**Project Title:** Real-time Exercise Coaching and Range of Motion Tracking using RTOS

**Authors:** Ibrahim Binmahfood & Robert Wilcox

**Date:** 3.21.2024

**Affiliation:** PSU ECE course 544

## Abstract

**Problem Statement:** Lack of real-time feedback on range of motion (ROM) during exercise can lead to improper form, reduced training efficiency, and increased risk of injury. Current form tracking solutions often rely heavily on visual feedback, which can be sub-optimal in a gym setting.

**Proposed Solution:** We aimed to develop a wearable, FreeRTOS-based device using IMU sensors and Bluetooth for real-time ROM monitoring and feedback. Due to technical challenges, we shifted our focus to creating a prototype demonstrating the concept's core value.

**Challenges and Pivot:** We encountered significant noise in the IMU's linear acceleration data, hindering accurate position estimation. To demonstrate our application's functionality, we pivoted to using simulated sensor data. We successfully developed a system that records a warm-up set to establish baseline ROM and provides real-time feedback if the user deviates from this range during working sets.

**Key Findings:** While current AHRS sensors might not provide the precision needed for accurate position tracking in this specific use case, our work highlights the potential of such a device to enhance exercise form. Additionally, we showcase the versatility of microcontrollers like the ESP32 for overcoming project challenges.

**Future Work:** We plan to investigate sensor fusion techniques, filtering algorithms, and other potential sensor inputs to increase positional accuracy. The prototype lays a foundation for further development of a commercially viable real-time ROM tracking device that could significantly improve exercise safety and effectiveness.

## Introduction

Proper exercise form is crucial for achieving optimal fitness results and minimizing injury risk. Maintaining a consistent and correct range of motion (ROM) throughout exercise sets is a key aspect of proper form. However, without real-time feedback, individuals may unknowingly deviate from their intended ROM, leading to reduced training effectiveness or even injuries. Existing form tracking systems frequently rely on visual aids or the expertise of coaches, which can be impractical or inaccessible in many workout settings.

This project explores the potential of real-time ROM tracking through a wearable device powered by a Real-Time Operating System (RTOS). We utilize inertial measurement unit (IMU) sensors with the aim of monitoring an individual's ROM during exercise.  While our initial goal was to provide instantaneous feedback to maintain optimal ROM, we faced challenges in obtaining accurate position data from the IMU sensors. We pivoted our approach, focusing on the following revised goals:

- **Proof-of-Concept Development:** Create a system demonstrating the potential of real-time ROM feedback, even with the use of simulated data.
- **Limitations Investigation:** Explore the challenges of using IMU sensors for accurate position and ROM determination in the context of exercise tracking.

## Project Overview

This project focuses on the design and development of a system exploring real-time exercise form monitoring. It combines a wearable device with IMU sensors and an FPGA running a Real-Time Operating System (RTOS).  While the primary goal was to track ROM using sensor data, technical challenges led to a shift in focus. We aimed to demonstrate the concept's value using simulated data and to investigate the complexities of sensor-based motion tracking.

The project highlights the link between proper exercise form and optimal results (Nippard, n.d.; Renaissance Periodization, n.d.; Team Full ROM, n.d.; Schoenfeld et al., 2021). It  addresses a gap in the market by  investigating  mechanisms for ROM feedback that go beyond  general health metrics in fitness trackers or impractical visual-based assessments.

## Objectives Addressed

- **Real-time ROM Concept:** Develop a system that records a warm-up ROM and provides feedback if the user deviates from this range during working sets, using simulated data to demonstrate functionality.
- **Sensor Challenges:** Investigate limitations of IMU sensors for accurate position estimation and ROM tracking in exercise scenarios.

## System Overview

The exercise monitoring system is composed of two primary units:

- **Wearable Unit:** A compact, battery-powered assembly containing an ESP32 microcontroller, an Adafruit BNO055 AHRS sensor, and supporting circuitry on a breadboard. This unit acquires motion data and transmits it wirelessly via Bluetooth Low Energy (BLE).
- **FPGA Processing Unit:** A RealDigital BooleanBoard FPGA development board running a MicroBlaze soft-processor system designed in Xilinx Vivado. This unit receives data over a wired UART connection from a secondary ESP32 microcontroller (used due to challenges with the board's onboard BLE module), processes it, and provides ROM feedback.

