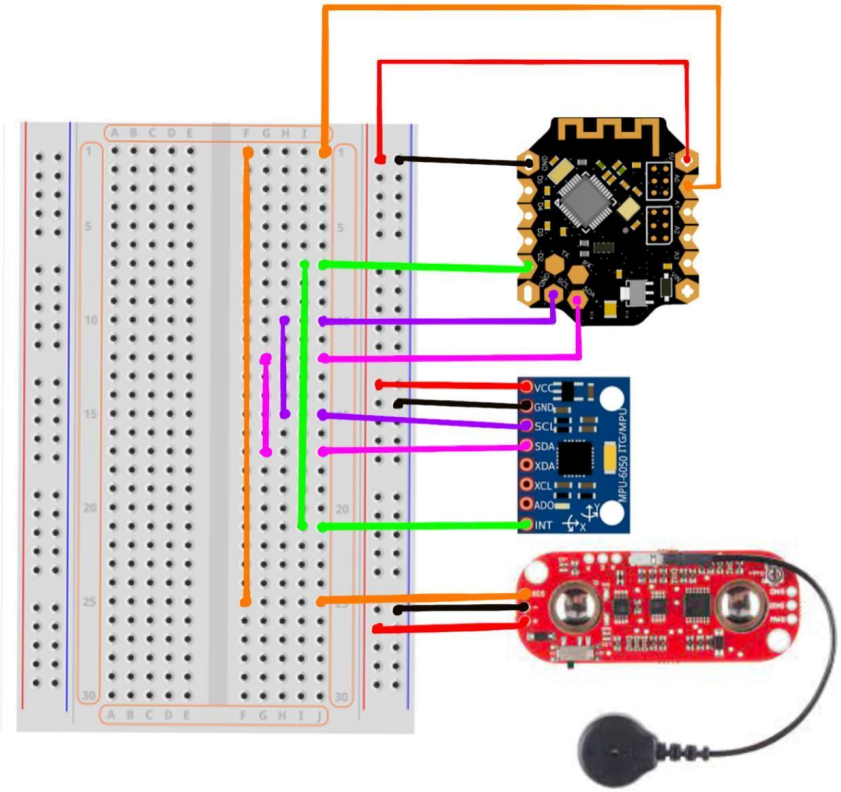


Figure 3.1.3.2: Schematics with the MyoWare Muscle Sensor

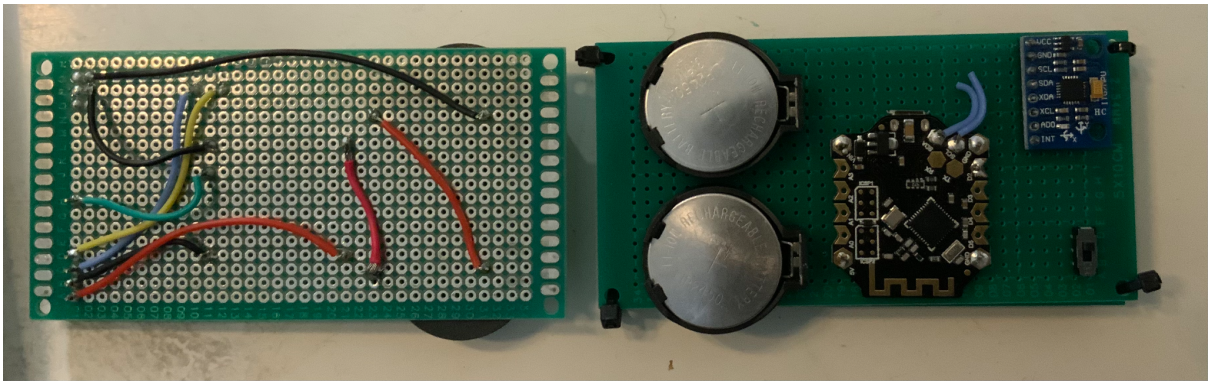


Figure 3.1.3.3: Physical schematics of both sides

3.1.4 Operating Voltage and Current

Bluno Beetle

- Operating Voltage: 5V
- Operating Current: 20mA

MPU6050

- Operating Voltage: 2.375V - 3.46V
- Operating Current: 3.8mA when accelerometer and gyroscope enabled and DMP disabled

MyoWare Muscle Sensor

- Operating Voltage: 3.3V or 5V
- Operating Current: 9mA

To ensure the power consumption is not too high, our system for one dancer contains one set with one inertial measurement unit (IMU) connected to a Bluno Beetle. The IMU modules have breakout boards with voltage regulators, so that the sensors can be powered using Beetle's power supply. To power them, the 5V pin of the Beetle is connected to VCC pins of MPU6050s and the "+" pin of the muscle sensor.

To power a Bluno Beetle, a USB cable connected or a 5-8V external power connected to VIN is needed. Hence, we chose an external power supply for a better wearable experience. We used two 3.6V rechargeable lithium coin cells LIR2450 as shown in Figure 3.1.4.1 for each set. Two battery holders as shown in Figure 3.1.4.2 are soldered on the breadboard, and a slider switch is soldered on as well. For each set, 2 batteries are connected to provide a 7.2V power supply for each set.

Figure 3.1.4.1: LIR2450 Rechargeable Battery

Figure 3.1.4.2: Battery Holder

Compared with other batteries, LIR2450 costs less and has a lighter weight in total for one set. The small size of 24mm diameter of the battery keeps the set light and easy to settle on the glove. A 7.2V power supply is able to support the set with the EMG sensor working

continuously for more than 1 hour. It takes less than 3 hours to fully charge the LIR2450. We also have extra batteries for replacement, so that we can test incessantly.

Changes in Power Supply

In the first version design of the power supply, we used two 3V lithium coin cells CR2032 as shown in Figure 3.1.4.3. The holder with a switch shown in Figure 3.1.4.4 is connected to Beetle and supplies 6V to each set. The coin cell is chosen for its smaller size and fewer number of cells required compared to AAA batteries.

Figure 3.1.4.3 The 3V lithium coin cell

Figure 3.1.4.4 The holder for 2 coin cells

However, during the test in week 8, we noticed that as the battery level depletes, the bluetooth connection of the Bluno Beetle becomes unstable. The batteries need to be replaced after roughly 4 hours of usage, otherwise the performance will be negatively affected. The cost was acceptable only when 1 dancer was dancing in Week 9.

However, when we tested with 3 dancers and prepared for the test in week 11, the cost of the batteries hugely increased as the batteries needed to be changed frequently. Hence, we decided to change the non-rechargeable battery CR2032 to the rechargeable battery LIR2450. Compared to CR2032, the size of the LIR2450 is a bit larger, hence we also had to change the battery holder and add on the slider switch to the circuit design. After changing the batteries to LIR2450, the connection became more stable and the batteries can even support up to a one-day-long test. The rechargeable battery is more environmentally friendly as well.

3.1.5 Algorithm and Libraries

This section will include the algorithm and libraries for gathering data from sensors. The basic functions or steps are shown in the flowchart in Figure 3.1.5.1.

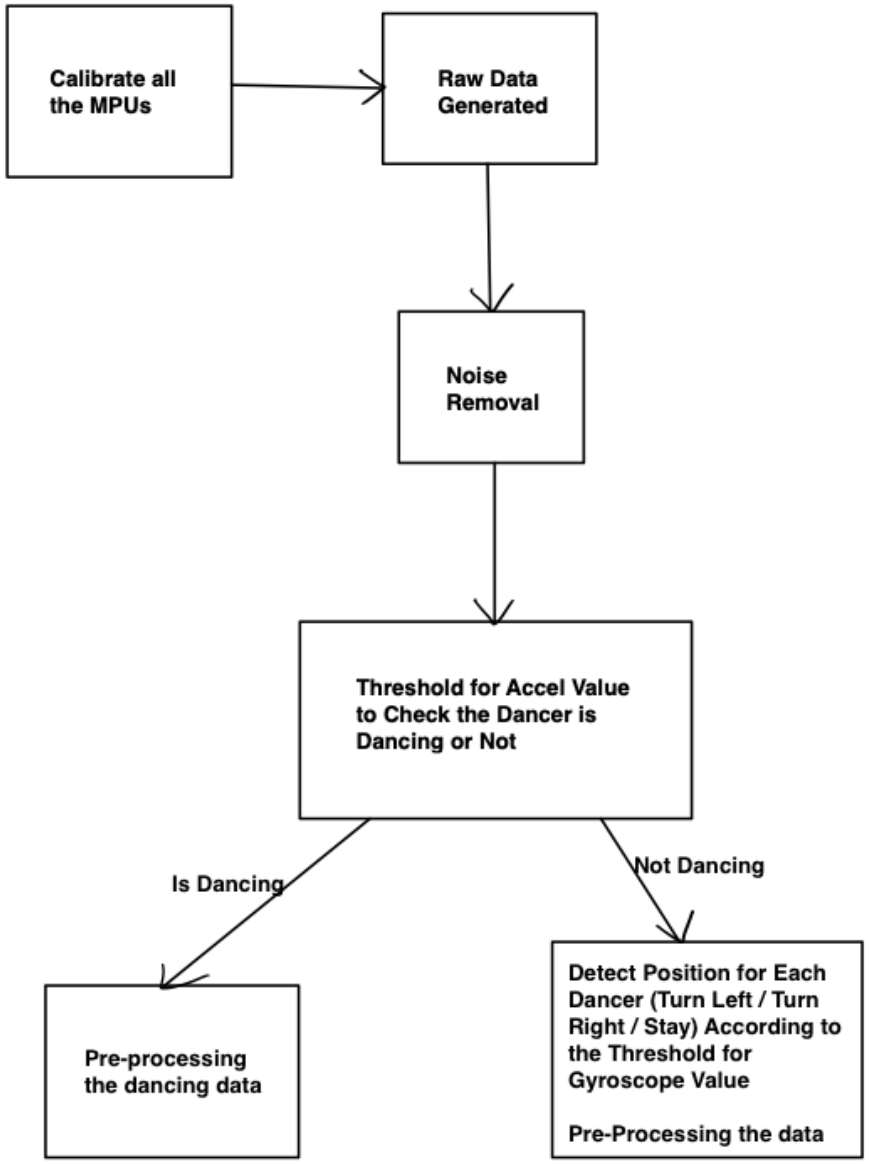


Figure 3.1.5.1: Flowchart of Sensors Code

We use the Arduino IDE to code since the Beetle is based on Arduino Uno. The I2C library we used for connecting I2C components is the Wire library (Wire, n.d.). This helps us extract the values of x, y, z axes from the accelerometer and the x,y,z axes from the gyroscope. The details about how this data is used will be explained in Section 5.1. We also used the MPU6050_tockn library from token to extract data from MPU6050s (MPU6050_tockn, n.d.). These values will be used to detect the dancer's movements and positions. We use the code shown in Figure 3.1.5.2 to test the initial values of x, y, z axes from the accelerometer and the gyroscope in the IMU.

```
#include <MPU6050_tockn.h>
#include <Wire.h>

MPU6050 mpu6050(Wire);

long timer = 0;

void setup() {
  Serial.begin(9600);
  Wire.begin();
  mpu6050.begin();
  mpu6050.calcGyroOffsets(true);
}

void loop() {
  mpu6050.update();

  if(millis() - timer > 100) {
    Serial.println("-------------------------");
    Serial.print("gyroX: ");
    Serial.print(mpu6050.getGyroX());
    Serial.print("  gyroY: ");
    Serial.print(mpu6050.getGyroY());
    Serial.print("  gyroZ: ");
    Serial.println(mpu6050.getGyroZ());

    Serial.print("accX: ");
    Serial.print(mpu6050.getAccX());
    Serial.print("  accY: ");
    Serial.print(mpu6050.getAccY());
    Serial.print("  accZ: ");
    Serial.println(mpu6050.getAccZ());

    timer = millis();
  }

}
```

Figure 3.1.5.2: Screenshot of testing code

We used the example code provided by the MPU library to calibrate the MPUs. For each MPU6050, we do this five times and set the mean value for each dimension as the offset value as shown in Figure 3.1.5.3.

```
// Calibrate offset values for different beetles
void setOffsetValues() {
  if (ID == 1) { // C1
    mpu.setXAccelOffset(-2203);
    mpu.setYAccelOffset(-211);
    mpu.setZAccelOffset(1198);

    mpu.setXGyroOffset(604);
    mpu.setYGyroOffset(792);
    mpu.setZGyroOffset(4);
  }else if (ID == 2) { // H1
    mpu.setXAccelOffset(-1088);
    mpu.setYAccelOffset(1678);
    mpu.setZAccelOffset(1593);

    mpu.setXGyroOffset(-82);
    mpu.setYGyroOffset(-18);
    mpu.setZGyroOffset(68);
  } else if (ID == 3) { // C2
    mpu.setXAccelOffset(-4415);
    mpu.setYAccelOffset(400);
    mpu.setZAccelOffset(969);

    mpu.setXGyroOffset(47);
    mpu.setYGyroOffset(-32);
    mpu.setZGyroOffset(-2);
  } else if (ID == 4) { //H2
    mpu.setXAccelOffset(-2970);
    mpu.setYAccelOffset(3988);
    mpu.setZAccelOffset(1568);

    mpu.setXGyroOffset(57);
    mpu.setYGyroOffset(121);
    mpu.setZGyroOffset(-64);

  } else if (ID == 5) { //C3
    mpu.setXAccelOffset(-310);
    mpu.setYAccelOffset(1716);
    mpu.setZAccelOffset(773);

    mpu.setXGyroOffset(119);
    mpu.setYGyroOffset(21);
    mpu.setZGyroOffset(-33);
  } else if (ID == 6) { // H3
    mpu.setXAccelOffset(1244);
    mpu.setYAccelOffset(79);
    mpu.setZAccelOffset(849);

    mpu.setXGyroOffset(110);
    mpu.setYGyroOffset(138);
    mpu.setZGyroOffset(33);
  }
}
```

Figure 3.1.5.3: Set the Offset Value for Each MPU6050

To detect if the dancer is dancing or not, we apply the noise removal first and then use the threshold to see whether the dancer has a big move. A dancing count and an idling count are set, so that the dancer is considered as dancing if he or she reaches the count threshold. This prevents a false detection of the state of dancing or not dancing, which can affect the accuracy of our system. Both thresholds are set according to the 8 dance movements and the ending pose. The code is shown in Figure 3.1.5.4.

```
//noise removal
if (abs(accelX) < 24576 && abs(accelY) < 24576 && abs(accelZ) < 24576) {
  accelXDiff[aCount] = accelX - preAX;
  accelYDiff[aCount] = accelY - preAY;
  accelZDiff[aCount] = accelZ - preAZ;
  preAX = accelX;
  preAY = accelY;
  preAZ = accelZ;
  aCount += 1;

  // Dancer moving status thresholding
  if (aCount >= 5) {
    for (int i = 0; i < SAMPLES; i++) {
      accelXDiffSum += abs(accelXDiff[i]);
      accelYDiffSum += abs(accelYDiff[i]);
      accelZDiffSum += abs(accelZDiff[i]);
    }
    if (accelXDiffSum >= 10000 || accelYDiffSum >= 10000 || accelZDiffSum >= 10000) {
      dancing_count += 1;
      idling_count = 0;

      // Detect state of motion as moving
      if (dancing_count >= 8) {
        veryFirst = false;
        dancing = true;
        idling = false;
      }
    } else {
      idling_count += 1;
    }

    // Detect state of motion as idling
    if (idling_count >= 5) {
      if (idling_count == 5) {
          firstClear = true;
          firstStop = true;
        }
        idling = true;
        dancing = false;
        dancing_count = 0;
      }

      aCount = 0;
      accelXDiffSum = 0;
      accelYDiffSum = 0;
      accelZDiffSum = 0;
    }
  }
```

Figure 3.1.5.4: Check the Dancer is Dancing or Not

Once the dancer is detected as not dancing, the Y axis gyroscope value is checked with the threshold for turning right or left. Similar to the dancing detection, left and right counts are set to determine whether the dancer reaches that count and then considered as turning left or right. If in a period of time, the dancer is not turning left or turning right, he or she will be considered to stay at the same position. We factored in more time for the first position in case the dancer is not ready or has a connection issue at the beginning. The turning detection code for each dancer is shown in Figure 3.1.5.5. After collecting 3 dancers' turns, a logical function will be called to derive the new position according to the previous position and the turning result from 3 dancers. This is done on the Ultra96.

```
// Position Detection
void detectPosition() {
    // Capture timestamp when dancer first stops moving
    if (firstStop) {
      stopTime = millis();
      firstStop = false;
    }
    elapsedTime = millis();

    // Already detected the new position
    if (positionDetected) {
      return;
    }

    // Clear existing Gyro values to accurately determine turning
    if (firstClear) {
      clearGyroSum();
      firstClear = false;
    }

    if (gyroYAvgT > 1600) {
      rMoveCnt += 1;
    } else if (gyroYAvgT < -1600) {
      lMoveCnt += 1;
    }

    // Position change for the very first dance move
    if (veryFirst) {
      if (lMoveCnt > 15) {
        lMoveCnt = 0;
        rMoveCnt = 0;
        positionDetected = true;
        newPosition = 'L';
        clearGyroSum();
      } else if (rMoveCnt > 15) {
        lMoveCnt = 0;
        rMoveCnt = 0;
        positionDetected = true;
        newPosition = 'R';
        clearGyroSum();
      } else if (elapsedTime - stopTime > 39000) {
        // Detect as stay if no left or right turning is detected
        lMoveCnt = 0;
        rMoveCnt = 0;
        positionDetected = true;
        newPosition = 'S';
        clearGyroSum();
      }

      // Position change for every other dance moves
      } else {
        if (lMoveCnt > 15) {
          lMoveCnt = 0;
          rMoveCnt = 0;
          sMoveCnt = 0;
          positionDetected = true;
          newPosition = 'L';
          clearGyroSum();
        } else if (rMoveCnt > 15) {
          lMoveCnt = 0;
          rMoveCnt = 0;
          sMoveCnt = 0;
          positionDetected = true;
          newPosition = 'R';
          clearGyroSum();
        } else if (elapsedTime - stopTime > 14000) {
          lMoveCnt = 0;
          rMoveCnt = 0;
          sMoveCnt = 0;
          positionDetected = true;
          newPosition = 'S';
          clearGyroSum();
        }
      }
  }
```

Figure 3.1.5.5: Detect Turning Position for Each Dancer

For MyoWare Muscle Sensor, it uses electromyography (EMG) to detect the electrical activity of the muscle, and then convert to a varying voltage. The max amplitude, mean amplitude, and root mean square amplitude are in the time domain and the mean frequency value is in the frequency domain. These data can be processed to extract features which are capable of observing muscle fatigue of the dancer (Toro et al., 2019). We used all of these when testing in week 10 and realized that the mean amplitude value is good enough to notice the muscle fatigue level. Therefore, we then cut down to using the mean amplitude value only for observing muscle fatigue.

The data generated from analogRead() will first be converted from the [0, 1023] range into the voltage range [0, 5]V. The mean amplitude value is then scaled, which will be explained in the Internal Communications section. The mean amplitude is calculated as shown in Figure 3.1.5.6.

The increase in the amplitude represents muscle fatigue, hence the mean amplitude feature is used to determine the muscle fatigue. Therefore, when the mean amplitude increases, the dancer wearing the muscle sensor is considered to have a higher level of muscle fatigue.

```
float calculateEMGData() {
  if (ID == 6) { // Only read EMG data if it is EMG set
    totalValue = 0;
    squareValue = 0;
    meanAmplitude = 0;

    // 35 samples with 1kHz frequency, tuned down to maintain data reading at 20 Hz
    for (int i = 0; i < EMG_SAMPLES; i++) {
      float sensorValue = analogRead(A0); //read the analog value
      float convertedValue = (sensorValue / 1024.0) * 5; //convert into voltage range

      totalValue += convertedValue;
      delay(1);
    }
    meanAmplitude = totalValue / (EMG_SAMPLES * 1.0);
    scaledMeanAmplitude = round(meanAmplitude * 100.0);
  } else {
    scaledMeanAmplitude = 0; // Send as 0 if it is not EMG set
  }
  return scaledMeanAmplitude;
}
```

Figure 3.1.5.6: EMG Data Collection

3.2: Hardware FPGA

3.2.1 Ultra96 Synthesis and Simulation

Ultra96-V2 is based on Xilinx Zynq UltraScale+ MPSoC ZU3EG A484, allowing for the implementation of machine learning algorithms and acceleration. We will use Vivado High-level Synthesis (HLS) and PYNQ (PYNQ, n.d.) for the development of the hardware accelerator. Vivado HLS will generate HDL IP cores, which will be implemented into the Zynq Programmable Logic (PL) using AXI4-Lite, a memory-mapped input/output (MMIO) interface. We would then use PYNQ overlays, runned on Zynq Processor System (PS), to control and operate the FPGA's programmed logics and IO.

Since our machine learning models would be designed, trained and tested using Python, we would use the cocotb library in Python for simulation and creating testbenches ("Welcome to cocotb's documentation!", n.d.). There are two benefits of using cocotb instead of Vivado HLS-based C++ testbenches: First, the same logic would not need to be duplicated into C++; Second, machine learning developers who might not be familiar with hardware programming would be able to operate as well.

PYNQ overlay is an abstraction that allows us to operate across the PS and PL (Overlay tutorial., n.d.). It allows application developers to make use of the hardware acceleration without understanding hardware programming. We will develop a PYNQ overlay to program the FPGA and handle low-level reads and writes to the FPGA.

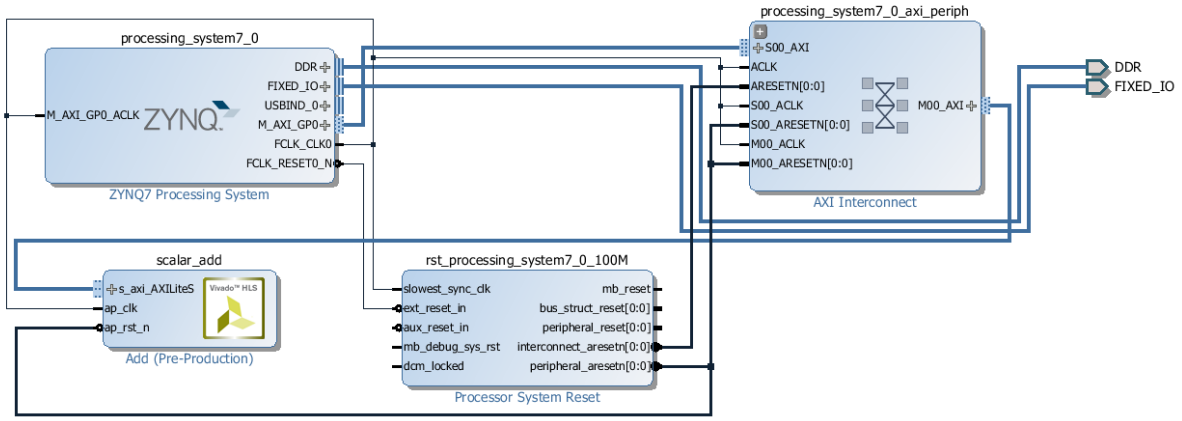


Figure 3.2.1.1: Example Block Diagram (Overlay tutorial. n.d.)

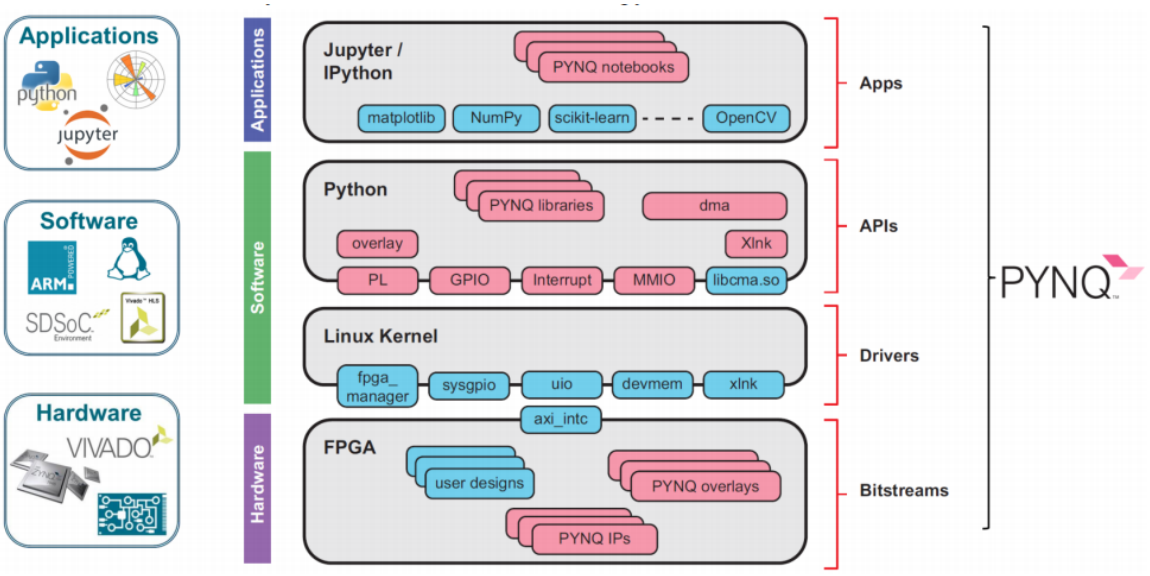


Figure 3.2.1.2: PYNQ Overlay Structure ("Know PYNQ in one article - code world." n.d.)

3.2.2 Neural Network Implementation

The two different machine learning models selected for recognizing dance moves are SVM and MLP (further discussed in Section 5.1). We will implement the machine learning models into a single IP core in C++ with reusable software components. There is another option of creating an independent IP core for each layer and stitching them together. However, although the alternative design would be easier to test and more flexible in structure, the overhead of streaming data could significantly increase. We decide to prioritize the effectiveness and efficiency of our system.

The model is implemented with HLS into an ip package, which was then used to generate a bitstream file that was later uploaded to the FPGA. The FPGA is then controlled with Python's PYNQ library.

Figure 3.2.2.1: Neuron Network Node Implementation in C++

3.2.3 System Design Evaluation

We have measured the recognition accuracy, speed and power efficiency of both of our algorithms. Taking the real time processing capabilities of the models into consideration, we have then come to a conclusion on the choice of our algorithm for the final product, which is the MLP model structure. This is because compared with the CNN model, the MLP model has a much lower power and computation resource consumption. The CNN model also did not produce a much higher dance move prediction accuracy. Therefore, we have decided to use the MLP model (further elaborated in section 5.1). Detailed comparison of both models are presented below.

The FPGA system will be evaluated based on the following aspects:

1. **Recognition accuracy**
   We have evaluated the algorithms and the system by analyzing the accuracy of the prediction results produced and comparing it with the accuracy of our model running on CPU.

Accuracy	CNN	MLP
No acceleration	98.4%	98.2%
FPGA acceleration	98.4%	98.2%

Table 3.2.3.1: Dance Move Prediction Accuracy Comparison

We see a similar accuracy between CNN and MLP models. The little difference in prediction accuracy also led to our decision of using MLP model instead of CNN model, as CNN models generally require higher power and computation resources, and are also more difficult to implement.

2. **Hardware latency and overhead**
   We have evaluated our system using the latency of the hardware implementation. The latency would be provided by Vivado HLS. We have also included the overhead of communication between Zynq PL and PS. This overhead would be accessible after the implementation of PYNQ overlay and actual testing of the system. The shorter latency there is, the better the performance.

Speed	CNN	MLP
CPU	6.2ms/input	5.5ms/input
FPGA	0.1ms/input	0.1ms/input

Table 3.2.3.2: Model Input Speed Comparison

We see a significant increase in speed with FPGA accelerations. The acceleration for the neural network model. However, we should also factor in the sampling frequency of our hardware sensors, which is 20Hz. This also limits our input speed. Increasing the input speed would also result in a higher power consumption.

3. **Power consumption**
   Further elaborated in Section 3.2.4

4. **Computation resource consumption**
   We measure the usage of various hardware resources, including computational power and usage of individual components. While some machine learning models lead to higher accuracy, they could also require much more hardware resources, leading to higher power consumption and cost.

Resource	CNN	MLP
LUT	51%	16%
LUTRAM	1%	1%
FF	12%	9%
BRAM	30%	14%
BUFG	1%	1%

Table 3.2.3.3: Computation Resource Consumption Comparison

Because CNN models require much higher computation resources and the excellent performances of MLP and SVM models, we have decided not to use CNN model for the dance move predictions.

5. **Implementation Efficiency**
   We also consider the efficiency of implementing certain designs. The balance between time spent and product effectiveness is also important for time management practices. We realized that CNN models are more difficult to implement and have a higher chance of producing unexpected mistakes.

3.2.4 Ultra96 Power Management

The Ultra96 board supports measuring power usage with PMBus (PMBus. 2021, May 3). We use this to measure the power consumption of the system whenever in use. Though we would try to reduce the power consumption as low as possible, we prioritize the performance of the dance move recognition.

After testing with both CNN and MLP models, we have recorded their respective power total on-chip power. When running the CNN model, the power is 2.315W, and the power is 2.159W for the MLP model. We can conclude that the higher power consumption of the CNN model is due to its more complex model structure, with more computation nodes and layers.

Reducing the CPU power usage:

1. Kill the unnecessary process and tasks.
2. Implement the machine learning model into FPGA for higher efficiency.

Reducing the FPGA power usage:

1. Control signals to indicate the start and end of a computation.
2. Balance the power usage if the performance of the machine learning model is adequate.

3.2.5 Changes Made

During our testing and development, we have made some changes from our initial design. One of the major changes was that we started to use DMA instead of MMIO for data transfer between the PL and PS sections on the Ultra96 (Figure 3.2.5.1). The main reason for this change is because we allow the CPU to perform other tasks while FPGA generates the output. This will reduce the time taken for predictions and increase the prediction speed of our entire

system. We were then able to shorten the time dedicated for dance move prediction and focus on other tasks such as position changing and communications.

```
def fpga_evaluate_dance(test):
    choose_model[0] = 0
    dma.sendchannel.transfer(choose_model)
    dma.sendchannel.wait()
    for j in range(100):
        input_buffer0[j] = test[j];
    dma.sendchannel.transfer(input_buffer0)
    dma.recvchannel.transfer(output_buffer0)
    dma.sendchannel.wait()
    dma.recvchannel.wait()
    return output_buffer0
```

Figure 3.2.5.1: DMA Implemented for FPGA I/Os

3.2.6 Potential Optimizations

Our current designs can be potentially improved in the following aspects:

Vivado HLS Compiler Directives
Vivado HLS provides various compiler directives allowing optimizations of generated hardware (e.g., loop unrolling and array partitioning). This could improve latency of the hardware accelerator.

4 Firmware & Communications Details

4.1 Internal Communications

4.1.1 Introduction

The purpose of internal communications is to establish and maintain the communication channel between the respective Bluno Beetles and laptops through the use of Bluetooth technology. Sensor data are first obtained from the MPU 6050 and MyoWare Muscle Sensor. The data are then processed and assembled into a packet that is properly formatted, before they are relayed to the laptops where these data are then transmitted to the Ultra96 and dashboard respectively. Hence, a stable and reliable mode of data transmission has to be ensured in this process so that the other system components can utilise the complete sensor data to generate useful output to the users.

