SENSING AND IOT COURSEWORK

IMPERIAL COLLEGE LONDON

DEPARTMENT OF DESIGN ENGINEERING

Smart Lift Tracker

Joseph Johnson

CID: 01850831

Video Submission

GitHub Repository

Abstract

This report details the development of the Smart Lift IoT sensor and mobile application, created as part of the Sensing and IoT course. The sensor and app allow a user to track their exercises within a gym environment, providing in depth metrics and progress tracking. The project is built upon cloud infrastructure, making it scalable to any number of users.

Contents

Chapter 1

Introduction

1.1 Background

When in the gym, current market offerings for recording your workouts are limited to users having to manually input their reps, sets, and weight lifted. While this is helpful, it would be useful to see more detailed statistics - in my case I plan to track the following for each rep:

- **Range of motion:** The percentage of the movement completed. For example, squat depth or lateral raise height.
- **Stability:** The smoothness of the motion. A less smooth rep is indicative of it being closer to failure, a target for building muscle.
- **Balance:** Applicable when using both arms or legs in free weight exercises. Nalance is a measure of how similar the movement is between the left and right sensor.
- **Force:** By using a user inputted weight and acceleration values, the force the exerted can be tracked to see figures such as peak or average force.

1.2 Project Goals

Task	**Sub Task**	Report Section
Sensing	Power Arduino using Battery	2.1.2
Sensing	Read IMU measurements	2.3
Sensing	Control RBG LED light	2.2.1
Sensing	Pair with mobile phone via Bluetooth	2.3.4
Sensing	Send and receive data via Bluetooth	2.3.4
Processing	Count the reps in a set	3.2
Processing	Calculate range of motion	3.4.1
Processing	Calculate stability	3.4.2
Processing	Calculate balance	3.4.3
Processing	Calculate force	3.4.4
Processing	Perform processing in the cloud	3.5
Mobile Application	Design front end	4.1
Mobile Application	Code front end	4.2.1
Mobile Application	Connect with back end	4.2.2
Mobile Application	Implement Authentication	4.2.2
Mobile Application	Connect to Arduino	4.3
Mobile Application	Visualise Data	4.4

Table 1.1: Project Task List

1.3 Technical Stack and System Design

1.3.1 Hardware

1.3.2 Mobile App

A mobile app, written in Flutter, is used to connect to the Arduino. This was chosen due to the improved user experience of using a phone in a gym compared to a laptop, and also due to my own desire to learn the Flutter language through practical application.

1.3.3 Firebase

Firebase, a backend as a service platform provided by google was used due to its ease of integration with Flutter. Firebase provides the following purposes:

- Authentication via Google sign in.
- Storing user data using a Firestore database.
- Calculating metrics from exercise data using Firebase Cloud Functions.

Chapter 2

Sensing

2.1 Hardware

2.1.1 Arduino Nano 33 BLE

Ardunio was chosen as the hardware platform for the project due to its high community support and low cost. Specifically the Arduino Nano 33 BLE was chosen as it was small, low energy, and included the 9-Axis IMU, reducing complexity, fragility and cost. It could also be powered easily from a pair of coin cell batteries, making it extremely easy to use as a portable, battery powered system.

2.1.2 Batteries

Two coin cell batteries are used to power the Arduino. These were chosen due to their slim, compact size and appropriate power delivery, providing the Arduino with the needed 3.3V. A battery housing was chosen that included a switch, allowing me to turn the devices on and off with ease.

2.1.3 Silicon Slap Band

In order to attach the micro controllers and batteries to dumbbells, barbells, machines, ankles or wrists, I chose to mount the electronics onto silicon slap bands. This solution provides the following benefits:

- **Grip:** The high friction silicone prevents the sensors from slipping, ensuring that the tracking is always relative to the target object.
- **Ease of use:** The slap on nature means that fitting and removing the devices takes a matter of seconds.
- **Portability:** When straightened, the devices are slim and can easily be slipped into any bag.
- **Low Weight:** The overall device is lightweight, making it unobtrusive on the user.

2.2 RGB LED

2.2.1 LED Control

The LED was controlled using using functions for setting a stable colour, and for setting a blinking colour, seen in C.1. These functions took values for red, blue, green and in the case of the blinking function, a time interval to set the frequency of the blinking, and a duration to set how long it should blink for. A table of different LED meanings can be found in B.1

2.3 IMU

2.3.1 Hardware

Onboard the Arduino Nano 33 BLE is the LSM9DS1 9-Axis IMU. This provides the following measurements:

- **Acceleration:** Used to determine lateral movements of the device
- **Gyroscope:** Used to determine rotational movements of the device
- **Magnetometer:** Generally unused in this application

2.3.2 Initialization of the IMU

When the device powers up, it performs an initial check to ensure it can access the Inertial Measurement Unit (IMU), as detailed in C.2. If the device encounters any issues during this stage, it responds by displaying a yellow LED and outputting an error message through the serial connection.

2.3.3 Data Collection

Once the IMU is successfully initialized, the device can enter the 'collecting' mode. In this mode, it actively records IMU readings, aggregating them into a temporary storage. This continuous data collection is essential for compiling comprehensive datasets, as shown in C.3. The data remains in this temporary storage until it is ready to be transmitted.

2.3.4 Bluetooth Communication for Data Transfer

The device utilizes Bluetooth for communication with a connected mobile phone. It stands by for specific commands – 'start' to commence data collection and 'stop' to cease it. On receiving the 'start' command, the device enters the 'collecting' mode. Conversely, the 'stop' command triggers the end of data collection and initiates the transfer of the accumulated data to the mobile phone.

The relevant code can be seen in C.4.

Chapter 3

Data Processing

3.1 Preprocessing

3.1.1 Noise Filter

A Butterworth filter was used to smooth out the IMU readings. It does this by removing high frequency noise. This was done in order to isolate the movement of the sensor from the noise of the sensor. A Butterworth filter

was chosen as it ensures the signals amplitude is not distorted.

