Gait Analysis by Angular Step Mapping for Fall Risk Prevention Research

A senior team project report
submitted in partial fulfillment of the requirement for the degree of
Bachelor of Science in Physics concentrating in
Engineering Physics and Applied Design
from the College of William & Mary in Virginia,

by

Lee Bradley
Martha Gizaw
Nate Winneg

Mentor: Dr. William Cooke

Dr. Jeffrey Nelson

Williamsburg, Virginia
April 30, 2020

Section 1: Overview:

The World Health Organization reports that falls are the second-leading cause of accidental death among senior adults around the world [1]. While individuals at any age can also fall, most are able to pick themselves up and move along with their days. A few of them have reached beyond age 60 and can face serious injuries even after only one fall.

Currently, a research team at William & Mary's Department of Kinesiology & Health Sciences attempts to recognize and correct aging-related factors that can result in falling. To meet this goal, that team has administered a battery of tests but wants to either improve or redesign those tests. Many of them have been videotaped to examine individual gait parameters of older subjects. Unfortunately, the team undergoes a slow, laborious process of analyzing video frame by video frame to measure step heights and angles without any way of automating this repetitive task.

Our team, namely the "Unstable Seniors", is a group of EPAD students whose mission is to develop a wireless, non-invasive product for the kinesiology team to improve and streamline the data derived from a gait analysis test. Our tasks included calibration, microcontroller circuiting and communications, CAD design, and time-series data processing. We want to use accelerometers strapped to the legs to quickly and wirelessly provide quantitative data on step height and total angular changes about a specific axis of a limb. Our collaboration with the Department of Kinesiology & Health Sciences is expected to inspire sports doctors, physical therapists, and other healthcare personnel to accurately and quantitatively describe how one walks.

Section 2: Project Planning

In this section, we outline the general timeline and planning in the fall semester and then in the spring semester. This section is intended to outline the general timeline of the project rather than to give technical details about each element. We will delve further into the technical details in the following section entitled: “Technical Specifications.”

Subsection 2.1: Downselection (Fall):

During the fall, this project was broken into two phases. The first phase that took us to about the midpoint in the fall semester was our initial idea phase to come up with some options for products that might be useful to the health sciences gait analysis effort in their fall risk prevention research. In order to do this, we met with the health sciences team and watched an hour or so of data collection in its current form for them. We then researched different ways that the health sciences team’s gait test could be changed. Each of the four team members researched a topic and we came up with four initial ideas to consider. These included measuring hip abduction and adduction strength, step pressure mapping, gait mapping with IMU sensors [2], and gait mapping with computer vision. Each team member researched a single initial topic for the first few weeks of the project until mid-October when we made our initial downselection. Due to concerns about the ability to test hip adduction/abduction strength of seniors and the lack of viability shown in the step pressure sensing efforts, the team chose to pursue the two gait mapping related efforts for the remainder of the fall semester.

In this second phase of the fall segment of this project, we took our two remaining ideas and added a team member to each since two team members had their ideas down selected. In this second phase, Nate and Lee worked on gait mapping with IMU sensors and Martha and Colm (who was not working with the team during the spring) worked on using computer vision to measure step height. We remained working in these teams of two to show the initial viability of each of these two ideas until the last week of classes in December where we made our final downselection. At that point, we decided to pursue a product based on Bosch BNO055 inertial measurement unit sensors. At this point, we broke up a few tasks that could be worked on during the winter break. Nate took the lead on developing an initial model of a mounting system for the device, Martha and Colm agreed to work on some initial data processing scripts written in Python, and Lee took the lead on writing the software for the microcontroller controlling the BNO055 sensor.

Subsection 2.2: Prototyping (Pre-COVID-19):

With the beginning of the second semester came the switch towards wireless communication. Leading this charge was the ESP8266 wireless microcontroller. The chip features an onboard WiFi chip, which we can implement alongside a Raspberry Pi to create a closed system in which we can wirelessly take data and transmit back to the host device. By implementing an MQTT protocol to connect the Pi to an array of ESP devices, our wireless prototype took shape. The ESP device was able to seamlessly connect to our Raspberry Pi's Network, and we were able to transmit data from the BNO055 through the ESP chip and across the MQTT network; however, the chip is, at this point, still relying on wired power, and is not free from USB tethers for serial communication. This challenge leads to the next prototyping stage, which began with the move to off-campus development due to Covid-19.

Subsection 2.3: Prototyping (Post-COVID-19):

Upon our transition to remote instruction, we reduced the number of wireless microcontroller sensors to one. We were able to have the Raspberry Pi establish its own WiFi protocols to transmit data with one or more ESP32 devices. Running an MQTT Python script inside the Pi serves as a way to start and stop the quaternion data collection via the ESP32. This step leads to data communications between ESP32 and the Pi for the publication of CSV files containing quaternion coordinates and rotation angle changes. At this time, we had the option to post-process that information before we can restart device communications as we were using a single sensor. Future prototyping will be up to next year's cohort of EPAD students.

Section 3: Technical Specifications

In this section, we go into greater detail about the technical elements of our project. We begin with section 3.1 talking about the initial ideas for potential technologies and how we narrowed them to a final idea to pursue in prototyping. We then delve deeper into the prototyping phases and the individual components involved in that process.