In the following sections, the initial design of the internal communications system that was planned for this project will be discussed and analysed. We will then share about the changes that were implemented after the individual evaluation in week 7 leading to the final evaluation in week 13 due to the limitations faced along the way. In the following sections, the management of tasks on the Beetles will first be discussed, before we move on to share about how the BLE interfaces will be set up and configured. The communication protocol between the Bluno Beetles and laptop will then be explained, before we elucidate on how reliability will be ensured in this communication channel.

4.1.2 Task Management on the Beetles

As there are a few tasks involved in ensuring the reliable and robust transmission of data from the Bluno Beetles to the laptops, it is critical to have a task management scheme in place to prioritise the necessary tasks over the more trivial ones. This can prevent the situation of task bottlenecks occurring especially when the number of dancers involved increases.

4.1.2.1 Initial Task Management Plan

Our initial hardware design plan was to use 2 Bluno Beetles per dancer — 1 on the chest and 1 on the hand. We had plans to utilise the FreeRTOS library that is readily available and easy to install through the Arduino Library Manager to manage tasks on the Bluno Beetle. This library provides the capability of multithreading on the Bluno Beetle and ensures that the tasks are executed in the order of priority as soon as they are triggered. This also improves the responsiveness of the system and enhances the overall performance of task execution.

The different tasks that we had original planned to run in the Bluno Beetle are as follows and they are listed in the order of priority:

- An ‘ACK’ packet to acknowledge the ‘TIME’ packet from the Bluno Beetle to complete the clock synchronization

- Data transmission of formatted packets from the Bluno Beetle to the laptop.
    - An ‘ACK’ packet to acknowledge the ‘HELLO’ packet from the laptop

    - A ‘TIME’ packet to carry out clock synchronization with the laptop
    - The MPU 6050 sensor readings
    - The MyoWare muscle sensor readings
- Reading and formatting of sensor data into data packets before they are transmitted to the laptop
- Receiving of data packets from the laptop
    - A ‘HELLO’ packet from the laptop during the initiation of a connection

However, we had noted that a potential point of concern is the considerably limited memory space of the Atmega328 chip that is used in the Bluno Beetle - 32KB of flash memory, 2KB of SRAM and 1KB of EEPROM (ATmega328P datasheet., n.d.). We suspected that this could limit the effectiveness of the FreeRTOS library in our project context depending on the memory demands of the library.

On the laptop, we felt that it was crucial to implement multithreading protocols to handle data from the 2 Bluno Beetles from each dancer efficiently. Within a thread, the functions that are handled include the handling of incoming data from the Bluno Beetle as well as the function to reset the Bluno Beetle. We employed the use of the Python Threading library which implements Thread-based parallelism (Python, n.d.). More specifically, we used the class threading.Thread to represent the activities that we want to run in a separate thread of control. Within the class threading.Thread, we override the *__init__()* and pass the beetle and characteristics object to the constructor as seen in Figure 4.1.2.1.1.

```
class myThread(threading.Thread):
    def __init__(self,beetle,characteristics):
        threading.Thread.__init__(self)
        self.beetle = beetle
        self.characteristics = characteristics
```

Figure 4.1.2.1.1: Overridden __init__() method of the threading.Thread class

Another method that we override in the threading.Thread class is the *run()* method, where we implement the main activities of receiving data from the Bluno Beetle, as well as sending data from the laptop to the external communications on the Ultra96(Figure 4.1.2.1.2). The specific details will be discussed in the sections below.

```
def run(self):
    global SEND_BUFFER
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect(("localhost", 8888))

        try:
            idle_count = 0
            # Only receive data after handshaking process is completed
            if (HANDSHAKE_BOOL_DICT[self.beetle.addr]):
                logger.info(f"Receiving data from {BEETLE_TYPE[self.beetle.addr]} Beetle...")
                while True:
                    try:
                        # Congestion control of data to external comms
                        if len(SEND_BUFFER) >= 3:
                                compiled_data = "".join(SEND_BUFFER)
                                sock.sendall(compiled_data.encode())
                                SEND_BUFFER = []
                        if self.beetle.waitForNotifications(2.0):
                            continue
                        idle_count += 1
                        logger.info(f"idle count: {idle_count}")
                        if idle_count == 1: # Beetle requires reset
                            break
                    except BTLEDisconnectError:
                        logger.error(f"BTLEDisconnect Error in Beetle {BEETLE_DICT[self.beetle.addr]}")
                        reconnectBeetle(self.beetle)
                        self.reset()
                        initHandshake(self.beetle,self.characteristics)
                        sock.close()
                        self.run()
                logger.error(f"Error in Beetle {BEETLE_DICT[self.beetle.addr]}")
                reconnectBeetle(self.beetle)
                self.reset()
                initHandshake(self.beetle,self.characteristics)
                sock.close()
                self.run()

        except Exception as e:
            logger.error(f"Error in Beetle {BEETLE_DICT[self.beetle.addr]}: {e}")
            reconnectBeetle(self.beetle)
            self.reset()
            initHandshake(self.beetle,self.characteristics)
            sock.close()
            self.run()
```

Figure 4.1.2.1.2: Overridden run() method of the threading.Thread class

The thread object is first created and is then started by calling the *start()* method in the threading library as seen in Figure 4.1.2.1.3. The *run()* method is then invoked in a separate thread of control for each object that calls the *start()* method. This ensures that the laptop is able to receive data from the chest and hand beetles simultaneously, and this doubles the overall efficiency of the system and ensures that the system is robust and effective.

```
myThread(beetle,characteristics).start()
```

Figure 4.1.2.1.3: start() method of the threading.Thread class

4.1.2.2 Final Task Management Plan

After extensive online research was carried out, we decided not to utilise the FreeRTOS library as the effectiveness of this library is limited by the small amount of memory space available for use by the library. Furthermore, we did not see the need to implement task management on the Bluno Beetle as data packets are rarely sent from the laptop apart from the initial handshaking process and the infrequent reset process. Hence, the main task that is running on the Bluno Beetle is the reading and formatting of sensor data into data packets, as well as the transmission of these formatted data packets to the laptop. The number of packet

types have also been reduced, hence there are not as many tasks to run on the Bluno Beetle. This will be further elucidated in section 4.1.4 below.

On the laptop, we decided to proceed with the use of the Python multithreading library as it is proven to be effective and greatly improves the performance of the system when multiple Bluno Beetles are used by a dancer. Although our final hardware implementation involves the dancer wearing only 1 beetle on the hand, we retained the use of this multithreading library in the code. This is to ensure that our system remains robust and versatile and allows for the increase in the number of Bluno Beetles to be used per dancer at any point in time in the event that 1 Bluno Beetle is insufficient in accurately determining the dance move or position change.

4.1.3 Setup and Configuration of the BLE interfaces

As the setup and configuration of the BLE interfaces have not changed throughout the project, this section will not be segmented into the initial and final implementation method.

The setup and configuration of the BLE interfaces was implemented in the Ubuntu Linux Operating System (OS). The choice of the Ubuntu Linux OS is justified by the need to run the necessary bluetooth libraries for the communication channel between the Bluno Beetles and the laptops. We used the Bluepy library, a Python module that enables communication with BLE devices (Harvey, I., 2014). We then configured the Bluno Beetles as peripheral objects using the MAC addresses of the respective Bluno Beetles.

We faced a deliberation over whether to use the Native Ubuntu or to run Ubuntu in VirtualBox. The former option will put the dancers through the hassle of having to manually install the OS, while the latter is more lightweight and convenient to install. However, we eventually chose to go with the option of installing Native Ubuntu after taking into consideration the relatively inferior Bluetooth capabilities for the latter option, especially when more beetles are being connected to the laptop. We were worried that the use of Ubuntu in VirtualBox would lead to incessant disconnection problems later on, which would severely disrupt the user experience of the dancers.

The following steps were carried out to configure the BLE on the Bluno Beetle on the Arduino IDE (DFRobot, n.d.).

1. In the Arduino IDE, select the serial port that is used by the Bluno Beetle (Tools > Port).
2. Open the serial monitor on the upper right hand corner of the Arduino IDE.
3. Select “No line ending” and “115200 baud” in the pull-down menus.
4. Type “+++” into the text box and click on the send button. The message “Enter AT Mode” will be displayed when you have entered the AT Command Mode.
5. Select “Both NL & CR” and “115200 baud” in the pull-down menus.
6. Type “AT+SETTING=DEFPERIPHERAL” and click on the send button. The message “Ok” will be displayed and the Bluno Beetle has been configured as a peripheral device.
7. Set “AT+MIN_INTERVAL=10” to set the minimum connection interval to 10ms.
8. Set “AT+MAX_INTERVAL=10” to set the maximum connection interval to 10ms.
9. Type “AT+EXIT” to exit AT mode.

To find out the MAC address, run the command “AT+MAC=?” in the serial monitor after having entered the AT Command Mode in the Arduino IDE. The MAC address of the Bluno Beetle will then be displayed in the serial monitor. An example is illustrated in Figure 4.1.3.1.

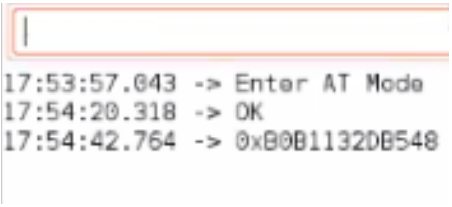


Figure 4.1.3.1: Extracting the MAC address of the Bluno Beetle

In the laptop code, we use the ScanDelegate class that is available from the Bluepy library to scan for BLE devices that are broadcasting advertising data. We are able to locate the Bluno Beetle as it has already been configured as a peripheral device in the Arduino IDE. In the main function, we instantiate the Scanner object and call the *withDelegate()* method to store a reference to a delegate object, in our case the *ScanDelegate* object. This is to receive callbacks when the broadcasts that are made by the BLE devices are received on the laptop. In the class *ScanDelegate*, we extract the MAC addresses of the surrounding BLE devices and determine whether the MAC address is that of a Bluno Beetle which we are using (Figure 4.1.3.2). If this is the case, we will log the information that this Bluno Beetle has been found. This is implemented by iterating through the keys of the Python dictionary *BEETLE_DICT[]*, which contains the MAC addresses of the 6 Bluno Beetles that are in our possession. To ensure that the discovery of a Bluno Beetle is only performed once, we use another Boolean Python dictionary *BEETLE_FOUND_DICT[]* to keep track of whether a Bluno Beetle has already previously been found (Figure 4.1.3.3).

```
class ScanDelegate(DefaultDelegate):
    def __init__(self):
        DefaultDelegate.__init__(self)

    def handleDiscovery(self, dev, isNewDev, isNewData):
            if dev.addr in BEETLE_DICT.keys():
                if not BEETLE_FOUND_DICT[dev.addr]:
                    BEETLE_FOUND_DICT[dev.addr] = True
                    logger.info(f"Beetle {BEETLE_DICT[dev.addr]} found!")
```

Figure 4.1.3.2: The ScanDelegate Class

```
BEETLE_ADDR_1 = "b0:b1:13:2d:b6:21" # SET 1 CHEST
BEETLE_ADDR_2 = "b0:b1:13:2d:b5:48" # SET 1 HAND
BEETLE_ADDR_3 = "b0:b1:13:2d:d6:75" # SET 2 CHEST
BEETLE_ADDR_4 = "2c:ab:33:cc:5f:45" # SET 2 HAND
BEETLE_ADDR_5 = "c8:df:84:fe:4c:19" # EMG SET CHEST
BEETLE_ADDR_6 = "b0:b1:13:2d:d6:7b" # EMG SET HAND

SERIVCE_ID = "0000dfb0-0000-1000-8000-00805f9b34fb"

BUFFER_DICT = {BEETLE_ADDR_1: b"", BEETLE_ADDR_2: b"", BEETLE_ADDR_3: b"", BEETLE_ADDR_4: b"", BEETLE_ADDR_5: b"", BEETLE_ADDR_6: b""}


BEETLE_ALL = [BEETLE_ADDR_1, BEETLE_ADDR_2, BEETLE_ADDR_3, BEETLE_ADDR_4, BEETLE_ADDR_5, BEETLE_ADDR_6]
BEETLE_DICT = {BEETLE_ADDR_1: 1, BEETLE_ADDR_2: 2, BEETLE_ADDR_3: 3, BEETLE_ADDR_4: 4, BEETLE_ADDR_5: 5, BEETLE_ADDR_6: 6}
BEETLE_TYPE = {BEETLE_ADDR_1: "HAND", BEETLE_ADDR_2: "HAND", BEETLE_ADDR_3: "HAND", BEETLE_ADDR_4: "HAND", BEETLE_ADDR_5: "HAND", BEETLE_ADDR_6: "HAND"}
BEETLE_FOUND_DICT = {BEETLE_ADDR_1: False, BEETLE_ADDR_2: False, BEETLE_ADDR_3: False, BEETLE_ADDR_4: False, BEETLE_ADDR_5: False, BEETLE_ADDR_6: False}
```

Figure 4.1.3.3: MAC Addresses of the Beetles and Different Python Dictionaries Implemented

We call the *scan()* method with a timeout of 3 seconds to allow for the laptop to scan for the nearby BLE devices. After which, we will attempt to establish a connection with the identified Bluno Beetle by calling the *establish_connection()* method. This can be seen in Figure 4.1.3.4.

```python
def main():
    scanner = Scanner().withDelegate(ScanDelegate())
    devices = scanner.scan(3.0)

    for discovered in devices:
        if discovered.addr in BEETLE_ALL:
            logger.info(f"Establishing connection with Beetle {BEETLE_DICT[discovered.addr]}")
            establish_connection(discovered.addr)


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

Figure 4.1.3.4: The main() function

In the *establish_connection()* method, the identified Bluno Beetle is set as a peripheral object with the MAC address specified. The list of Service and Characteristic objects that are offered by the Bluno Beetle are subsequently extracted and stored in the variables *service* and *characteristics*. Thereafter, the *setDelegate()* method is called to store a reference to the *MyDelegate* object, which handles asynchronous events like Bluetooth notifications. The handshaking process is also carried out and once the handshaking process is completed, the multithreaded object is then created and started as the *start()* method is called, as explained previously in Section 4.1.2.1. In the event that any exception occurs during the establishing of connection to the Bluno Beetle, a safety catch is implemented to ensure that the laptop reattempts to connect to the Bluno Beetle until this process is successfully completed. Figure 4.1.3.5 details the specifics of the *establish_connection()* method.

```python
# Function to establish peripheral connection with beetle
def establish_connection(addr):
    try:
        beetle = btle.Peripheral(addr)
        logger.info(f"Established peripheral connection with Beetle {BEETLE_DICT[addr]}")
        service = beetle.getServiceByUUID(SERIVCE_ID)
        characteristics = service.getCharacteristics()[0]
        beetle.setDelegate(MyDelegate(beetle.addr))
        handshake_status = initHandshake(beetle,characteristics)
        while handshake_status == False:
            initHandshake(beetle,characteristics)
        myThread(beetle,characteristics).start()
    except Exception as e:
        logger.error(f"Error encountered while establish peripheral connection: {e}")
        logger.info(f"Reattempting to establish connection with Beetle {BEETLE_DICT[addr]}")
        sleep(1.0)
        establish_connection(addr)
```

Figure 4.1.3.5: The establish_connection() function

4.1.4 Communication Protocol between the Beetles and Laptop

4.1.4.1 Packet Types

Initial Packet Types

Table 4.1.4.1.1 details the respective initial packet types that we had planned to implement and their purpose in our communication protocol. It is important to properly define these packet types as there are various types of data that is transmitted between the Bluno Beetle and laptop.

Packet Type	Value	Significance
HELLO	'H'	This packet is sent from the laptop to the Bluno Beetle during the 3-way handshake.
ACK	'A'	This packet is sent from the Bluno Beetle to the laptop and vice versa during the 3-way handshake.
IMU	'I'	This packet will include the MPU 6050 sensor readings.
EMG	'E'	This packet will include the MyoWare muscle sensor readings.
TIME	'T'	This packet will include the timestamp data to be sent from the Bluno Beetle to the laptop during the clock synchronization.

Table 4.1.4.1.1: List of Initial Packet Types

Final Packet Types

Table 4.1.4.1.2 shows the final packet types that we had decided to use in our implementation after changes were made along the way.

Packet Type	Value	Significance
HELLO	'H'	This packet is sent from the laptop to the Bluno Beetle during the 3-way handshake.
ACK	'A'	This packet is sent from the Bluno Beetle to the laptop and vice versa during the 3-way handshake.
DATA	'D'	This packet will include the MPU 6050 and EMG sensor readings.
POSITION	'P'	This packet will include the new position change of the dancer.
RESET	'R'	This packet will be sent from the laptop to the Bluno Beetle in the event that it is necessary to reset the Bluno Beetle.

Table 4.1.4.1.2: List of Final Packet Types

Towards the week 9 checkpoint where we integrated the dancer data of the 3 dancers, we had decided that timestamp data from the Bluno Beetle was no longer necessary in calculating the sync delay across the 3 dancers. This is because instead of using the timestamp on the Bluno Beetles, we are capturing the timestamp of the start moves of each dancer on the laptop. This reduces the need for us to implement time synchronisation across the 3 Bluno Beetles since the laptop time is universal. Hence, we decided to send both the MPU 6050 and EMG sensor readings in the same packet, with packet type 'D'. The specific packet details will be discussed in the section below.

Next, we also decided to implement the detection of dancers turning on the Bluno Beetle instead of the ML model, hence we added the packet type ‘P’ to send the packet of the detected position changes of the dancers.

Finally, the Reset packet ‘R’ is implemented as well. This is necessary because there are instances where the laptop will instruct the Bluno Beetle to carry out a reset, in the event that a disconnection takes place and reconnection follows. Unlike the Data and Position packets, this Reset packet will be sent from the laptop to the Bluno Beetle.

4.1.4.2 Establishing Communication

Initial Establishing Communication Process

The initial plan to establish a communication channel between the laptop and the Bluno Beetle involves a 3-way handshake, followed by a clock synchronisation process to synchronise the clocks in the Bluno Beetles.

In the 3-way handshake protocol, the laptop will first send a ‘HELLO’ packet to the Bluno Beetle, and the Bluno Beetle will acknowledge the connection by sending a ‘ACK’ packet back to the laptop. Finally, the laptop will send a ‘ACK’ packet to the Bluno Beetle to complete the 3-way handshake. The 3-way handshake protocol is illustrated in Figure 4.1.4.2.1.

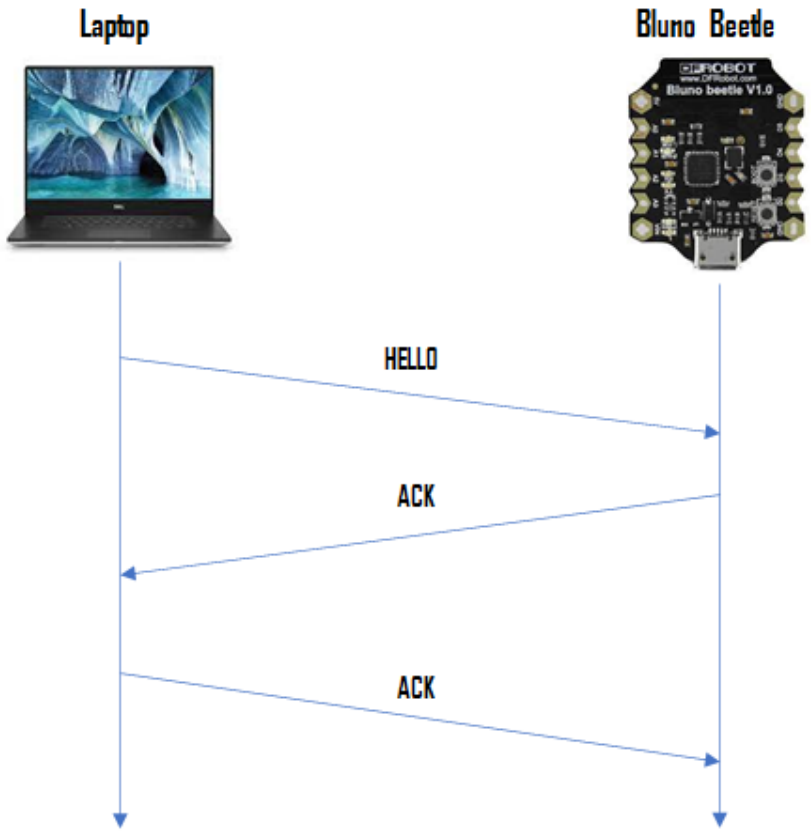


Figure 4.1.4.2.1: 3-way Handshake Protocol between Laptop and Bluno Beetle

Once the 3-way handshake has been completed, the Bluno Beetle will conduct a clock synchronisation with the laptop before any data transmission commences by sending a TIME packet to the laptop. Upon successful clock synchronization, the laptop will send an ACK packet to the Bluno Beetle. To ensure that the Bluno Beetle clock is closely synchronized with the laptop, this process of clock synchronization will take place at an interval to be determined.

Final Establishing Communication Process

In our final implementation of the establishing communication, the 3-way handshake process remains the same as elucidated above. However, as it is no longer necessary to capture and send any timestamps from the Bluno Beetle to the laptop, we decided to remove the clock synchronisation process and the sending of a TIME packet to the laptop. This slightly simplifies and expedites the process of establishing the communication channel.

The *initHandshake()* method is invoked in *establish_connection* after the laptop has set the identified Bluno Beetle as a peripheral object and has started to handle asynchronous events from the Bluno Beetle. The laptop will first send a Reset packet to the Bluno Beetle so that any previous data that remains in the Bluno Beetle is flushed out, as seen in Figure 4.1.4.2.2 and Figure 4.1.4.2.3. The laptop will then send a 'H' character to the Bluno Beetle to initiate the handshake process (Figure 4.1.4.2.4), and once the Bluno Beetle receives this 'H' character, it will proceed to send a 'A' character back to the laptop through Serial.write in Figure 4.1.4.2.5. Meanwhile, the laptop will await the 'A' character from the Bluno Beetle for up to 3 seconds with the *waitForNotifications()* method (Figure 4.1.4.2.6).

```
# Function to carry out handshaking process between beetle and laptop
def initHandshake(beetle, characteristics):
    logger.info(f"Initializing Handshake Sequence with Beetle {BEETLE_DICT[beetle.addr]}")

    logger.info("Resetting Device")
    characteristics.write(bytes(RESET,'utf-8'), withResponse = False)
```

Figure 4.1.4.2.2: Reset Packet Sent to Bluno Beetle

```
void(* resetFunc) (void) = 0;
```

Figure 4.1.4.2.3: Function to Reset Beetle

```
try:
    # Attempt up to 3 handshake attempts
    while (handshake_attempts <= 3 and not HANDSHAKE_BOOL_DICT[beetle.addr]):
        characteristics.write(bytes(HELLO,'utf-8'), withResponse = False)
        logger.info(f"HANDSHAKE ATTEMPT {handshake_attempts}: HELLO packet sent to Beetle {BEETLE_DICT[beetle.addr]}")
```

Figure 4.1.4.2.4: Hello Packet Sent to Bluno Beetle

```
void setup() {
  Serial.begin(115200);
  Wire.begin(); //initialise I2C comm
  serialNo = EEPROM.read(SERIAL_NUMBER_ADDRESS);
  ID = int(serialNo);
  setupMPU();
}

void loop() {
  if (Serial.available() > 0) {
    // Read data sent from laptop
    byte cmd = Serial.read();
    switch(char(cmd)) {
      // Send ACK back to laptop
      case 'H':
        Serial.write('A');
        handshake_completed = false;
        break;
      // Received ACK from laptop, handshake process is completed
      case 'A':
        handshake_completed = true;
        break;
      // Received instruction from laptop to reset beetle
      case 'R':
        resetFunc();
        break;
    }
  }
```

Figure 4.1.4.2.5: Hello Packet Received from Laptop

```
if (beetle.waitForNotifications(3.0) and (ACK_BOOL_DICT[beetle.addr] == True)):
    characteristics.write(bytes(ACK,'utf-8'),withResponse = False)
    logger.info(f"HANDSHAKE ATTEMPT {handshake_attempts}: ACK packet sent to Beetle {BEETLE_DICT[beetle.addr]}")

    HANDSHAKE_BOOL_DICT[beetle.addr] = True
    logger.info(f"Handshake sequence completed. Laptop is successfully connected to Beetle {BEETLE_DICT[beetle.addr]}")
    return True
handshake_attempts += 1
```

Figure 4.1.4.2.6: Laptop Waiting for Ack Packet from Bluno Beetle

The *handleNotification()* method in class *MyDelegate* will verify that the 'A' character has been received by the laptop ((Figure 4.1.4.2.7) and the *initHandshake()* method will proceed to send the final 'A' character back to the Bluno Beetle to end the 3-way handshake process (Figure 4.1.4.2.6).

```
class MyDelegate(btle.DefaultDelegate):
    def __init__(self,addr):
        btle.DefaultDelegate.__init__(self)
        self.beetle_addr = addr
        self.buffer = b''

    def handleNotification(self,cHandle,fragment):
        packet_size = len(fragment)

        # Handshake Process
        if not HANDSHAKE_BOOL_DICT[self.beetle_addr]:
            if packet_size == 1:
                packet_type = struct.unpack('<c',fragment)[0]
                # ACK packet for handshake
                if packet_type == b'A':
                    logger.info(f"ACK packet has been received from Beetle {BEETLE_DICT[self.beetle_addr]}.")
                    ACK_BOOL_DICT[self.beetle_addr] = True
```

Figure 4.1.4.2.7: Laptop Receiving Ack Packet from Bluno Beetle

The laptop will carry out this handshake process for up to 3 attempts, following which the handshake process will be deemed to have failed and the laptop will proceed to disconnect and reconnect with the Bluno Beetle before initialising handshake again as seen in Figure

4.1.4.2.8. Once the handshake process has completed, the laptop is now ready to receive data from the Bluno Beetle. The Bluno Beetle is also ready to send data to the laptop.

```
    # Reconnect to beetle if handshaking fails after 3 attempts
    if handshake_attempts > 3:
        logger.error(f"Handshake sequence with Beetle {BEETLE_DICT[beetle.addr]} failed. Reattempting handshake...")
        reconnectBeetle(beetle)
        initHandshake(beetle,characteristics)

except Exception as e:
    logger.error(f"{e} Error handshaking with Beetle {BEETLE_DICT[beetle.addr]}")
    reconnectBeetle(beetle)
    initHandshake(beetle,characteristics)
```

Figure 4.1.4.2.8: Laptop Reattempting Handshake Process

4.1.4.3 Packet Format

Initial Packet Format

The maximum recommended payload size for the Bluno Beetle is 20 bytes as the Bluno Beetle uses the BLE, which is a Bluetooth 4.0 technology (Stack Overflow, 2016). Hence, we must ensure that the total packet size transmitted is kept within the size limit of 20 bytes.

As explained in Section 4.1.4.1, the initial plan was to have 3 different packet types to be sent from the Bluno Beetle to the laptop — IMU, EMG and TIME packet. The TIME packet consisted of the 1 byte packet type which identifies the packet as a TIME packet, the 1 byte device ID, an unsigned long timestamp which is 4 bytes, as well as a 1 byte checksum at the end of the packet. This packet was initially planned for the purpose of clock synchronization. The TIME packet format is as listed in Figure 4.1.4.3.1 below.

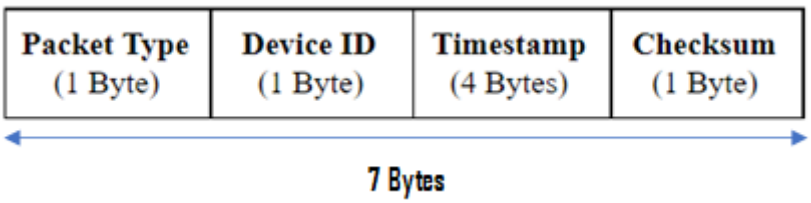


Figure 4.1.4.3.1: TIME Packet Format

Once the clock synchronization has been completed, the Bluno Beetle would then start sending the sensor data to the laptop after it has properly formatted the data into data packets. As there are two sensors involved, there are two different data packet formats that the Bluno Beetle can send to the laptop. We had decided to adopt the sampling rate of 20 Hz in measuring the sensor readings as research has shown that 20 Hz allows for good recognition performance for a wearable (H. Junker et al., 2004)

The first data packet was the IMU packet, which contains the dataset from the MYU 6050 sensor. The data that the sensor reads are the X,Y and Z axis from both the accelerometer and gyroscope respectively. The IMU packet consisted of the 1 byte packet type, 1 byte start flag to indicate the start of a dance move, the 1 byte device ID, the 4 byte timestamp and the 1 byte checksum. The sensor readings from the accelerometer and the gyroscope each transmit 6 bytes worth of data, which equates to 2 bytes per axis. Figure 4.1.4.3.2 illustrates the format of the IMU packet.

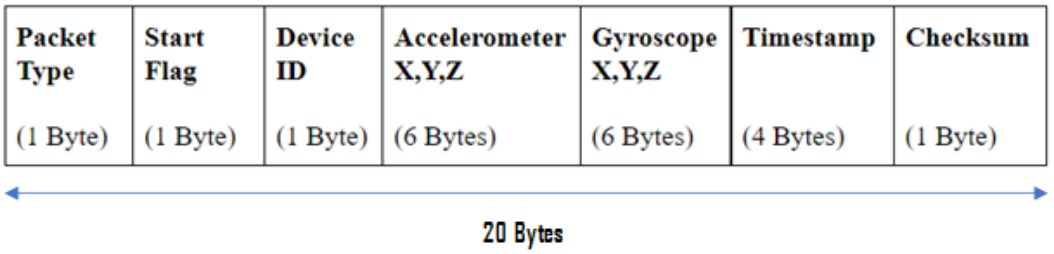


Figure 4.1.4.3.2: IMU Packet Format

The second data packet is the EMG packet, which contains the dataset captured from the MyoWare Muscle Sensor. Back then, we had tentatively assigned 6 bytes to the data readings from the muscle sensor, with 2 bytes each for the max amplitude, mean amplitude, and root mean square amplitude in the time domain. The rest of the EMG packet consisted of the 1 byte packet type, 1 byte device ID, as well as the 1 byte checksum. The EMG packet format is illustrated in Figure 4.1.4.3.3.

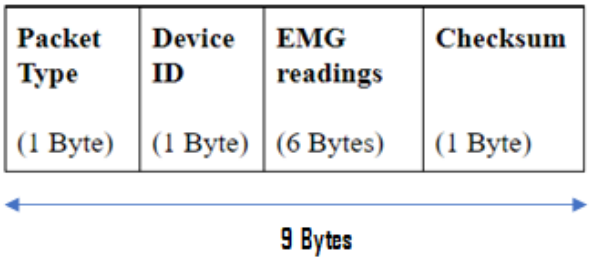


Figure 4.1.4.3.3: EMG Packet Format

Final Packet Format

In our final implementation, we only had 2 different types of packet that were sent from the Bluno Beetle to the laptop — DATA and POSITION packets. We also decided to stick with the sampling frequency of 20 Hz.

For the DATA packet, it no longer contains the 4 byte timestamp, 1 byte start flag, and 1 byte device ID as seen in Figure 4.1.4.3.4. The start of a new move is instead detected on the laptop, and the device ID is now defined in the OS environment when running the laptop code. Instead, we have added a 1 byte moving flag to inform the laptop of whether a dancer is moving or in the idling state. Additionally, we also combined the EMG data with the MPU6050 sensor readings. This is because we realised that we only require the 2 byte mean amplitude from the EMG to display the fatigue level of the dancer. Furthermore, this EMG data is only read from the dancer who is wearing the EMG set, while the mean amplitude values for the other dancers are returned as value 0 as seen in Figure 4.1.4.3.5. Finally, to ensure that the DATA packet is of size 20 bytes, we also add a 2 byte padding to the packet. We used struct to define the DATA packet and the respective content that it holds, as seen in Figure 4.1.4.3.6.

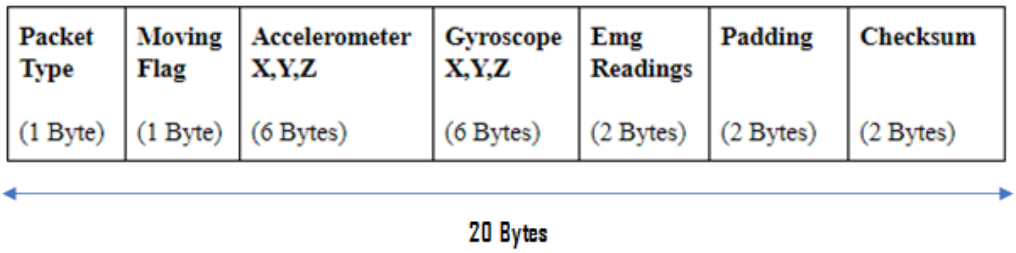


Figure 4.1.4.3.4: DATA Packet Format

```
float calculateEMGData() {
  if (ID == 6) { // Only read EMG data if it is EMG set
    totalValue = 0;
    meanAmplitude = 0;

    // 35 samples with 1kHz frequency, tuned down to maintain data reading at 20 Hz
    for (int i = 0; i < EMG_SAMPLES; i++) {
      float sensorValue = analogRead(A0); //read the analog value
      float convertedValue = (sensorValue / 1024.0) * 5; //convert into voltage range

      totalValue += convertedValue;
      delay(1);
    }
    meanAmplitude = totalValue / (EMG_SAMPLES * 1.0);
    scaledMeanAmplitude = round(meanAmplitude * 100.0);
  } else {
    scaledMeanAmplitude = 0; // Send as 0 if it is not EMG set
  }
  return scaledMeanAmplitude;
}
```

Figure 4.1.4.3.5: Reading of EMG Data Only From the EMG Set

```
// Define Data packet format
struct Data_Packet {
  char type;
  char moving;
  int accx;
  int accy;
  int accz;
  int gyrox;
  int gyroy;
  int gyroz;
  int emg;
  int padding;
  int checksum;
};
```

Figure 4.1.4.3.6: DATA Struct Format

As the data size of a float is 4 bytes which is very expensive, we decided to scale the EMG and the X,Y and Z axis readings from the MPU 6050 accordingly and convert the readings to 2 byte int type. This can be seen in Figure 4.1.4.3.7.