## Block Diagram

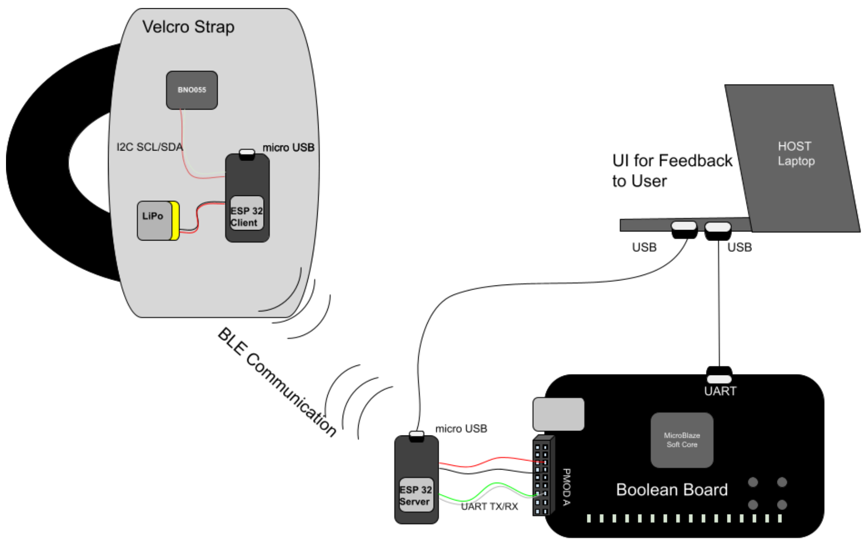


Fig. 1

- **Wearable Unit**
  - ESP32 Microcontroller
  - Adafruit BNO055 AHRS Sensor (with I2C connection)
  - LiPo Battery (3.7V 2000mAh 7.4Wh)
- **FPGA Processing Unit**
  - RealDigital BooleanBoard FPGA
    - MicroBlaze Soft Processor
    - N4IO Module

        - 2 AXI UARTLite IP Blocks (one for terminal I/O, one for external ESP32 communication)
    - External ESP32 Microcontroller (with PMOD connection)

## Component Explanations

- **FPGAs (BooleanBoard):** The core of the processing unit. The FPGA's flexibility allows the implementation of the MicroBlaze processor, I/O interfaces, and custom logic for ROM tracking algorithms.
- **MicroBlaze Processor:** A soft-core processor implemented within the FPGA fabric, providing a platform for software execution and system control.
- **ESP32 Microcontrollers:** Low-power microcontrollers with integrated Bluetooth capabilities. One ESP32 handles sensor communication and data preprocessing on the wearable unit, while the second acts as a bridge between the BLE interface and the FPGA's UART.
- **Adafruit BNO055 Sensor:** An AHRS sensor providing orientation and acceleration data, critical for the initial goal of position estimation.
- **UART Communication:** Serial communication protocol used for data exchange between the FPGA and the external ESP32.
- **Bluetooth Low Energy (BLE):** Wireless protocol enabling communication between the wearable unit and the FPGA processing unit.

## Circuit Functionality

1. **Sensor Data Acquisition (Wearable):** The BNO055 sensor continuously provides orientation and acceleration data to the ESP32 on the wearable unit.
2. **Data Preprocessing (Wearable):** Initial attempts involved the ESP32 performing sensor fusion and attempting to estimate position. When noise proved to be a challenge, data spoofing functions were added here for demonstration purposes.
3. **Wireless Transmission (Wearable):** The preprocessed sensor data (or spoofed data) is transmitted wirelessly over BLE to the secondary ESP32.
4. **Data Reception (FPGA):** The secondary ESP32 receives the BLE data and forwards it to the FPGA via the UART connection.
5. **Data Processing (FPGA):** The MicroBlaze processor executes ROM comparison algorithms (using current values against a recorded warm-up set) to determine if the user deviates from their intended range of motion.
6. **Feedback Generation (FPGA – Conceptual):** The original design intended for feedback mechanisms (audio, visual, etc.) to be triggered on the FPGA, alerting the user to ROM deviations.

## Hardware Design

- **Wearable Unit**

    - **Adafruit Feather ESP32 Microcontroller:** Provides Bluetooth communication, sensor data preprocessing (or spoofing), and a compact form factor for wearable applications.
    - **Adafruit BNO055 AHRS Sensor:** Offers pre-calibrated sensor fusion for orientation and acceleration data.
    - **Power:** LiPo battery for portability and extended runtime.
- **FPGA Processing Unit**
    - **RealDigital BooleanBoard:** Flexible platform with MicroBlaze soft-core processor and configurable I/O.
    - **External Adafruit Feather ESP32 Microcontroller:** Facilitates BLE communication workaround and relays preprocessed (or spoofed) data to the FPGA.