Subsection 3.1: Downselection

In order to best satisfy the client's desire for improved data collection and analysis, several methods of automated testing were considered. Each test was designed to output quantitative data for nearly instantaneous analysis. These ideas will be discussed in-depth, as well as the review process for selecting a single procedure to produce data of interest to our client.

It has been shown that the strength of the hip abductor muscle groups is correlated with balance and support [3]. In order to test a subject's strength in this area, a sitting test was proposed in which a dynamometer would be used to the maximum torque a subject could produce from their hip abductors. From this data, a model could be created to illustrate the correlation between applied torque and force per unit length and propensity for falling accidents.

The client also provided a GAITRite mat for potential use. This mat consists of a densely packed array of pressure sensors that are able to map a person's gait and the relative pressure on different locations of their feet during footfall. It was thought that this data might be captured from the proprietary system and used in further analysis. If possible, this data could be stored, and easily added to a model to predict falling accidents based on anomalies in gait patterns.

The main dataset that the client wishes to gather is on step height. The team has devised two potential methods for measuring this parameter. The first is a system utilizing small cameras mounted to the foot, followed by post-processing using computer vision to determine the step height using natural rulers in the foreground and background of the image. The use of computer vision algorithms would allow research teams to quickly gather qualitative data from existing video.

The second method for gathering step height data was devised by mounting an array of inertial measurement sensors along a subject's legs. By collecting a time series of angle measurements along the quad, calf, and foot, a digital representation of a person's gait may be created, and by creating the right trigonometric model, a person's step height may be calculated at any point.

Throughout the semester, the research team has slowly narrowed its focus down to only a single system. Firstly, while the research behind the hip abductor test points towards a good indication of stability, it was determined that the test would not prove applicable in the needed context. In order to accurately measure the correct muscle groups, the subject must be laying down; merely sitting in a chair would offer brace points, and the data collected would not accurately reflect the strength of the subject's hip abductors. The subject must be laying down to isolate the abductor muscle group, which is not feasible given the potential lack of mobility in the subjects. As such, this test was dropped in pursuit of better options. The GAITRite mat provided by the client also proved to be a difficult endeavor. The propriety program would not allow data extraction outside of the GAITRite environment, so the mat will have to remain outside of the developing tests.

Finally, after exploring the possibility of using computer vision throughout the first half of the project timeline, potential pitfalls of the system became apparent. The system offered too much variation in background and camera mounting position, as well as physical limitations in size of camera and needed refresh rates and resolutions. It was simply too difficult to get usable footage, and accurately analyze footage consistently. Thus this idea, while offering the benefit of integrating with the existing dataset, was not feasible.

Our final system, relying on wireless sensor units transmitting spatial data to build a digital representation of one's gait, proved to be the most viable. The system uses small, coin-sized sensors, and offers a non-invasive way to gather high-resolution data. The Inertial Measurement devices we chose, the BNO055, offers data stream at 20Hz, without output in Euler angles (3-dimensional representation of rotation around 3 orthogonal axis, with the z-axis directed through the Earth's center of gravity) and/or Quaternions (a system using 3 real axis and a fourth imaginary axis of rotation). In order to escape phenomena such as gimbal lock- the alignment of axes during rotation, and subsequent data loss, quaternions are chosen as measurement values. This also allows angle changes to be immediately calculated, as the dot product of 2 quaternions results in the half-angle rotation between them. This promising method of data-collection allowed the team to push forward with prototyping a wireless system to deliver real-time, accurate data for computational analysis.

Subsection 3.2: Prototyping

This section will be broken into three as there were three main components to the prototyping phase of our project. These are hardware development which includes, first, the mounting system and microcontroller chip selection. Second, software/firmware development which includes the code used to control the Raspberry Pi, the microcontroller + sensor configuration, and the data collection algorithm in general. Finally, data processing and analysis which includes the post processing and plotting of data once it had been taken. As mentioned in the project planning section, these were the lines across which we divided the workload during the prototyping section of the year.

Section 3.2.1: Hardware Development:

The first two iterations of the mounting system were designed to house a coin cell battery [4], an mBed NXP lcp1768 microcontroller [5], and the BNO055 [6] breakout board. These are shown in the appendix in figure 3. The first iteration (version 1) features a self-locking mechanism where the top of the model slides on and twists to lock over the bottom. In theory, the BNO055 board would be stacked on top of the mBed microcontroller in the larger slot and the coin cell would slide into the smaller slot. This self-locking proved to be difficult to manufacture without significant post processing after printing and it was bulkier than necessary. The second iteration (version 2) locks with screws in threaded slots in the corners of the model and contains embedded slots for Velcro straps rather than extruded handles. It did, however, feature the same stacking layout of components. This proved to be more robust but the slots for Velcro straps proved difficult to 3D print and the stacking layout made it much taller than necessary leading to bouncing when in use.