```
  //process the moving data
  if (dancing) {
    //process Accel Data
    //convert data to g. LSB per g = 16384.0 from the datasheet.
    //MPU6050 has the range of +-2g.
    gForceX = accelX / 16384.0;
    gForceY = accelY / 16384.0;
    gForceZ = accelZ / 16384.0;
    scaledgForceX = (round(gForceX * 100.0));
    scaledgForceY = (round(gForceY * 100.0));
    scaledgForceZ = (round(gForceZ * 100.0));
  } else { //process the walking/stop data
    gForceX = accelX / 16384.0;
    gForceY = accelY / 16384.0;
    gForceZ = accelZ / 16384.0;
    scaledgForceX = (round(gForceX * 100.0));
    scaledgForceY = (round(gForceY * 100.0));
    scaledgForceZ = (round(gForceZ * 100.0));
  }
}
```

Figure 4.1.4.3.7: Scaling the Sensor Readings

For the POSITION packet, it only consists of a 1 byte packet type and a 1 byte position data (Figure 4.1.4.3.8). The position data can be a char 'S', 'L' or 'R', depending on whether the detected movement of the dancer is a Stay, Left or Right turn. The rest of the packet is formed up with the padding and checksum to ensure that the packet is of size 20 bytes. The packet format in Struct can be seen in Figure 4.1.4.3.9.

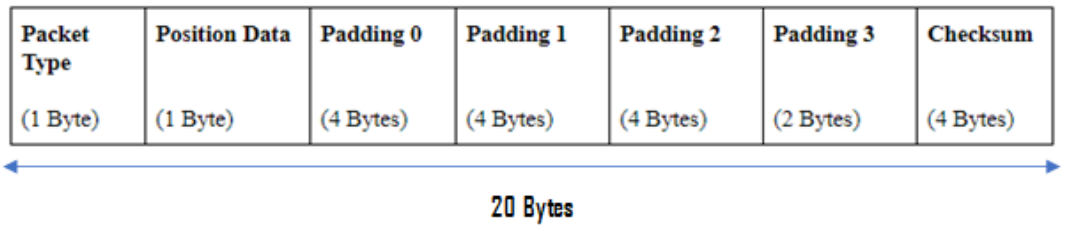


Figure 4.1.4.3.8: POSITION Packet Format

```
// Define Position packet format
struct Position_Packet {
  char type;
  char newPosition;
  long padding0;
  long padding1;
  long padding2;
  int padding3;
  long checksum;
};
```

Figure 4.1.4.3.9: POSITION Struct Format

4.1.4.4 Baud Rate

Baud rate determines the rate of data transmission over serial communication, and the baud rate has to be the same on both the Bluno Beetle and the laptop in order for data transmission between them to take place successfully. The baud rate that we used in this project is 115200 bps. A higher baud rate would result in more errors in data transmission as the sender may be unable to send a complete data packet in the short amount of time. A lower baud rate would cause an increase in latency of the system, which makes the data received to be less real-time.

Throughout our entire testing experience, the chosen baud rate of 115200 bps did not cause any problems. Hence, we decided to stick to 115200 bps as the chosen final baud rate.

4.1.4.5 Receiving and Sending Data

After the connection between the laptop and Bluno Beetle has been established in the *establish_connection()* function and a new thread has been started, the *run()* method in the threading.Thread class will be executed. A socket connection is established with the server that has been started by the external communications, and this connection channel serves to facilitate data transmission between the laptop and the external communications code. Figure 4.1.4.5.1 shows the socket connection as explained above.

```
def run(self):
    global SEND_BUFFER
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect(("localhost", 8888))
```

Figure 4.1.4.5.1: Establishing Socket Connection

To prevent an overflow of data at the external communications, we have implemented a congestion control system to dictate the flow of data from the laptop to the external communications. The data that the laptop receives from the Bluno Beetle will be appended to a buffer array *SEND_BUFFER*, and once the length of this buffer has increased to 3 or more, the data in *SEND_BUFFER* is joined into a String and sent over to the external communications. *SEND_BUFFER* is then cleared to continue receiving data. This can be seen in Figure 4.1.4.5.2.

```
# Congestion control of data to external comms
if len(SEND_BUFFER) >= 3:
        compiled_data = "".join(SEND_BUFFER)
        sock.sendall(compiled_data.encode())
        SEND_BUFFER = []
```

Figure 4.1.4.5.2: Data Transmission from Laptop to External Communications

The *waitForNotifications()* method in *run()* allows the laptop to receive data from the Bluno Beetle continuously until there is a disruption. This disruption could be caused by exceptions such as the BTLEDisconnectError exception, and in that case the laptop attempts to reconnect to the Beetle through the *reconnectBeetle()* method. Thereafter, the laptop calls for the reset of the Bluno Beetle before conducting the 3-way handshaking process with the Beetle. The current socket connection is then terminated and a new socket connection is initiated and data transmission resumes. Figure 4.1.4.5.3 illustrates the above flow.

```
        if self.beetle.waitForNotifications(2.0):
            continue
        idle_count += 1
        logger.info(f"idle count: {idle_count}")
        if idle_count == 1: # Beetle requires reset
            break
    except BTLEDisconnectError:
        logger.error(f"BTLEDisconnect Error in Beetle {BEETLE_DICT[self.beetle.addr]}")
        reconnectBeetle(self.beetle)
        self.reset()
        initHandshake(self.beetle,self.characteristics)
        sock.close()
        self.run()
logger.error(f"Error in Beetle {BEETLE_DICT[self.beetle.addr]}")
reconnectBeetle(self.beetle)
self.reset()
initHandshake(self.beetle,self.characteristics)
sock.close()
```

Figure 4.1.4.5.3: Receiving Data and Reconnection

Data Packet

Once a packet has been deemed as a complete packet, this packet is identified by its first element, which could be a 'D' or 'P' character. A Data packet's first element is a 'D' character, and *struct.unpack()* is then invoked to convert the C structs that have been represented as Python bytes objects into their respective original data type as seen in Figure 4.1.4.5.4. The checksum of the unpacked packet is then tabulated through the *calcDataChecksum()* function. This will be elaborated in the section below.

```
# DATA Packet
if fragment[0] == 68: # If fragment[0] is char 'D'
    packet = struct.unpack('<cchhhhhhhhh', fragment)
```

Figure 4.1.4.5.4: Struct.unpack Function Invoked

Once it has been verified that the received packet is indeed the intended packet that was sent from the Bluno Beetle, the laptop proceeds to extract the respective data from the unpacked Data packet, as shown in Figure 4.1.4.5.5. In our final implementation, we are only sending the Data packets when the dancer is moving, and the data sent to the external communications is formatted as shown in Figure 4.1.5.5.6.

```
arduino_checksum = packet[-1]
checkSumState = calcDataChecksum(packet,arduino_checksum)
if (checkSumState):
    beetle_pos = 0 # Hand Beetle
    moving = packet[1]
    accx = packet[2]
    accy = packet[3]
    accz = packet[4]
    gyrox = packet[5]
    gyroy = packet[6]
    gyroz = packet[7]
    emg = packet[8] / 100.0
```

Figure 4.1.4.5.5: Extracting the Data Packet

```
# Send dance move data when dancer is moving
if moving_packet == 1:
    final_data = f"#{DANCER_ID},{beetle_pos},{gyrox},{gyroy},{gyroz},{accx},{accy},{accz},{emg},{moving_packet}\n"
    SEND_BUFFER.append(final_data)
```

Figure 4.1.4.5.6: Formatted Data Sent to External Communications

As mentioned earlier, the laptop handles the identification of the start of a new dance move, hence this is implemented with the use of the *START_MOVE* flag which detects whether a Data packet is the start of a new dance move. The *START_MOVE_TIME* flag is used to detect whether the timestamp of the start of a new dance move has been sent to the external communications. When a dancer first stops moving, the *START_MOVE* flag is set to False. When a dancer starts moving, the *START_MOVE* and *START_MOVE_TIME* flag are then set to True and the timestamp of that move is captured instantaneously. Once the timestamp has been sent to the external communications, the *START_MOVE_TIME* flag is then set to False. This ensures that the start timestamp is only sent once. On the external communications end, the 3 dancers' start timestamp are then used to compute the sync delay for each move. The above flow is illustrated in Figure 4.1.4.5.7.

```
if (moving == b'Y' and beetle_pos == 0):
    moving_status = "Hand Moving"
    moving_packet = 1
    VERY_FIRST = False
    POS_DATA_SENT = True

    # Capture start of new dance move timestamp
    if not START_MOVE and not START_MOVE_TIME:
        START_MOVE = True
        start_time = time.time()
        START_MOVE_TIME = True

    # Reset flag status that keeps track of dancer movement status
    if not STOP_FIRST_BOOL:
        STOP_FIRST_BOOL = True

elif (moving == b'N' and beetle_pos == 0):
    moving_status = "Hand Not Moving"
    moving_packet = 0

    # Reset flag status that keeps track of first move of each dance move
    if START_MOVE:
        START_MOVE = False

    # Keep track of time elapsed since dancer stops moving for position data sending
    if STOP_FIRST_BOOL:
        STOP_FIRST_TIME = time.time()
        STOP_FIRST_BOOL = False
        POS_DATA_SENT = False
```

Figure 4.1.4.5.6: Logic to Detect Start Move

Position Packet

Similar to the Data packet, the Position packet is identified by the packet's first element as a 'P' character. Struct.unpack() is then invoked to convert the received data into its original data type, before the checksum is tabulated. Once the packet has been deemed as the correct packet, the position data is extracted and the *POS_DETECTED_BOOL* flag is then set to True to indicate that the dancer's new position has been detected. The laptop then uses the *STOP_FIRST_TIME* variable that stores the timestamp when the dancer first stops dancing as shown in Figure 4.1.4.5.6, as well as the current time, to determine if the position packet

should be sent to the external communications. The sending of the position data to the external communications can be seen in Figure 4.1.4.5.7.

```
# Very first dance move
if VERY_FIRST:
    # If 41 seconds has passed and position data is not sent yet
    if round(time.time() - STOP_FIRST_TIME) >= 41 and not POS_DATA_SENT:

        # Send S as position change if no position change is detected
        if not POS_DETECTED_BOOL:
            POS_DETECTED = "S"

        try:
            position_data = f"${DANCER_ID},{POS_DETECTED}\n"
            SEND_BUFFER.append(position_data)
            STOP_FIRST_TIME = 9999999999
            logger.info(f"{BEETLE_TYPE[self.beetle_addr]} BEETLE NEW POSITION: {POS_DETECTED}")
            POS_DATA_SENT = True
            POS_DETECTED_BOOL = False
        except Exception:
            print(traceback.format_exc())

# For every other dance moves
else:
    # If 16 seconds has passed and position data is not sent yet
    if round(time.time() - STOP_FIRST_TIME) >= 16 and not POS_DATA_SENT:

        # Send S as position change if no position change is detected
        if not POS_DETECTED_BOOL:
            POS_DETECTED = "S"

        try:
            position_data = f"${DANCER_ID},{POS_DETECTED}\n"
            SEND_BUFFER.append(position_data)
            STOP_FIRST_TIME = 9999999999
            logger.info(f"{BEETLE_TYPE[self.beetle_addr]} BEETLE NEW POSITION: {POS_DETECTED}")
            POS_DATA_SENT = True
            POS_DETECTED_BOOL = False
        except Exception:
            print(traceback.format_exc())
```

Figure 4.1.4.5.7: Logic to Send Position Data

The time threshold set for the very first dance move is 41 seconds and 16 seconds for all the other dance moves. The purpose of this time threshold is to allow the dancer to have an adequate amount of time to execute the position change after the new dance move has been generated by the eval_server. The time threshold of the very first move is larger to factor in the connection time of the dancers to the laptop, where potential disconnection can happen during the initial establishment of connection.

The sending of the position change to the external communications also has a ‘safety’ system in place, where in the event that no position packet is sent from the dancer, it will be deemed as a Stay position change for the dancer. This is because our group had encountered situations where the position packet was strangely not sent from the Bluno Beetle to the laptop. This system also ensures that even if the dancer’s beetle disconnects from the laptop during the position change, the correct position change that is detected on the Bluno Beetle will still be sent to the laptop once the connection has been reestablished. This helps to prevent any loss of important data and enhances the reliability of the system.

4.1.5 Handling Reliability Issues

It is of paramount importance to ensure that the communication protocol between the Bluno Beetles and laptops is as reliable and resistant against errors that may occur during the data

transmission process. Hence, several functions have been implemented to enhance the overall reliability of the communication system.

Checksum

Firstly, in ensuring that data is accurately transmitted, a checksum is appended to the end of all data packets. This checksum value will then be verified on the laptop by comparing the computed checksum value with the checksum that it receives. The checksum value is calculated through the XOR operation of every byte in the data packet on both the Bluno Beetle and the laptop. If the computed checksum value is different from the checksum that is sent from the Bluno Beetle, that indicates a possible corruption of data. The implementation of the checksum has not changed throughout the project as it has served its purpose very well. Figure 4.1.5.1 illustrates the checksum functions that we had implemented on the Bluno Beetle in the Arduino IDE for both the Data and Position packet types.

```
// Calculate Data packet checksum
int calcDataChecksum(Data_Packet packet) {
  return packet.type ^ packet.moving ^ packet.accx ^ packet.accy ^ packet.accz ^ packet.gyrox ^ packet.gyroy ^ packet.gyroz ^ packet.emg ^ packet.padding;
}

// Calculate Position packet checksum
long calcPositionChecksum(Position_Packet packet) {
  return packet.type ^ packet.newPosition ^ packet.padding0 ^ packet.padding1 ^ packet.padding2 ^ packet.padding3;
}
```

Figure 4.1.5.1: Checksum Function on the Arduino IDE

```
def calcDataChecksum(data_packet,arduino_checksum):
    try:
        data_check = data_packet[0:len(data_packet)-1]
        checksum = ord(data_check[0])
        for i in range(1,len(data_check)):
            # Convert bytes to int type
            if type(data_check[i]) == bytes:
                checksum ^= ord(data_check[i])
            else:
                checksum ^= data_check[i]

        # Received packet is the correct intended packet
        if (checksum == arduino_checksum):
            return True
        else:
            return False

    except ValueError:
        return False
```

Figure 4.1.5.2: Checksum Function on the Laptop

As seen in Figure 4.1.5.2, the checksum function on the laptop iterates through the unpacked packet and if the element is of data type bytes, we use the Python *ord()* function to convert it to int type. Thereafter, we carry out the XOR operation using this element with the checksum variable. If the final calculated checksum is of the same value as the arduino checksum value, we will return True to indicate that this packet is accurately sent. Else, this packet will be passed.

However, we have also bear in mind that the use of checksum value to verify the correctness of data received may be limited by the fact that checksum value can only detect single single-bit errors. Hence, some data corruption may go unnoticed by the protocol. However, after carrying out observations on the use of checksum to detect data corruption, we realised that the limitations are not apparent and have not disrupted our system. Hence, we did not implement alternative checks such as the Cyclic Redundancy Check (CRC).

Disconnection and Reconnection

To prepare for the event that the Bluno Beetle gets disconnected from the laptop, a reconnect function was implemented with the try-except logic. Under this logic, when the Bluno Beetle abruptly disconnects from the laptop, the disconnect exception will be thrown and the reconnect function will be executed. The reconnect function will then set the *HANDSHAKE_BOOL_DICT* variable to False as the Bluno Beetle will undergo the 3-way handshake process when it has reconnected to the laptop. Once the Bluno Beetle has been reconnected to the laptop, the laptop will send a Reset packet to the Bluno to reset its status. The 3-way handshake process then takes place and data transmission resumes thereafter.

By implementing the try-except logic, this automates the reconnection of the Bluno Beetle to the laptop when it is disconnected which enhances reliability and efficiency. This also ensures that data loss is kept to the minimum so that the system remains robust and effective. The *reconnectBeetle()* function can be seen in Figure 4.1.5.3.

```
# Function to reconnect laptop to beetle
def reconnectBeetle(beetle):
    HANDSHAKE_BOOL_DICT[beetle.addr] = False
    beetle._stopHelper()
    beetle.disconnect()
    sleep(2.0)
    try:
        logger.info(f"Reconnecting to Beetle {BEETLE_DICT[beetle.addr]}")
        beetle.connect(beetle.addr)
        sleep(1.0)
        beetle.withDelegate(MyDelegate(beetle.addr))
        logger.info(f"Reconnected to Beetle {BEETLE_DICT[beetle.addr]}")

    except Exception as e:
        logger.error(f"{e}  Error reconnecting with Beetle {BEETLE_DICT[beetle.addr]}")
        reconnectBeetle(beetle)
```

Figure 4.1.5.3: The Reconnect Function

Data Fragmentation

Data fragmentation was an issue that we had commonly faced after we had implemented the basic data transmission. Due to the high speed that data is being sent from the Bluno Beetle to the laptop, there are cases where the data of 20 bytes is broken up into multiple fragments and sent in separate transmission instances. This posed a huge problem when we did not have any data fragmentation handling protocols in place, especially when the laptop attempted to unpack this packet using *struct.unpack()* which returns an error.

Hence, we followed through with our initial plan to counter this issue by implementing a data buffer. If the data buffer is empty and the received data packet from the Bluno Beetle is less than 20 bytes, the incomplete data packet is then appended into the data buffer. If the resulting data buffer is of size 20 bytes, this entails that a complete packet of size 20 bytes has been reassembled and the packet is ready to be unpacked.

If the data buffer exceeds 20 bytes, the first 20 bytes of the buffer are extracted and this forms the complete packet of size 20 bytes. The remaining bytes in the buffer are retained for future assembling purposes. The data buffer will be cleared once a complete packet has been assembled and no data is left unassembled. The data fragmentation handling protocol can be seen in Figure 4.1.5.4.

```
# Data assembling is required
else:
    existing_fragmented_data = BUFFER_DICT[self.beetle_addr]

    # No data fragments currently present
    if existing_fragmented_data == b'':
        existing_fragmented_data = fragment
        fragmented_data_length = len(existing_fragmented_data)
    else:
        existing_fragmented_data += fragment
        fragmented_data_length = len(existing_fragmented_data)

    # Assembled data fragments form exactly 1 packet
    if fragmented_data_length == 20:
        fragment = existing_fragmented_data
        # Clear assembled data fragments
        BUFFER_DICT[self.beetle_addr] = b''

        # Send for data reading directly since it is a complete packet
        self.handleData(fragment)

    # Current data fragments form more than 1 complete packet
    elif fragmented_data_length > 20:
        # Extract only the first 20 bytes for a complete packet
        fragment = existing_fragmented_data[0:20]

        # Store rest of fragment into buffer dictionary
        BUFFER_DICT[self.beetle_addr] = existing_fragmented_data[20:]
        # Send for data reading directly
        self.handleData(fragment)

    else:
        BUFFER_DICT[self.beetle_addr] += fragment
```

Figure 4.1.5.4: The Data Fragmentation Handling Protocol

Overall, this data fragmentation handling protocol has been proven to help ensure that data corruption is kept to the minimum and under good control. This prevents a minor corrupted data problem from snowballing into a system failure. This protocol has therefore ensured that our system is functioning with the highest reliability in place, and users are provided with the optimal experience of accurate data transmission and minimized system disruptions.

4.2 External Communications

4.2.1 Introduction

External Communications is a necessary component in order to facilitate data transfer between all the different components. The plan is to have the data from dancer's laptops sent to the Ultra96 via the External Communications, and then that data is encrypted and sent to the evaluation server. This section will explain how we will form stable connections between the dancers' laptops and the Ultra96, and the Ultra96 and the laptop. Additionally, we will discuss how we settled the synchronization delay between dancers and the various libraries and APIs used. In this revised edition, we will go about explaining in significantly more detail how we went about achieving the aforementioned connections while noting the changes we carried out along the way. The name of the main External Communications code is ExtComms.py.

4.2.2 Communication between laptops and Ultra96

Transmission Control Protocol (TCP) will be used to facilitate connection between the laptops assigned to each dancer and Ultra96. Each laptop will have a client that connects to the Ultra96 via TCP socket to ensure stable and reliable transference of dancer's movement data along with timestamps to the Ultra96. We run the client on the Ultra96 first and then run the respective client codes on the laptop. The data that is sent will be mainly raw data of gyro x, y, z and acceleration x, y, z from each dancer. The Ultra96 will use threading and the Twisted library in order to handle concurrency between the different laptops.

4.2.2.1 Threading between laptops and Ultra96

Since multi-threaded programming is often tricky, even with high level abstractions, and since forking Python processes has many disadvantages, like Python's reference counting not playing well with copy-on-write and problems with shared state, it was felt the best option was an event-driven framework. A benefit of such approach is that by letting other event-driven frameworks take over the main loop, server and client code are essentially the same - making peer-to-peer a reality. While Twisted includes its own event loop, Twisted can already interoperate with GTK+'s and Tk's main loops, as well as provide an emulation of event-based I/O for Jython (specific support for the Swing toolkit is planned). Client code is never aware of the loop it is running under, as long as it is using Twisted's interface for registering for interesting events.

Having said that, with the Twisted library, I did not actually have to do multithreading in order to manage the 3 connections between each laptop and the ExtComms.py. Initially with native Python multithreading, I would have had to set up an individual thread for each connection of a laptop, and with the Global interpreter Lock (GIL), which prevents multiple native threads from executing Python bytecodes at once (Jain, n.d), I would have had to worry about synchronizing the data packets from each laptop due to the non thread safe nature of some of the functions.

Hence, by using the Twisted library, I can effectively circumvent this problem by simply using only 1 thread to manage all 3 connections. Due to the presence of non-thread-safe

pieces of code, methods within Twisted will generally only be called by the reactor thread itself as can be seen here:

```
try:
        reactor.listenTCP(8888, LaptopFactory())
        thread = threading.Thread(target=reactor.run, args=(False,))
        thread.start()

        mqueue = Queue() #to communicate with thread
```

Figure 4.2.2.1.1: Reactor thread code snippet

Here we will be calling 2 classes. One is seen here, specifically the LaptopFactory() class that handles initial setup, disconnection and the calling of the second or our main class LaptopSide() as can be seen here:

```
class LaptopFactory(Factory):

    def __init__(self,):

        self.dancernumber = 0  #number of dancers currently connected
        self.skip_initial_readings_cnt = [[50, 50], [50, 50], [50, 50]]
        self.counter = 0
        self.init_counter = 24
        self.start_time = time.time()
        self.is_idle = True
        self.Dancer1mlclass = ML()
        self.Dancer2mlclass = ML()
        self.Dancer3mlclass = ML()

    #protocol = EchoClient

    def clientConnectionFailed(self, connector, reason):
        print("Connection failed - goodbye!")
        reactor.stop()

    def clientConnectionLost(self, connector, reason):
        print("Connection lost - goodbye!")
        reactor.stop()

    def buildProtocol(self, addr):
        return LaptopSide(self)
```

Figure 4.2.2.1.2: LaptopFactory code snippet

The 3 Dancer_ml classes are for interacting with the Machine Learning (ML) model and will be explained below.

For the second class, LaptopSide(), we handle what will be shown on the terminal that is running the ExtComms.py code and the processing of the different packets received from the laptops.

```
class LaptopSide(LineReceiver):
    """Once connected, send relevant data."""

    delimiter = b"\n"

    def __init__(self, persistent_data):
        self.persistent_data = persistent_data
        self.clearLineBuffer()

    def connectionMade(self):
        print("A New Dancer is on the Stage!")
        self.persistent_data.dancernumber += 1
        self.printDancerNumber()

    def connectionLost(self, reason):
        print("A Dancer has left the Stage!")
        self.persistent_data.dancernumber -= 1
        self.printDancerNumber()

    def printDancerNumber(self):
        print(
            "There are currently %d connected dancers."
            % self.persistent_data.dancernumber
        )
```

Figure 4.2.2.1.3: LaptopSide code snippet(top)

Here, whenever we have a laptop connected or disconnected, the corresponding message will be printed on the terminal, while also updating the number of connected dancers.

4.2.2.2 Packet processing

```
def lineReceived(self, line):  # line is data received
        global dancer1list
        global dancer2list
        global dancer3list
        #global begin_time
        try:
            line = line.decode()

            self.getFrequency()

            if line[0] == "#":  # add case for dummy data

                (
                    dancer_id,
                    data_type,
                    gyrox,
                    gyroy,
                    gyroz,
                    accx,
                    accy,
                    accz,
                    emg,
                    moving,
                ) = line[1:].split(",")
                # appends data for each dancer to window
                dancer_id = int(dancer_id)
                gyrox, gyroy, gyroz, accx, accy, accz= (
                    float(gyrox),
                    float(gyroy),
                    float(gyroz),
                    float(accx),
                    float(accy),
                    float(accz),
                    #float(starttime),
                )
```

Figure 4.2.2.2.1: LaptopSide code snippet(lineReceived func.)

With regards to the processing of data packets from the beetles via the laptops, this is done by our lineReceived function as can be seen in the above image. The function will expect 3 different kinds of packets, specifically sensor data packets marked by a "#" at the beginning as seen above, a time stamp packet marked by a "!", and a position packet marked by a "$". The sensor data packet, received as "#dancer_id,data_type,gyrox,gyroy,gyroz,accx,accy,accz,emg,moving", is processed as seen above.

```
elif line[0] == "!":
      (
      dancer_id,
      starttime,
      ) = line[1:].split(",")
      dancer_id = int(dancer_id)
      starttime = float(starttime)
      if (dancer_id == 1) and TIMESTAMP_BOOL_DICT["Dancer 1"] == False:
      TIMESTAMP_DICT["Dancer 1"] = starttime
      TIMESTAMP_BOOL_DICT["Dancer 1"] = True
      elif dancer_id == 2 and TIMESTAMP_BOOL_DICT["Dancer 2"] == False:
      TIMESTAMP_DICT["Dancer 2"] = starttime
      TIMESTAMP_BOOL_DICT["Dancer 2"] = True
      elif dancer_id == 3 and TIMESTAMP_BOOL_DICT["Dancer 3"] == False:
      TIMESTAMP_DICT["Dancer 3"] = starttime
      TIMESTAMP_BOOL_DICT["Dancer 3"] = True
```

Figure 4.2.2.2.2: LaptopSide code snippet(lineReceived func., timestamp packet)

This is the processing for the timestamp packets which we will be using for the sync delay calculation later on. In this case, we used dictionaries (TIMESTAMP_DICT) for each dancer in order to store their individual time stamps. To keep track of when to update them, we use another dictionary (TIMESTAMP_BOOL_DICT) in which we set it to True when the related TIMESTAMP_DICT has been updated, and False after sending a prediction to the evaluation server. The TIMESTAMP_DICT can only be updated to True if TIMESTAMP_BOOL_DICT is False.

```
elif line[0] == "$":
      (
      dancer_id,
      newPosition,
      ) = line[1:].split(",")
      dancer_id = int(dancer_id)
      newPosition = str(newPosition)
      if dancer_id == 1 and POSITION_BOOL_DICT["Dancer 1"] == False:
      POSITION_DICT["Dancer 1"] = newPosition
      POSITION_BOOL_DICT["Dancer 1"] = True
      dancer1list.clear()

      elif dancer_id == 2 and POSITION_BOOL_DICT["Dancer 2"] == False:
      POSITION_DICT["Dancer 2"] = newPosition
      POSITION_BOOL_DICT["Dancer 2"] = True
           dancer2list.clear()

      elif dancer_id == 3 and POSITION_BOOL_DICT["Dancer 3"] == False:
           POSITION_DICT["Dancer 3"] = newPosition
           POSITION_BOOL_DICT["Dancer 3"] = True
           dancer3list.clear()
```

Figure 4.2.2.2.3: LaptopSide code snippet(lineReceived func., position packet)

For the processing of position packets, it is largely the same as that of the timestamp packets. The clear of the respective lists is for the dance predictions from the ML model later. From these packets, we expect a position change string character of either “S” for stay or no change, “R” for go right or “L” for go left. They are then stored in POSITION_DICT per their respective dancers.

4.2.3 Communication between laptops and Dashboard server

Transmission Control Protocol (TCP) will be used to facilitate connection between the laptops assigned to each dancer and MongoDB Dashboard server. Each laptop will have a client that connects to the Ultra96 via TCP socket to ensure stable and reliable transference of dancer’s movement data along with timestamps to the Ultra96. We run the client on the Ultra96 first and then run the respective client codes on the laptop.

In order to be able to connect to the MongoDB server run by my teammate, I first had to specify which database to connect to and which collections I had to send the data too as can be seen in the image below:

```
client = MongoClient(

   "mongodb://172.31.20.67:2717/data"
)

db = client
data = db.data
```

Figure 4.2.3.1: MongoDB specification code snippet

The plan was that the laptop will receive the raw data of gyro x, y, z and acceleration x, y, z from each dancer and send it to the dashboard server. The Ultra96 will also have the data processed and once processed, sent to the dashboard server., just as with the evaluation server. In our current implementation, there are 2 different formats being used for each sending of information due to the schema used for their respective collections.