The code for this can be seen in in C.5

3.1.2 Removing Outliers

Additionally, outliers from the time series are removed in order to improve the accuracy of the tracking. The mean of the dataset is calculated, and any values that are exceed a threshold *(I found a threshold of 3 to be best)*, are removed. For my purposes, I found this a useful addition to the preprocessing process that dealt with exceptionally high readings from the sensors in cases such as dropping the weights.

The code for this can be seen in C.6

3.1.3 Upright Transformation

To make the system more tolerant of the sensor being positioned at different rotations around bar, dataset is rotated so that it always begins with the z-axis pointing directly up. This is done by assuming the sensor is at rest at first. As the accelerometer detects the acceleration due to gravity, the direction of the magnitude is assumed to be be facing down, towards the ground. The whole dataset is rotated accordingly such that this acceleration is aligned with negative z.

This creates a much more tolerant system, that is is reliable regardless of how the sensor is mounted. However, I did find that this was not always as reliable on exercises where dumbbells or wrists were tracked, as the tracker was not as stable as when a barbell starts on a rack or when a machine is at rest, ensuring it is absolutely stationary.

The code for this can be seen in C.7

3.2 Detecting Repetitions

The first challenge for processing the data was splitting the time series data into repetitions. For my purposes, each repetition is made up of the following phases:

- **Phase 1:** The first movement of the lift, either the eccentric or concentric depending on the type of exercise.
- **Phase 2:** The isometric, or hold between phase 1 and 2.
- **Phase 3:** The second movement of the lift, the inverse of phase 1.

In order to split this, first a compound measurement of movement was determined, calculated as shown below.

$$c = \left(\frac{a}{a_{\text{max}}} \times 100\right) \times \left(\frac{g}{g_{\text{max}}} \times 100\right) \times \left(\frac{m}{m_{\text{max}}} \times 100\right) \tag{3.1}$$

where c is the compound measure of movement ,
a is the magnitude of the accelerometer values ,
g is the magnitude of the gyroscope values,
m is the magnitude of the rate of change of magnetometer values ,

This compound magnitude was smoothed using a rolling window of 50ms to make it more resilient to noise, with the result of this shown below in figure A.1.

From here, the compound magnitude was differentiated in order to find the turning points. A threshold for the movement to be classified as stationary was applied, and manually tuned in order to achieve accurate classification of phases. The results of this are shown in figure A.2.

The time series is then split when the derivative moves upwards past the upper or lower threshold. Data before the first crossing is disregarded, as this is when the user is preparing to lift, and data between sets is discarded, as this is when the user is resting between reps. This gives the result as shown in figure A.3, where each rep is represented as a different colour, split by phase.

3.3 Creating Calibration Datasets

When a user adds a new exercise through the app, they are asked to create a calibration dataset, performing the exercise with little or no weight in order to ensure they are using optimal form and a full range of motion. The recorded data is then split by rep. Each phase of the rep is then scaled to 3 seconds, and an average accelerometer, gyroscope and magnetometer reading is calculated for each phase. This is the calibration dataset, unique for each user performing a given exercise such that it fits the individual bio mechanics. A sample of this calibration dataset can be seen in figure A.4.

3.4 Calculating Metrics

From here, when a user records an exercise as part of a workout, metrics can be calculated. These can be tracked over time in order to measure process beyond weight and repetitions.

3.4.1 Range of motion

Range of motion is calculated using the following equation:

$$\text{ROM Score} = \left(\frac{\frac{D_{r1}+D_{r3}}{2}}{\frac{D_{c1}+D_{c3}}{2}} \right) \times 100 \tag{3.2}$$

where ROM Score is the Range of Motion score,
D_{c1} is the displacement magnitude for phase 1 in calibration data,
D_{c3} is the displacement magnitude for phase 3 in calibration data,
D_{r1} is the displacement magnitude for phase 1 in rep data,
D_{r3} is the displacement magnitude for phase 3 in rep data.

This scoring method rewards the user for achieving a range of motion close to the calibration dataset, and penalises them for over or under extending.

The code for this can be seen in C.8

3.4.2 Stability

The stability, or smoothness of motion is determined by fitting a 2nd order polynomial curve to the time series data of both accelerometer and gyroscope axes (ax, ay, az, gx, gy, gz), and then assessing the deviation of the actual data from this fitted curve. The score for each axis is normalized and aggregated to derive an overall smoothness score. I chose this method by evaluating data from many exercises, and noticing that a second order polynomial was a good fit for the curve of the data. The scoring equation is as follows:

$$S = \frac{1}{N} \sum_{i=1}^{N} \left(1 - \frac{\text{MSE}_i}{\text{Var}_i} \right) \times 100 \tag{3.3}$$

where S is the average smoothness score across all axes, N is the number of axes under consideration, MSE_i is the mean

The code for this can be seen in C.10

3.4.3 Balance

Balance compares the compound movement measurement across the two sensors, in order to determine the difference between them. This is useful for determining imbalance between left and right limbs in exercises such as the bench press to ensure that they are working in unison and evenly distributing the lifting load.

In order to get this score, the time series datasets are scaled between 0 and 100, and the average of the difference at all time points is used to provide the score. This can be written as:

$$S = \frac{1}{n}\sum_{i=1}^{n}\left|\left(\frac{x_i - \min(X)}{\max(X) - \min(X)} \times 100\right) - \left(\frac{y_i - \min(Y)}{\max(Y) - \min(Y)} \times 100\right)\right| \tag{3.4}$$

where S is the balance score,
n is the number of time points,
x_i is the value at time point i in dataset X,
y_i is the value at time point i in dataset Y,
$\min(X), \min(Y)$ are the minimum values in datasets X, Y respectively,
$\max(X), \max(Y)$ are the maximum values in datasets X, Y respectively.

The code for this can be seen in C.11

3.4.4 Force

A time series for the force is determined as below, using the recorded acceleration and the user inputted weight.

$$f = \frac{w}{g}a \tag{3.5}$$

where f is the force ,
w is the lifted weight ,
g is the gravitational constant, 9.81 ,
a is the acceleration,

From this, an average force, and maximum force can be determined. Force curves are also provided within the mobile app, giving users more insights into their lift.