We then moved on to the second phase of prototyping. In this phase we began looking at wireless data collection. We made the decision to move to a different mBed based microcontroller called an mBed MAX32630FTHR[7]. This microcontroller is designed to run on battery power, contains a port to plug in a rechargeable battery, and supports Bluetooth and BLE communication making it an ideal choice for a wearable sensor configuration. After considerable work on the part of Lee and Dr. Cooke to try to make the mBed MAX32630FTHR chip function with BLE communication, we were forced to abandon the idea due to lack of functionality and time constraints. We were also able to fabricate another iteration of a mounting system as shown in figure 5 in the appendix. This version was configured to fit 2 coin cells in series in the circular slot, the mBed MAX32630FTHR, and the BNO055 board in different slots rather than stacked. This version is shorter and less subject to bounce as a person walks. It contains springs to hold batteries, sensor, and microcontroller more firmly in place. Finally, it contains outward slots for Velcro straps that are more feasible to produce and reuse than the inward, rounded slots on the previous model. The microcontroller and sensor are both shown in figure 5 mounted inside the model.

In the third phase of prototyping, we switched our microcontroller to a Wifi based chip called an ESP8266 NodeMCU 12-E [8] and then later to an updated version of the same chip called an ESP32 DevKit 3C [9]. These were necessary to make wireless communication between microcontroller and raspberry Pi function successfully. We also switched to a Lithium Polymer battery[10] after the first few tests at fully wireless data collection and learning that the coin cell circuit could not provide the necessary current to the microcontroller system. This battery is shown in figure 14 along with its part number. To go along with these last hardware changes, we fabricated a final mounting system model as shown in figure 6. This model is shorter than the previous as it stacks the BNO055 board and the Lithium Polymer battery and has larger arms holding the Velcro straps in place than the previous model. This allows it to be both more robust than previous models as well as slightly smaller.

Section 3.2.2: Software/Firmware Development:

In our first phase of prototyping, we worked with an mBed NXP lcp1768 microcontroller and the BNO055 breakout board. In this initial phase, the only code used was the firmware written to control the microcontroller. This consisted of the use of 4 libraries in an Arduino file and some base code to get sensor readings. The first of these libraries was wire.h which initializes I2C communication between the computer and the microcontroller. The second was Adafruit_Sensor.h which is adafruit's sensor driver library. This allows the program to communicate with the adafruit sensor. The third library was Adafruit_BNO055.h which contains functions specific to the initialization and use of the BNO055 chip that is embedded on the Adafruit breakout board we were using. Finally, we included utility/imumaths.h which allows the Arduino script to understand the output of the BNO055 output.[11]

In the second phase of prototyping, not much changed on the software side as we were unable to get the wireless capabilities of the mBed MAX32630FTHR to function. In the third phase, however, we had to change the Arduino code dramatically to incorporate the Wifi communication. We chose to use MQTT broker/client protocols as our mode of Wifi based communication between the microcontroller and a Raspberry Pi 4 to do our data collection. MQTT communication works by configuring devices as clients all connected to the same network as each other and as the central broker. Clients have the capability to publish messages to a topic as well as to subscribe to topics and receive messages sent by other clients to those topics. When a message is published, it is sent first to the broker which determines the topic of the message and which clients should receive the message depending on the topic. In our case, we configured the ESP microcontroller as a client and the Raspberry Pi to broadcast a network over which to communicate as well as acting as the broker and a client. This way, we can broadcast commands from the Pi to the ESP's wirelessly and receive data, also wirelessly, from the ESP on the Raspberry Pi. As far as the code we used, this first consisted of adding two libraries. These were Wifi.h to give us functions to control connection to a Wifi network and PubSubClient.h to control the MQTT protocols. A full flow chart of the code used on the ESP32 DevkitC is shown in the appendix. In addition to the microcontroller firmware, we also wrote a software script in python to control the MQTT protocols on the Raspberry Pi. In this script, we imported paho.mqtt.client [12] as a library for functions to control the MQTT protocols, numpy to do the give us array appending capabilities necessary for transporting data to CSV files, math in order to convert between data types, and CSV to give us read/write capabilities on CSV files. In the script itself, we have three main functions: on_message that deals with when a message is received, on_connect to handle when the pi connects to the MQTT broker, and on_log to print what's going on. The bulk of the code happens in on_message allowing us to use different messages being sent from the ESP32 to trigger protocols like adding data to a CSV file, converting data between data types, and plotting.

Section 3.2.3: Data Processing/Analysis

In our first phase of prototyping with the mBed NXP lcp1768 setup, we were able to take data relating to the angle change of the calf and thigh while walking and convert them to data regarding the height of the foot while walking. A plot of thigh and calf angles in degrees as well as the calculated step height data in centimeters is shown in the appendix in figure 1. The x-axis of all plots is a count of data points taken at approximately 100Hz.