```
                raw_data = {
                    "userID": str(dancer_id),
                    "aX": str(accx),
                    "aY": str(accy),
                    "aZ": str(accz),
                    "gX": str(gyrox),
                    "gY": str(gyroy),
                    "gZ": str(gyroz),
                }

                emg_data = {"emgMean": str(emg)}
                if self.persistent_data.counter % 20 == 0:
                    self.persistent_data.counter += 1
                    if dancer_id == 1:
                        if data_type == 0:
                            data.d1_raw_hand_datas.insert(raw_data)
                    elif dancer_id == 2:
                        if data_type == 0:
                            data.d2_raw_hand_datas.insert(raw_data)
                    elif dancer_id == 3:
                        if data_type == 0:
                            data.d3_raw_hand_datas.insert(raw_data)

                    if emg != 0.0:
                        data.emg_datas.insert(emgdata)
```

Figure 4.2.3.2: MongoDB raw data code snippet

For the raw data collections, after we have processed the data packet as seen in Figure 4.2.2.2.1, we organize it into the schema of raw_data as seen in Figure 4.2.3.2. If this is not done properly, the database on the Mongo server side will be unable to correctly process this data. The same is done for the emg data that was in the same data packet (if there is any), but has its own separate collections. Due to the frequency of data being sent per laptop (20Hz), it was decided that we would only send the readings to the collections every 20 counts, where depending on the dancer id within the data packet would determine whose collections the data would be sent to.

```
def format_results(positions, dance_moves, dance_move, pos, sync_delay): #, positions, pos, sync_delay):
    new_positions = dancemoves.swap_positions(positions, pos)
    eval_results = f"{new_positions[0]} {new_positions[1]} {new_positions[2]}|{dance_move}|{sync_delay}|"
    if len(dance_moves) == 3:
        dashboard_results = {"predictedDance1": dance_moves[0],
                             "predictedDance2": dance_moves[1],
                             "predictedDance3": dance_moves[2],
                             "predictedPos": f"{new_positions[0]}|{new_positions[1]}|{new_positions[2]}",
                             "syncDelay" : str(sync_delay)}  #will update later depending on how you set schema amir
    elif len(dance_moves) == 2:
        dashboard_results = {"predictedDance1": dance_moves[0],
                             "predictedDance2": dance_moves[1],
                             "predictedDance3": dance_moves[1],
                             "predictedPos": f"{new_positions[0]}|{new_positions[1]}|{new_positions[2]}",
                             "syncDelay" : str(sync_delay)}

    elif len(dance_moves) == 1:
        dashboard_results = {"predictedDance1": dance_moves[0],
                             "predictedDance2": dance_moves[0],
                             "predictedDance3": dance_moves[0],
                             "predictedPos": f"{new_positions[0]}|{new_positions[1]}|{new_positions[2]}",
                             "syncDelay" : str(sync_delay)}
    #dashboard_results = f"{positions[0]} {positions[1]} {positions[2]}|{dance_move}|{new_positions[0]} {new_posit

    return eval_results, dashboard_results
```

Figure 4.2.3.3: processed data formatting code snippet

```
data.processed_datas.insert(dashboard_results)
```

Figure 4.2.3.4: MongoDB processed data insert code snippet

For the processed data collections, after we get the finalized predictions for each dancer from the ML model, we format the results as seen in Figure 4.2.3.3 to the schema designed for the processed data collections. Additionally, the new positions and sync delay are included as part of this schema. Later when sending the prediction to the eval server, the formatted dashboard results are sent to the MongoDB server via the line of code in Figure 4.2.3.4.

4.2.4 Communication between Ultra96 and evaluation server

We used the Transmission Control Protocol (TCP) to form a connection between the Ultra96 and the evaluation server. Once the dancers start to dance, the laptop will receive the raw data of gyro x, y, z and acceleration x, y, z from each dancer and with the machine learning model, process it.

4.2.4.1 Processing data into ML model

```
if data_type == 0 and moving == 1:

      if dancer_id == 1 and (POSITION_BOOL_DICT["Dancer 1"] == True):
      self.persistent_data.Dancer1mlclass.write_data(
                  dancer_id,
                  [gyrox, gyroy, gyroz, accx, accy, accz],
            )
      elif dancer_id == 2 and (POSITION_BOOL_DICT["Dancer 2"] == True):
            self.persistent_data.Dancer2mlclass.write_data(
                  dancer_id,
                  [gyrox, gyroy, gyroz, accx, accy, accz],
            )
      elif dancer_id == 3 and (POSITION_BOOL_DICT["Dancer 3"] == True):
            self.persistent_data.Dancer3mlclass.write_data(
                  dancer_id,
                  [gyrox, gyroy, gyroz, accx, accy, accz],
            )

self.handleMainLogic(dancer_id)
```

Figure 4.2.4.1.1: Passing data into the ML model snippet

To pass in the raw data into the ML model, several conditions have to be met first as seen in the above figure. The data type variable has to be 0 to signify its from the hand beetle, and the moving variable from the same packet has to be 1 to indicate that the dancer is moving. After that, based on the dancer_id in the packet, we write to the respective dancer's ML class the sensor data from the packet. However, it is only written when the dancers have updated their positions via sending a position packet and having POSITION_BOOL_DICT for the respective dancer to become True. This is to prevent the backlog of data for the previous move from being written into the ML model for the prediction of the next move and result in the model making a wrong prediction.

```
def handleMainLogic(self, dancer_id):  #receive ai prediction data

    global dancer1list
    global dancer2list
    global dancer3list

    if dancer_id ==1:
        pred1 = self.persistent_data.Dancer1mlclass.predict(dancer1list)  #give me predictions, if none give me none
        #print(pred1)
        if pred1 is not None:
            logger.info(pred1)
            mqueue.put((pred1))
            dancer1list.clear()
            self.clearLineBuffer()

    elif dancer_id ==2:
        pred2 = self.persistent_data.Dancer2mlclass.predict(dancer2list)  #give me predictions, if none give me none

        if pred2 is not None:
            logger.info(pred2)
            mqueue.put((pred2))
            dancer2list.clear()
            self.clearLineBuffer()

    elif dancer_id ==3:
        pred3 = self.persistent_data.Dancer3mlclass.predict(dancer3list)  #give me predictions, if none give me none
        if pred3 is not None:
            logger.info(pred3)
            mqueue.put((pred3))
            dancer3list.clear()
            self.clearLineBuffer()
```

Figure 4.2.4.1.2: Receiving prediction from ML model snippet

In this function HandleMainLogic, we retrieve the predictions for each dancer based on dancer_id via their respective list as seen in the above figure. So long as the pred has a value, we put said value into a queue in order to retrieve them later. After that, we clear the list and the buffer data to make sure we do not use any of this data for the next prediction.

4.2.4.2 Sending prediction to the evaluation server

```
if __name__ == "__main__":
    #logger.info("Started server on port %d" % DANCE_PORT)

    dashboarddata = {"predictedDance1": 'inactive', "predictedDance2": 'inactive', "predictedDance3": 'inactive', "predicte

    #mqueue = Queue() #to communicate with thread
    positions = [1, 2, 3]
    pos = ["S", "S", "S"]
    sync_delay = 0
    try:
        reactor.listenTCP(8888, LaptopFactory())
        thread = threading.Thread(target=reactor.run, args=(False,))
        thread.start()

        mqueue = Queue() #to communicate with thread

        input("Press any input to start evaluation server")  #press enter

        group_id = "B01"
        secret_key = "1234123412341234"
        my_client = Client(IP_ADDRESS, EVAL_PORT, group_id, secret_key) #IP_ADDRESS = 137.132.92.127, EVAL_PORT = 5000
        my_client.send_message("1 2 3" + "|" + "start" + "|" + "1.5" + "|")
        #logger.info(f"received positions: {positions}")
        counter = 1
```

Figure 4.2.4.2.1: Initial set-up for connection to eval_server snippet

We first set up the base positions, position changes and the mqueue as seen in the above figure when we first run the ExtComms.py code. After we start the thread, we would have to press enter on our side in order to initiate connection to the evaluation server (eval_servr.py) via eval_client.py which we have imported into ExtComms.py. We will explain more about eval_client.py in the Data Encryption section. Once there is the "enter" input, we are essentially entering preset values of group_id and secret key as if we were directly running eval_client.py. For the IP_ADDRESS and EVAL_PORT that are also required, we enter them manually by inputting them before calling ExtComms.py like this:

IP_ADDRESS=localhost EVAL_PORT=5000 python3 Extcomms.py

Once we attempt the connection, we just wait for the person running the eval_server.py to enter the secret key we gave them. Once the connection is finalized we have the dancers connect to the ExtComms.py.

```
while True:
    current_time = time.time()
    if counter == 1:
        if (mqueue.qsize() >= 3) or ((mqueue.qsize() <= 2 and mqueue.qsize() > 0) and (time.time() - begin_time >= 85.0)):   #should we lower a bit see how?
            logger.info("bigger than 121")
            dance_moves = list(mqueue.queue) #dance_moves1, 2,3
            dance_move = stats.mode(dance_moves)[0][0]

            if (POSITION_BOOL_DICT["Dancer 1"] == True):
                pos[0] = POSITION_DICT["Dancer 1"]
            if (POSITION_BOOL_DICT["Dancer 2"] == True):
                pos[1] = POSITION_DICT["Dancer 2"]
            if (POSITION_BOOL_DICT["Dancer 3"] == True):
                pos[2] = POSITION_DICT["Dancer 3"]

            POSITION_BOOL_DICT["Dancer 1"] = False
            POSITION_BOOL_DICT["Dancer 2"] = False
            POSITION_BOOL_DICT["Dancer 3"] = False

            if (TIMESTAMP_BOOL_DICT["Dancer 1"] == True) and (TIMESTAMP_BOOL_DICT["Dancer 2"] == True) and (TIMESTAMP_BOOL_DICT["Dancer 3"] == True):
                sync_delay = sync.calculate_sync_delay(TIMESTAMP_DICT["Dancer 1"], TIMESTAMP_DICT["Dancer 2"], TIMESTAMP_DICT["Dancer 3"])

            TIMESTAMP_BOOL_DICT["Dancer 1"] = False
            TIMESTAMP_BOOL_DICT["Dancer 2"] = False
            TIMESTAMP_BOOL_DICT["Dancer 3"] = False
            logger.info(f"predictions: {(sync_delay)}")

            logger.info(f"predictions: {(dance_move)}")
            eval_results, dashboard_results = format_results(
                positions, dance_moves, dance_move, pos, sync_delay
            )

            logger.info(f"eval_results: {eval_results}")
            logger.info(f"dashboard_results: {dashboard_results}")

            data.processed_datas.insert(dashboard_results)

            my_client.send_message(eval_results)
            positions = my_client.receive_dancer_position()
            positions = [int(position) for position in positions.split(" ")]
            logger.info(f"received positions: {positions}")
            begin_time = time.time()
            counter += 1
            dancer1list.clear()
            dancer2list.clear()
            dancer3list.clear()
            mqueue.queue.clear()
```

Figure 4.2.4.2.2: Retrieving predictions and sync delay snippet

When retrieving the predictions, the size of the mqueue has to be greater than or equal to 3 to signify that there are 3 predictions for the 3 dancers. We take the moves from the queue and see which move occurs the most to set as the main move to be sent to the evaluation server as seen in the above figure. In the event that the queue has less than 3 entries and we are about to hit the 1 minute time limit for dancing, we will send what we currently have, which is usually correct due to our model's high accuracy. This prevents action timeout from occurring. Note that the timeout is different between the 1st move and subsequent moves as the begin_time that we take reference to is different to later ones which are updated when we receive the new ground truth. To differentiate, we use a counter that starts at 1 for the 1st move. At counter = 1, the timeout is longer. When the counter is not equal to 1, the timeout is now shorter to around 51 seconds.

For the position change, so long as the respective POSITION_BOOL_DICT is True, we store the related POSITION_DICT value in pos[] array as seen in the image above. The pos[] array will hold the position change data of the dancer, which could be character 'S', 'L' or 'R'. We then set all the POSITION_BOOL_DICT values to be False to show that the positions can be updated for the next move. The position change uses the pos array to determine the new positions from the current ground truth. This is done in the format_results function in Figure 4.2.3.3 via the function swap_positions from a separate file I created: dancemoves.py. The specific function definition can be seen here:

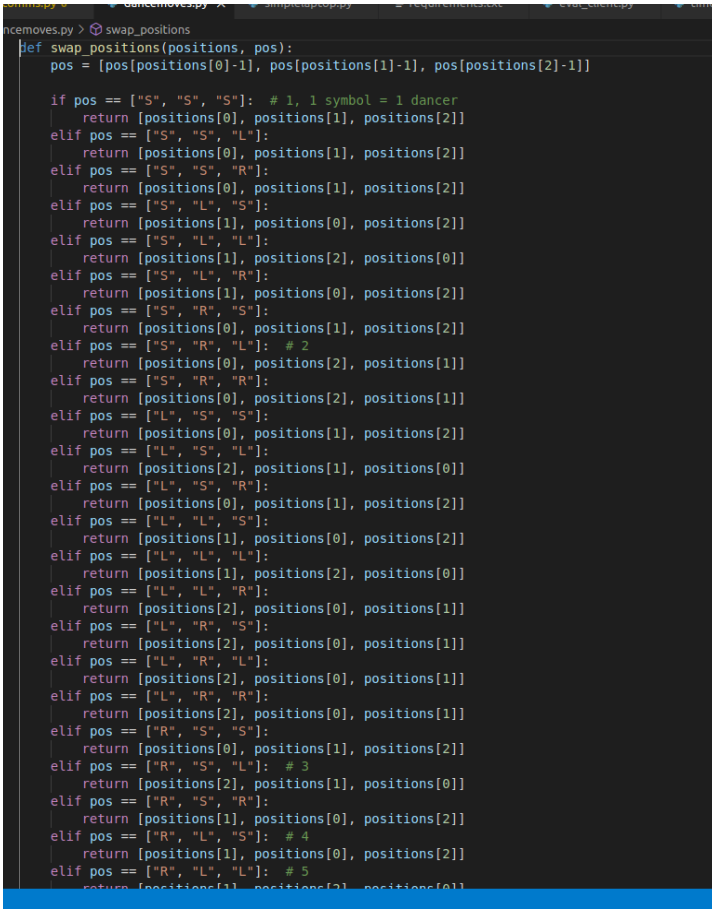


Figure 4.2.4.2.3: Retrieving predictions and sync delay snippet

While this is not the complete code, the basic idea is that there is a finite number of different orders to expect, so depending on the combination of position packet symbols, we can accurately predict the position change like this

For the sync delay, we require that all of TIMESTAMP_BOOL_DICT has to be True in order to run the sync delay calculations. I will go into more detail on how the sync delay is calculated in the Synchronization Delay section. After the calculation, all of TIMESTAMP_BOOL_DICT is set to False to allow all of TIMESTAMP_DICT to be updated again.

After that we format the data using the function format_results seen in Figure 4.2.3.3, the message format to be sent to the evaluation server looks like this:

"#POSITIONS|ACTIONS|sync delay"

where POSITIONS refers to the various positions of the 3 dancers, ACTIONS are the various dance moves that will be done, and sync delay stands for the synchronization delay between dancers in milliseconds. After sending to the evaluation server, we receive the new ground truth positions, update the begin_time and counter, and clear the dancer list and mqueue.

4.2.5 Data Encryption

We implemented AES Encryption in order to encrypt our data that is sent to the evaluation and dashboard servers. We first pad the data, then we carry out the cypher encryption on the padded data. After that, together with an initialization vector, we carry out base 64 encoding on the message and it is then ready to be sent. Below is general diagram explaining the AES encryption:

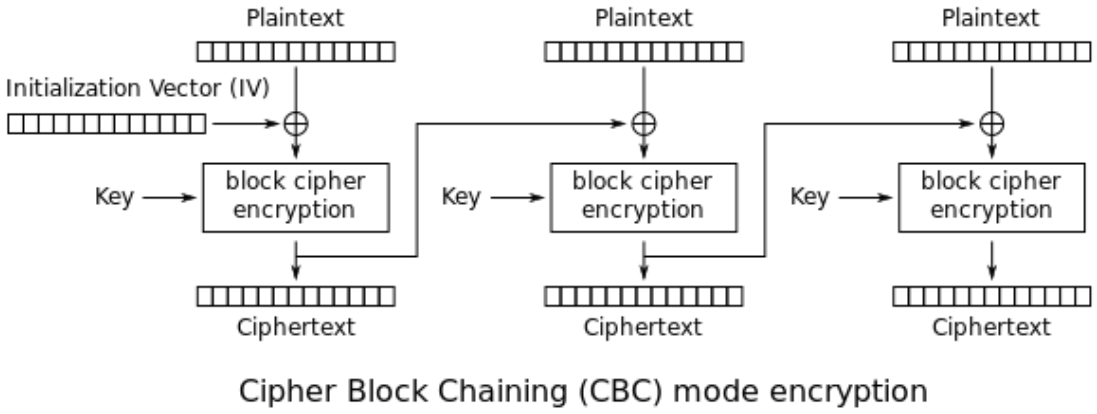
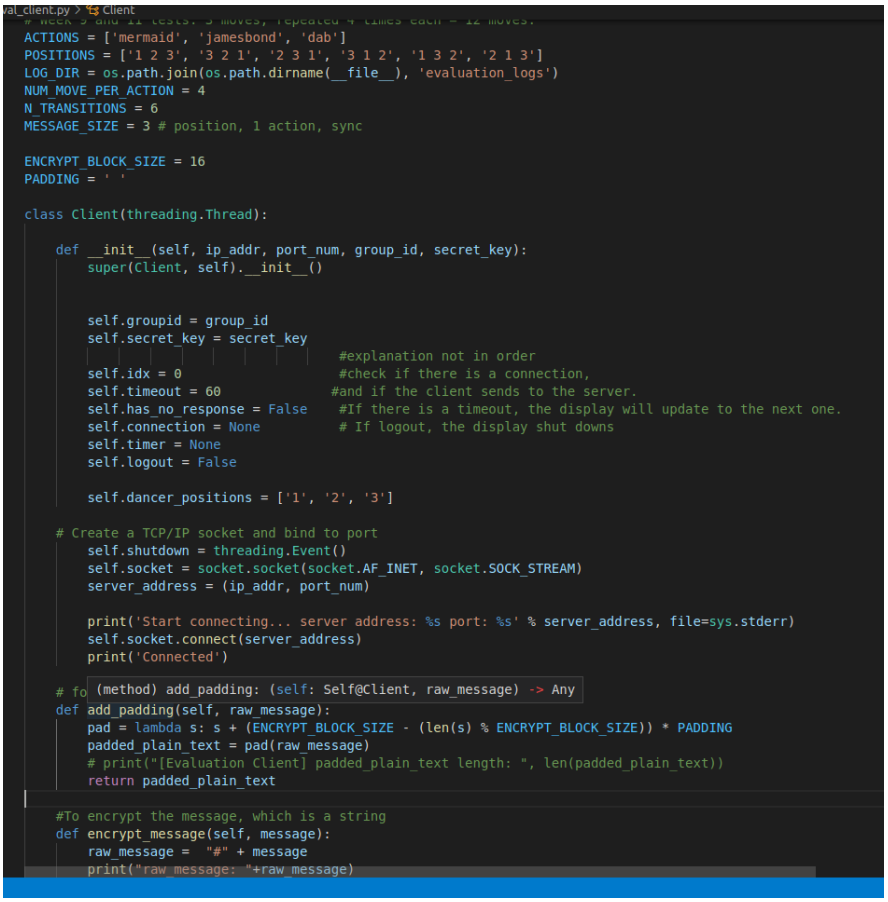


Figure 4.2.5.1: Encryption process

With that said, here is our detailed implementation and explanation of our eval_client.py that contains the encryption, starting with the 2 images below:



```
val_client.py > Client
# week 9 and 11 tests: 3 moves, repeated 4 times each = 12 moves.
ACTIONS = ['mermaid', 'jamesbond', 'dab']
POSITIONS = ['1 2 3', '3 2 1', '2 3 1', '3 1 2', '1 3 2', '2 1 3']
LOG_DIR = os.path.join(os.path.dirname(__file__), 'evaluation_logs')
NUM_MOVE_PER_ACTION = 4
N_TRANSITIONS = 6
MESSAGE_SIZE = 3 # position, 1 action, sync

ENCRYPT_BLOCK_SIZE = 16
PADDING = ' '

class Client(threading.Thread):

    def __init__(self, ip_addr, port_num, group_id, secret_key):
        super(Client, self).__init__()


        self.groupid = group_id
        self.secret_key = secret_key
                                    #explanation not in order
        self.idx = 0                #check if there is a connection,
        self.timeout = 60           #and if the client sends to the server.
        self.has_no_response = False  #If there is a timeout, the display will update to the next one.
        self.connection = None        # If logout, the display shut downs
        self.timer = None
        self.logout = False

        self.dancer_positions = ['1', '2', '3']

    # Create a TCP/IP socket and bind to port
        self.shutdown = threading.Event()
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_address = (ip_addr, port_num)

        print('Start connecting... server address: %s port: %s' % server_address, file=sys.stderr)
        self.socket.connect(server_address)
        print('Connected')

    # fo (method) add_padding: (self: Self@Client, raw_message) -> Any
    def add_padding(self, raw_message):
        pad = lambda s: s + (ENCRYPT_BLOCK_SIZE - (len(s) % ENCRYPT_BLOCK_SIZE)) * PADDING
        padded_plain_text = pad(raw_message)
        # print("[Evaluation Client] padded_plain_text length: ", len(padded_plain_text))
        return padded_plain_text

    #To encrypt the message, which is a string
    def encrypt_message(self, message):
        raw_message =  "#" + message
        print("raw_message: "+raw_message)
```

Figure 4.2.5.2: Encryption process part 1 snippet

```
#To encrypt the message, which is a string
def encrypt_message(self, message):
    raw_message =  "#" + message
    print("raw_message: "+raw_message)
    padded_raw_message = self.add_padding(raw_message)
    print("padded_raw_message: " + padded_raw_message)
    iv = Random.new().read(AES.block_size)
    aes_key = bytes(str(self.secret_key), encoding="utf8")
    cipher = AES.new(aes_key, AES.MODE_CBC, iv)
    encrypted_message = base64.b64encode(iv + cipher.encrypt(bytes(padded_raw_message, "utf8")))
    print("[Evaluation Client] encrypted_message: ", encrypted_message)
    return encrypted_message

#To send the message to the sever
def send_message(self, message):
    encrypted_text = self.encrypt_message(message)
    print("[Evaluation Client] encrypted_text: ", encrypted_text)
    self.socket.sendall(encrypted_text)

def receive_dancer_position(self):
    dancer_position = self.socket.recv(1024)
    msg = dancer_position.decode("utf8")
    return msg

def stop(self):
    self.connection.close()
    self.shutdown.set()
    self.timer.cancel()
```

Figure 4.2.5.3: Encryption process part 2 snippet

In the first image, we define the Client class with some dummy values, while also defining the add_padding function which is necessary for encryption in the next function, encrypt_message. In this function in Figure 4.2.5.1, we first attach the "#" to the message, then pad it. We then create the AES cipher using the aes_key based on our secret key (1234123412341234), setting the mode to cipher block chaining and iv variable from the AES.block_size. Then we encode the message in base64 with iv variable and utf8 encryption, resulting in the encrypted message. The send_message function is then used by eval_client.py and by extension ExtComms.py to send the message to the evaluation server and the receive_dancer_position function is used to get the new ground truth.

4.2.6 Synchronization delay

In order to account for the synchronization delay, we were initially planning on using the Network Time Protocol (NTP) in order to account for the delay between the dancers. The plan was to use the internal clock of the Ultra96 as the reference clock. This is due to the fact that we have only 1 Ultra96, so it will be easier to sync the Bluno Beetles with different time delays to the single Ultra96.

Below is a diagram and formula explaining how the time delay will be calculated.

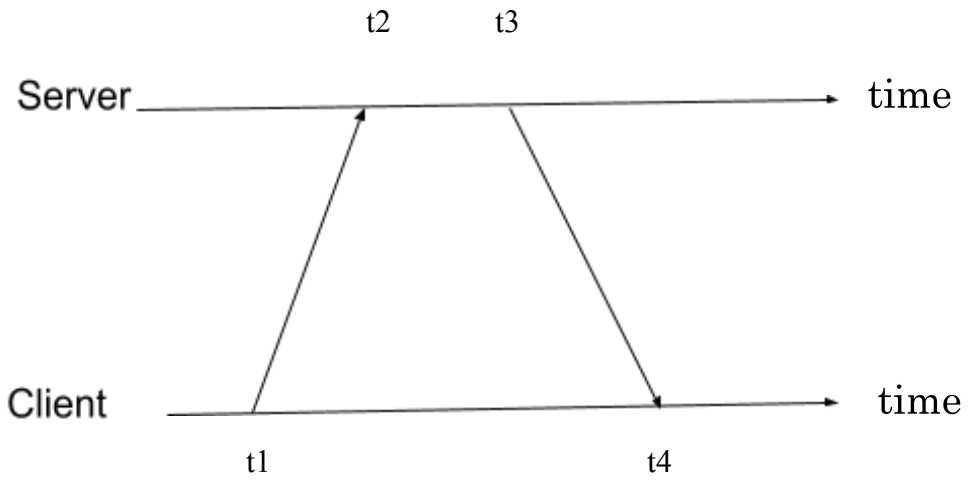


Figure 4.2.6.1 Time calibration via NTP

$$Round\ Trip\ Time\ (RTT) = t4 - t3 + t2 - t1$$
$$Clock\ Offset(O) = t2 - t1 - (RTT/2)$$

When the client(Beetle/laptop) first sends a packet, the timestamp is recorded at T1. At T2, the server(Ultra96) receives the packet and saves that timestamp. At T3, the server sends back the timestamp and the client receives it at T4. The Round Trip Time (RTT) is then calculated with the above equation.

From there, to account for the synchronization delay (sync delay), the plan was to record the start time for each dancer which was recorded by the time stamp, add their individual beetles' offsets to the respective start times, and then have the maximum of the 3 sums to minus the minimum of the 3 sums. The result was the sync delay that is sent to the evaluation server and dashboard server. Aside from that, the RTT for each laptop would have constantly updated due to the constantly changing positions of the dancers. Below is the formula that explains the general idea.

$$Start\ timestamps: T1, T2, T3$$
$$Start\ time\ with\ offset: T1 + O1, T2 + O2, T3 + O3$$
$$Synchronisation\ delay\ (sync\ delay) =$$
$$max(T1 + O1, T2 + O2, T3 + O3) - min(T1 + O1, T2 + O2, T3 + O3)$$

However, we had adopted a different approach in our final implementation. Instead of recording the timestamps on the Bluno Beetle which have different clock time, we ended up recording the timestamps on the laptops as explained in the Internal Communications section. Hence, we ended up using the laptop's Universal Time as a reference for the dancer's timestamp. This implementation allows us to do away with the offset calculation and only use the very last equation in order to calculate the sync delay. The entire sync delay calculation was therefore much more simplified. This can be seen in the code below:

```
def calculate_sync_delay(beetle1_time_ultra96, beetle2_time_ultra96, beetle3_time_ultra96):
    sync_delay = 850
    if beetle1_time_ultra96 is not None and beetle2_time_ultra96 is not None and beetle3_time_ultra96 is not Non
        sync_delay = max(beetle1_time_ultra96, beetle2_time_ultra96, beetle3_time_ultra96) \
                   - min(beetle1_time_ultra96, beetle2_time_ultra96, beetle3_time_ultra96)
    elif beetle1_time_ultra96 is None:
        sync_delay = max(beetle2_time_ultra96, beetle3_time_ultra96) - min(beetle2_time_ultra96, beetle3_time_ul
    elif beetle2_time_ultra96 is None:
        sync_delay = max(beetle1_time_ultra96, beetle3_time_ultra96) - min(beetle1_time_ultra96, beetle3_time_ul
    elif beetle3_time_ultra96 is None:
        sync_delay = max(beetle1_time_ultra96, beetle2_time_ultra96) - min(beetle1_time_ultra96, beetle2_time_ul
    return sync_delay
```

Figure 4.2.6.2 sync_delay code snippet

The code above follows the earlier equation, but if we are missing one of the time stamps, we will just take the maximum of the 2 timestamps to minus the minimum of the 2 timestamps in order to obtain the sync delay.

4.2.7 Libraries and APIs to be used

The libraries and relevant APIs used are listed in this table.

Library	**API**	**Function**
socket	socket.socket, socket.AF_INET, socket.SOCK_STREAM, socket.connect, socket.sendall, socket.recv	To facilitate TCP connection between laptops and Ultra96, Ultra96 and evaluation server by building server socket and multiple client sockets
threading	threading.Thread, threading.Event	To create initial server threads for the 3 different laptops that will be sending information simultaneously Help with the TCP connection, while also having only 1 thread in one process
time	time.time	To create timestamps in

		order to be able to calculate the synchronization delay.
twisted	twisted.internet, twisted.internet.protocol, twisted.protocols.basic	To handle the different sockets, and make maintaining the TCP connection easier by carrying out auto-reconnection when clients(laptops) disconnect
crypto	AES, random	To carry out the encryption of the messages to be sent from Ultra96 to evaluation server and dashboard server

Table 4.2.7.1: List of libraries and APIs used and the reasonings

5 Software Details

5.1 Software Machine Learning

5.1.1 Overview

The machine learning process is used to predict the dance move of the dancer based on the data gathered from the sensors. In our project, the problem that we have to overcome is in the form of a multiclass classification. The goal is to construct a function which, given a new data point, will correctly predict the class to which the new point belongs (Multiclass classification - MIT).

The diagram below shows the overall machine learning pipeline that we are using.

Figure 5.1.1.1: Machine Learning Workflow

The raw data are first collected from the sensors in the form of Gyro X, Y, Z and Accelerometer X, Y, Z. The data will then be segmented into different windows and we will then preprocess and engineer new features based on these windows. Next, the data will be normalised before splitting the data into the training and testing sets. Finally, the model will then be trained and validated. The hyperparameters are then tuned and evaluated using the test set. The model will then be deployed after we reach a satisfactory performance on our evaluation metrics.

5.1.2 Data Collection

Our system records sensor readings at a frequency of 20Hz, which means that 20 packets of readings will be received every second. This frequency was determined after numerous trials as 20Hz gave rather reliable results, having fewer bad packets.

To generate a robust model with lower probability of overfitting, we would require ample data to train our model on, thus our whole group will be dancing to generate data for the model. Each dancer will dance for 8000 data points, with a frequency of 20Hz, this would mean that each dancer will dance continuously for 400 seconds. We decided to do this continuously to account for fatigue during evaluation where the dancers would have to dance 33 dance moves. Thus to ensure our model is able to predict accurately even when the dancer is fatigued, we decided to record training dance data even when the dancers are fatigued. Furthermore to vary our dataset, we decided to dance at 4 different speeds, this allows our model to be able to predict dance moves even if they are executed at a faster or slower pace. Due to the restrictions and limitations, we are unable to gather more data from dancers outside of our group, however if the guideline permits, we would have gathered data from another 6 individuals outside our group to ensure that we have ample and a varied dataset.

In our initial design, we planned to have 2 sensors, one on the hand and the other on the chest, however, after testing it on our machine learning model, we observed that the additional chest sensor was not necessary as there was not much improvement in the performance of the machine learning model. Thus we decided to use a single sensor on the hand for the rest of our project.

5.1.3 Data Preprocessing & Segmentation

Data received will first be preprocessed, filtering the noise caused by involuntary movements during data collection. This is done through the use of Butterworth and Moving Average filters. (Lima et al, 2019)
Butterworth filters are a type of signal processing filter used to have a frequency response as flat as possible in the passband (Control system design guide, 2012). It has a flat frequency response in the passband and roll-offs towards zero in the stop band.
Moving Average filters are a type of low pass finite impulse response filter. It takes samples of an input and calculates the average to output a single point. It is simple yet effective in reducing random white noises while keeping the sharpest step response.(Smith, 1997)

Categorical variables will also be encoded using a OneHotEncoder as an ordinal relationship might be imposed when we use ordinal encoding. It removes the problem when an unnecessary relationship is introduced which might be captured by the machine learning model. A binary column is created for each dance move, 1 will be assigned to the column corresponding to that particular dance move and 0 otherwise.

Segmentation is the process of splitting the data into separate groups depending on their attributes or behavior. This process is usually used in unsupervised learning where the data is separated into their respective clusters. This is usually seen in K-means Clustering where each data is grouped based on their respective clusters.

In our project, segmentation is required as our model has to identify different dance moves by the dancers while they are dancing with multiple dance moves at a time. It is extremely inefficient if we have to run the model on every data we obtain which is computationally expensive as well. The most common method for data segmentation in Human Activity Recognition Tasks is the use of a sliding window approach (Myroniv et al., 2017). Continuous data that we get from the sensors are first segmented into different windows based on time intervals. There are 2 different types of window, static sized window and dynamically sized window. For static sized window, there are 2 algorithms that we can consider, with overlap and without overlap. In the dynamic window approach, the data is segmented into different window sizes according to specific features.

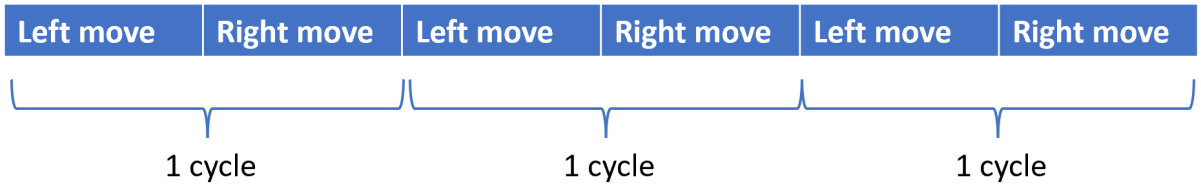


Figure 5.1.3.1: Visual representation of dance data

For our project, as each dance move takes approximately the same amount of time to execute (2 sec), we decide to use the static size window approach. 1 cycle in a dance move would include both the move for the left and for the right. Thus the window size will be the time taken to complete 1 cycle of a dance move as seen in Fig 5.1.3.1. A static sized window with overlap could provide more data for our model to train on, however this could cause our model to overfit as the same data would be used to train twice. We first attempted using a static sized window without overlap and this gave us excellent results as seen in our preliminary testing, thus we decided to stick with it for the rest of the project.

5.1.4 Feature Engineering & Selection

Our current data includes the 3 different axes from the gyroscope and accelerometer which might be insufficient to train a model that is able to generalize well. (Zhu et al., 2017) To develop a better model, our team decided to generate new features to explore and test on our model. There are both time domain features and frequency domain features that we can explore.

For time domain, we can estimate:

- Mean
- Variance
- Median
- Maximum
- Minimum
- Kurtosis
- Median Absolute Deviation
- Interquartile Range
- Correlation

For frequency domain, we can estimate:

- Spectral Energy
- Peak Frequency
- Entropy
- Fast Fourier Transform

Selection of features can be classified under a few methods, filter, wrapper, embedded and hybrid. For the purpose of our project, we will pick the filter methods which are computationally less expensive relative to the other methods. Filter methods include information gain, chi-square test, Fisher's score and correlation coefficient. (Feature selection techniques in machine learning, 2020)

In our final implementation, we decided upon these features:

- Minimum
- Maximum
- Mean
- Variance
- Median Absolute Deviation
- Kurtosis
- Interquartile Range
- Correlation
- Fast Fourier Transform

The feature Minimum refers to the minimum gyroscope X, Y, Z and accelerometer X, Y, Z values within the window of size 40.

The feature Maximum refers to the maximum gyroscope X, Y, Z and accelerometer X, Y, Z values within the window of size 40.

The feature Mean refers to the average gyroscope X, Y, Z and accelerometer X, Y, Z values within the window of size 40. The formula is shown below.

$$\mu = \frac{1}{N}\sum_{i=0}^{N-1} x_i \quad where\ N = window\ size$$

The feature Variance refers to the average gyroscope X, Y, Z and accelerometer X, Y, Z values within the window of size 40. The formula for variance is given below.

$$s^2 = \frac{1}{N-1}\sum_{i=0}^{N-1} (x_i - \mu)^2 \quad where\ N = window\ size$$

The feature Mean Absolute Deviation is the average distance between the data and the mean value. The formula is given below.

$$MAD = \frac{1}{N}\sum_{i=0}^{N-1} |x_i - \mu| \quad where\ N = window\ size$$

The feature Kurtosis describes the distribution of the data, measuring the extreme values in either tail. The formula for Kurtosis is given below.

$$Kurt = \frac{\mu_4}{\sigma^4} \quad where\ \mu_4 = fourth\ central\ moment$$

The feature Interquartile Range refers to the difference between the 25th and 75th percentile of the data in the window size.

The feature Correlation refers to the cross-correlations between Gyroscope X, Y, Z and Accelerometer X, Y, Z.

The feature Fast Fourier Transform refers to the fourier transform of the 6 original features, Gyroscope X, Y, Z and Accelerometer X, Y, Z which convert them to the frequency domain.

5.1.5 Data Normalization

Normalization is the process of ensuring each numeric column of our dataset is of a common scale, without affecting the difference in the range of values. This is essential to ensure that no particular column has a higher influence on the output before the model is trained. For our project, the accelerometer has a resting value of $10 m/s^2$ due to the Earth's gravitational acceleration, an unnormalized accelerometer value will have a high influence on the model due to its larger absolute value. The function used to normalize the data to a range of [0,1] is the following:

$$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

where $x = (x_1, x_2, x_3, \ldots, x_n)$

Figure 5.1.5.1: Normalization Formula

5.1.6 Models Exploration & Selection

We will be exploring both traditional machine learning models and deep learning models for our project. Deep learning models are able to train and classify without the need for feature engineering, however, one drawback that deep learning models have is the need for large amounts of data to train on. Traditional machine learning models on the other hand require sufficient features to train on even though it requires lesser data compared to deep learning models.

Traditional Machine Learning Models

Support Vector Machines

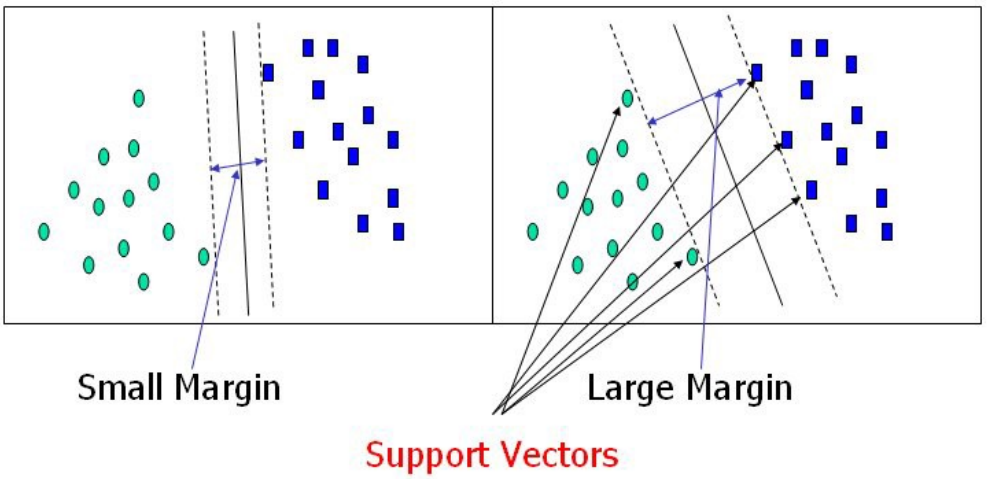


Figure 5.1.6.1: Hyperplanes of SVM (Gandhi, 2018)

Support Vector Machines (SVM) is a traditional machine learning model that can be used for classification and regression tasks. SVM classifies data by finding a hyperplane in an N-dimensional space that maximises the distance between the data points from the different classes, also known as margin. (Gandhi, 2018)

K-Nearest Neighbours

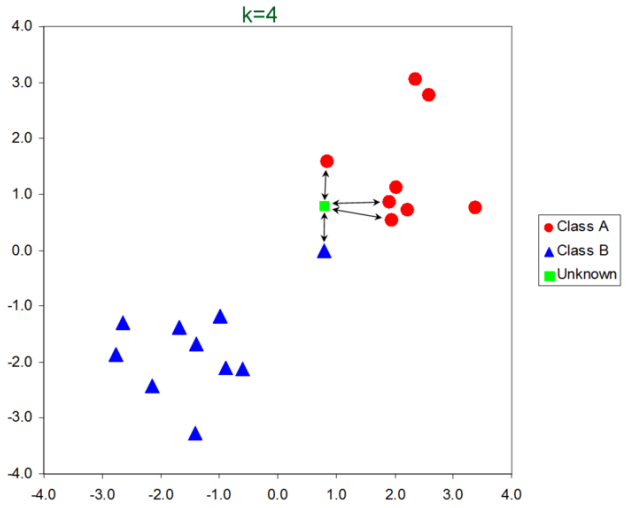


Figure 5.1.6.2: 2D Visual Representation of KNN (Peterson, 2019)

K-Nearest Neighbours (KNN) is a non-parametric model that is used for classification and regression. During training, the data will be represented in a multidimensional space with each data having a class label. When new data is tested on the model like the green square in the above figure, the resulting class of the new data point will be the majority class of its K nearest neighbours which is the red circle in the above example. Minkowski distance is used as the metric to calculate the distance between the new data point and its neighbours.

Deep Learning Models

Convolutional Neural Networks

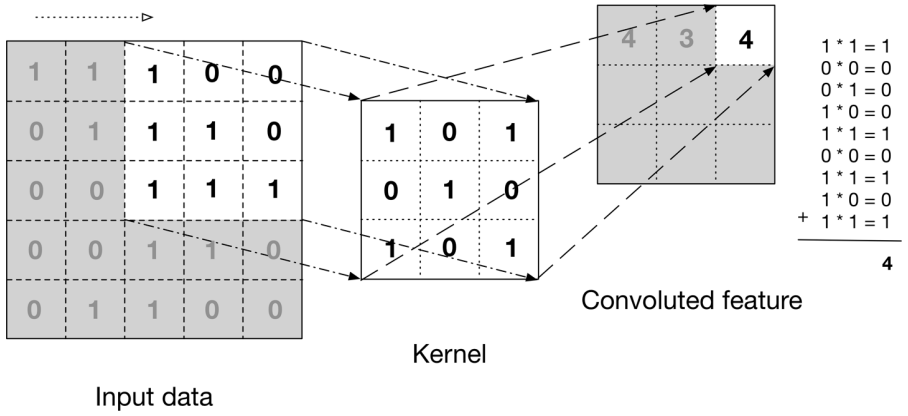


Figure 5.1.6.3: Example of a Convolution with a Filter Kernel (CNN for deep learning: Convolutional Neural Networks 2021)

Convolutional Neural Networks (CNN) are neural networks designed usually for image data. They consist of convolutional layers which are filter kernels used to capture spatial information and temporal dependencies of the data. The kernel will convolve with the input data which essentially is a matrix multiplication across the input. As the filter kernel is traversing the input data, the filter is applied in an overlapping way such that surrounding information of a particular data point is captured as well. The output feature map will then be put through a pooling layer where important features are extracted and the dimension of the feature map will be reduced. There are 3 different types of pooling, max pooling, sum

pooling and mean pooling. Max pooling takes the largest element from the specified filter size, sum pooling takes the sum of all the elements from the specified filter size and mean pooling takes the average. The output will then be flattened and fed to a fully connected neural network to output the class of the input.

Multi-Layer Perceptron

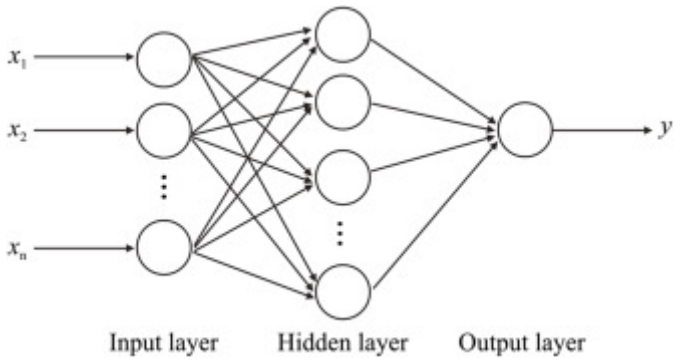


Figure 5.1.6.4: MLP Architecture (Fath et al., 2018)

Multi-Layer Perceptron (MLP) is a feed-forward neural network suitable for classification problems. MLP consists of an input layer, hidden layer and output layer. Inputs from the input layer are fed to the neurons in the hidden layer. Each neuron weighs the values from the previous layer and adds a bias which will then be summed and passed into an activation function to introduce non linearity. The neurons will continue to feed the output forward and the final output will then be compared to the expected output. This is done using a loss function like cross entropy loss. The error calculated will then be back-propagated through the network updating the weights and bias in the process. The objective is to minimise the loss function and this is done using an optimiser. Adam is currently the most popular optimiser, leveraging on an adaptive learning rate and momentum to update the weights and bias.

5.1.7 Model Training

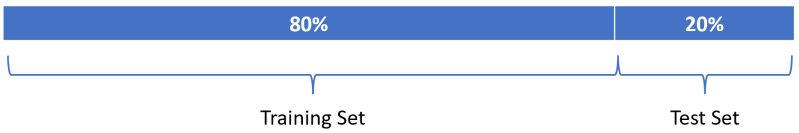


Figure 5.1.7.1: 80/20 Train Test Split

We will split our data into a training set and test set using the Pareto Principle also known as the 80/20 rule, 80% used for training and 20% used for testing. Furthermore, it is important that the training and testing sets are representative of the data set in general, thus instead of doing a random 80/20 split, we will use a stratified split based on the class labels.

5.1.8 Model Validation

There are a few validation approaches that we can explore, k-fold cross validation and leave one out cross validation.

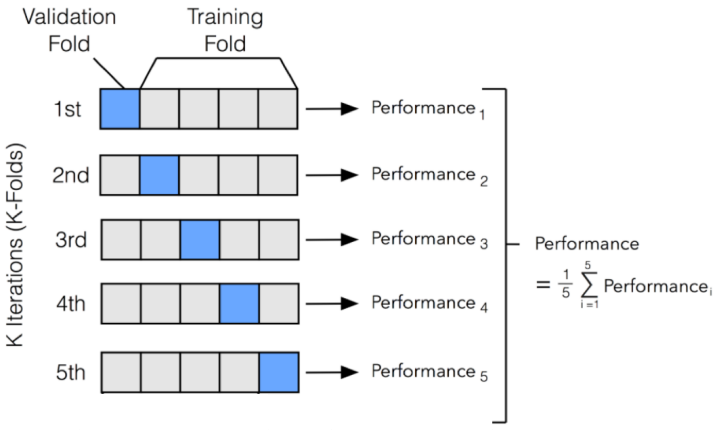


Figure 5.1.8.1: Example of 5-Fold Cross Validation (Importance of cross validation: Are Evaluation Metrics enough? 2021)

K-fold cross validation first splits the training data into k groups, each group will be used as the validation set while the other groups will be used to train the model. The mean of all k validation accuracies will then be obtained and used as the validation accuracy.

Leave one out cross validation (LOOCV) is a configuration of k-fold cross validation where k = number of examples in the training data set. Each example will be used as the validation set while the rest of the examples will be used to train the model. One benefit of using LOOCV is that the estimated performance of the model will be more robust, however this comes with a huge computational cost.

One metric to determine overfitting and underfitting in machine learning is the training validation loss graph. Training loss indicates how well the model is fitting the training data while the validation loss indicates how well the model fits new data.

Underfitting occurs when the algorithm is unable to model the training or new data, giving high error values that do not decrease over epochs. Therefore, there is high bias and the algorithm is unable to represent the complexity of the data. An example of a training validation loss graph that shows underfitting is the following.

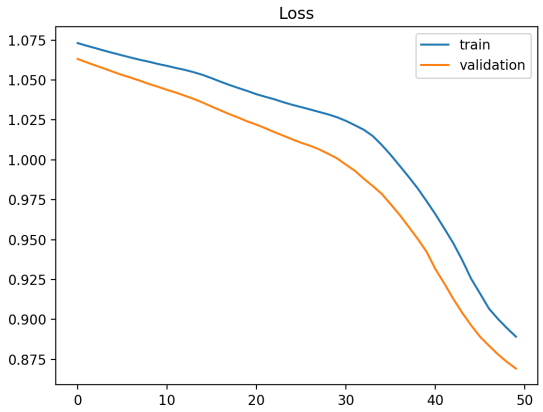


Figure 5.1.8.2: An Example Underfitted Graph (Brownlee, 2019)

Overfitting occurs when the algorithm represents the training data too well, and performs poorly on new data, resulting in poor generalization. The model is too complex and it is not

able to recognise simpler patterns in the data thus unable to perform well on new data. AN example of an overfitted training validation loss graph would be the following.

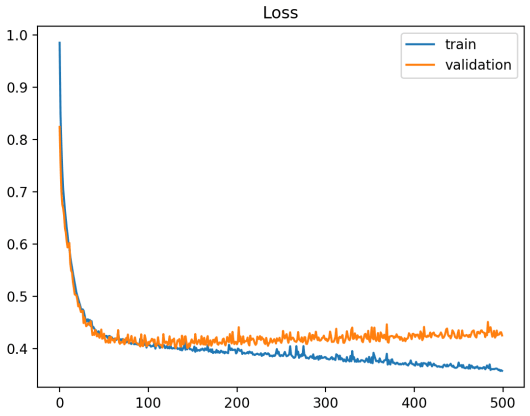


Figure 5.1.8.3: An Example Overfitted Graph (Brownlee, 2019)

A model with a good fit would need to have a balance between an overfitted and inderfitted model. The training and validation loss has to decrease to a point of stability with a minimal gap between the two final losses. Continued training of a well fitted model will lead to overfitting thus early stopping is used to avoid an overfitted model. An example of a well fitted model is shown below.

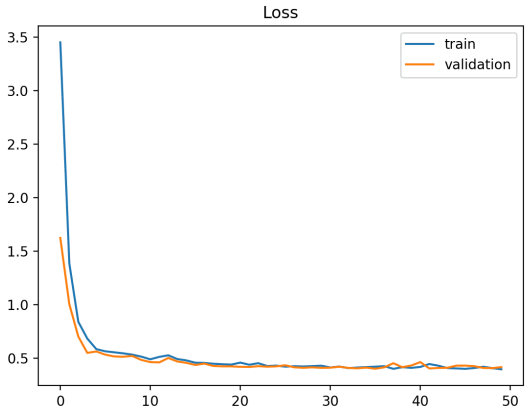


Figure 5.1.8.4: An Example Well Fitted Graph (Brownlee, 2019)

5.1.9 Model Testing

5.1.9.1 Evaluation Metrics

In classification problems we first have to understand what a confusion matrix is. A confusion matrix is a N x N matrix, where N is the number of classes being predicted. A simple example would be a binary classification task as shown below.

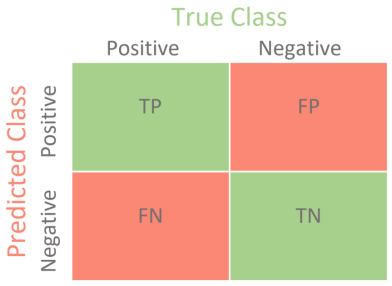


Figure 5.1.9.1.1: Binary Class Confusion Matrix (Mohajon, 2021)

There are 4 different metric for every prediction, True Positive (TP), False Positive (FP), False Negative (FN) and True Negative (TN).
True Positive refers to the number of predictions where the classifier correctly predicts the positive class as positive.
True Negative refers to the number of predictions where the classifier correctly predicts the negative class as negative.
False Positive refers to the number of predictions where the classifier incorrectly predicts the negative class as positive.
False Negative refers to the number of predictions where the classifier incorrectly predicts the positive class as negative.

Apart from these metrics, we can also determine the Precision, Recall, Specificity and Accuracy of the model by using the following formulas:

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{Specificity} = \frac{TN}{TN + FP}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + FN + TN + FP}$$

Precision tells us the fraction of predictions that are positive are actually positive.
Recall tells us the fraction of all positives that were predicted as positive.
Specificity tells us the fraction of all negatives that were predicted as negative.
Accuracy gives us the fraction of total samples that were correctly predicted by the model.

In our project, accuracy is important as we want a high number of samples to be correctly predicted. Moreover, both precision and recall are also important, thus to evaluate our model we decided to use accuracy and the F1-score as it is a harmonic mean between precision and recall.

$$F_1 = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} = \frac{\mathrm{TP}}{\mathrm{TP} + \frac{1}{2}(\mathrm{FP} + \mathrm{FN})}$$

Figure 5.1.9.1.2: F1 Score Formula

When a particular dance move A is being performed and the dance move B is predicted, this will show a case of False Negative (FN). When another dance move C is being performed and it gets predicted as dance move A, this represents a case of False Positive (FP). For both cases of dance move detection and location detection, we would want to minimise the number of FP and FN and the F1 score is able to represent this as seen in the formula above. As FP and FN decrease, the overall F1 score will increase.

We will first test our model on a sample data set found from kaggle. Based on the results from our preliminary testing, we will then choose models to focus on and use for our project.

5.1.9.2 Preliminary Testing

To perform preliminary model testing for our project, we used a Human Activity Recognition dataset from Kaggle (Learning, U. C. I. M. Human activity recognition with smartphones. Kaggle. 2019)
to train and test our models on. The data set was built from 30 participants performing daily living activities with an inertial sensor embedded smart phone mounted on their waist. The sensor is able to capture 3-axial linear acceleration and velocity at a rate of 50Hz. The data were then labelled manually. The collected sensor signals were pre-processed by applying noise filters such as the Butterworth low pass-filter of 0.3Hz cut-off frequency was used to separate body acceleration and gravity. The signal was then sampled using a static sized sliding window of 2.56 sec with no overlap.

	tBodyAcc-mean()-X	tBodyAcc-mean()-Y	tBodyAcc-mean()-Z	tBodyAcc-std()-X	tBodyAcc-std()-Y	tBodyAcc-std()-Z	tBodyAcc-mad()-X	tBodyAcc-mad()-Y	tBodyAcc-mad()-Z	tBodyAcc-max()-X	...	fBodyBodyGyroJerkMag-meanFreq()
0	0.288585	-0.020294	-0.132905	-0.995279	-0.983111	-0.913526	-0.995112	-0.983185	-0.923527	-0.934724	...	-0.074323
1	0.278419	-0.016411	-0.123520	-0.998245	-0.975300	-0.960322	-0.998807	-0.974914	-0.957686	-0.943068	...	0.158075
2	0.279653	-0.019467	-0.113462	-0.995380	-0.967187	-0.978944	-0.996520	-0.963668	-0.977469	-0.938692	...	0.414503
3	0.279174	-0.026201	-0.123283	-0.996091	-0.983403	-0.990675	-0.997099	-0.982750	-0.989302	-0.938692	...	0.404573
4	0.276629	-0.016570	-0.115362	-0.998139	-0.980817	-0.990482	-0.998321	-0.979672	-0.990441	-0.942469	...	0.087753

5 rows × 561 columns

Figure 5.1.9.2.1: First 5 Rows of Human Activity Recognition Dataset

Each record in the dataset represents an activity performed during a 2.56 sec time interval. The features for our dataset are:

- Triaxial acceleration from the accelerometer and estimated body acceleration.
- Triaxial Angular velocity from gyroscope.
- A 561-feature vector with time and frequency domain variables.
- Activity label.

Hyperparameter Tuning

The main hyperparameter tuning algorithm that we adopted is the Hyperband method. It is a variation of Random Search with the use of early stoppage and adaptive resource allocation. It approaches hyperparameter optimization as a pure-exploration non stochastic infinite-armed bandit problem.

```
from keras.models import Sequential
from keras.layers import Dense,Dropout
import keras_tuner as kt
from tensorflow import keras
import hyperopt as hp

def model_builder(hp):
    model=Sequential()
    model.add(Dense(units=hp.Int('units', min_value=32, max_value=512, step=32),
                    kernel_initializer='uniform',activation='relu',input_dim=X_train.shape[1]))

    model.add(Dense(units=hp.Int('units', min_value=32, max_value=512, step=32),
                    kernel_initializer='uniform',activation='relu'))

    model.add(Dense(units=hp.Int('units', min_value=32, max_value=512, step=32),
                    kernel_initializer='uniform',activation='relu'))

    model.add(Dense(units=6,kernel_initializer='uniform',activation='softmax'))

    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

    model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp_learning_rate),
                  loss='categorical_crossentropy',metrics=['accuracy'])

    return model

tuner = kt.Hyperband(model_builder,
                     objective='val_accuracy',
                     max_epochs=10,
                     factor=3)
stop_early = keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
tuner.search(X_train, Y_train, epochs=50, validation_split=0.2, callbacks=[stop_early])
```

Figure 5.1.9.2.2: Hyperband Tuning Hyperparameters for MLP

```
The hyperparameter search is complete.
The optimal number of units in the first densely-connected
layer is 64
The optimal number of units in the second densely-connected
layer is 32
The optimal number of units in the third densely-connected
layer is 512
The optimal learning rate for the optimizer
is 0.001.
```

Figure 5.1.9.2.3: Hyperparameters for MLP

Results for SVM

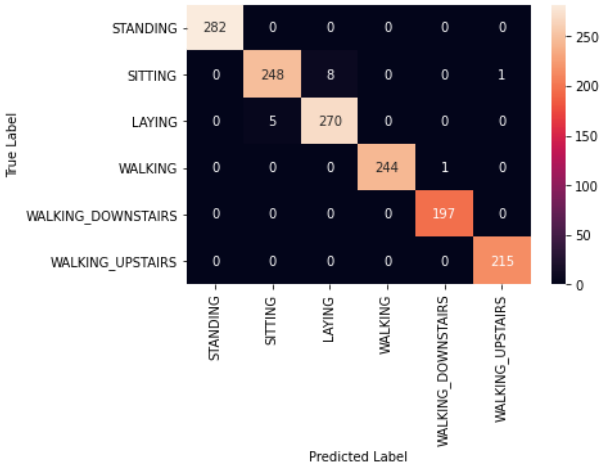


Figure 5.1.9.2.4: Confusion Matrix of SVM

	precision	recall	f1-score	support
LAYING	1.00	1.00	1.00	282
SITTING	0.98	0.96	0.97	257
STANDING	0.97	0.98	0.98	275
WALKING	1.00	1.00	1.00	245
WALKING_DOWNSTAIRS	0.99	1.00	1.00	197
WALKING_UPSTAIRS	1.00	1.00	1.00	215
accuracy			0.99	1471
macro avg	0.99	0.99	0.99	1471
weighted avg	0.99	0.99	0.99	1471

Figure 5.1.9.2.5: Classification Report for SVM

Results for KNN

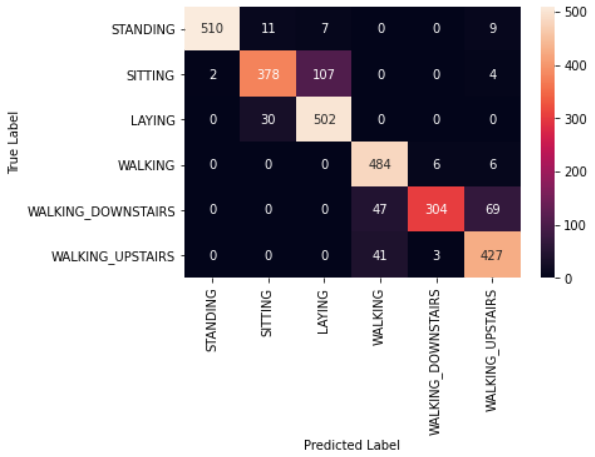


Figure 5.1.9.2.6: Confusion Matrix of KNN

	precision	recall	f1-score	support
LAYING	1.00	0.95	0.97	537
SITTING	0.90	0.77	0.83	491
STANDING	0.81	0.94	0.87	532
WALKING	0.85	0.98	0.91	496
WALKING_DOWNSTAIRS	0.97	0.72	0.83	420
WALKING_UPSTAIRS	0.83	0.91	0.87	471
accuracy			0.88	2947
macro avg	0.89	0.88	0.88	2947
weighted avg	0.89	0.88	0.88	2947

Figure 5.1.9.2.7: Classification Report for KNN

Results for MLP

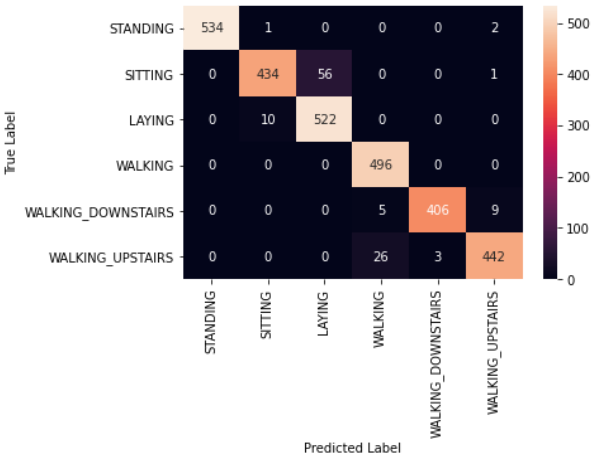


Figure 5.1.9.2.8: Confusion Matrix of MLP

	precision	recall	f1-score	support
LAYING	1.00	0.99	1.00	537
SITTING	0.98	0.88	0.93	491
STANDING	0.90	0.98	0.94	532
WALKING	0.94	1.00	0.97	496
WALKING_DOWNSTAIRS	0.99	0.97	0.98	420
WALKING_UPSTAIRS	0.97	0.94	0.96	471
accuracy			0.96	2947
macro avg	0.96	0.96	0.96	2947
weighted avg	0.96	0.96	0.96	2947

Figure 5.1.9.2.9: Classification Report for MLP

Results for CNN

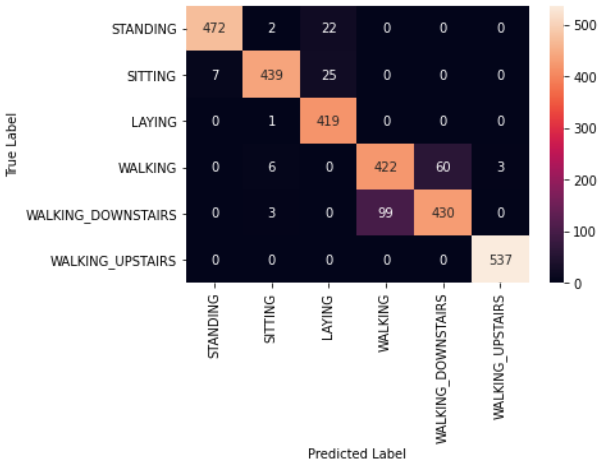


Figure 5.1.9.2.10: Confusion Matrix of CNN

	precision	recall	f1-score	support
LAYING	0.99	0.95	0.97	496
SITTING	0.97	0.93	0.95	471
STANDING	0.90	1.00	0.95	420
WALKING	0.81	0.86	0.83	491
WALKING_DOWNSTAIRS	0.88	0.81	0.84	532
WALKING_UPSTAIRS	0.99	1.00	1.00	537
accuracy			0.92	2947
macro avg	0.92	0.92	0.92	2947
weighted avg	0.92	0.92	0.92	2947

Figure 5.1.9.2.11: Classification Report for CNN

From our preliminary testing results, we can see that of the 4 models that were tested, SVM and MLP gave the highest f1-score. Therefore, the models that we choose to focus on for our practical testing would be the above-mentioned models.

5.1.9.3 Practical Training & Testing

This section shows the results from our test using real time dance data that we recorded ourselves. 3 dancers were picked from our group to dance and test the model that we implemented.

SVM

For our Support Vector Classifier we used the sk-learn svm library. Hyperparameters were tuned using RandomisedSearch where a fixed number of parameters are sampled from a specified distribution.

Our tuned parameters are as follow:
Best C: 1000
Best gamma: 0.0001
Best kernel: rbf

Results:

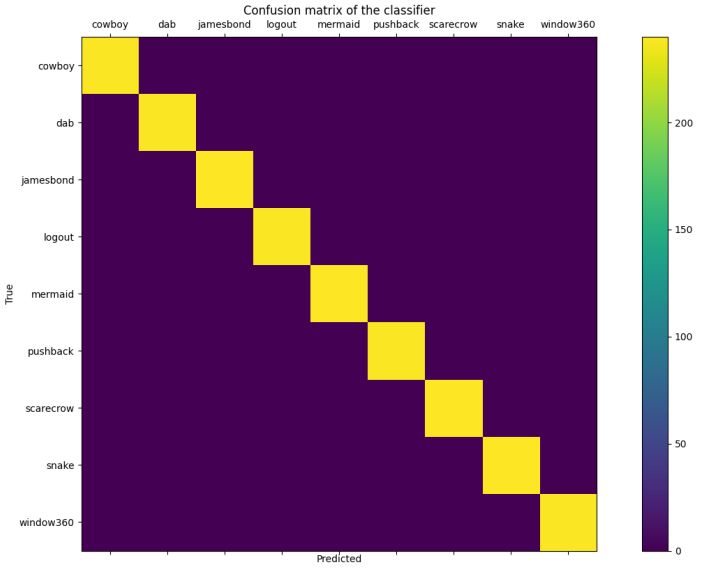


Figure 5.1.9.3.1: Confusion Matrix of SVM

	precision	recall	f1-score	support
cowboy	1.00	1.00	1.00	239
dab	1.00	1.00	1.00	239
jamesbond	1.00	1.00	1.00	240
logout	1.00	1.00	1.00	239
mermaid	1.00	0.99	1.00	239
pushback	1.00	1.00	1.00	240
scarecrow	1.00	1.00	1.00	239
snake	0.99	1.00	0.99	240
window360	1.00	1.00	1.00	240
accuracy			1.00	2155
macro avg	1.00	1.00	1.00	2155
weighted avg	1.00	1.00	1.00	2155

Figure 5.1.9.3.2: Classification Report for SVM

MLP

The framework we chose to use was PyTorch instead of Tensorflow due to compatibility issues with the Ultra96 architecture. Our neural network uses Relu as the activation function and it is a simple 2 layer network with 8 neurons in the dense layer. Early stopping of Epoch 15 was used as a form of regularisation to ensure there the model does not overfit.