## Software Design

- **FreeRTOS:** Real-time operating system for task management, synchronization, and responsiveness; crucial for handling preprocessed sensor data and potential feedback on the FPGA side.
- **Key Software Components**
    - **Sensor Data Preprocessing/Spoofing Task (Wearable ESP32):** Acquires raw data from the BNO055 sensor, performs preprocessing (or generates spoofed data for demonstration), and sends it via BLE.
        - The program to acquire the data from the sensor was originally set up to get quaternion, linear acceleration, and gravity data from the sensor and transmit that to the FPGA. The FPGA would then be responsible for calculating the position based on this data. However, the delays in the transmission of the data caused an extra level of error in the position calculation, so the position calculation step was moved to the ESP32. This way, the ESP32 could read the raw data from the sensor, process it into a position, and send the position to the FPGA.
        - A lot of work was put into the data processing functions. The BNO055 sensor has built in fusion algorithms, which take readings from the IMU's several different sensors and give orientation (in quaternions or Euler angles), linear acceleration data, and gravitational acceleration data. To estimate position from this data, first we must rotate the sensor's data into our reference frame. We chose to use quaternions for this purpose, as Euler angles can suffer from a phenomenon known as gimbal lock. The quaternion data is essentially a representation of how the sensor's reference frame relates to our own. Using some matrix transformations, we can rotate the sensor data from its reference frame into our own. This makes the data "easy" to operate on and calculate position from.

```
// Function to convert quaternion orientation data into a 3x3 rotation matrix.
// This matrix can then be used to rotate vectors from the local sensor frame to the global
frame.
```

```
void quaternionToRotationMatrix(myQuaternion q, float R[3][3]) {
    float qw = q.w, qx = q.x, qy = q.y, qz = q.z; // Extract quaternion components for easier
usage.

    // The following computations are based on quaternion-to-rotation matrix equations.
    // The resulting matrix will be used to rotate the sensor's reference frame into our
reference frame,
    // allowing us to get x, y, and z data relative to gravity instead of relative to the
sensor's axes
    R[0][0] = 1 - 2*qy*qy - 2*qz*qz;
    R[0][1] = 2*qx*qy - 2*qz*qw;
    R[0][2] = 2*qx*qz + 2*qy*qw;

    R[1][0] = 2*qx*qy + 2*qz*qw;
    R[1][1] = 1 - 2*qx*qx - 2*qz*qz;
    R[1][2] = 2*qy*qz - 2*qx*qw;

    R[2][0] = 2*qx*qz - 2*qy*qw;
    R[2][1] = 2*qy*qz + 2*qx*qw;
    R[2][2] = 1 - 2*qx*qx - 2*qy*qy;
}
```

- After the sensor data is multiplied by this rotation matrix, it is in our reference frame. We can then multiply the linear acceleration in each cardinal direction by the time passed between each measurement to get the instantaneous velocity. This velocity can be integrated again to get position. This type of position estimation is known as “dead reckoning,” because it is the position relative to a starting place in 3D space. Dead reckoning is quite challenging, and is susceptible to drift due to unstable sensor data. We found the drift to be quite extreme in our case. While the orientation data was fairly stable, the linear acceleration data from our sensor was very noisy, causing our position readings to drift all over the place, and make extreme jumps. A low pass filter was attempted in order to smooth out the linear acceleration data, and while it helped, it did not fully solve our issues.

```
 // Low-pass filter applied to 3D vector data
LinearAcceleration applyLowPassFilter(LinearAcceleration currentData, LinearAcceleration
previousFilteredData, float alpha) {
    LinearAcceleration filteredData;
    filteredData.linAccX = alpha * previousFilteredData.linAccX + (1 - alpha) *
currentData.linAccX;
    filteredData.linAccY = alpha * previousFilteredData.linAccY + (1 - alpha) *
currentData.linAccY;
    filteredData.linAccZ = alpha * previousFilteredData.linAccZ + (1 - alpha) *
currentData.linAccZ;
    return filteredData;
}
```

- We also implemented functionality to cut off any linear acceleration values beneath a certain threshold, and decide they were negligible. This helped us stop the sensor from drifting out of control after moving it.
- After various different filtering attempts, we tried a Kalman filter. This is a more advanced mathematical filter which uses a state transition system to estimate the next state based on various tuning parameters. Kalman filters work best when you have real input data you can feed back into the system in order to train and correct it. We did not have any way to do this, so we were feeding back in estimated position data, which makes the filter very difficult to tune. We got the sensor to stop drifting so much, and to have more stable measurements, but it seems in doing so it is very unresponsive now. We do not see an accurate translation of the sensor's position into estimated position.
- After all of this, we decided to abandon attempts to get the sensor to work and spoof the data. The ESP32 would now be responsible for generating different types of data: warmup data and working set data. The warm up data would follow regular and smooth intervals, and the working set data would deviate as time increases, allowing us to demonstrate the corrective features of our application.
    - Both of these data sets were generated using trig functions, as they model periodic movement well. A small random element was added to the frequency and amplitude to simulate real conditions where every rep isn't identical. For the warmup data:
    -

```
case SQUAT:
            amplitudeX = 8.0 * randomAmplitudeFactor; // Max deviation in cm, forward/backward
            amplitudeY = 4.0 * randomAmplitudeFactor; // Max deviation in cm, left/right
            amplitudeZ = 33.0 * randomAmplitudeFactor; // Half range for movement from +1 to -65
cm
            frequencyX = frequencyY = frequencyZ = 0.25 * randomFrequencyFactor; // Assuming a
complete squat takes about 4 seconds

            // Simulating slight movements in x and y directions
            pos.x = amplitudeX * sin(TWO_PI * frequencyX * t);
            pos.y = amplitudeY * sin(TWO_PI * frequencyY * t);

            // For z-axis: simulate standing up to squatting down and back up
            // The range is adjusted from +1 to -65 cm, which is a total movement of 66 cm.
            // Since cos(0) = 1, starting value of pos.z should be +1 cm when t=0.
            // To achieve this, we adjust the cos wave to oscillate between 0 (standing) to -66
(squatting down),
            // then we add +1 to shift the entire range up by 1 cm.
            pos.z = -amplitudeZ * cos(TWO_PI * frequencyZ * t) + 1; // Correct formula


            // Velocity calculation as the derivative of position
            vel.x = amplitudeX * TWO_PI * frequencyX * cos(TWO_PI * frequencyX * t);
            vel.y = amplitudeY * TWO_PI * frequencyY * cos(TWO_PI * frequencyY * t);
            vel.z = amplitudeZ * TWO_PI * frequencyZ * sin(TWO_PI * frequencyZ * t);
            break;
```