The code for this can be seen in C.9

3.5 Cloud Computation

The processing for these metrics is performed using Firebase Cloud Functions. When data has been received by the mobile phone from the Arduino, a cloud function is called where metrics are calculated and the data is stored in the Firestore. This method was chosen initially because I had prototyped these methods within python and did not want to replicate this work by rewriting it in Flutter, however it also has the added benefit of improving application performance as these computations are offloaded.

Chapter 4

Mobile Application

I chose to spend a majority of my time to learn and apply mobile application development. This was shaped both by my targeted professional direction of becoming a software developer, and also by the inherent benefits to the user experience a mobile application delivers in this application.

4.1 UI/UX Design

I began by designing the applications front end within Figma. This allowed me to rapidly create visually appealing designs, and iterate on them until I achieved a user experience and visual style that I was happy with without the need to continuously rewrite code.

This process included mapping out the users journey between the different pages of the application, designing the components, and creating a layout that was intuitive and easy to navigate. A.5

From the front end design of the app, I determined the necessary back end functions needed for the user to interact with the database.

4.2 Technical Implementation

4.2.1 Flutter

Once I had finalised my UI and UX designs, I wrote the application within the Flutter framework. I initially found this a fun and efficient task due to the hot reloading features of Flutter that let developers see the impact of their code changes in real time on an emulated device. I made use of the Flutter Flow tool in order to rapidly develop the visual aspects of my application in a no code environment, and then exported the code and wrote the custom logic by hand. I found this workflow effective given the time constraints of the project, and found Flutter Flow an effective tool.

However, once I began interacting with the sensors, I found that I had to compile the APK for the application and upload it to my device every time I wanted to test my code. This greatly slowed down development, as it would take a few minutes to compile the code and upload it every time.

4.2.2 Firebase

Firebase was used as the backend service due to its ease of integration with Flutter using the FlutterFire library. Both Flutter and Firebase are developed by Google, and as such have been designed to work well together.

Firestore

Data is stored within a Firestore. Firestores are document based, NoSQL cloud storage options that provide fast and flexible storage of any type of data. The schema for my application can be seen below in figure A.6.

Authentication

In order to design the application with scaleability in mind, I began the development process by creating the account creation and log in flow. One of the major advantages of using Firebase are its tools for providing an authentication service, both with accounts localised to the application, and third party sign in providers allowing users to sign in with their existing Facebook, Google, Twitter account and more. For my purposes, I chose to use a Google Account as the sign in method due to personal preference, although future additions of other providers would be simple to implement.

Cloud Functions

Data processing was performed int he cloud in order to reduce the computational load on the mobile device and Arduino. I used the cloud functions tool within the Firebase suite in order to run do this. These are called within the application in order to process sensor data.

4.3 Communicating with the Arduino

In the development of the project, communicating between the Arduino and the mobile application was the major difficulty.

Pairing

Upon being powered, the devices enter a pairing mode, signified by the onboard RGB led flashing blue. The user can then navigate to the devices page within the app, where they can see a list of all available Bluetooth

devices, and the devices appear as 'Lift Sensor Left' and 'Lift Sensor Right'. When these are selected, they will be paired to the phone and the LED's will transition to solid blue.

Recording Data

When the user begins the exercise, by pressing the associated button on the mobile app, the phone broadcasts a "start" message to the Arduino which will start the data collection process. When the user has completed the exercise, pressing the stop button broadcasts the "stop" message, and the app listens for the data sent by the Arduino. Once this has been received, a cloud function is called in order to perform preprocessing on the data, calculate metrics and store the data and results within a 'reps' document, which is in turn related to its parent set. *(Refer to the database schema, A.6, for how this fits within the broader application.)*.

4.4 Data Representation

4.4.1 Exercise Graphs

For each exercise, graphs can be seen to view details about the specific exercise. This is broken down into sets, and then reps respectively. Sets metrics are the mean of rep metrics, and exercise metrics are the mean of set metrics.

This allows the user to breakdown their workout to answer questions such as:

- "*Did I achieve close to failure by having lower scores at the end of my set?*"
- "*Did I rest a sufficient amount of time between sets to ensure that I had recovered?*"

Examples of these graphs can be seen in figure **ADD REFERENCE HERE**.

4.4.2 Progress Graphs

Long term trends in the data can also be viewed within the application. This gives the user insights into their progress beyond traditional methods, and allows the user to answer questions such as:

- "*Am I increasing the weight too rapidly at the expense of poor form?*"
- "*Am I consistently paying attention to my form?*"

Examples of these graphs can be seen in figures A.7

Chapter 5

Conclusion

5.1 Evaluation

5.1.1 Accuracy

I found rep counting to be very accurate, with over 95 percent accuracy in my testing across exercises.

I found that when calculating metric data, the accuracy differed between types of exercise. Machine based exercises such as the leg press and pectoral fly gave the best performance - this is to be expected as the movement is restricted to a repeatable and linear fashion. Following this, linear free weight exercises such as barbell squats were fairly accurate, with scores aligning with how the exercise felt. I found that the worst performance was, as expected, in exercises where there was high variance in path between reps - for example, when performing pull-ups I attached the sensors to my ankles and found these to have high variation.

5.1.2 Software Performance

I found that the performance of the application was very smooth, with little delay or lag. Given the compute time of data processing within my python testing *(a few seconds)*, I assume that offloading the computation to a cloud function was beneficial in ensuring a smooth and enjoyable user experience.

After thorough bug testing and iteration, I found the application to work reliably and perform as expected when in use.

5.1.3 Security

Although the nature of the system is not particularly sensitive, system security was still considered. The implementation of Google Authentication provides industry standard protocols for authentication, and user data is secured using a role based access method such that data can only be requested by its associated user account.