```
class NNModel(nn.Module):
    def __init__(self, num_units=8, nonlin=nn.ReLU()):
        super(NNModel, self).__init__()

        self.dense0 = nn.Linear(config.input_size, num_units)
        self.nonlin = nonlin
        self.output = nn.Linear(num_units, config.output_size)

    def forward(self, X, **kwargs):
        X = self.nonlin(self.dense0(X))
        X = self.output(X)
        return X
```

Figure 5.1.9.3.3: MLP Architecture

Results:

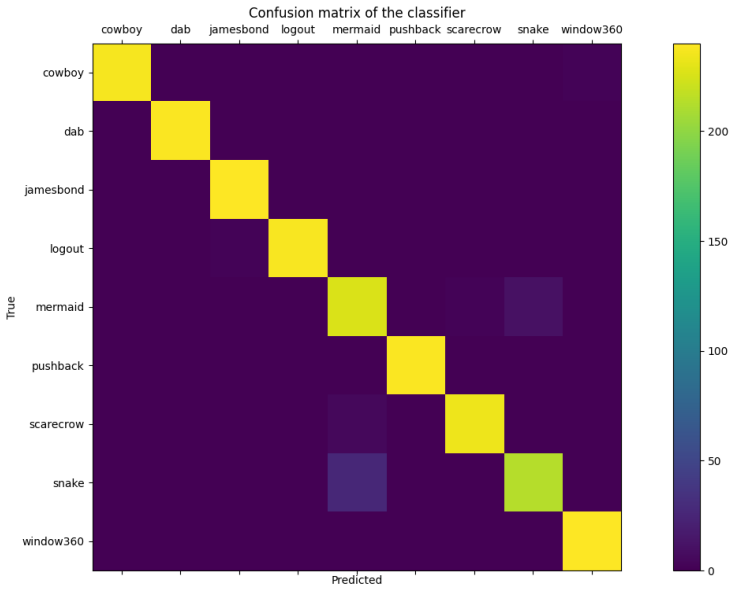


Figure 5.1.9.3.4: Confusion Matrix of MLP

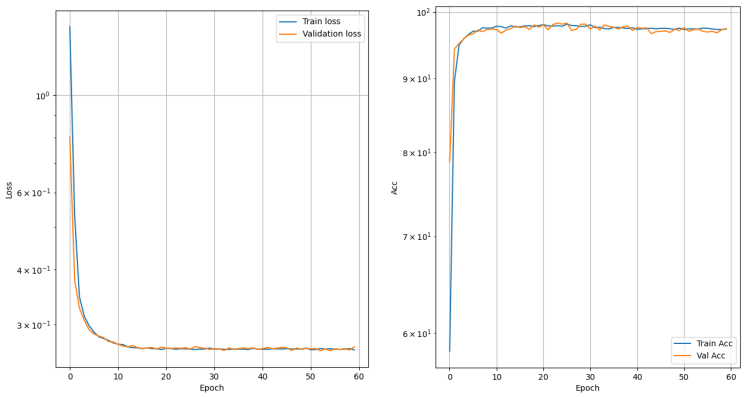


Figure 5.1.9.3.5: Training Validation Loss and Accuracy Graph for MLP

	precision	recall	f1-score	support
cowboy	1.00	1.00	1.00	239
dab	1.00	1.00	1.00	240
jamesbond	1.00	1.00	1.00	240
logout	1.00	1.00	1.00	240
mermaid	0.87	0.98	0.92	240
pushback	1.00	1.00	1.00	239
scarecrow	0.99	1.00	0.99	239
snake	0.99	0.85	0.91	239
window360	1.00	1.00	1.00	239
accuracy			0.98	2155
macro avg	0.98	0.98	0.98	2155
weighted avg	0.98	0.98	0.98	2155

Figure 5.1.9.3.6: Classification Report for MLP

Although from the above results, we can see that the SVM model has a slightly better overall F1 score and a higher accuracy compared to the MLP model, our team decided to use the MLP model as the processing speed for the SVM model was significantly slower compared to the MLP model. It is especially crucial for our model to be fast as we are required to make a prediction in real time, therefore, MLP was the chosen model for our project.

5.2 Software Dashboard

This section gives an overview of the software dashboard that will be used by both coaches and trainers. The following subsections will detail (1) the proposed tech stack, (2) mockups of the dashboard design, and(3) a survey for user feedback gathering.

5.2.1 Tech Stack

The software dashboard will utilise the MERN stack - MongoDB, ExpressJS, ReactJS and NodeJS. MERN is a JavaScript stack used for deployments of full-stack web applications, providing an end-to-end framework that covers the backend to the frontend.

The MERN stack requires the use of only 2 languages in its entire development cycle - JavaScript and JSON (JavaScript Object Notation). This allows the data to flow naturally front to back, making it fast to build on and reasonably simple to debug. Additionally, from the standpoint of a developer, it only requires us to be proficient with a single programming language (JavaScript) and the JSON document structure, to be able to understand and develop the system (MongoDB, n.d).

Additionally, The MERN stack is an open-source framework with an active community and excellent developer support online, providing us developers with as many guides and documentations that we need. This will definitely ease and speed up the development cycle.

Database: MongoDB

When choosing a database, the backbone of any full stack web application, our biggest consideration was the ease of integration with the other components of the tech stack and the Ultra96.

MongoDB is a popular document-oriented NoSQL database. Unlike traditional databases, MongoDB represents its information in a schemaless series of JSON-like documents (actually stored as binary JSON, or BSON), as opposed to the table and row format of relational systems such as PostgreSQL or MySQL. The rigidity of having database schemas therefore makes MongoDB easier and more flexible to build with full-stack applications. In an environment with a large flow of data inflow and outflow, it would be beneficial for the database to be able to easily serve semi-structured, or structured data. This will provide the application with room for any potential errors (MongoDB, n.d).

Having to receive the data from the Ultra96, it is wise to use a database that can provide us with an array of drivers, for different programming languages. This will give the Ultra96 developer flexibility when it comes to calling the MongoDB service. In our case, PyMongo provides us with exactly that.

For this application, we will be running MongoDB locally on a separate laptop. In the initial design, we suggested using MongoDB's cloud database service, MongoDB Atlas. which handles all the complexity of deploying, managing, and healing our deployments on a cloud service provider of our choice. However, we faced a big issue of the Ultra96 not being able to communicate with the cloud database due to NUS's network firewall (despite including whitelisting in the

database configurations). As such, the database was set up locally, on 3 different ports to allow for the use of Replica Sets to aid in real time streaming.

Backend Framework: ExpressJS

ExpressJS is a minimal web application framework that works in tandem with NodeJS. It provides us with a myriad of HTTP utility methods and middleware, to ease the creation of APIs. This removes the need for developers to write their own code to build routing components which tend to be time-consuming and tedious. This helps us cut down the coding time and reduce the effort needed, especially important in this capstone project (StrongLoop, n.d).

As mentioned, being a framework of NodeJS, ExpressJS is also highly extensible and can be easily modified or expanded if required. It is also able to manage the efficient handling of concurrent requests, by extension of NodeJS's ability to process multiple requests simultaneously. This will definitely provide us developers with the flexibility and efficiency needed in a real-time dashboard.

Frontend Library: ReactJS

Deciding on a framework or library to meet our front-end needs was also a crucial step in our development process. We wanted a tool that can help us create interactive User Interfaces, with ease.

ReactJS is a JavaScript library which allows that, by being component-based. In essence, it gives the developers the ability to build encapsulated components that are entirely reusable. Similar to the use of ExpressJS, this could also cut down the development process since developers are able to use 'templates' instead of re-writing the same lines of code (Facebook Inc, n.d).

In our use case of being a real-time application, that has to constantly update the data on the interface, it is an advantage that ReactJS implements a Virtual Document Object Model (DOM) system. A Virtual DOM system creates a lightweight copy of the actual DOM of the browser, which can be easily manipulated and changed as compared to Real DOM systems in frameworks such as AngularJS and VueJS. Although the Real DOM will eventually be updated as well, by using a Virtual DOM Tree, ReactJS is able to decide the best ways to do so. This allows only the updated elements of the UI to be re-rendered, instead of the entire DOM, which helps with efficiency.

From the perspective of a user, this makes the application run much smoother whenever there is a state change. In our use case, an example of a state change is when we receive a new data packet and have to update the information shown on the dashboard. This helps to reduce any delay when re-rendering the data.

Runtime Environment: NodeJS

After considering all the other aspects of the stack, it would make sense for NodeJS to be our runtime environment of choice. NodeJS allows for the "JavaScript Everywhere" paradigm, allowing for the unification of all the aforementioned components around a single programming language (OpenJS Foundation, n.d).

Another big advantage of using NodeJS in our stack is its ability to run in a single process, without creating a new thread for every request. In short, it is able to handle thousands of concurrent connections without needing to block a thread of waste CPU cycles waiting. This feature of NodeJS makes it perfect in a real-time setting as data can be processed quicker when transitioning between the client and server.

The Node Package Manager (npm), the default packet manager for NodeJS, provides developers with a database of millions of public packages, free for use. This allows for extensibility of the other components in the stack such as being able to install libraries that can add pre-written components into our ReactJS frontend or install Mongoose which helps to ease the integration of MongoDB in the tech stack. npm is a wonderful tool that will help us scale and improve the application easily, with minimal time and effort.

Summary

The figure 5.2.1.1 below summarizes where each component of the tech stack is positioned relative to each other.

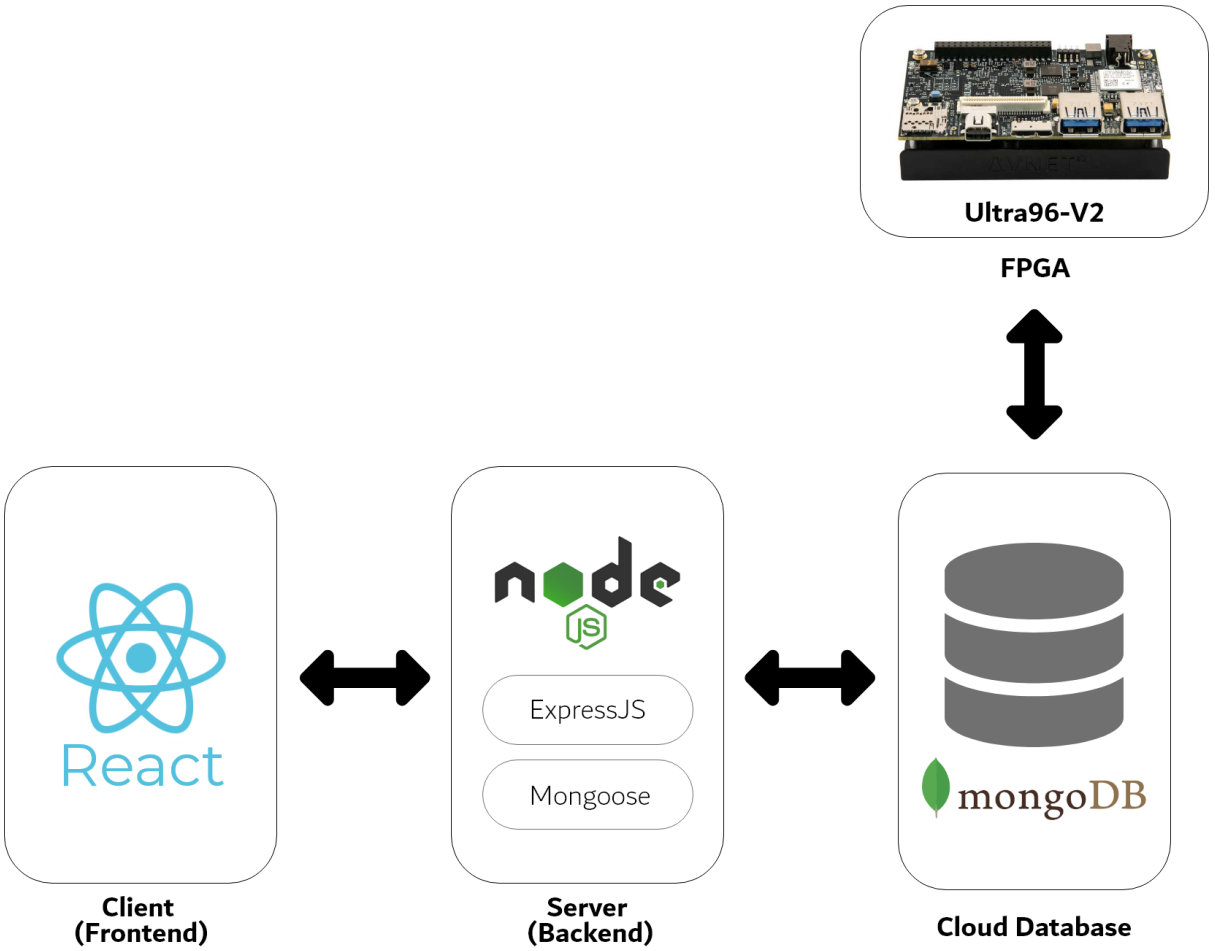


Figure 5.2.1.1: Diagram of the tech stack

5.2.2 Dashboard Design

5.2.2.1 Dashboard Mockups

Login/Registration Screen

As briefly discussed in Section 1, the dashboard will comprise of 2 sections, (1) the login and (2) the actual dashboard for use during dance lessons.

The login section will consist of a login screen and a sign up screen. As seen from Figure 5.2.2.1.1, the login will allow users to access their accounts and corresponding features. However, if they have yet to have their own personal accounts, the sign up screen as shown in Figure 5.2.2.1.2 will allow users to do just that.

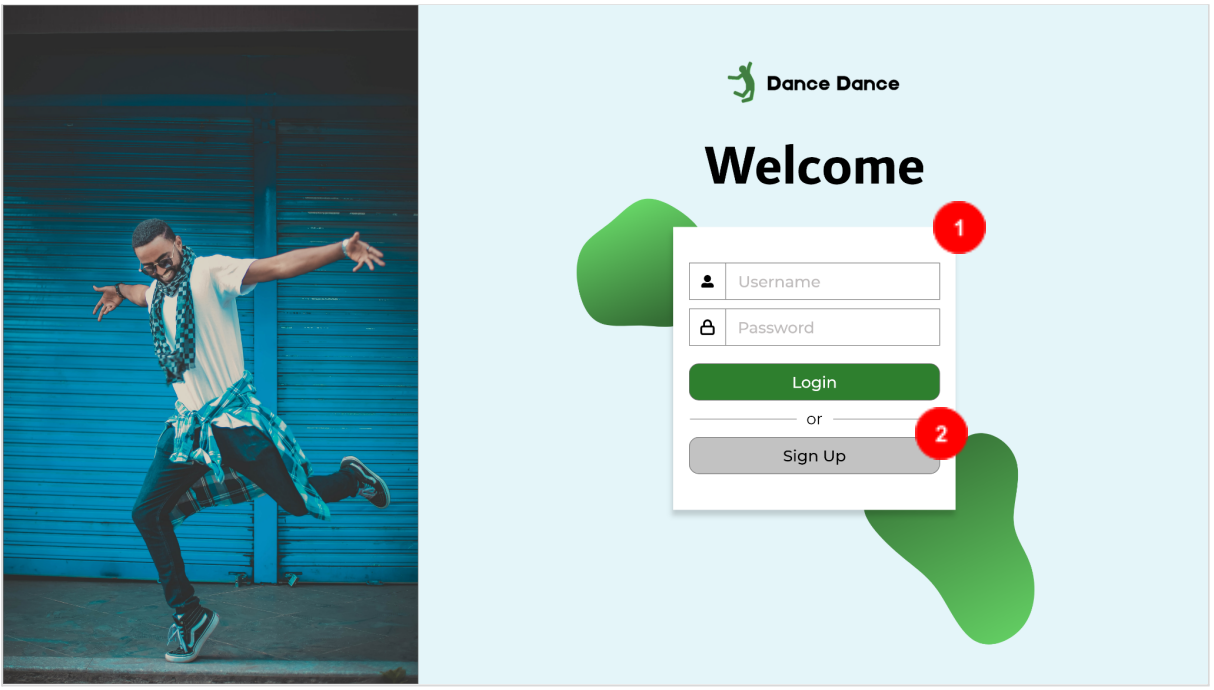


Figure 5.2.2.1.1: Mockup of the login screen for existing users

1	The login form component which requires the username and password before a user can access the dashboard
2	The Sign Up button which brings a user to the registration page in the event that a user does not already have a user account

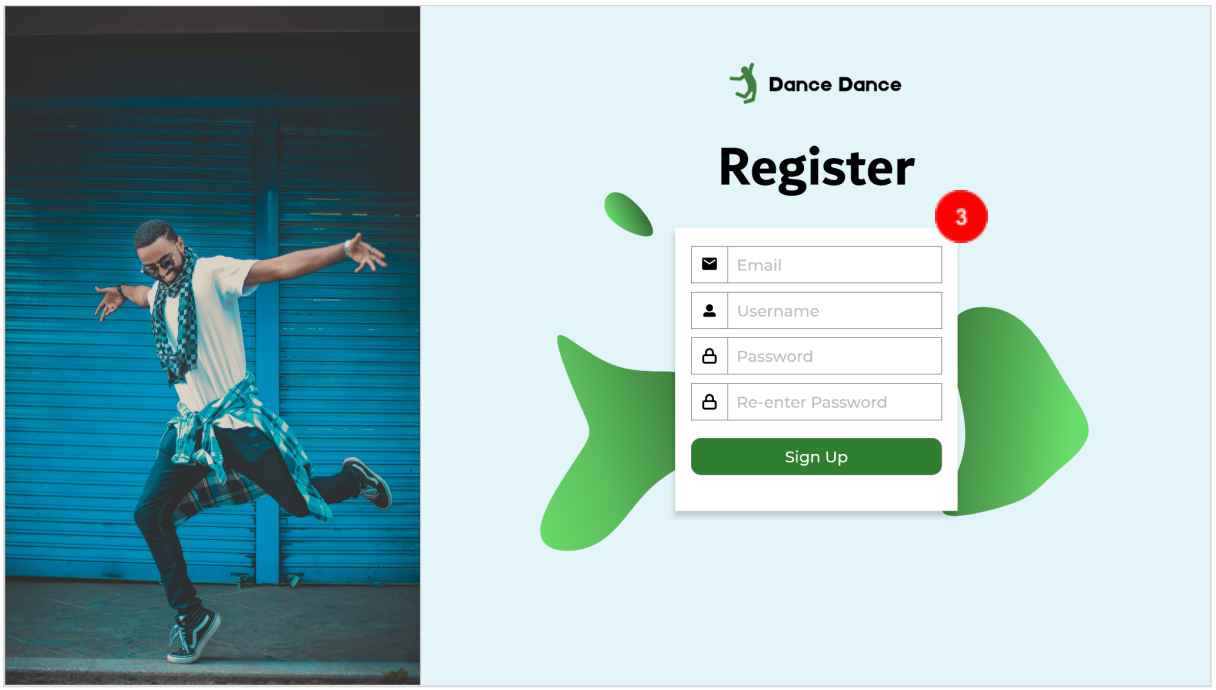


Figure 5.2.2.1.2: Mockup of the sign up screen for existing users

3	The registration form component which requires the user's email, username and password to create a new account.

Moving forward to the main application, it comprises 2 main tabs - (1) dashboard and (2) history of past data collected.

Dashboard Screen

The real-time data tab aims to provide an overview of the dancer's performance by displaying the raw data from sensors and by visualising the data through line graphs, as shown in Figure 5.2.2.1.3.

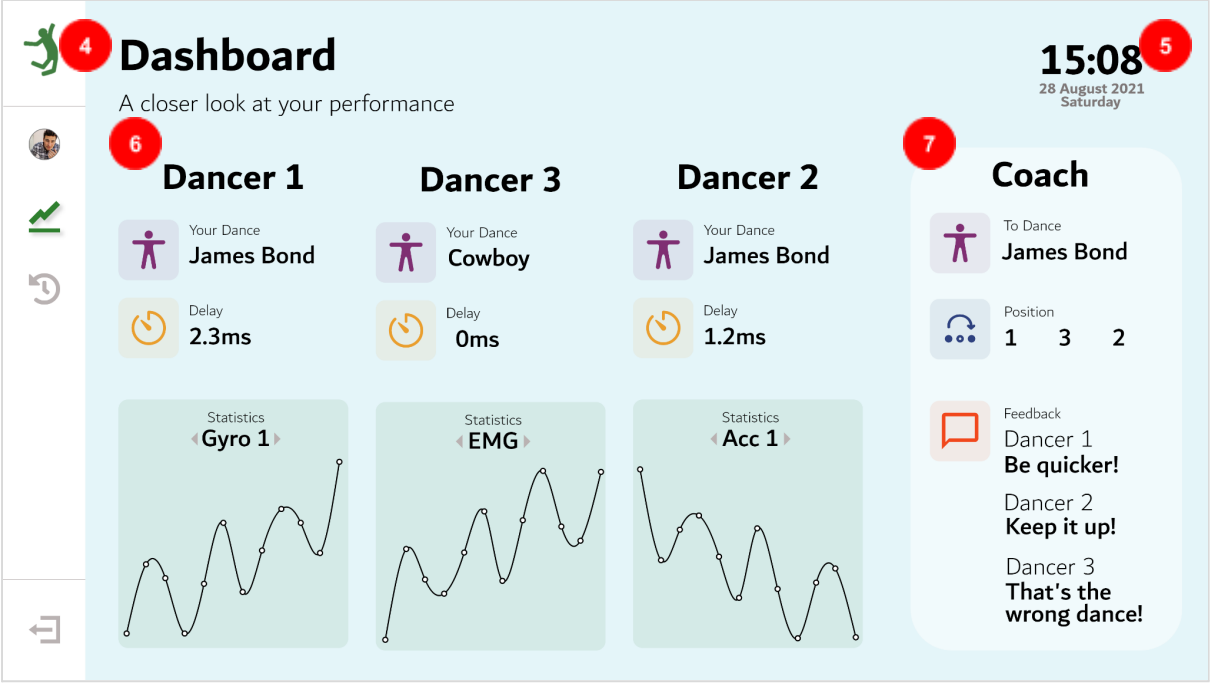


Figure 5.2.2.1.3: Mockup of the dashboard screen

4	The navigation bar component which is always present at the side of the screen. The navigation bar includes 3 clickable buttons that bring the user to the Dashboard Screen, History Screen or to Logout respectively.
5	The current time and date.
6	The dancer components which show a summary of a dancer's performance. It includes the dance the dancer is currently performing and the synchrony delay. It also includes a line graph where users can easily choose between the type of data to visualize - Gyro 1, Acc 1, Gyro 2, Acc 2 or EMD. It is important to note that the order of the 3 components will shift around accordingly when the corresponding dancers shift positions (Left to right on in real life is represented by left to right on dashboard)