This step height data contained a relatively high level of uncertainty with errors in the centimeter range. This data for step height was calculated by measuring (by hand) the length of the thigh and calf and using the angle change of the thigh and the calf to measure the height that the foot has left the ground. This equation is shown in equation 1 where h is step height, T is the length of the thigh, and C is the length of the calf. A model of the step that this equation corresponds to is visually shown in figure 2 in the appendix where θ and φ refer to the angle change of the thigh and calf respectively.

$$h = T * Cos(\theta) + C * Cos(\varphi) \qquad \text{(Equation 1)}$$

In phase two of prototyping, in addition to switching to a wireless capable microcontroller, we also realized that our data had a large amount of error due to the effects of poor alignment and, in some cases, gimbal lock which occurs when two of the three degrees of freedom are driven into a parallel configuration. We made the decision to switch from measuring the euler pitch to measuring quaternions. Quaternions measure linear motion in the x, y, and z axis as well as rotation w about an axis. By taking the dot product of two quaternions, we can get the cosine of half of the angle between the two. If we then take the arccosine and multiply by 2, we can get the angle change about the axis of rotation between the two quaternions. This is shown in equation 2 below where Q is a quaternion consisting of w, x, y, and z coordinates.

$$Angle\ Change = cos^{-1}[2 * (Q_{initial} \cdot Q_{final})] \qquad \text{(Equation 2)}$$

This is also shown visually in the appendix in figure 4 of the 30º angle change between parent and child quaternions about the depicted axis. This measurement allows us to bypass the problem of missing some angle data that was picked up in roll and yaw instead of pitch due to misalignment of the sensor on the leg.

Regarding data analysis in this final phase of prototyping, we collected the CSV files that resulted from the wireless communications between the Raspberry Pi and the microcontrollers. In addition, we wrote an additional Python script that can take in and plot the data with a library called Matplotlib. Note that the BNO device can report either the Euler angles or the quaternions. As 4-coordinate descriptions of the rotation angles and axis orientations, quaternions have been useful for directly measuring the net angle change about a specific axis. If we have two quaternions--one at time = 0 ($Q_1 = Q_0$) and one at any time ranging from 0 to 3000 ($Q_2 = Q_{[0-300\ ]}$)--we can take their dot product (see Equation 2, Figure 8) to obtain the cosine of half of the net angle change between them. That is:

$$\cos(\frac{\alpha}{2}) = Q_1 \cdot Q_2 = w_1 w_2 + x_1 x_2 + y_1 y_2 + z_1 z_2 \qquad \text{(Equation 3)}$$

All of the quaternions are normalized so that $w^2 + x^2 + y^2 + z^2 = 1$, where the angle between one orientation and itself is zero [13].

So far, the multiplication of two quaternions gives us the thigh and calf angles that change over the course of walking activity. Figure 9 displays such changes. Both plots not only are representative of the typical angles each part of the leg makes, but they also display the changes in one variable. Unlike Euler angles, the quaternions are simpler at measuring the rotation angles that we need to accurately calculate step height and other gait parameters.

Euler angles are representative of a rotation that is about one of the main Euler axes: roll (ϕ), pitch (Θ), or yaw (Ψ). As we focus mainly on the pitch, if that is the only angle changing dramatically at the calf, it can clearly demonstrate how much a limb can rotate, especially via the net angle change (see Figure 10). The roll and yaw angles do not make as much movement as the pitch is the primary axis that we rotate about. If we have both the pitch and roll changing significantly at the thigh (see Figure 11), we will need to combine those two Euler angles or even all three of them. We will encounter two challenges that come with Euler angles combinations. First, we must find the dot product of each rotation matrix per angle, and we must extract the Euler angle representations from the resulting rotation matrix.

$$
\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R_z(\psi)R_y(\theta)R_x(\phi)\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}
$$

$$
= \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}
$$

$$
= \begin{bmatrix} \cos\theta\cos\psi & -\cos\phi\sin\psi + \sin\phi\sin\theta\cos\psi & \sin\phi\sin\psi + \cos\phi\sin\theta\cos\psi \\ \cos\theta\sin\psi & \cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi & -\sin\phi\cos\psi + \cos\phi\sin\theta\sin\psi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad \text{(Equation 4)}
$$

$$
\varphi_R = arctan2(R_{32}, R_{33}) \quad \text{(Equation 5)}
$$

$$
\theta_R = arcsin(R_{13}) \quad \text{(Equation 6)}
$$

$$
\psi_R = arctan2(R_{21}, R_{11}) \quad \text{(Equation 7)}
$$

$$
\begin{aligned}
arctan2(y, x) &= arctan(y/x), if\ x > 0 \quad \text{(Equation 8)} \\
&= arctan(y/x) + \pi, if\ x < 0, y \geq 0 \\
&= arctan(y/x) - \pi, if\ x < 0, y < 0 \\
&= +\pi/2, if\ x = 0, y > 0 \\
&= -\pi/2, if\ x = 0, y < 0 \\
&= undefined, if\ x = 0, y = 0
\end{aligned}
$$

Second, if we let combined rotation angle $\alpha_R = R_z(\psi)R_y(\theta)R_x(\varphi)$ and unit vector $\hat{r}$ be the axis about which the rotation occurs, we will need to use $cos(\frac{\alpha_R}{2}), sin(\frac{\alpha_R}{2})\hat{r}$ to calculate all four coordinates of each quaternion. Theoretically,

$$\underline{q} \equiv \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \cos\left(\frac{\alpha}{2}\right) \\ \sin\left(\frac{\alpha}{2}\right)\sin\theta\cos\phi \\ \sin\left(\frac{\alpha}{2}\right)\sin\theta\sin\phi \\ \sin\left(\frac{\alpha}{2}\right)\cos\theta \end{pmatrix} = \begin{pmatrix} \cos\left(\frac{\alpha}{2}\right) \\ \sin\left(\frac{\alpha}{2}\right)\hat{\mathbf{r}} \end{pmatrix}$$