- And for the working set data factors which increase the variability with time were added to force our data to deviate with time:

```
// Fatigue impact: As time increases, increase both the amplitude and variability
    float timeBasedIncreaseFactor = min(t / 60.0, 1.0); // Caps at 100% increase at or beyond 1
minute
    float variabilityAmplitude = baseVariabilityAmplitude * (1.0 + timeBasedIncreaseFactor); //
Increases up to 10%
    float variabilityFrequency = baseVariabilityFrequency * (1.0 + timeBasedIncreaseFactor); //
Increases up to 10%
    float amplitudeIncreaseFactor = 1.0 + timeBasedIncreaseFactor; // Amplitude can double

    // Adjust amplitude and frequency based on exercise and time-based fatigue simulation
    switch (exercise) {
        case BENCH_PRESS:
            amplitudeX = 17.0 * amplitudeIncreaseFactor; // Example: average of initial range,
then increase
            amplitudeY = 4.0 * amplitudeIncreaseFactor;
            amplitudeZ = 32.5 * amplitudeIncreaseFactor;
            frequencyX = frequencyY = frequencyZ = 0.33; // Keeping base frequency, could vary if
desired
            break;
```

    - The ESP32 then sends the calculated position data over BLE. The Bytes 0x02 and 0x03 are appended to the front and read of the packets as delimiters to indicate packet boundaries. A simple XOR checksum of the bits in the packet is also done, and stored in the second to last byte. This will allow us to check the packet integrity once it is decoded on the FPGA.
    - The ESP32 has several control words which control the data output. Initially, the ESP32 will not transmit data, but simply establish the BLE connection. Once it receives the START command, it will start generating data and sending it over BLE. If at any time it receives STOP, it will stop. WARMUP and WORKING will switch to generating those datasets, respectively. SQUAT, BENCH_PRESS, or BICEP_CURL will change the data being sent over BLE to one of those datasets. The spoofed data is different to represent the real motion of these 3 exercises.

- **BLE Communication Tasks (Both ESP32s):** Manage reliable data exchange between the wearable ESP32 and the intermediary ESP32.
    - The second ESP32 is responsible for forwarding any data received over BLE to the FPGA, as well as forwarding data from the FPGA over BLE. This was relatively simple to accomplish by setting up a client BLE server

on the ESP32. Arduino IDE provides several BLE libraries which have extensive functionality. For our program we first establish a client server, and then scan for any available BLE devices. Each of the BLE devices scanned is compared to a known device name for our other ESP32. When the name is found, the ESP32s are connected to the same client server, allowing transfers between both devices.

```
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {
  void onResult(BLEAdvertisedDevice advertisedDevice) {
    Serial.print("BLE Device found: ");
    Serial.println(advertisedDevice.toString().c_str());

    // Check if this is the device we want to connect to
    if (advertisedDevice.getName() == myDeviceName) {
      Serial.println("Device found. Connecting!");
      BLEDevice::getScan()->stop();
      pServerAddress = new BLEAddress(advertisedDevice.getAddress());
      doConnect = true;
      doScan = true;
    }
  }
};
```

- ■ The data received/sent by this ESP32 is forwarded directly to the UART pins on the Adafruit feather board, which are in turn connected to the PMOD pins on our FPGA.