5.1.4 Scalability

The solution was developed with scalability in mind. The use of cloud authentication and a robust database schema allows multiple users to install and use the app, each seeing only their own data. The use of cloud functions also makes the backend architecture more scalable, with compute power being adaptively provisioned by Google servers as and when it is needed.

5.2 Future Improvements

If I were to work further on the project, I would make the following improvements:

5.2.1 Device Housing

It would be beneficial to design a casing for the tracker. This would improve the durability of the device, protecting the Arduino from any damage that may occur from being thrown into a bag. This would also improve the visual aspect of the device and give it a more polished look. For personal purpose, a 3D printed case would be ideal due to the low cost and rapid speed of manufacture, although if this were to be manufactured on scale, then an injection moulded case would be more suitable due to the low cost and rate of manufacture at a production scale.

5.2.2 Automatic Pairing

It would improve the user experience if the application could remember the device so that it does not have to be manually paired every time.

5.2.3 Automatic Exercise Classification

Using machine learning techniques, each exercise could be automatically classified. This would allow the user to workout without needing to use their phone at all, and have all exercises automatically logged to view later.

5.2.4 Real Time Feedback

Instead of only providing insights after the exercise has been completed, it would be beneficial to show metrics in real time. This would be best achieved with on device computation, and as such the python functions would need to be converted to Flutter code and optimisation would need to occur. Metrics with low computational strain such as force would be best suited to this.

5.2.5 Form Mistake Classification

Machine learning could be applied in order to estimate the causes of the device straying from its calibration path - for example, if it is rotating too much during a squat, the back is likely rounding. This could be used to provide real time guidance using a text to speech engine such as Eleven Labs Generative Voice AI in order to guide the user on how to improve on mid-exercise.

However, the feasibility of achieving accurate predictions using my current tracking methods is questionable, and would require further investigation and potentially the use of additional sensors or image based methods such as pose estimation through cameras.

5.2.6 Social Integration

Similar to other applications such as Strava or Hevy, a social integration feature that lets users view their friends activity and progress would also be a useful feature, giving further motivation to users in the gym by holding them accountable to their friends.