7	The coach component which shows the move that the dancers are expected to perform, including the relative positions of the dancer. It also includes simple feedback that is catered for each individual dancer depending on their performance. This feature is to be implemented in the later stages of the project.

History Screen

The history data aims to allow users to take a look at the data from past dances as shown in Figure 5.2.2.1.4.

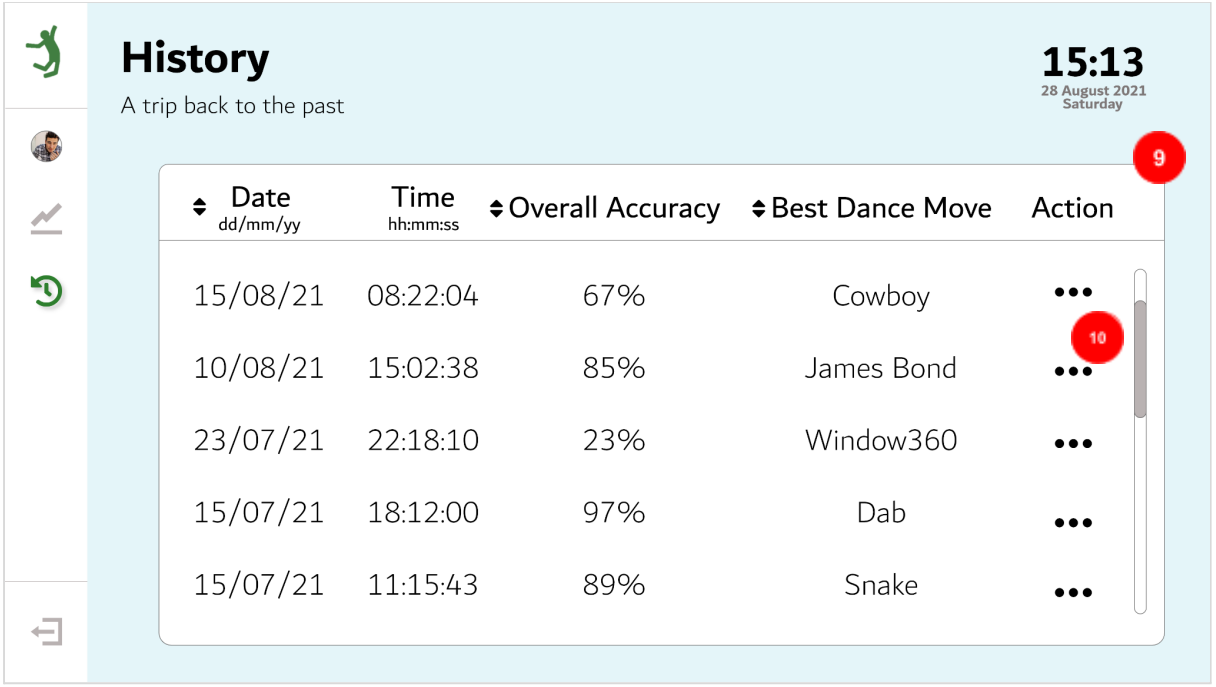


Figure 5.2.2.1.4: Mockup of the History screen

9	The table component which showcases the summary of past dances, easily identified by the data and time. The table can be easily traversed through easy sorting of the respective columns.
10	The action button will lead the user to a modal which will provide users with a more in depth look at the past data. This modal will include the graphs of the accelerometer and gyroscope captured during the dance.

5.2.2.2 Final Dashboard Implementation

Although the final implementation of the frontend design remains largely similar, we had to make some minor changes to accommodate the needs of the users.

Login/Registration Screen

As seen from Figures 5.2.2.2.1 and 5.2.2.2.2, the login and registration pages had very minimal changes. The only notable change worth mentioning, is the removal of needing a username when registering. As such, when a user signs in, he/she will sign in by simply using the email instead. The biggest reason as to why we removed the need for usernames when registering is that the emails are already going to be unique and can be used for identification of the users. Hence, we consider it to be redundant.

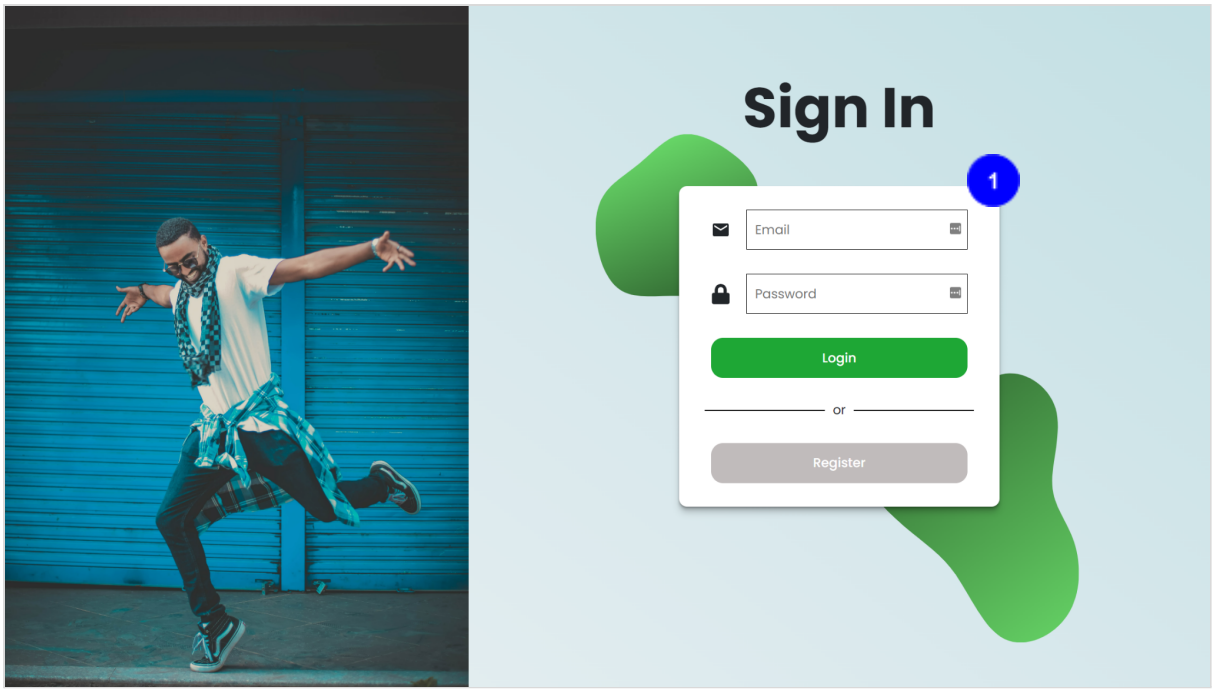


Figure 5.2.2.2.1: Final implementation of login page

Changes Made
1 The sign in page which previously required a username and password to login, now requires an email and password instead.

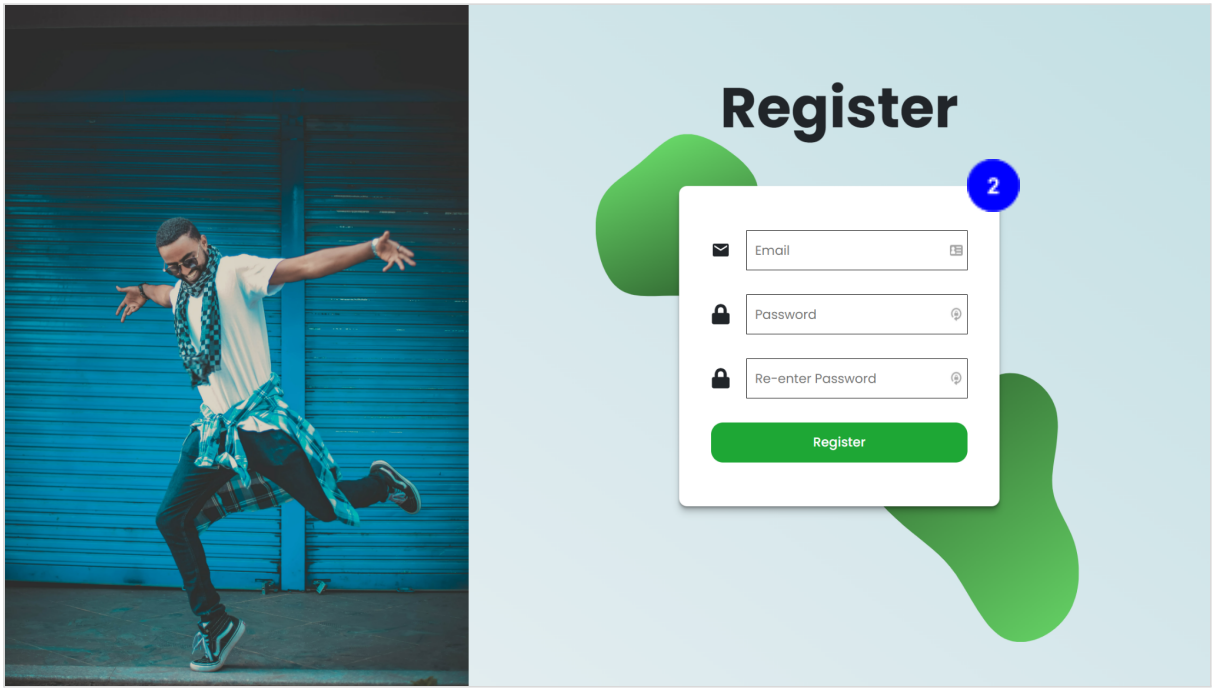


Figure 5.2.2.2.2: Final implementation of registration page

Changes Made
2 The registration page which previously required a username to register, now only requires an email for identification, with the reason discussed above.

Moving forward to the main application, apart from the originally suggested tabs - dashboard and history screens, we also implemented a new screen, the tutorial screen

Dashboard Screen

There were many changes that were made to the design of the dashboard which streams real time data to the users. The final design implementation of the dashboard can be seen in figure 5.2.2.2.3. The reasons for the individual changes will be discussed in the table below the figure.

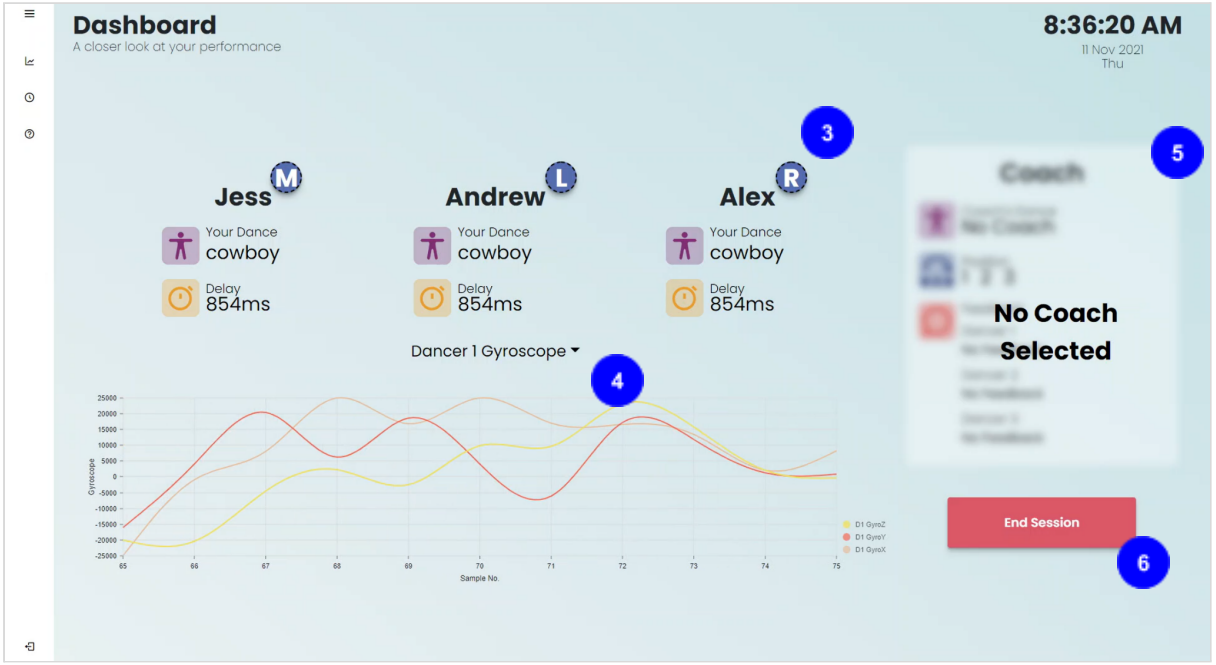


Figure 5.2.2.2.3: Final implementation of Dashboard Screen

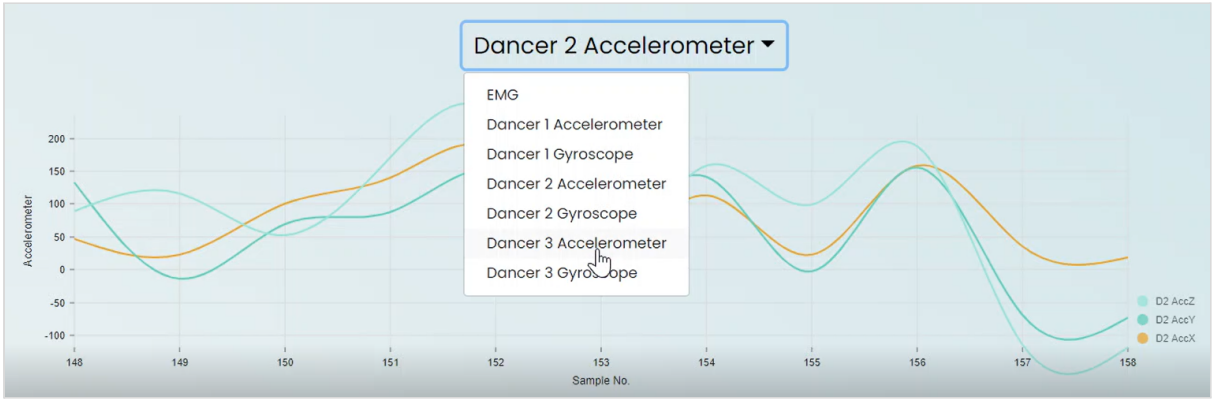


Figure 5.2.2.2.4: Closeup of graph dropdown

Changes Made	
3	The original design to show the dancer position was to dynamically move the positions of these cards according to the position. However, we feel that it may be too distracting to have moving components on the web page and may even contribute to latency on the frontend.
3	Aside from the key information tagged to each individual dancer, such as the dance being performed and the sync delay, the actual positions of the dancers can also now be clearly seen represented by the letters 'L' , 'M' and 'R', which represent Left, Middle and Right respectively. We believe that the letter positions are more intuitive as the dancers simply have to just focus on their individual positions that is tagged to their name
4	The original design utilises 3 different graphs that will showcase the incoming real time data. However, we feel that it may be slightly too crammed to have so many graphs and information moving around as the dancers carry out their sessions. Additionally, this may add to latency to the frontend. A single graph is now used instead, which spans almost 70% of the screen, making it much more visible to the users. Additionally, users can simply choose which graphs to look at by clicking on the drop down beside the graph title, to select other graphs. The users will be able to choose which graph to look at, including accelerometer and gyrometer readings of each dancer, and the emg data of dancer 2. This can be seen in Figure 5.2.2.2.4. Additionally, we chose to display the EMG data to the users. The EMG has been pre-processed server side to show the gradient of the signal received. This gradient is a good representation of the fatigue level of a dance - the steeper the gradient, the more exhausted a dancer. This EMG graph can then be used by a dancer to gauge if he/she is pushing themselves to the limit during the dance session
5	Previously, it was assumed that the dashboard would be able to receive the ground truth of the dancers and positions to perform. However, after knowing that this is not true, we decided to scrap the coach card and blur out the card, with the text 'No Coach Selected' overlaying. The reason for this overlay will be further elaborated in **Section 5.2.5 Future Features**.
6	Another component that was not inside the original design was the 'End Session' button. This button will allow users to to manually 'logout' and end a dance session.

In addition to the dashboard, an overlay as shown in Figure 5.2.2.2.5 will show up on the screen whenever a session ends. This screen can be triggered by 2 ways - (1) by clicking the end session button above, if the user wants to end it manually, and (2) when the server receives a logout move prediction from the Ultra96. In both cases, the users will be given the option to end the ssion, which brings them to the History Screen, or to continue dancing, just in case there was a false prediction (or a misclick).

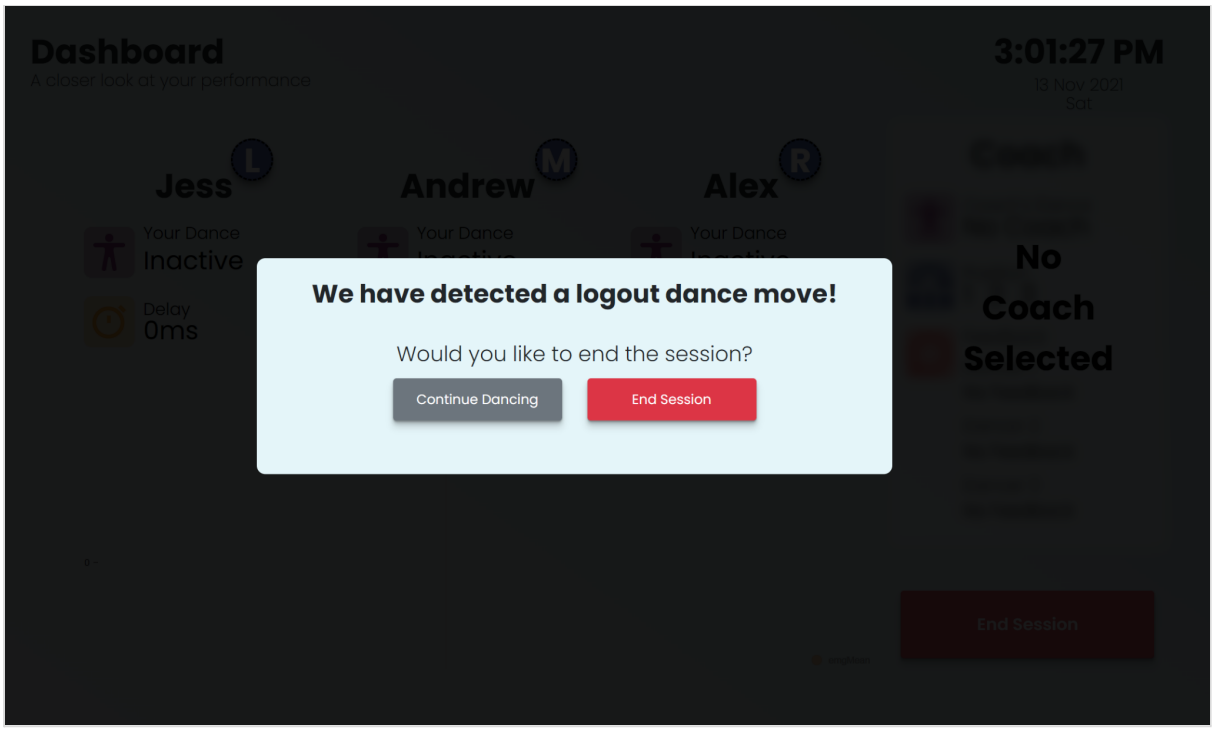


Figure 5.2.2.2.5: Logout overlay

History Screen

The core component of the History Screen is the same, which is a table of the past dance sessions. As seen in Figure 5.2.2.2.6, it still showcases some key information such as date, time and duration of the dance session. For each row of data, users will have the option to expand on the row, which will open up a model, as shown in Figure 5.2.2.2.7 that shows some analytics of the dance session. The specific of the analytics will be shown below.

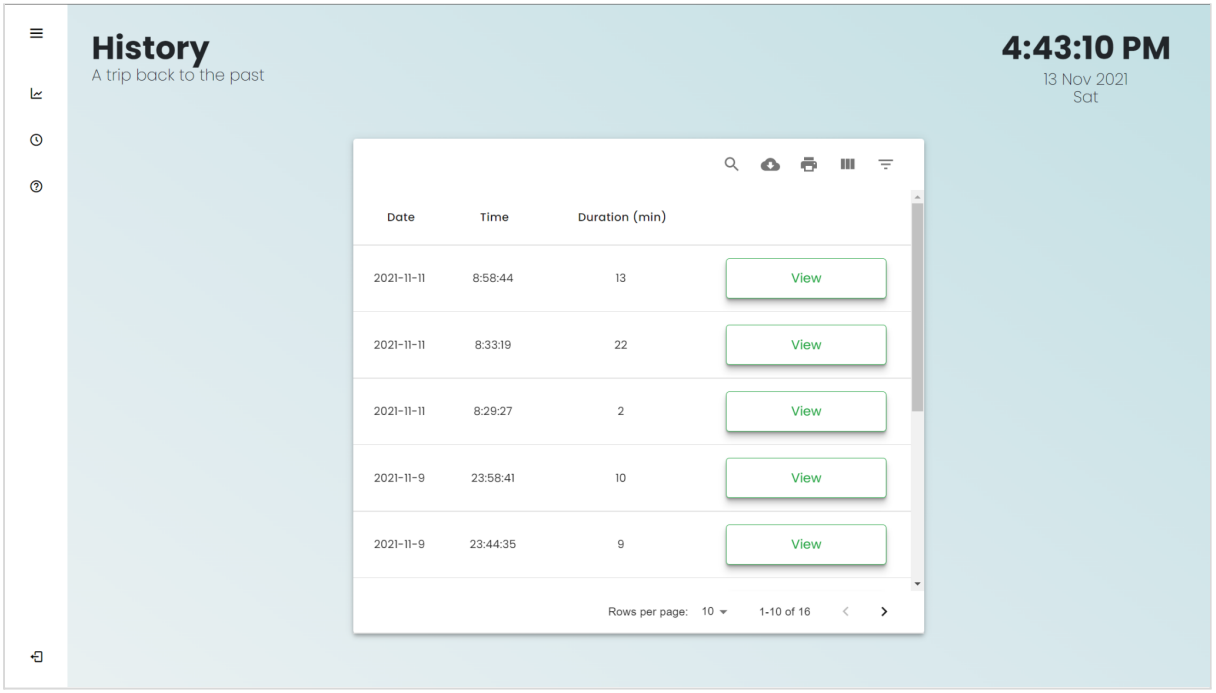


Figure 5.2.2.2.6: Final implementation of History Screen

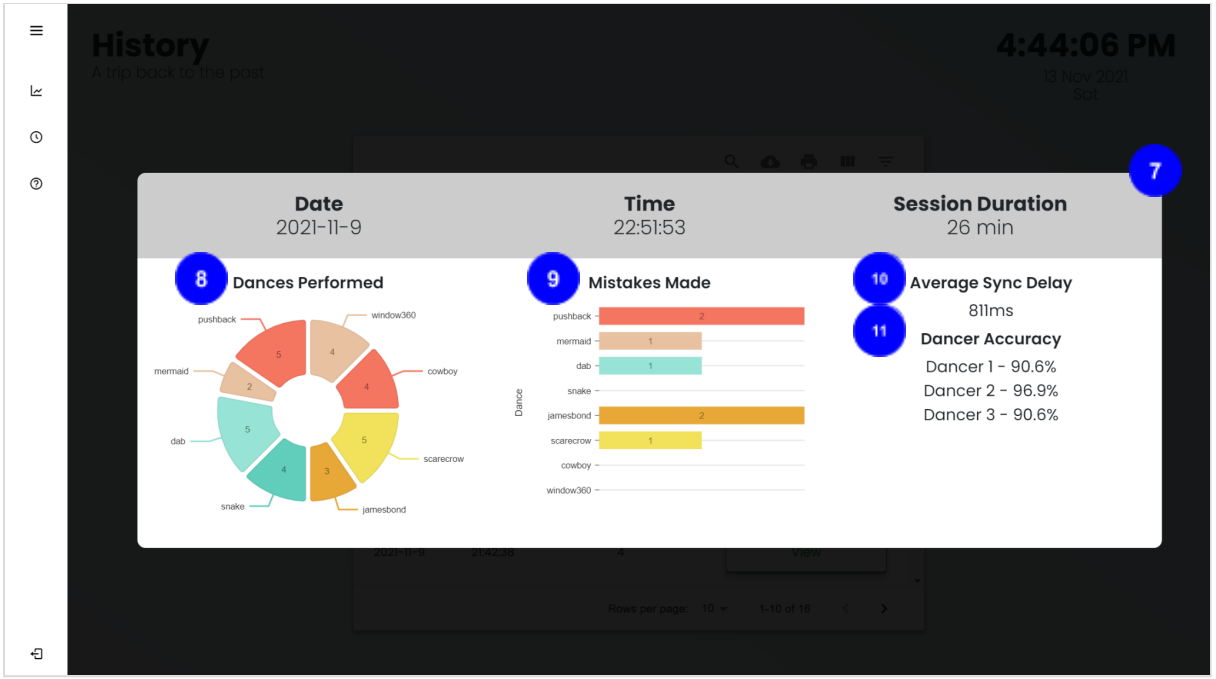


Figure 5.2.2.2.7: Modal showing analytics of dance sessions

Changes Made	
7	The header of the model simply shows information that as already shown in the original table - date, time and duration of session.
8	The pie chart is a representation of all the dance moves performed during the entire session, showing the counts for each specific dance move. This will give dancers a general overview of the session.
9	The bar graph showcases the mistakes that were made during the entire session. In this case, since we do not have a ground truth, a mistake* is taken to be when a dancer's predicted dance does not align with the mode of all the dancers' prediction. This will provide users with some information on which dance moves are the most mistake-prone, such that they can improve on these dance moves in the future.
10	Users will also be able to see the average sync delay of the entire session, to get a good gauge on how in sync they were during the entire dance session.
11	Additionally, dancers' accuracy is a simple gauge on who were better performing dancers. The accuracy is calculated based on the mistakes made by each dancer. Similar to the bar graph, the mistake* is taken to be when a dancer's predicted dance does not align with the mode of all the dancers' predictions.

* *It is to note that this method of calculating mistakes and accuracy is not ideal, but we were limited by the lack of access to the ground truth. The ideal implementation will be elaborated in* ***Section 5.2.5 Future Features.***

Tutorial Screen

The tutorial screen serves as a guide for dancers who want to learn how to perform specific dance moves. Users will simply have to click on a dance move on the right, and a looping GIF will play on the left, showing dancers how to dance the move. This can be seen in Figure 5.2.2.2.8.

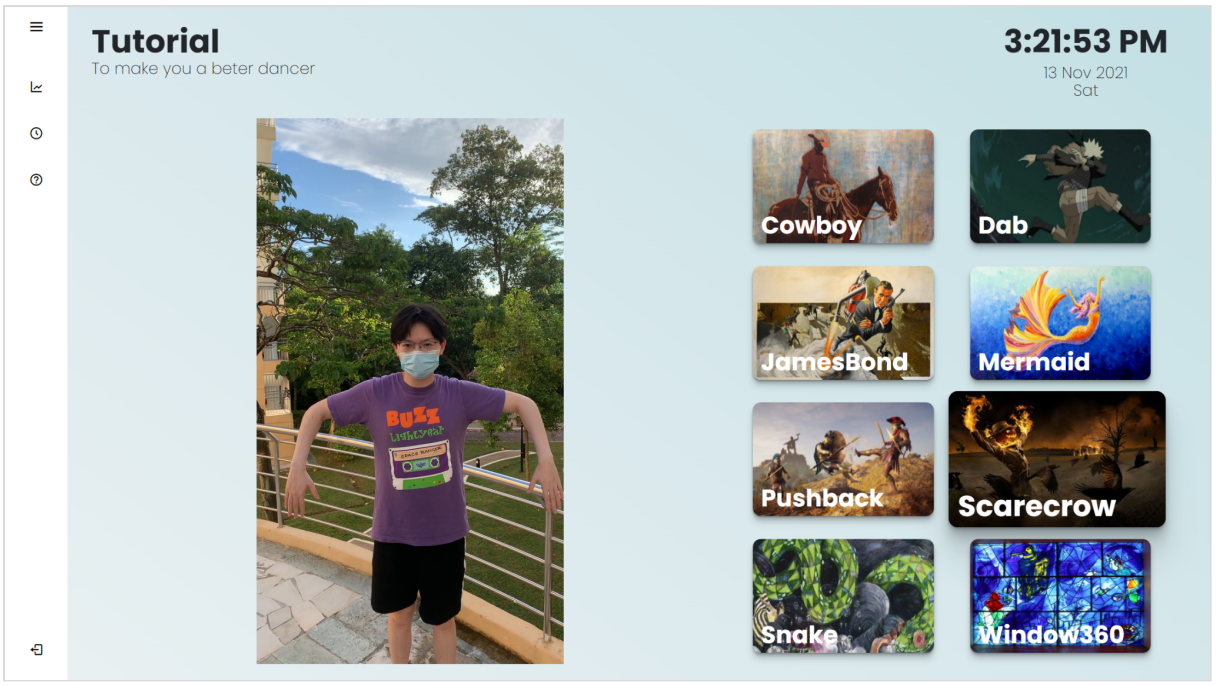


Figure 5.2.2.2.8: Final implementation of Tutorial Screen

Navigation Bar

In all the screens above, a navigation bar (navbar) can be seen on the left hand side of the screen. The navbar serves as a way for users to easily navigate between the different screens.

The navbar consists of 2 ‘modes’ - collapsed and expanded which is demonstrated in Figure 5.2.2.2.9. The collapse version is much smaller which gives users a much larger view of the main content on screen. Apart from the icons to help users know what they will be clicking on, there are also tooltips which appear on hover, showing where the button will link to.

When expanded, users can now easily see the different screens that they can navigate to. In addition to this, the bottom of the navbar will also show the logged in user, alongside a profile picture and ‘Dancing since YYYY’ which indicates when the user started this account.

Lastly, in both modes of the navbar, users can easily logout by clicking on the logout button, which will bring the user back to the login screen.

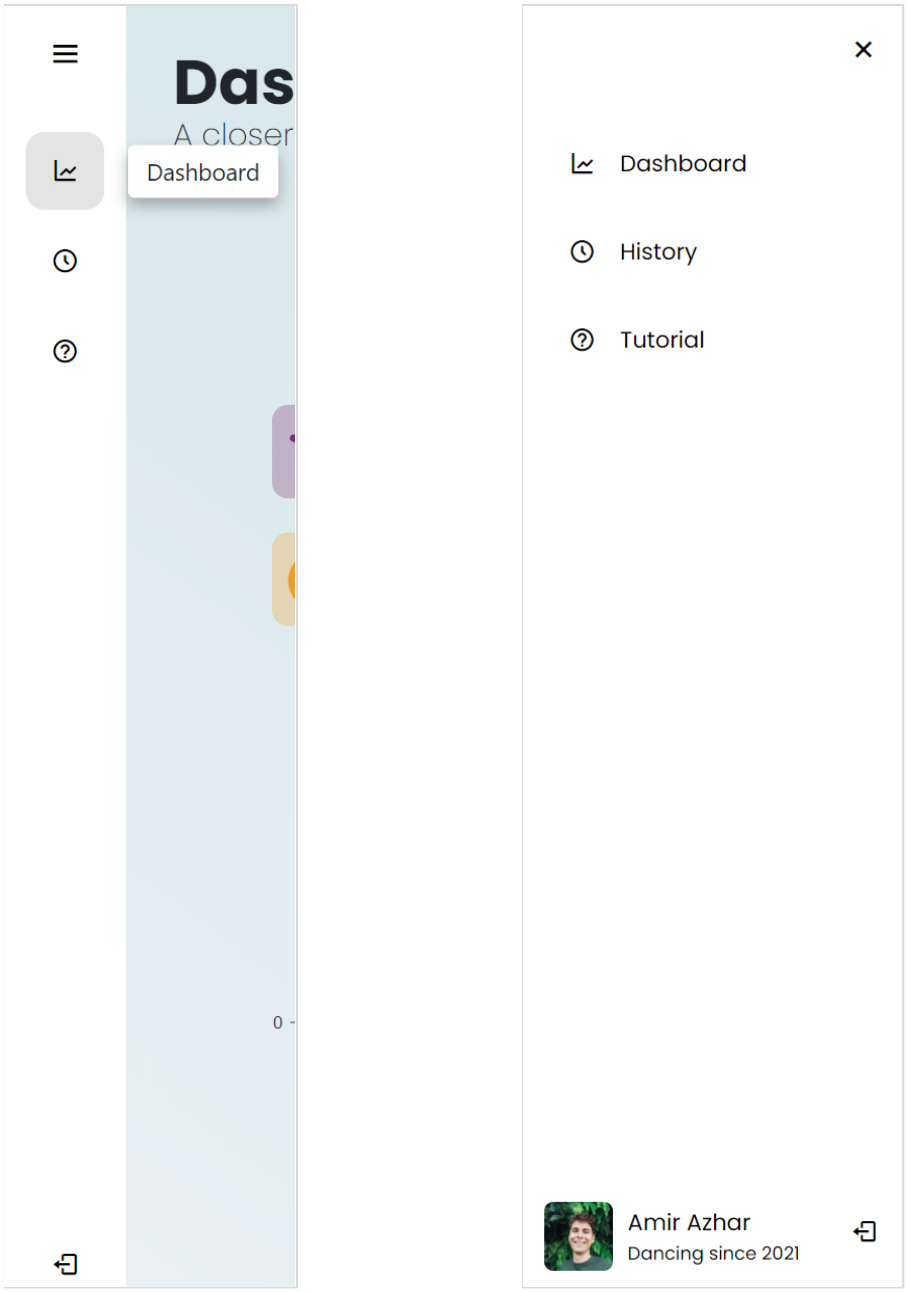


Figure 5.2.2.2.9: The collapsed (left) and expanded (right) navbar

5.2.3 Storing of Incoming Data

As previously explained, we will be storing the data in a MongoDB database. With the help of mongoose, a MongoDB object modelling tool for NodeJS, we are able to create schemas to model the incoming data from the Ultra96.

Figure 5.2.3.1 shows the schema containing raw sensor data from the accelerometer, gyroscope. 3 different models are being created with the same schema, to represent all 3 dancers. Each schema is also tagged with a user ID for easier identification. The chema also includes the tag, which was previously used to identify if the data is coming from the hand or chest sensors. Since our final implementation only has 1 sensor on the hand, the tag property is unused.

```
const mongoose = require("mongoose");

const rawDataSchema = mongoose.Schema({
  userID: { type: String, required: true, trim: true },
  tag: { type: String, required: true, trim: true },
  aX: { type: String, required: true, trim: true },
  aY: { type: String, required: true, trim: true },
  aZ: { type: String, required: true, trim: true },
  gX: { type: String, required: true, trim: true },
  gY: { type: String, required: true, trim: true },
  gZ: { type: String, required: true, trim: true },
});

const D1RawHandData = mongoose.model("d1_raw_hand_data", rawDataSchema);
const D2RawHandData = mongoose.model("d2_raw_hand_data", rawDataSchema);
const D3RawHandData = mongoose.model("d3_raw_hand_data", rawDataSchema);

module.exports = {
  D1RawHandData,
  D2RawHandData,
  D3RawHandData,
};
```

Figure 5.2.3.1: rawDataSchema for accelerometer and gyro data

Figure 5.2.3.2 shows the schema containing the predicted dance move, predicted position and sync delay.