- ○ **Data Reception Task (FPGA):** Receives the preprocessed (or spoofed) sensor data via the UART interface.
  - ■ Reading the data on the FPGA consists of several different programs which facilitate packet transfer and ensure data integrity. The first is esp32_BLE_uart.c, which simply provides an interface to send or receive a single byte from the UART. The UART base address in the file is tied to the appropriate PMOD pins in the constraints file for our system.
  - ■ The next program in the data reception pipeline is packetRader.c, which uses the functions in esp32_BLE_uart.c to read a packet from the ESP32. The function receives bytes from the UART until it detects the start delimiter, at which point it starts collecting and storing the data in an array. Once the end delimiter is detected, the packet checksum is calculated and compared to the checksum bit. If the checksum is valid, the packet is copied into a variable which holds the most recent valid packet. This way, the read packet function can be called, but we will be able to get data regardless of whether or not the read was successful (it’ll just be old data).

```
// Main function to read and process packets
bool readPacket() {
    uint8_t byte;
    while (receiveByteFromESP32(&byte)) {
        if (processReceivedByte(byte)) {
            // Packet received, validate checksum
            if (validateChecksum()) {
                // Valid packet received

                memcpy(lastValidPacket, packetBuffer, packetIndex);
                lastValidPacketLength = packetIndex;

                //xil_printf("Valid packet received\r\n");
                return true;
            } else {
                // Checksum mismatch
                xil_printf("Checksum mismatch\r\n");
                resetPacketCollection();
                return false;
            }
        }
    }
    return false; // No complete packet received yet
}
```

- The processPacket function serves as a way to get the data into a struct which contains both the velocity and position data. Each packet is limited to 16 bytes by the UART, so we could not send both the position and velocity data in the same packet. The processPacket function, through the updatePandV.c file, allows us to copy the most recent valid packets into the velocity and data fields of a global struct, which will be used for getting data by our main program. Process packet deserializes the data and copies it into the struct elements.

```
void processPacket(Velocity * velocity, Position* position) {
    uint8_t length;
    const uint8_t* packet = getLastValidPacket(&length);
    char packetType = getPacketType();

    switch(packetType) {
        case PACKET_TYPE_VELOCITY: {
            // Deserialize velocity
            if(length >= 3 + sizeof(Velocity)) {
                memcpy(velocity, packet + 2, sizeof(Velocity));
            }
            break;
        }
        case PACKET_TYPE_POSITON: {
            // Deserialize position
            if(length >= 3 + sizeof(Position)) {
                memcpy(position, packet + 2, sizeof(Position));
            }
            break;
        }
        default: {
           // xil_printf("Unknown packet type: %c\n", packetType);
            break;
        }
    }
}
```

- The last piece of data received on the FPGA is in updataPandV.c, which uses local variable copies to copy the received data into the global struct used in the main program. In this way, hopefully, the main program can use data protection methods, and because the rest of the levels down are not directly accessing the same struct, we should be safe.

```
void getPositionAndVelocity(DataPoint *dataPoint) {

    processPacket(&myVelocity, &myPosition);

    dataPoint->x_pos = myPosition.xPos;
    dataPoint->y_pos = myPosition.yPos;
    dataPoint->z_pos = myPosition.zPos;

    dataPoint->x_veloc = myVelocity.xVel;
    dataPoint->y_veloc = myVelocity.yVel;
    dataPoint->z_veloc = myVelocity.zVel;

}
```

- **ROM Comparison Task (FPGA):** Implements algorithms to compare the incoming data against the recorded warm-up ROM baseline. This includes printing the data to the terminal as well as updating the min/max values in the Extremes struct.

```
               ... rest of loop
            }

            // Check IF Current System State in Working mode
            if(currentState == STATE_WORKING) {
                ...
                // Check IF current readings are outside of recorded extremes
                // then Alert user via UART console
                if (tempData.x_pos < dataExtremes.minX || tempData.x_pos > dataExtremes.maxX)
                    print("ALERT: X position out of range!\r\n");
                ...
            }
```

[redacted]

**Theory of Operation**

1. **Initialization:** The system powers on, FreeRTOS initializes tasks, and hardware connections are established (I2C for the BNO055, UART between ESP32s, BLE).
2. **Sensor Data Preprocessing/Spoofing (Wearable ESP32):** The sensor-side ESP32 acquires data from the BNO055, performs preprocessing or generates spoofed data.
3. **BLE Transmission (Wearable ESP32):** The preprocessed (or spoofed) data is transmitted over BLE to the intermediary ESP32.
4. **Data Relay (Intermediary ESP32):** The second ESP32 receives the BLE data and forwards it to the FPGA via the UART connection.
5. **Data Reception (FPGA):** The FPGA receives the preprocessed (or spoofed) sensor data.
6. **Warm-up Recording (FPGA - vMenuTask):** The user performs a warm-up set, and the system captures the baseline ROM data, storing it in the Extremes struct.
7. **Working Set Monitoring (FPGA - DataPrintTask):** During working sets, continuous sensor data is compared against the baseline ROM.
8. **ROM Comparison (FPGA - DataPrintTask):** The incoming data is checked against the stored Extremes values to identify if the user is deviating from the established ROM.

**Challenges:**

- **Sensor Data Noise:** The BNO055 sensor's linear acceleration data was very noisy, causing drift and inaccurate position calculations in the dead reckoning approach. This is a frequent issue with inertial measurement units (IMUs).
- **Real-time Calculation Delays:** Initially, position calculation was offloaded to an FPGA, but transmission delays introduced further errors. This highlights the challenge of synchronization and latency in distributed sensor systems.
- **Kalman Filter Tuning:** Kalman filters can be powerful for refining sensor data, but they work best with real-world feedback for tuning. Using estimated position data made the filter difficult to optimize, leading to an unresponsive sensor.
- **RTOS System Hangs:** The system experienced unpredictable hangs, often during data transactions. This suggests a potential memory issue with the Real-Time Operating System (RTOS). Stack overflow errors and malloc failures further support this theory.