Appendix A

Figures

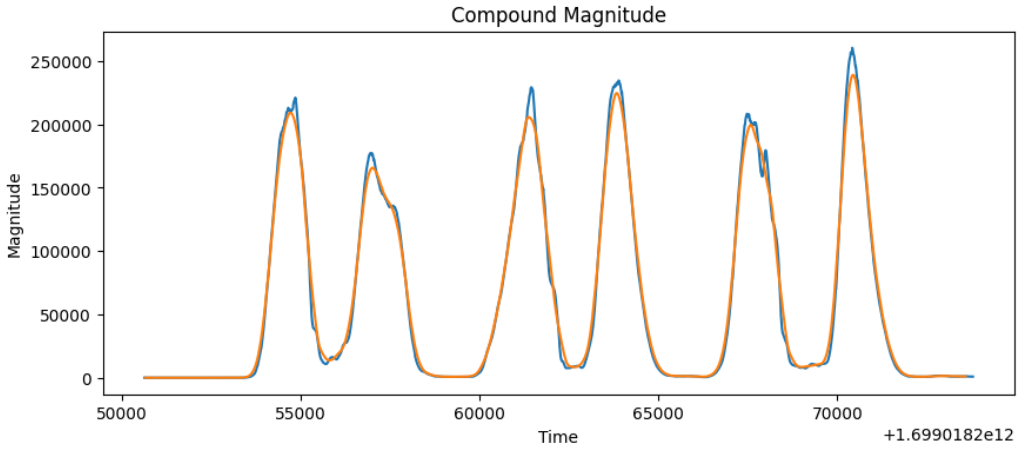

Figure A.1: Graph of Compound Magnitude

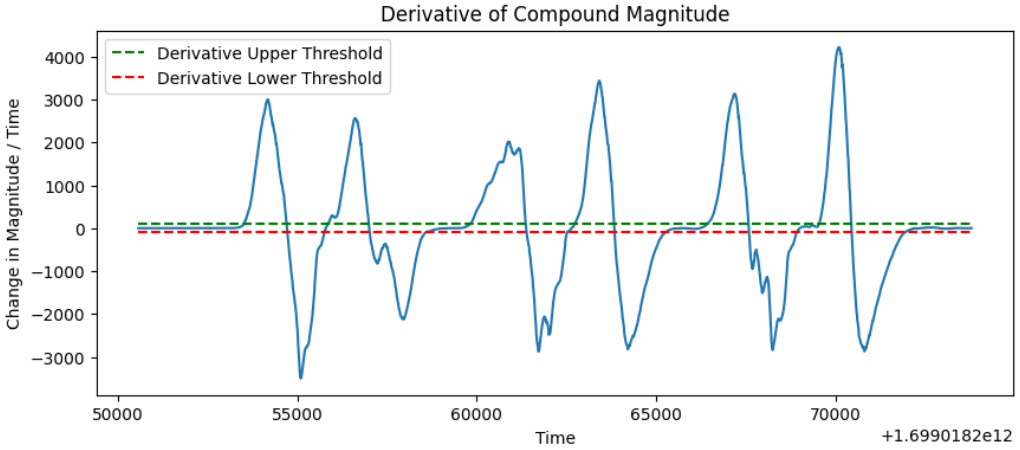

Figure A.2: Graph of the derivative of the compound magnitude

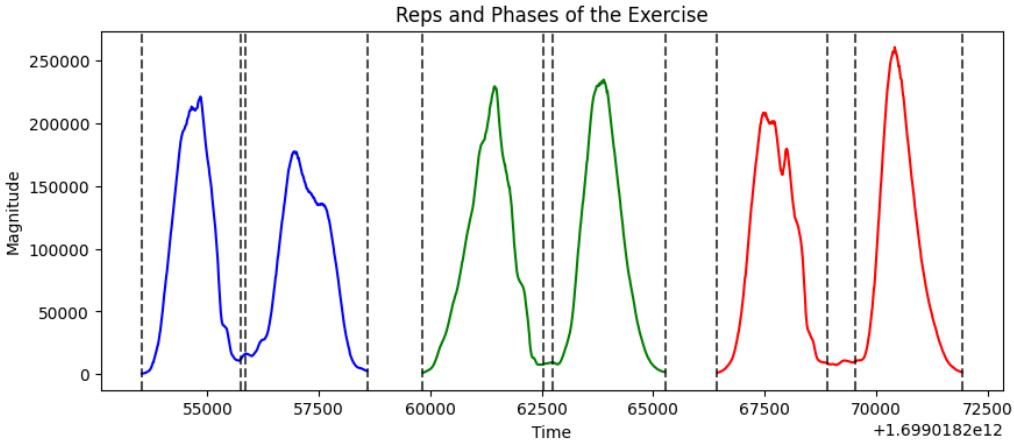

Figure A.3: Graph showing the breakdown of the reps and phases within a set

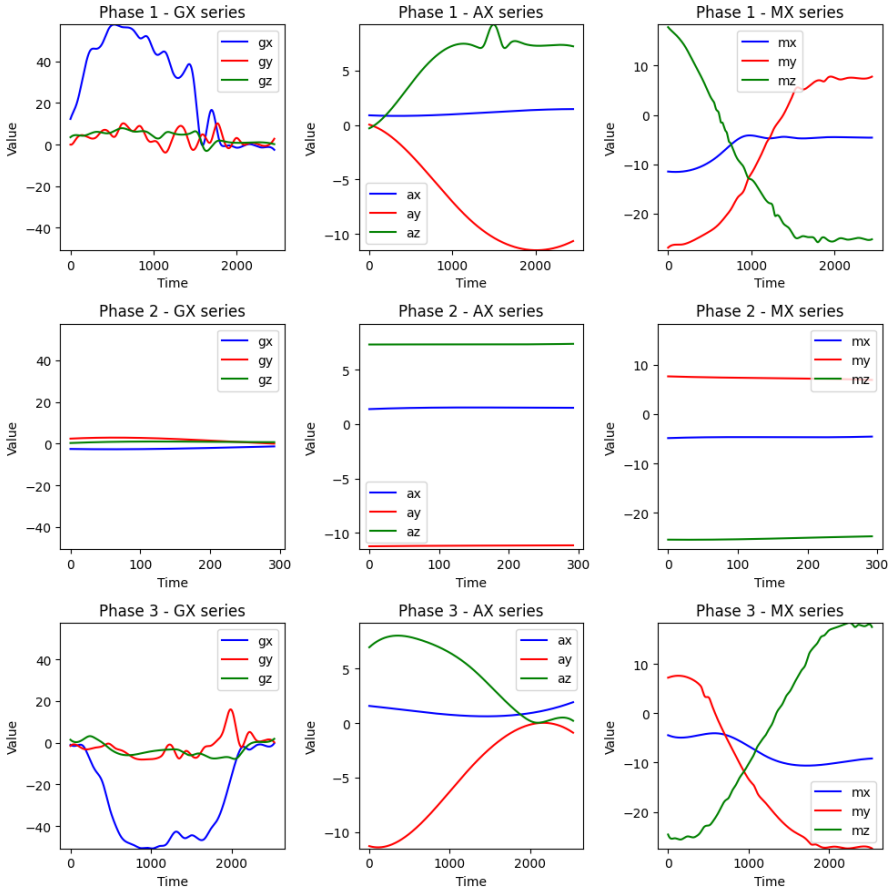

Figure A.4: Sample Calibration Dataset

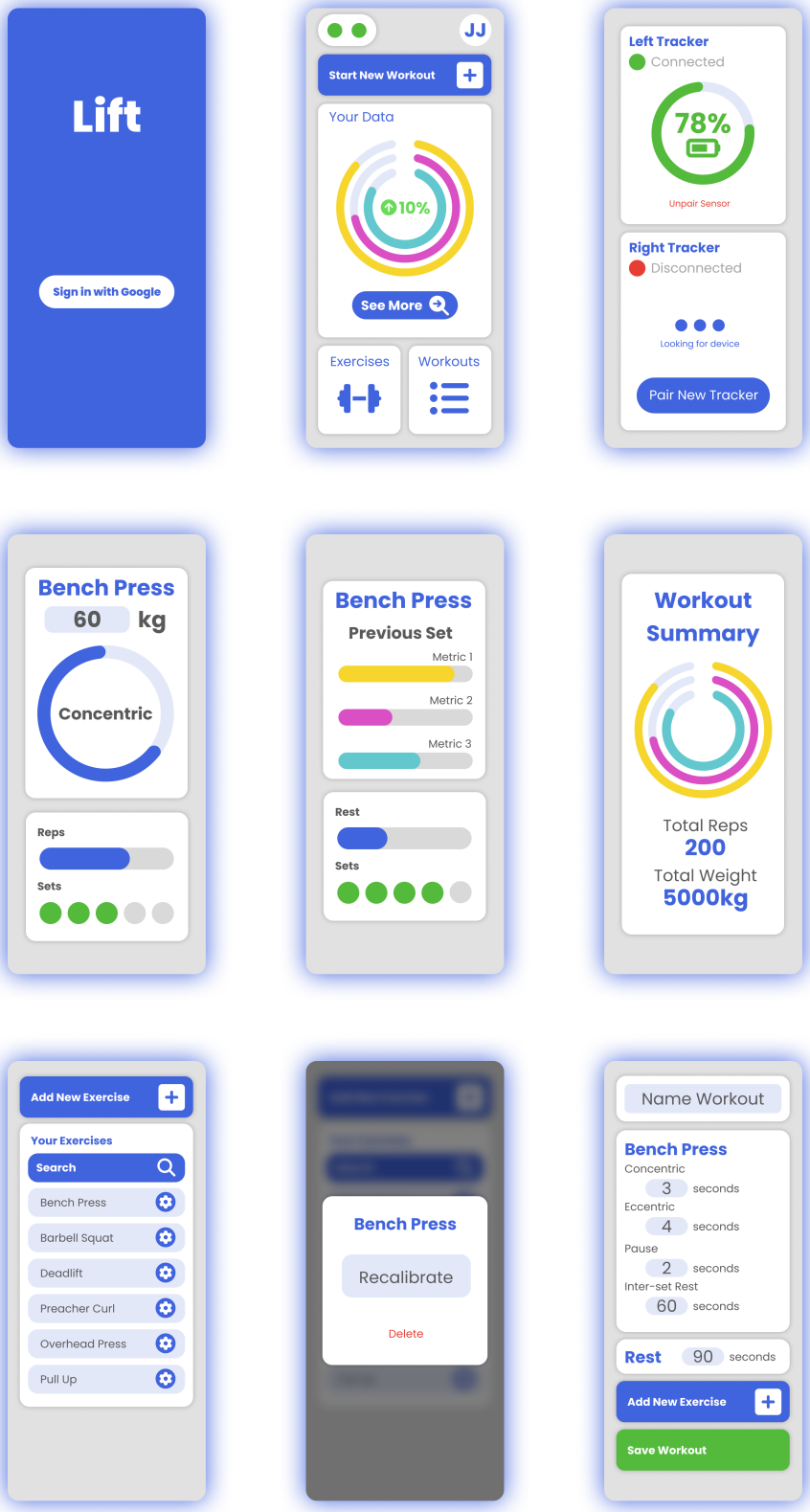

Figure A.5: Extract of Initial Figma Prototypes

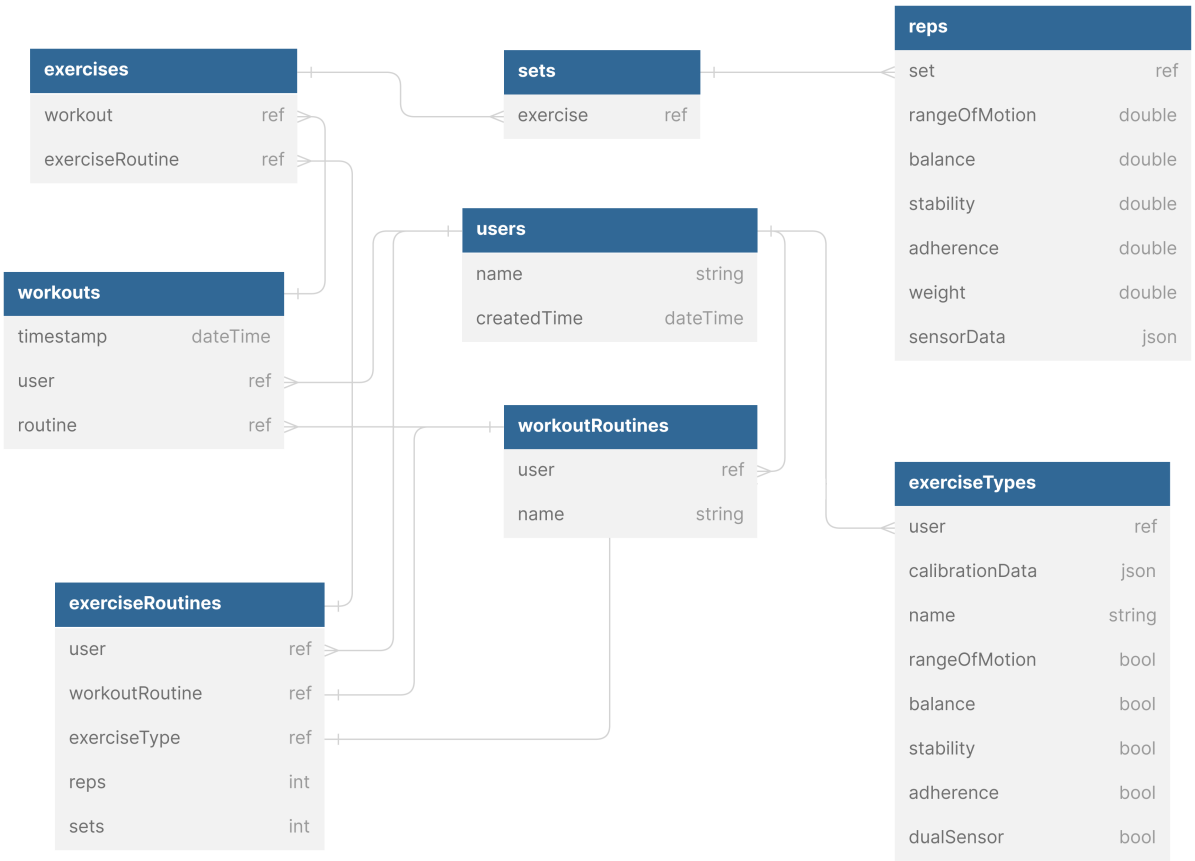

Figure A.6: Database Schema

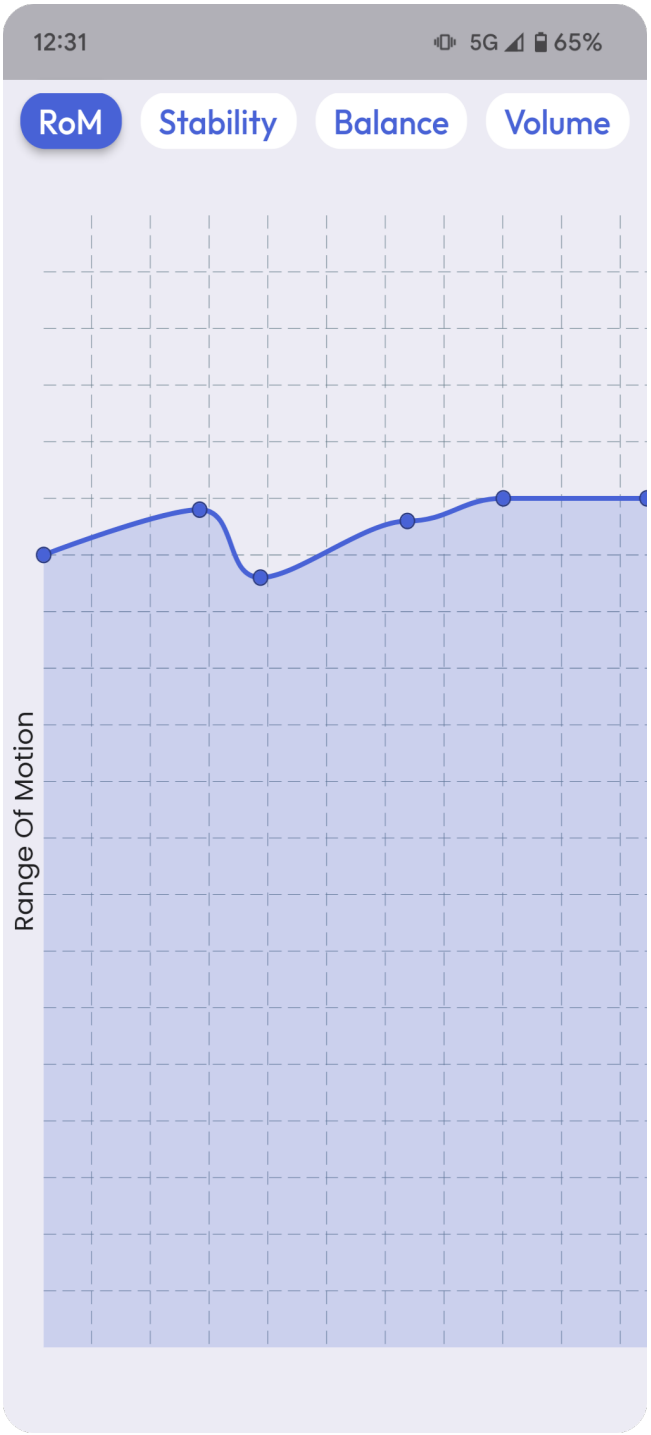

Figure A.7: Range of Motion Graph

Appendix B

Tables

LED Color	On/Off Pattern	Meaning
Red	Blink	Indicates a failure in initializing the IMU or BLE.
Green	On (Steady)	Signals the start of data collection from the IMU.
Blue	On (Steady)	Signals the stop of data collection and the start of data transmission.
Green	Blink	Indicates the completion of the data transmission process.

Table B.1: Meaning of RGB LED modes

Appendix C

Code

```
void setLEDColor(int color, bool on) {
    digitalWrite(LED_RED, color == COLOR_RED && on ? LOW : HIGH);
    digitalWrite(LED_GREEN, color == COLOR_GREEN && on ? LOW : HIGH);
    digitalWrite(LED_BLUE, color == COLOR_BLUE && on ? LOW : HIGH);
}

void blinkLED(int color, int interval, int duration) {
    long startTime = millis();
    while (millis() - startTime < duration) {
        setLEDColor(color, true);