(Equation 9)

For the purpose of this project, however, we calculated them in a way that allows us to produce the quaternions that are similar to those produced by the BNO device. Hence, for the quaternion of combined rotation angle $Q_R = [w_R, x_R, y_R, z_R]$,

$$w_R = cos(\frac{\alpha_R}{2}) \quad \text{(Equation 10)}$$

$$x_R = sin(\frac{\alpha_R}{2})sin(90)cos(180) \quad \text{(Equation 11)}$$

$$y_R = 0.01 * (\frac{y}{2}) * cos(\frac{y}{2si\ \ (\frac{\alpha_R}{2})}) \quad \text{(Equation 12)}$$

$$z_R = 0.01 * (\frac{z}{2}) * cos(\frac{z}{2si\ \ (\frac{\alpha_R}{2})}) \quad \text{(Equation 13)}$$

Figure 12 compares two plots with quaternion-based and Euler-based net angle changes between two quaternions. If we look closer at the slight changes in the plot for the angular changes between quaternions of a combined Euler angle, we can argue that that plot is a result of the complicated math that we can avoid if the BNO quaternions are better at measuring angle change. It is imperative to know this concept because we are rotating the initial axis of a limb with the possibility of gimbal lock and other axes rotating. We would then have to waste time doing the math to determine the most useful results for the net angle change.

The last task to complete under data processing is to obtain the total angle change applied to a leg being lifted upward while the subject is sitting. This is based on the wireless quaternion data from a single sensor device. Imagine sitting with a leg resting at 90 degrees; if we choose to raise the lower part of the leg below the knee, we can typically say that our leg becomes horizontal at 180 degrees. We can then visualize this change of up to 90 degrees when we calculate the dot products of quaternions (see Figure 13). To explain the smaller dips at time counts 50-60 and 70-80, the leg appears to be swinging at smaller angles as a warmup between two complete cycles of leg-raising activity. If we confirm that the leg in a sitting position can change angles from 90 to 180 degrees, then we should stress that quaternions are simple enough to describe the rotations that usually occur in leg movements.

Section 4: Looking Forward

Due to time constraints and the necessity to work remotely for the last month or so of the project, we were not able to deliver a finished and functioning product. Because of that, we would like to take this section to outline the steps that we would have taken had we not been time-constrained. In addition, as there is another group working on this effort for the 2020-2021 academic year, we hope to give them a sense of our thoughts on how to best complete the project.

The first major component that we ran out of time completing is integrating multiple sensors using MQTT Wifi communication between the Raspberry Pi and the ESP32 microcontroller. This process includes modifying both the ESP32 code and the MQTT python script slightly in order to identify which sensor configuration each set of coordinates that is sent to the Pi is coming from. Our idea for this was to publish coordinates from each ESP to a topic labeled with that ESP's location, for example: "Left Calf." This way the data can be saved and manipulated for each leg segment and then combined later for step height calculations. The important component when integrating multiple sensors and microcontrollers into the system is to ensure that all microcontrollers are subscribed to the "cmd" topic in order for synchronization of starting and stopping data collection.

The second major component that we were unable to complete was an automated calibration step that allowed the system to calculate the length of a subject's calf and thigh leg segments from a step. Our method for this was to have a specific routine outlined on the microcontroller that takes quaternion data and measures the angle change when a subject steps onto a block of known height and distance from the leg's starting position. We can then use the angle change information to calculate the length of the calf and thigh leg segments. A model of this calibration step is shown in the appendix in figure 8 where h is the known height of the block and d is the known distance from the leg's starting position. In addition, the equations for C and T are shown in equations 3 and 4.

$$C = \frac{\left(\frac{d}{sin(\theta)} - \frac{h}{1-cos(\theta)}\right)}{\left[\left(\frac{cos(\varphi)-1}{1-cos(\theta)}\right) + \left(\frac{sin(\varphi)}{sin(\theta)}\right)\right]} \qquad \text{(Equation 14)}$$

$$T = \frac{d - Csin(\varphi)}{sin(\theta)} \qquad \text{(Equation 15)}$$

Once the system is calibrated and T and C are known, we can use them to determine step height by subtracting T$cos(\theta)$and C$cos(\varphi)$from T + C giving us h at every point in the dataset where angle change is measured against the starting position. The final state of this data analysis that we believe is most valuable to the health sciences effort is to look at the maximum step height when the foot is parallel to the floor and present those values as individual step height measurements for each step in a walk. This can be determined using a third sensor on the foot to determine when the foot flex has near-zero angle change relative to the starting position.