```
const mongoose = require("mongoose");

const processedDataSchema = mongoose.Schema({
  predictedDance1: { type: String, required: true, trim: true }, // Dance
  predictedDance2: { type: String, required: true, trim: true }, // Dance
  predictedDance3: { type: String, required: true, trim: true }, // Dance
  predictedPos: { type: String, required: true, trim: true }, // pos1 | pos2 | pos3
  syncDelay: { type: String, required: true, trim: true }, // delay
});

const ProcessedData = mongoose.model("processed_data", processedDataSchema);

module.exports = ProcessedData;
```

Figure 5.2.3.2: processedDataSchema for dance and positions prediction, and sync delay

The last schema shown in Figure 5.2.3.3 shows the schema used for EMG data. As seen, the EMG data is separate from the accelerometer and gyro data, just to create segregation in case we decide to remove the EMG graph feature.

```
const mongoose = require("mongoose");

const emgDataSchema = mongoose.Schema({
  emgMean: { type: String, required: true, trim: true },
});

const EmgData = mongoose.model("emg_data", emgDataSchema);

module.exports = EmgData;
```

Figure 5.2.3.3: emgDataSchema used for EMG data

5.2.4 Real-time Streaming

When trying to understand how real-time streaming is performed from the standpoint the dashboard, there are 3 major links that we have to analyse - (1) Ultra96 → MongoDB, (2) MongoDB → Server, and lastly (3) Server ↔ Client.

Ultra96 → MongoDB

As mentioned in the earlier section, we will be using the PyMongo python driver to send any data from the Ultra96 to the local database. PyMongo is the official MongoDB Python Driver for MongoDB. This therefore ensures that the data travelling between these 2 mediums is as reliable and efficient as possible.

However, during testing, we faced latency issues on the server and client to display the real time data which was constantly coming in at 60Hz (3 dancers). As such, the external communications component pre-processed the data by sampling the incoming data, and only sending data to the database at 5Hz. This significantly improved the latency of the dashboard.

MongoDB → Server

To boost the real-time capabilities of MongoDB, we will be utilising MongoDB's 'Change Streams' API. This API allows the server to subscribe to all data changes on the MongoDB Atlas database and immediately react to them (MongoDB, n.d). This ensures that the server will receive the data with as little latency as possible.

Server ↔ Client

Socket.IO is a JavaScript library which enables real-time, bidirectional and event-based communication (Automattic, n.d). It utilises the WebSocket communication protocol (HTTP long polling for backup) as a means of transporting data, providing a full-duplex and low-latency channel between the server and the browser. This makes it much easier when trying to push the data to the client, to be represented on real-time counters such as logs, charts or graphs.

5.2.5 Initial User Survey

A major step of a UI/UX sketching stage is understanding the user and his/her needs. To get a hold of this, we should ideally perform market research/analysis. Due to the constraints of time, a more cost and time efficient method would be by conducting user interviews (minor design choices such as color schemes or fonts will be decided through market research). The upcoming sub-sections detail the survey that will help us decide the screens, components and features required for our application.

Target Audience

Dance coaches and trainees of varying ages

Survey Goals

To understand the needs and wants of a user, in a dance aid application so that we will be able to implement the most user-friendly and thoughtful UI and features

Survey Methodology

Due to the constraints of Covid-19, it will not be feasible and safe to interview face-to-face. As such, the survey will be sent through multiple online mediums. The survey will be made using Google Forms and distributed mainly to 'Telegram' and 'WhatsApp', through chat groups with high volume of users and traffic such as 'National University of Singapore Chat' or 'Memes n Dreams'. The responses of the survey will then be automatically stored and hopefully be ready for the next step of our survey - the analysis.

Survey Questions

The survey questions will be categorized into 2 - (1) understanding the user and his/her needs and (2) understanding user preferences for design choices for this application. These questions are formatted in Table 5.2.5.1.

Category	Question	Intended Purpose
User and his/her needs	How would you categorize a fruitful dance session for you?	To understand what dancers prioritise during a dance session.
	What are some difficulties you face doing dance classes over video conferencing?	To understand the current shortcomings of video conferencing dance sessions.
	What features/functionalities do you hope to have when carrying out dance classes over video conferencing?	To see what users look out for and hope to have during these dance sessions.
	Based on the features shown on the current design mockup of the	To see if the current mockup provides users

	dashboard, how do you think the data and information will help you in your dance sessions?	with the required information and/or if more is needed.
	Apart from video conferencing dance sessions, what other alternatives did you go for to meet your dance needs?	To also take a look at the other solutions in the market and learn from them.
Preferences for design choices	What do you think is the best way to display/represent data from sensors? e.g. graphs, pie charts, pure numerical data	To see which type of data visualisation is best. for users.
	Do you think the current mockup of the dashboard is aesthetically pleasing and also very informative?	To understand if the current design mockup is satisfactory.
	Are the text easily readable and are the icons/images easily recognizable?	To see if users can easily read and understand the components on screen.
	Would the dashboard be easy to navigate around?	To see if users are able to traverse around the webpage without any guidance.

Table 5.2.5.1: User survey questions

5.2.6 Other Notable Information

In this section, we will be elaborating on specific features in this fullstack application that was not previously discussed in depth since it is not a technical requirement on the project/dashboard. These features were added to improve the functionality of the website.

User Authentication

This feature is integral when users are logging in, logging out and registering. This feature was implemented using NPM packages such as jsonwebtoken (JWT), brcypt and redux.

JWT is the core of user authentication as it helps to define a “compact and self-contained way for securely transmitting information between parties as a JSON object”. This information is then used as a signature to verify if a user is authenticated or not, and its signature is also protected using RSA and ECDSA (JWT, n.d.).

Additionally, bcrypt is used to store sensitive information in the database. Since the database will be storing sensitive information such as user password, the password will have to be encrypted first using bcrypt, to protect the privacy of users. As seen in figure 5.2.6.1, the passwords of the users stored in the database are all encrypted, improving the security of the system (Arias, 2021)

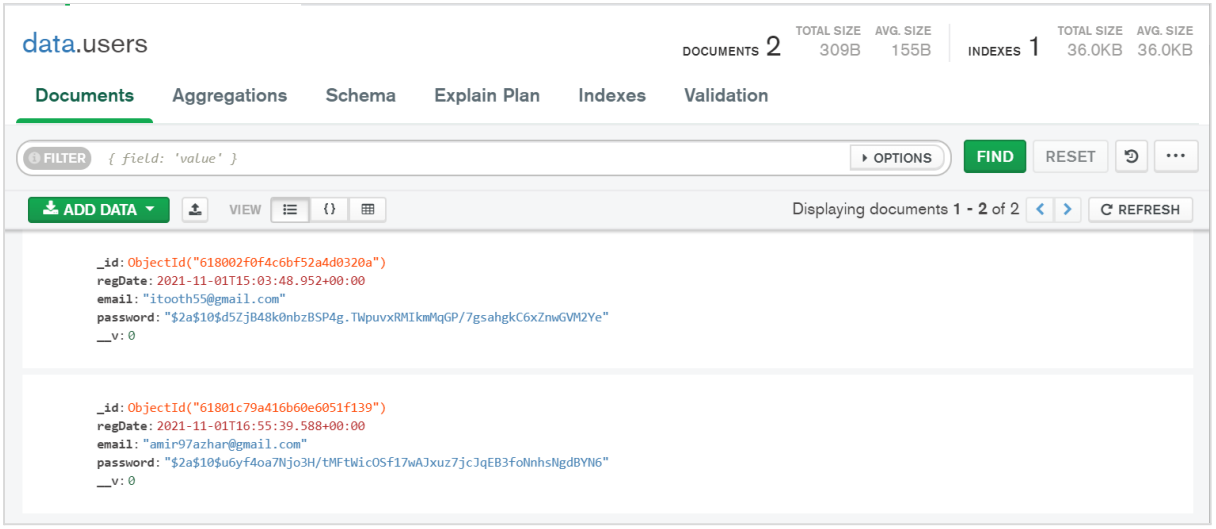


Figure 5.2.6.1: User collection in the MongoDB

Redux was also key in the implementation of the user authentication feature. Redux is used as a state management tool to keep track of which users are being logged in or out. To look at all the components from a larger point of view, JWT informs the server that a user is authenticated while Redux will ensure that that an authenticated user will stay logged in when he/she signs in (Redux, n.d.)

5.2.7 Future Features

This section will highlight the key features that we would like to implement to the dashboard in the future, but we were not able to due to the lack of time.

Coach Card

As previously mentioned in the description of the dashboard design, the original design included a coach card that showed the ground truth. This was removed as we found out that the dashboard will not be receiving any of this from the evaluation server.

Despite this, we decided to keep the card, only with an overlay that states “No Coach Selected”. A feature that we would like to implement (outside of this assessment) is the ability for users to pick a coach, which is essentially a dance routine to follow. Users will then be given instructions on the coach card based on the set dance routine that was created. Additionally, the coach card would be able to provide automated feedback to the dancers depending on the dancer’s predicted dance and position. With this, the analytics on the History Screen will be more accurate, as it will then use the coach data as the ground truth.

User Profile Customisation

It can be seen from the navbar that there is a profile photo and username that is tagged to the current logged in user. In the screenshot, this photo and username has been hardcoded for demonstrative purposes. A future feature would include users being able to click on their profile photo and edit account specific information such as username, password and email.

Detailed Information in Tutorial Screen

Currently the tutorial page seems less of a tutorial, and more of just simply showing the dances. In the future, we aim to include information on how a dance move can be performed,

by stating the different steps a user can take to achieve a dance move or even how to move the different body parts e.g. rotate hips, sway the arms at a 180 degree angle etc.

6 Project Management Plan

6.1 Project Management Tools

Trello

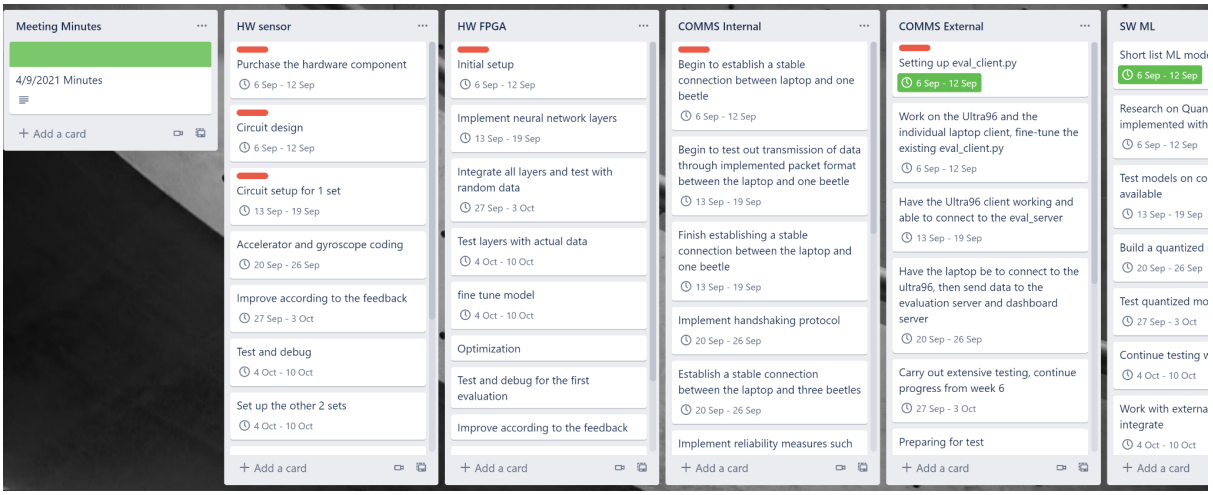


Figure 6.1.1: Screenshot of Trello Dashboard

To track and manage our project, our team decided to use Trello as our project management tool. Trello is a visual project management tool that empowers teams to ideate, plan, manage, and celebrate their work together in a collaborative, productive, and organized way (*Learn Trello Board basics*). Furthermore, we can track one another's progress, ensuring that we are on track with our schedule and avoiding the need to rush our work.

Github

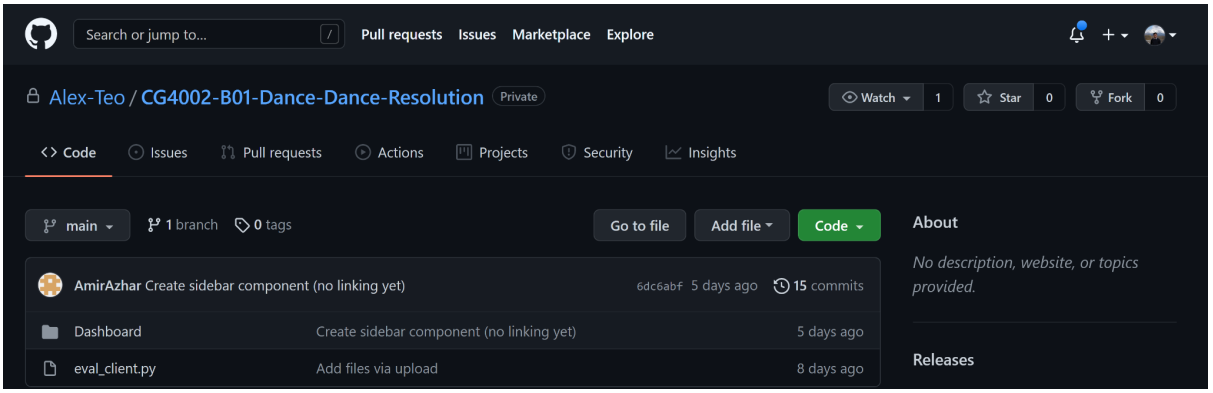


Figure 6.1.2: Screenshot of Github Repository

Version control and collaboration of code is done using a Github repository. Commits are done with descriptive messages allowing us to track each commit easily. Each feature is also done on a separate branch and a pull request is required to merge with the master branch. Every pull request would also require a reviewer to allow us to comment on the changes proposed in the pull request.

6.2 Project Schedule

Week	HW sensors	HW FPGA	COMMS internal	COMMS external	SW ML	SW Dashboard
3	- Research on power design	- Research on FPGA	- Research on comms principles	- Setting up eval_client.py	- Research on Traditional Machine Learning models and Deep Learning models - Preliminary testing with online dataset	- Set up boiler plate for MERN stack and create a MongoDB Atlas database
4	Design report					
	- Purchase the hardware component - Circuit design	- Initial setup	- Begin to establish a stable connection between the laptop and one beetle	- Work on the Ultra96 and the individual laptop client, fine-tune the existing eval_client.py	- Short list ML models to use - Research on Quantization to be implemented with Neural Networks	- Create basic mockups of the different screens using AdobeXd
5	- Circuit setup for 1 set	- Implement neural network layers	- Finish establishing a stable connection between the laptop and one beetle - Begin to test out transmission of data through implemented packet format between the laptop and one beetle	- Have the Ultra96 client working and able to connect to the eval_server	- Test models on collected data if available - Build a quantized model	- Begin converting the design mockups into actual code and setup the links between frontend and backend

6	Progress checkpoint (individual)					
	- Accelerator and gyroscope coding	- Implement neural network layers	- Establish a stable connection between the laptop and three beetles - Implement handshaking protocol	- Have the laptop be to connect to the ultra96, then send data to the evaluation server and dashboard server	- Test quantized model on FPGA	- Using dummy data in the database to test the dashboard
Recess	- Improve according to the feedback	- Integrate all layers and test with random data	- Implement reliability measures such as the reconnect function - Test for errors and fix any issues in data transmission - Implement improvements in areas highlighted during feedback session	- Carry out extensive testing, continue progress from week 6	- Continue testing with FPGA	- More testing of the dashboard with dummy data and fine tuning
7	Individual subcomponent tests					
	- Test and debug - Set up the other 2 sets	- Integrate all layers and test with random data	- Prepare for test and debug for evaluation - Fix issues based on feedback from evaluation	- Preparing for test -Fix any bugs -start trying to integrate	- Work with external comms to integrate	- Optimization
8	- Test and debug for the first evaluation	- Test layers with actual data, fine-tune	- Integrate with the hardware	- Monitor progress, troubleshootin	- Identify areas to improve on	- Testing dashboard with real-time

		model	sensor and external comms components - Improve on the speed and test out the reliability of the system after integrating with other components	g	- Continue testing and optimize	data
9	First evaluation test					
	- Improve according to the feedback - Purchase Rechargeable Batteries	- Optimize latency	- Prepare and debug for first evaluation test - Fix issues based on feedback from evaluation	- Troubleshooting and monitoring	- Identify problems and rectify	- Testing dashboard with real-time data
10	- Test and debug for all sets - Add position detection code	- Optimization	- Perform rigorous testing and rectify any issues	- Troubleshooting and monitoring	- Fine tune and test	- Testing dashboard with real-time data and optimizing
11	Second evaluation test Peer evaluation					
	- Final test and finetune - Change to rechargeable batteries	- Optimization	- Prepare and debug for second evaluation test - Fix issues based on feedback from evaluation	- Troubleshooting and monitoring	- Rectify issues and test	- Testing dashboard with real-time data and optimizing
12	- Final test and finetune for turning position	- Optimization	- Final testing and fine tuning - Prepare for	- Troubleshooting and monitoring	- Continue testing and perform stress test	- Final testing and debugging

	detection		final evaluation			
13	Final evaluation Final report					
	- Final testing and debugging - Finalize report	- Final testing and debugging - Finalize report	- Final testing and debugging - Finalize report	- Final testing and debugging - Finalize report	- Final testing and debugging - Finalize report	- Final testing and debugging - Finalize report

7 Ethical and Societal Impacts

7.1 Future Use Cases

In its current implementation, our device is mainly limited to dancing and tracking dance moves. With that said, there are many possible areas we could possibly expand to. With the addition of new sensors and chips, as well as the power of the machine learning model, there are truly endless possibilities to the future of our device. We will discuss these potential developments in the following sections.

7.1.1 Entertainment

Given how the connection is formed between the beetle and the laptop, it is possible to implement a version of the hand beetle as a form of simplified motion controller not unlike the Playstation Move or Wii remote. Further enhancement to the system, such as having sensors to detect finger/digit movement could also contribute greatly to the development of more advanced VR controllers, which are not exactly the most organic or form fitting in appearance today as can be seen in this image (Amazon, n.d):

Figure 7.1.1.1: AMVR Touch Controller for Oculus Rift CV1

With more improvements to the glove, we could make a form fitting and easy to use controller that could be rather ergonomic in design. Moving your hands in VR would feel as natural as moving your fingers and hands normally in the real world.

7.1.2 Sports

Aside from dancing, we could use the hand tracking to study movements in sports that heavily feature the usage of hands such as soccer goalkeepers, volleyball and tennis players.

This way we can get data from top players and use their data to help refine the movements of less skilled athletes, not unlike what we are doing with our current implementation.

7.1.3 Medical

In terms of medical usages, we could modify the current wearable to be akin to something similar to the Apple watch or other fitness trackers so that we could use it to monitor the elderly. Since our wearable can detect sudden changes in acceleration and gyroscope, we could use it to monitor the elderly in the event that they were to fall or collapse without warning. The wearable could then automatically send a signal to the nearest emergency response team in order to reach the affected elderly as soon as possible. This would be especially useful for elderlies who are residing alone.

7.1.4 Miscellaneous uses

Other uses we could experiment with would be a gesture recognition system that uses the beetle to recognise different gestures via changes in acceleration and gyroscope. Aside from that, we can also use the tracking system for recognising hand movements when carrying out activities like drawing. We could also use the EMG in a way to measure how much stress is being applied by hands and fingers during certain parts of the drawing.

7.2 Privacy Concerns

There were previous accounts of trouble in connecting the Bluno Beetles and the laptops due to the bluetooth interference from numerous Bluno Beetles of other groups around in previous semesters. However, due to the virtual testing nature of our evaluation, we did not encounter this problem. This could pose a problem in the event that the MAC addresses of the Bluno Beetles happened to match or were purposefully spoofed, where the messages being sent via Bluetooth could be intercepted by malicious personnel without consent. Additionally, messages with malicious intent could be injected, resulting in the server side receiving messages that could very well break the server, or even leak important information. Possible ways to circumvent this include password mandates when connecting, or some other authentication channels to confirm the connecting party.

With regards to data collection, the physical date of anyone's data being collected should be subject to the relevant data protection laws in any country. This is to prevent such personal data from being misappropriated by any parties with malicious intent or otherwise, such as being used by technology companies without consent. Unless the user has willingly given his or her consent, the data should be stored locally on the device and not be sent to any external parties. This means discrete leakage of data of any form should not be tolerated.

In order to address the first concern, for the sensors, they could each have their own in-house security key to prevent unauthorised reading or writing of data unless allowed to do so. This should prevent some passive leakage of data. Additionally we could also encrypt the data sent between Beetle and laptop, while also having some sort of checksum in the file to serve as proof that this file was not injected from a malicious party in transit. The malicious files would then be discarded.

For the second concern, the relevant government agencies are needed to enforce the penalties should any of the relevant be broken. Additionally, affected parties could also help to report said violation of laws to the government agencies should they chance upon it. Users are, as last resort, possibly allowed to "hack" or modify their wearables to make it more secure if need be, not unlike how in America some people hacked their own insulin pumps.

7.3 Ethical Considerations

In this case, there are at least 3 main key stakeholders here we should look into.

7.3.1 Dancers/Testers

Due to the expedited production process, there could be some fallacies when we developed the ML model and the wearable. This meant that it is possible that during the training of the model or during testing of the full product, testers may have taken shortcuts or utilised loopholes that made it such that while the moves looked like they were being correctly performed, they were actually not. This could potentially affect future users as they would be unable to accurately tell if their moves were correct and the whole system would be rendered meaningless. In that case, we should evaluate the role that testers had to play here. On one hand, by discovering these exploits in the testing phases, these could have been reported early to the developers so as to fix during this stage. However, if this was not relayed to the development team, why would the testers not do so? In that case, testers failed to live up to the requirements of their jobs due to this failure of disclosure, which would mean violation of duty ethics.

7.3.2 Coaches

The coaches here would be acting as the intermediary between the dancers and the ML model feedback. In that sense, they guide the dancers to execute the correct moves via the feedback. While the device is technically supposed to guide the dancers to the correct performance of the moves, what if the feedback was wrong? If the coach lacks experience, would he be aware that he is giving wrong instructions to his dancers. Would it be correct to lay the entire blame on them for blindly following the model? Or what if the coach was seasoned? Even if he knew it was wrong to follow the feedback given by the model, could he still be contractually obligated to do so and doing otherwise by giving his own commands could go against his directives as the coach thereby violating duty ethics? Here the coach has the right to ensure that the dancers are dancing properly, and so could ignore the prediction results if they are wrong in the sense that there is a mismatch between right and wrong. With that, we need to weigh the duty and rights ethics here before we can definitively label the nature of his actions.

7.3.3 Manufacturers

When it comes to delivering a product under very short time constraints, it is expected that there would be some bugs. However, if the manufacturers failed to disclose said redundancies to the consumers and painted the illusion that the system was infallible, then that would be a major ethics problem. Of course, there are cases where even the designers are unable to identify all the bugs and they would only be discovered once the product has

been released to the mass public, thereupon some of the blame can be directed away from them. Regardless, it is up to the manufacturers themselves to own up to their mistakes, otherwise they would fail to meet their responsibilities which would fall under duty ethics..

7.4 Conclusion

When it comes to developing new technologies at a fast pace, there will definitely be some areas where we will overlook when it comes to ethics and the like. However, among the things we do not forget about, we must ensure that at the very least safety, privacy and health of the users and testers are maintained. As for other areas where we may overlook such as reliability, these failures should be minimized as much as we can before we finalize our product. If a flaw we did not spot earlier comes up, we must take responsibility for it, apologise and fix it as soon as we can. When all these things come together, we can then truly maintain our ethical obligations for the product.

References

Analytics Vidhya. (2021, July 23). CNN for deep learning: Convolutional Neural Networks. https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/.

Analytics Vidhya. (2021, May 21). Importance of cross validation: Are Evaluation Metrics enough? https://www.analyticsvidhya.com/blog/2021/05/importance-of-cross-validation-are-evaluation-metrics-enough/.

Analytics Vidhya. (2020, December 2). Feature selection techniques in machine learning. https://www.analyticsvidhya.com/blog/2020/10/feature-selection-techniques-in-machine-learning/.

Arduino. (n.d.). *MPU6050_tockn.* https://www.arduino.cc/reference/en/libraries/mpu6050_tockn/

Arduino. (n.d.). Wire. https://www.arduino.cc/en/reference/wire/

Automattic. (n.d.). *Socket.io 4.0.* https://socket.io/

Amazon. (n.d.). AMVR Touch Controller grip cover for Oculus rift cv1 anti-throw handle protective sleeve (only work with Rift CV1). Amazon.sg: Video Games. Retrieved November 14, 2021, from https://www.amazon.sg/AMVR-Controller-Oculus-Anti-Throw-Protective/dp/B07ZKQ5LP8/ref=asc_df_B07ZKQ5LP8/

ATmega328P datasheet. (n.d.). https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf.

Beedle, M. (2001). *Manifesto for agile software development*. https://agilemanifesto.org/.

BlunoBeetle datasheet. (n.d.). https://media.digikey.com/pdf/Data%20Sheets/DFRobot%20PDFs/DFR0339_Web.pdf.

Brownlee, J. (2019, August 6). *How to use learning curves to diagnose machine learning model performance*. Machine Learning Mastery. Retrieved November 14, 2021, from https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/.

Dan Arias. (2021). *Hashing in Action: Understanding bcrypt.* https://auth0.com/blog/hashing-in-action-understanding-bcrypt/

DFRobot. (n.d.). *Bluno Beetle Simple Tutorial With Explanatory Images*. https://www.dfrobot.com/blog-283.html

DFRobot. (n.d.). *SKU:DFR0339.* https://wiki.dfrobot.com/Bluno_Beetle_SKU_DFR0339

Elsevier Science & Technology. (2012). *Control system design guide*.

Facebook Inc (n.d.). *React.* https://reactjs.org/

Fath, A. H., Madanifar, F., & Abbasi, M. (2018, December 12). Implementation of multilayer PERCEPTRON (MLP) and radial basis FUNCTION (RBF) neural networks to PREDICT SOLUTION gas-oil ratio of crude oil systems. Petroleum. https://www.sciencedirect.com/science/article/pii/S2405656118301020.

Gandhi, R. (2018, July 5). Support Vector Machine - introduction to machine learning algorithms. Medium. https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.

Harvey, I. (2014). *Bluepy - a Bluetooth Le interface For python*. bluepy - a Bluetooth LE interface for Python - bluepy 0.9.11 documentation http://ianharvey.github.io/bluepy-doc/.

H. Junker, P. Lukowicz and G. Troster. (2004). *Sampling frequency, signal resolution and the accuracy of wearable context recognition systems*. IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/1364710.

Jain, A. (n.d.). Problem with multithreading in python ? Anurag Jain. Retrieved November 14, 2021, from http://anuragjain67.github.io/writing/2016/01/15/problem-with-multithreading-in-python.

JWT (n.d.). *Introduction to JSON Web Token.* https://jwt.io/introduction

Know PYNQ in one article - code world. (n.d.). https://www.codetd.com/en/article/12698501.

Learning, U. C. I. M. (2019, November 13). *Human activity recognition with smartphones. Kaggle.* https://www.kaggle.com/uciml/human-activity-recognition-with-smartphones.

Lima, W. S., Souto, E., El-Khatib, K., Jalali, R., & Gama, J. (2019)). (rep.). *Human Activity Recognition Using Inertial Sensors in a Smartphone: An Overview.*

Mohajon, J. (2021, July 24). *Confusion matrix for your multi-class machine learning model*. https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826.

MongoDB Inc (n.d.). *Change streams*. Change Streams - MongoDB Manual. https://docs.mongodb.com/manual/changeStreams/.

MongoDB Inc (n.d.). C*omparing the differences - mongodb vs mysql*. MongoDB. https://www.mongodb.com/compare/mongodb-mysql.

MongoDB Inc (n.d.). *What is the MERN Stack? Introduction & examples.* MongoDB. https://www.mongodb.com/mern-stack.

MyoWare Muscle Sensor Kit. (n.d.). https://learn.sparkfun.com/tutorials/myoware-muscle-sensor-kit/all

Myoware User Manual . (n.d.). https://cdn.sparkfun.com/datasheets/Sensors/Biometric/MyowareUserManualAT-04-001.pdf.

Myroniv, Bohdan & Wu, Cheng-Wei & Ren, Yi & Christian, Albert & Bajo, Ensa & Tseng, Yu-chee. (2017). *Analyzing User Emotions via Physiology Signals. Data Science and Pattern Recognition.* 2.

MIT. (n.d.). https://www.mit.edu/~9.520/spring09/Classes/multiclass.pdf.

MPU 6050 datasheet. (n.d.). https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf.

Naik, A. (2019, October 28). *How to Interface Arduino and THE MPU6050 SENSOR. Epictec.* https://epictecs.wordpress.com/2018/05/28/how-to-interface-arduino-and-the-mpu6050-sensor/.

OpenJS Foundation (n.d.). *Introduction to Node.js.* Node.js https://nodejs.dev/learn.

Overlay tutorial. (n.d.). *Overlay Tutorial - Python productivity for Zynq (Pynq) v1.0.* https://pynq.readthedocs.io/en/v2.0/overlay_design_methodology/overlay_tutorial.html.

Peterson, L. E. (2019). *K-Nearest Neighbor*. Scholarpedia. http://www.scholarpedia.org/article/K-nearest_neighbor.

Python. (n.d.). *Thread-based parallelism - Python 3.10.0 documentation.* https://docs.python.org/3/library/threading.html.

PMBus. (2021, May 3). *Pmbus® home.* https://pmbus.org/.

PYNQ. (n.d.). *Python productivity for Zynq*. http://www.pynq.io/.

Redux (n.d.). Redux. https://redux.js.org/

Smith, S. W. (1997). *The scientist and engineer's guide to digital signal processing*. California Technical Pub.

Stack Overflow. (2016, August 12). *Maximum packet length for Bluetooth LE?* Stack Overflow. https://stackoverflow.com/questions/38913743/maximum-packet-length-for-bluetooth-le.

StrongLoop (n.d.). *Node.js web application framework.* Express. https://expressjs.com/.

Toro, S. F. D., Santos-Cuadros, S., Olmeda, E., Álvarez-Caldas, C., Díaz, V., & San Román, J. L. (2019, July 20). Is the use of a low-cost SEMG sensor valid to measure muscle fatigue? Sensors (Basel, Switzerland). https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6679263/.

Trello (n.d.). *Learn Trello Board basics*. Trello 101: How to Use Trello Boards & Cards. https://trello.com/en/guide/trello-101.

Ultra96-V2. (n.d.). https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/ultra96-v2/

Welcome to cocotb's documentation! Welcome to cocotb's documentation! - cocotb 1.6.0.dev0+gec99a877.d20210503 documentation. (n.d.). https://docs.cocotb.org/en/stable/#.

Zhu, J., San-Segundo, R., & Pardo, J. M. (2017, June 2). Feature extraction for robust physical activity recognition. Human-centric Computing and Information Sciences. https://hcis-journal.springeropen.com/articles/10.1186/s13673-017-0097-2.