**Solutions:**

- **Data Filtering:**

    - **Low-pass filter:** A basic attempt was made to smooth out the noisy acceleration data. While it helped somewhat, it didn't fully solve the drift issue.
    - **Kalman Filter:** An attempt was made to implement this filter, but the lack of reliable feedback limited its effectiveness.
- **Data Spoofing:** To sidestep the sensor issues, the focus shifted to generating simulated exercise data.
    - **Trigonometric Functions:** Warm-up and working set data were modeled using trigonometric functions, providing controlled and predictable movement patterns.
    - **Randomness:** Random elements were added to simulate realistic variability in exercise execution.
    - **Fatigue Simulation:** The working set data incorporated factors to increase amplitude and variability over time, mimicking the effects of fatigue.
- **RTOS system Hangs**: With the time constraints of this project, we completed a more simple version of the program, which only has a few tasks and just one queue for data. This reduced the stack size needed by the program, and we stopped seeing stack overflow errors.

**Diagrams:**

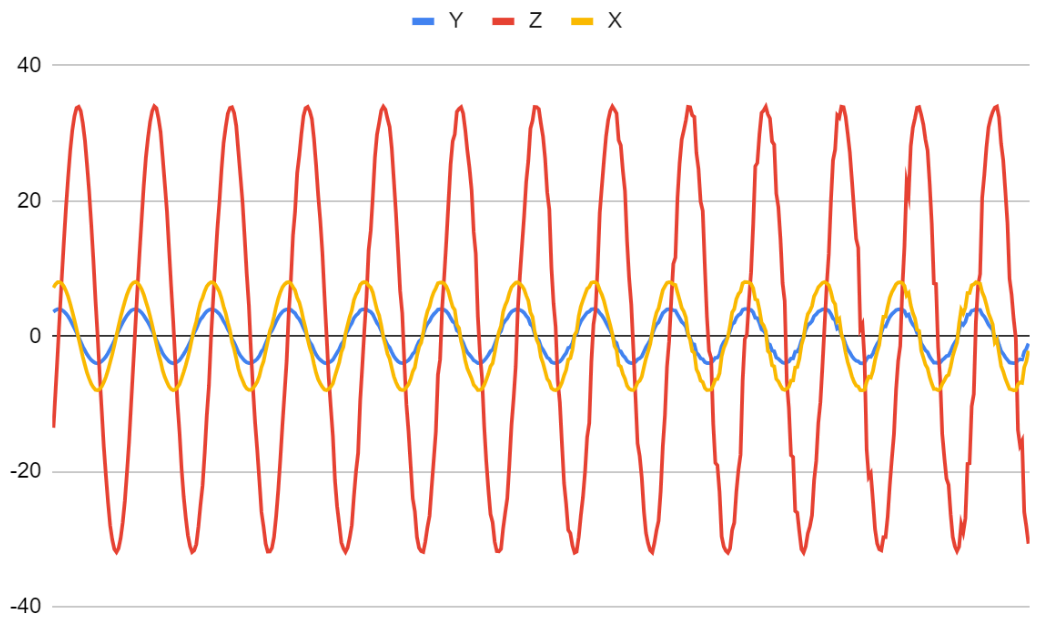


**Fig 1.** Warm up data spoofed by our sensor-side ESP32. There is a randomness factor added to the warm up to simulate fatigue, but it has a much smaller effect