Appendix:

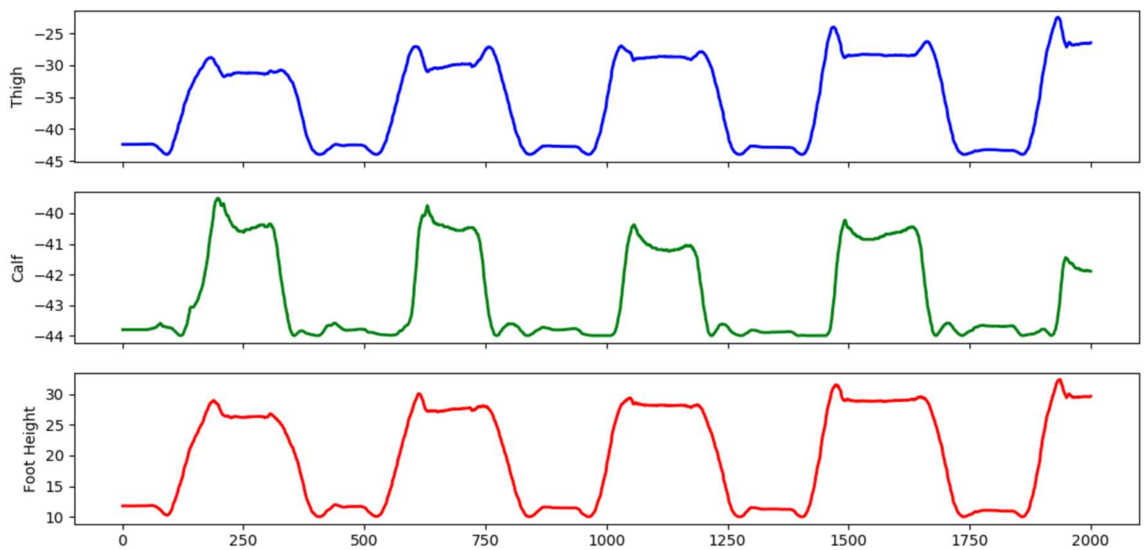

Figure 1: Initial step height measurements.

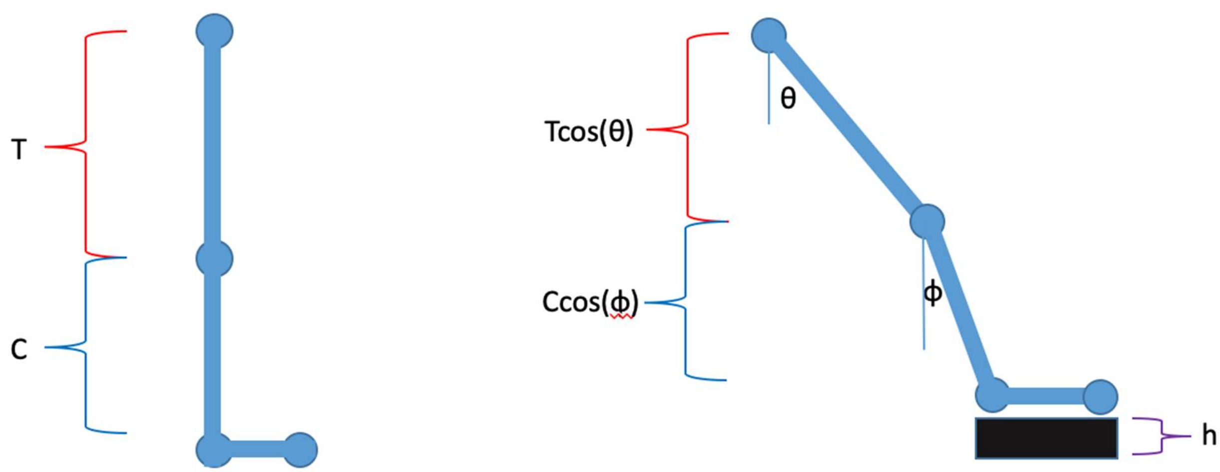

Figure 2: Model of initial step height motion.

Figure 3: Prototype mounting systems versions 1 and 2

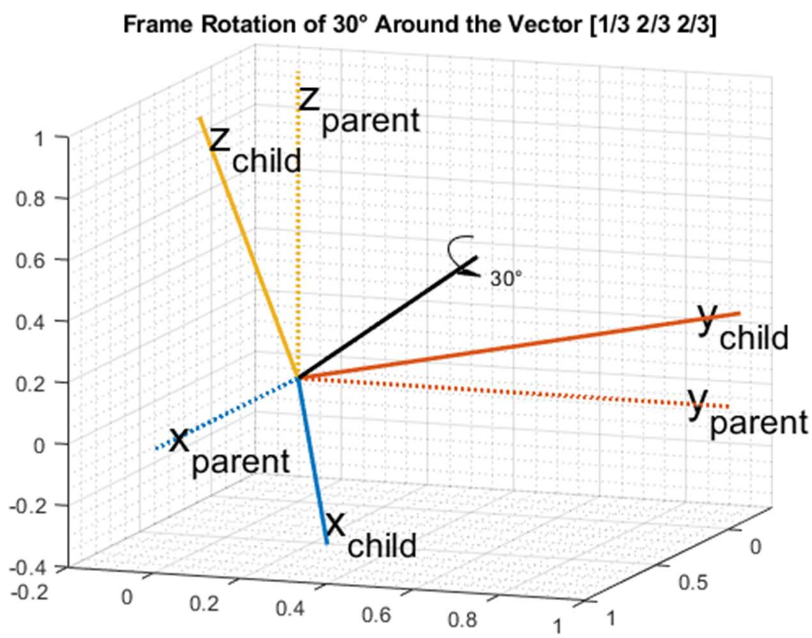

Figure 4: Quaternion coordinate visual representation.