        delay(interval);
        setLEDColor(color, false);
        delay(interval);
    }
}
```

Listing C.1: LED Control Code

```
if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    blinkLED(COLOR_RED, 500, 5000);
    return;
} else {
    Serial.println("IMU initialized.");
}
```

Listing C.2: IMU Initialization Code

```
if (collecting && dataIndex < MAX_DATA_POINTS) {
    if (IMU.accelerationAvailable() && IMU.gyroscopeAvailable() && IMU.magneticFieldAvailable
        ()) {
        IMUData currentData;
        currentData.timestamp = millis();
        IMU.readAcceleration(currentData.ax, currentData.ay, currentData.az);
        IMU.readGyroscope(currentData.gx, currentData.gy, currentData.gz);
        IMU.readMagneticField(currentData.mx, currentData.my, currentData.mz);
        data[dataIndex++] = currentData;
    }
}
```

Listing C.3: IMU Reading Code

```
if (inputValue == "start") {
            setLEDColor(COLOR_GREEN, true);
            imuReader.start();
        } else if (inputValue == "stop") {
            setLEDColor(COLOR_BLUE, true);
            blinkLED(COLOR_GREEN, 100, 1000);
            imuReader.stop();

            auto dataSize = imuReader.getDataSize();
            for (int i = 0; i < dataSize; i++) {
                const auto& d = imuReader.getData()[i];
                String dataString = String(d.timestamp) + "," + String(d.ax) + "," + String(d.