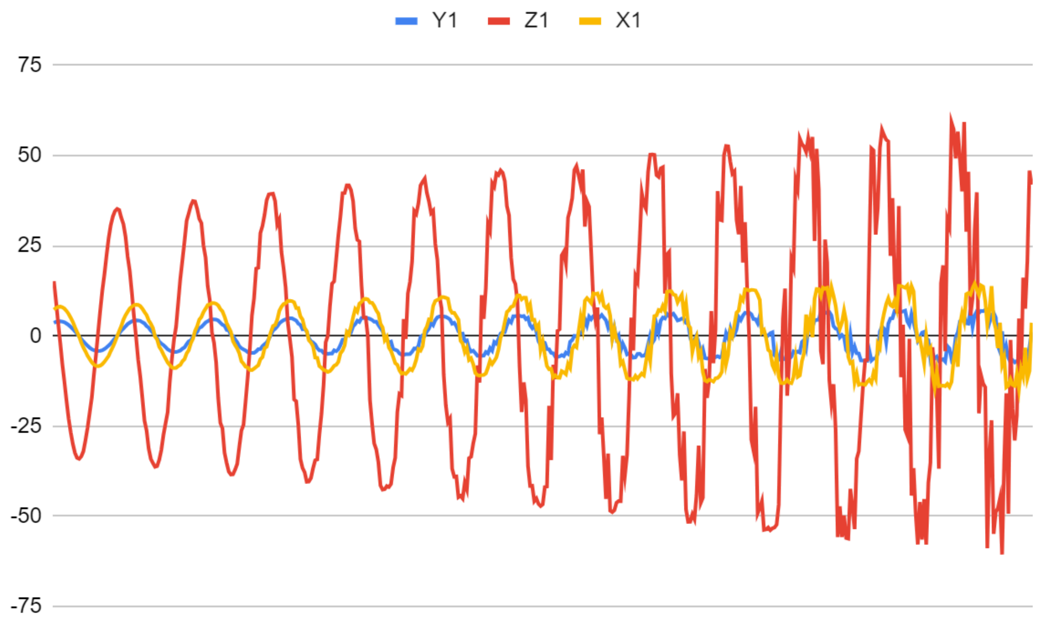


**Fig 2.** Working set data spoofed by our sensor-side ESP332. There is much more randomness and a greater difference in amplitude as time progresses. This doesn't perfectly simulate real

world motion, but it is sufficient to test our application and is an easily replicable way to generate different types of data.

## Conclusion

This project explored the potential of real-time exercise coaching and Range of Motion (ROM) tracking using an RTOS-based wearable device. While our initial goal was to utilize IMU sensors for accurate position estimation, technical challenges highlighted the complexities of this task within a dynamic exercise context. The noisy linear acceleration data from the IMU hindered reliable dead reckoning calculations.

To overcome these obstacles, we successfully pivoted our approach. We demonstrated the core value proposition of a real-time ROM feedback system by developing a prototype using simulated sensor data. This strategic shift allowed us to investigate the following:

- **Sensor-Based Monitoring Limitations:** We gained valuable insights into the limitations of AHRS sensors, specifically the challenges of achieving the precision required for accurate real-time position tracking in exercise scenarios.
- **ROM Tracking Concept Validation:** Our system successfully recorded a warm-up ROM baseline and provided feedback when simulated working sets deviated from the baseline. This demonstrates the potential of real-time ROM feedback for enhancing exercise form and safety.
- **Microcontroller Versatility:** The ESP32 microcontroller's flexibility enabled us to address both sensor data handling (or spoofing) and wireless communication, underscoring its suitability for similar embedded projects.

In addressing the RTOS memory issues, we gained practical experience in resource optimization. While a definitive solution wasn't reached within the project's timeframe, the troubleshooting process reinforced the importance of careful memory management in embedded systems.

### Future Work

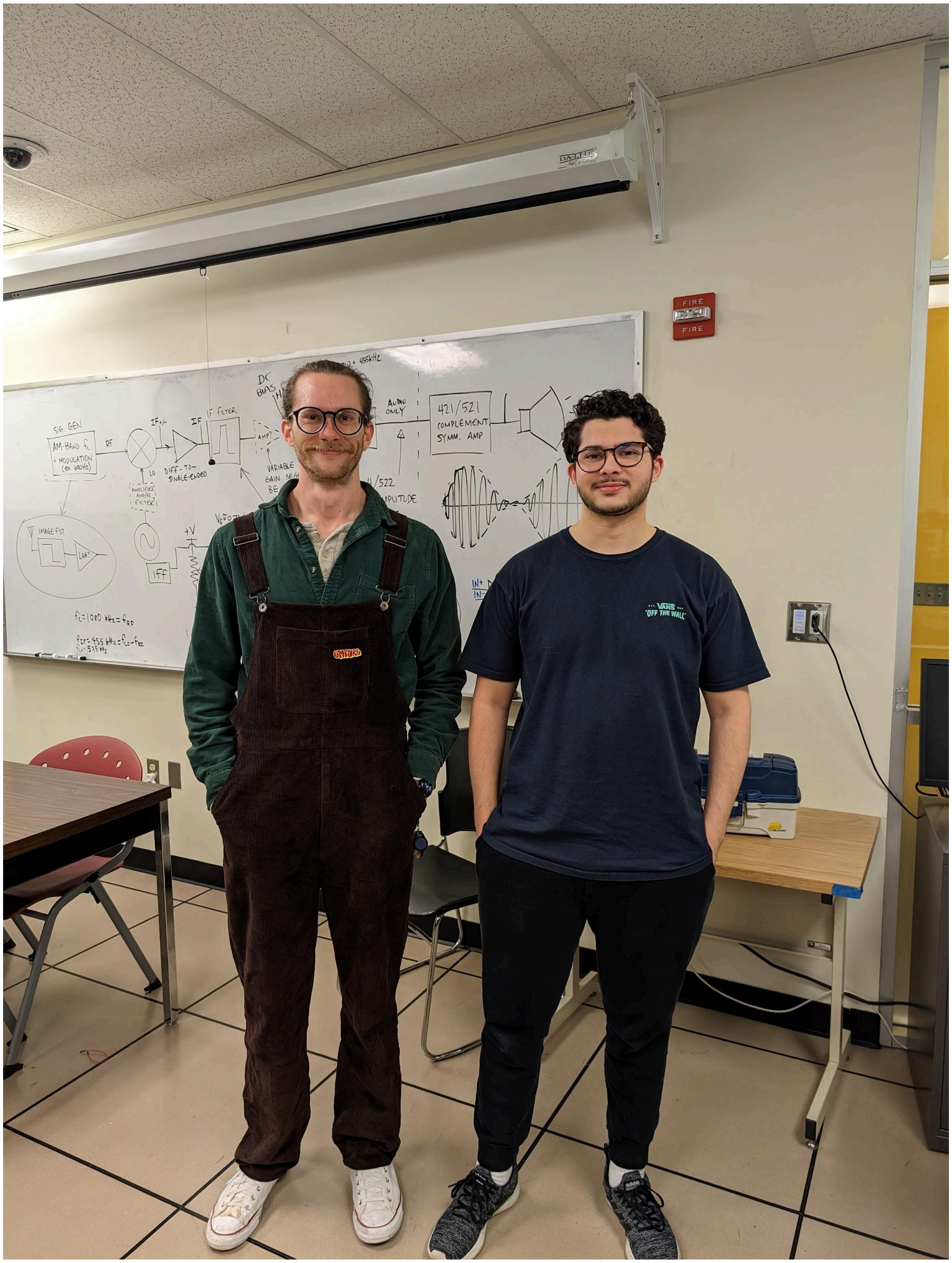
This project lays a strong foundation for further development. We propose the following avenues of exploration:

- **Sensor Augmentation and Fusion:** Investigate the integration of additional sensor inputs (e.g., force sensors, optical markers) and explore advanced sensor fusion techniques to improve position accuracy.
- **Refined Filtering Algorithms:** Research and implement more sophisticated filtering algorithms specifically tailored for exercise motion data.
- **User Testing and Feedback:** Conduct thorough user testing to evaluate the usability, effectiveness, and perceived value of the prototype. Gather feedback on data visualization and feedback mechanisms.

In conclusion, although sensor-related challenges necessitated a change of focus, this project provided a robust proof-of-concept for real-time ROM feedback.  It emphasizes the importance of iterative development and strategic adaptations in the face of technical hurdles. This work has the potential to significantly improve exercise form, safety, and training outcomes.

**Reference List**

- Adafruit Industries. (2014). BST_BNO055_DS000_12 [Data sheet]. Retrieved from https://cdn-shop.adafruit.com/datasheets/BST_BNO055_DS000_12.pdf
- automaticaddison. (2020, September 24). How to convert a quaternion to a rotation matrix [Blog post]. Retrieved March 7, 2024, from https://automaticaddison.com/how-to-convert-a-quaternion-to-a-rotation-matrix/
- denyssene. (n.d.). SimpleKalmanFilter [Code repository]. Retrieved March 7, 2024, from https://github.com/denyssene/SimpleKalmanFilter
- Instructables. (n.d.). BlueTooth link with auto-detect connect. Retrieved March 7, 2024, from https://www.instructables.com/BlueTooth-Link-with-auto-detect-connect/
- Nippard, J. (n.d.). Junk volume: Why you must avoid it for max muscle growth. jeffnippard.com. Retrieved March 7, 2024, from https://jeffnippard.com/blogs/news/junk-volume-why-you-must-avoid-it-for-max-muscle
- Real Digital. (n.d.). BooleanBoard: Real Digital Development board. Retrieved March 7, 2024, from https://www.realdigital.org/doc/02013cd17602c8af749f00561f88ae21
- Renaissance Periodization (n.d.). Training volume landmarks for muscle growth. RP Strength. Retrieved March 7, 2024, from https://rpstrength.com/blogs/articles/training-volume-landmarks-muscle-growth
- Schoenfeld, B. J., Grgic, J., Van Every, D. W., & Plotkin, D. L. (2021). Loading recommendations for muscle strength, hypertrophy, and local endurance: A re-examination of the repetition continuum. *Sports*, 9(2), 32. https://doi.org/10.3390/sports9020032
- Team Full ROM. (n.d.). Hypertrophy training periodization. teamfullrom.com. Retrieved March 7, 2024, from https://teamfullrom.com/blogs/news/hypertrophy-training-periodization
- Wright, L. (2016). *Design of a wearable device to monitor shoulder flexion and extension angle* [Master's thesis, University of Vermont]. ScholarWorks@UVM. https://scholarworks.uvm.edu/cgi/viewcontent.cgi?article=1449&context=graddis


SIG. GEN
AM-BAND fc + MODULATION (ex 600Hz)
RF
IF FILTER
DIFF-TO-SINGLE-ENDED
AUDIO ONLY
421/521 COMPLEMENT SYMM. AMP
FIRE
FIRE
VANS "OFF THE WALL"