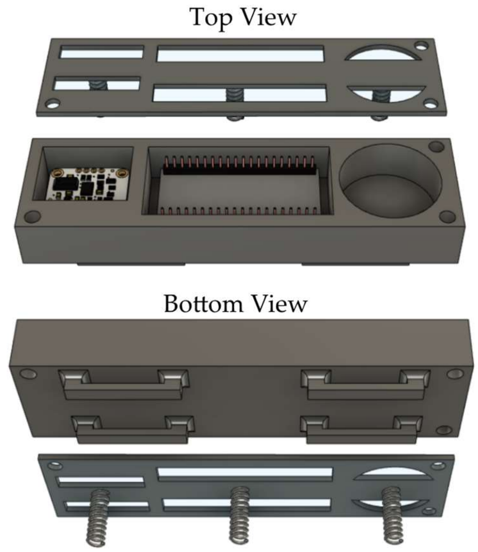

Figure 5: Prototype Mounting System Version 3.

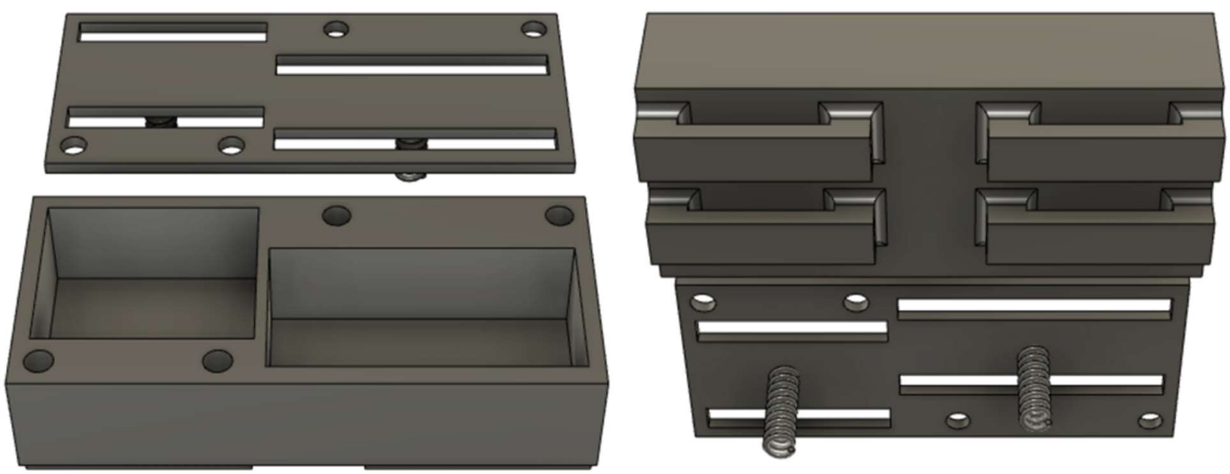
Figure 6: Final mounting system.