                    ay) + "," + String(d.az) + "," + String(d.gx) + "," + String(d.gy) + "," +
                     String(d.gz) + "," + String(d.mx) + "," + String(d.my) + "," + String(d.
                    mz);
                outputChar.writeValue(dataString.c_str());
                delay(100);
            }
            setLEDColor(COLOR_BLUE, false);
        }
```

Listing C.4: IMU Sending Code

```
def filter_noise(self, df, fc=0.1):
    # Create a low-pass Butterworth filter
    b, a = butter(10, fc, btype='low')

    # Apply the filter to the accelerometer and gyroscope data
    df.iloc[:, 3:6] = filtfilt(b, a, df.iloc[:, 3:6], axis=0)  # Accelerometer data
    df.iloc[:, 0:3] = filtfilt(b, a, df.iloc[:, 0:3], axis=0)  # Gyroscope data
    df.iloc[:, 6:9] = filtfilt(b, a, df.iloc[:, 6:9], axis=0)  # Magnetometer data

    return df
```

Listing C.5: Butterworth Filtering

```
def handle_outliers(self, df, threshold=3):
    # Columns for accelerometer, gyroscope, and magnetometer data
    sensor_columns = ['ax', 'ay', 'az', 'gx', 'gy', 'gz', 'mx', 'my', 'mz']

    # Calculate mean and standard deviation for each sensor data column
    mean = df[sensor_columns].mean()
    std_dev = df[sensor_columns].std()

    # Calculate z-scores for each sensor data column
    z_scores = (df[sensor_columns] - mean) / std_dev

    # Filter out data points where any of the z-scores exceed the threshold
    return df[(z_scores.abs() < threshold).all(axis=1)]
```

Listing C.6: Removing Outliers

```
    def transformToUpright(self, data):
        # Calculate the magnitude of the accelerometer data
        data['a_magnitude'] = np.sqrt(
            data['ax']**2 + data['ay']**2 + data['az']**2)

        # Find the index of the row where the accelerometer magnitude is closest to 9.81 m/s^2
        upright_idx = (data['a_magnitude'] - 9.81).abs().idxmin()
        acc_reading = data.loc[upright_idx, ['ax', 'ay', 'az']]

        # Calculate the rotation needed
        current_orientation = acc_reading / np.linalg.norm(acc_reading)
        desired_orientation = np.array([0, 0, 1])
        rotation = R.align_vectors([desired_orientation], [
                                   current_orientation])[0]

        # Apply the rotation to the accelerometer, gyroscope, and magnetometer data
        for sensor_data in [['ax', 'ay', 'az'], ['gx', 'gy', 'gz'], ['mx', 'my', 'mz']]:
            data[sensor_data] = rotation.apply(data[sensor_data])

        # Drop the auxiliary magnitude column
        data.drop('a_magnitude', axis=1, inplace=True)

        return data
```

Listing C.7: Transforming to upright

```
    def calculateRangeOfMotionScore(self, repData, plot=False):
        calibrationPhase1Displacement = self.calculate_displacement(
            self.callibrationData[0], title="Calibration Phase 1")
        calibrationPhase3Displacement = self.calculate_displacement(
            self.callibrationData[2], title="calibration Phase 3")

        calibrationPhase1DisplacementMagnitude = np.linalg.norm(
            calibrationPhase1Displacement)
        calibrationPhase3DisplacementMagnitude = np.linalg.norm(
            calibrationPhase3Displacement)

        averageCalibrationDisplacement = (
            calibrationPhase1DisplacementMagnitude + calibrationPhase3DisplacementMagnitude) /
                2

        repPhase1Displacement = self.calculate_displacement(
            repData[0], title="Rep Phase 1")
        repPhase3Displacement = self.calculate_displacement(
            repData[2], title="Rep Phase 3")
```

```
        repPhase1DisplacementMagnitude = np.linalg.norm(
            repPhase1Displacement)
        repPhase3DisplacementMagnitude = np.linalg.norm(
            repPhase3Displacement)

        averageRepDisplacement = (
            repPhase1DisplacementMagnitude + repPhase3DisplacementMagnitude) / 2


        score = (averageRepDisplacement / averageCalibrationDisplacement) * 100

        return score

    def calculate_displacement(self, data, title="", plot=False):

        time_seconds = (data.index - data.index[0]) / 1000

        accelerations = []
        velocities = []
        displacements = []

        for axis in ['ax', 'ay', 'az']:
            acceleration = data[axis].values
            accelerations.append(acceleration)

            velocity = cumtrapz(acceleration, time_seconds, initial=0)
            velocities.append(velocity)

            displacement = cumtrapz(velocity, time_seconds, initial=0)
            displacements.append(displacement)

        displacement_array = np.vstack(displacements).T

        maxDisp = np.max(displacement_array, axis=0)
        minDisp = np.min(displacement_array, axis=0)
        return maxDisp - minDisp
```

Listing C.8: Range of Motion

```
    def acceleration_magnitude(self, data):
        return np.sqrt(data[:, 0]**2 + data[:, 1]**2 + data[:, 2]**2)

    def averageForce(self, data, weight):
        # Calculate the magnitude of the acceleration vector
        acceleration_magnitude = self.acceleration_magnitude(data)

        # Calculate the force (F = ma)
        force_data = weight * acceleration_magnitude

        # Calculate and return the average force
        average_force = force_data.mean()
        return average_force

    def maxForce(self, data, weight):
        # Calculate the magnitude of the acceleration vector
        acceleration_magnitude = self.acceleration_magnitude(data)

        # Calculate the force (F = ma)
        force_data = weight * acceleration_magnitude

        # Calculate and return the maximum force
        max_force = force_data.max()
        return max_force
```

Listing C.9: Calculating Force

```
    def calculateStabilityScore(self, data, axes=['ax', 'ay', 'az', 'gx', 'gy', 'gz'], plot=
        False):
        scores = []

            for axis in axes:
                y_data = data[axis].values
                x_data = np.arange(len(y_data))
                p = Polynomial.fit(x_data, y_data, 2)
                fitted_values = p(x_data)
```

```
                mse = mean_squared_error(y_data, fitted_values)
                max_mse = np.var(y_data)
                normalized_mse = mse / max_mse
                score = (1 - normalized_mse) * 100
                scores.append(score)

                if plot:
                    plt.figure(figsize=(10, 4))
                    plt.plot(y_data, label='Actual Data')
                    plt.plot(fitted_values, label='Fitted Curve')
                    plt.title(f'Actual Data vs Fitted Curve - {axis}')
                    plt.xlabel('Time')
                    plt.ylabel(axis)
                    plt.legend()
                    plt.show()

            average_score = np.mean(scores)
            return average_score
```

Listing C.10: Calculating Stability

```
    def scaleData(self, data):
        scaled_data = 100 * (data - data.min()) / (data.max() - data.min())
        return scaled_data

    def calculateBalanceScore(self, df1, df2):
        # Ensure that both dataframes have the same columns and length
        if df1.shape != df2.shape or not df1.columns.equals(df2.columns):
            raise ValueError("Dataframes must have the same shape and columns")

        # Scale both dataframes
        scaled_df1 = self.scaleData(df1)
        scaled_df2 = self.scaleData(df2)

        # Calculate the absolute differences
        differences = abs(scaled_df1 - scaled_df2)

        # Calculate the average of these differences
        balance_score = differences.mean().mean()

        return balance_score
```

Listing C.11: Calculating Balance