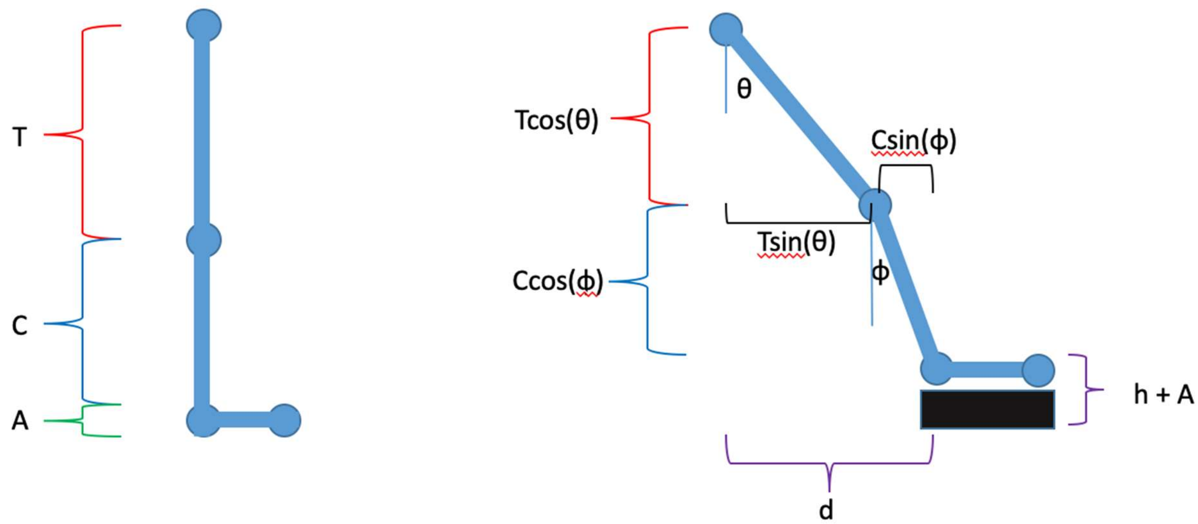

Figure 7: Calibration step model

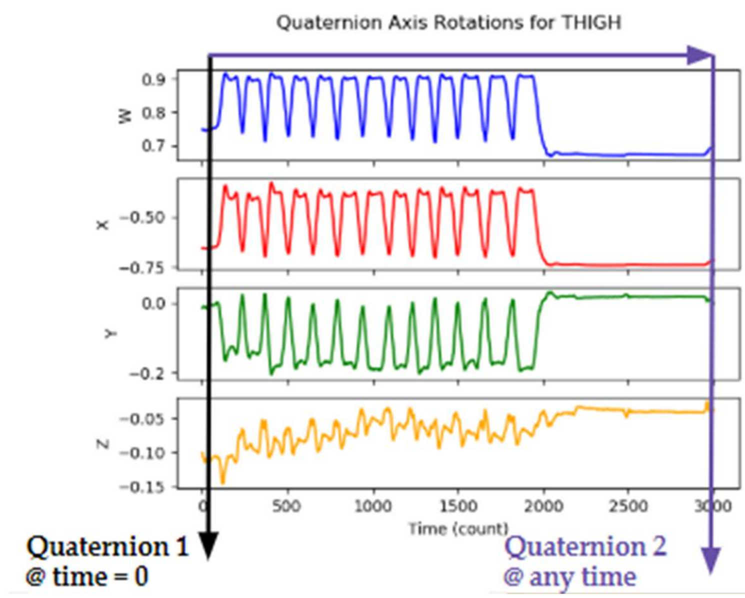

Figure 8: A visual way of finding and multiplying two quaternions over walking time.

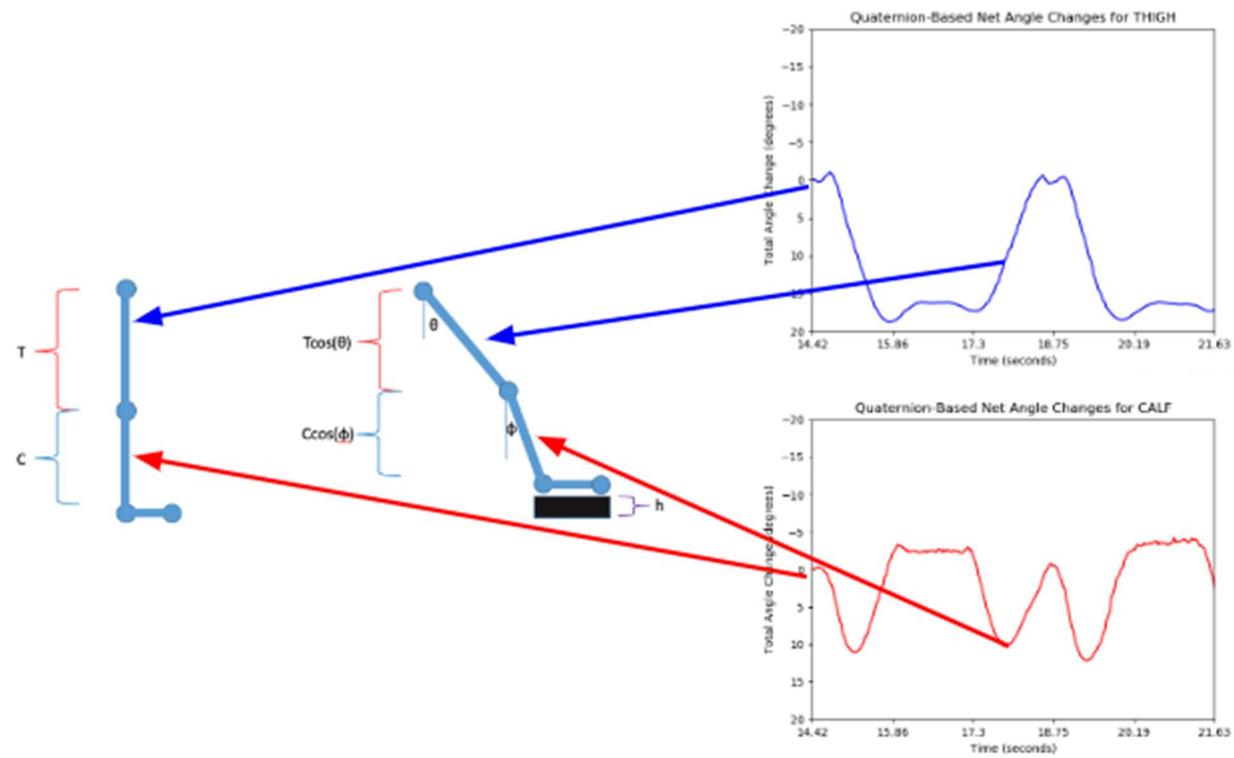

Figure 9: Quaternion-based net angle changes at the thigh (blue) and calf (red); 0 degrees signifies standing straight, other angles indicate walking movement.

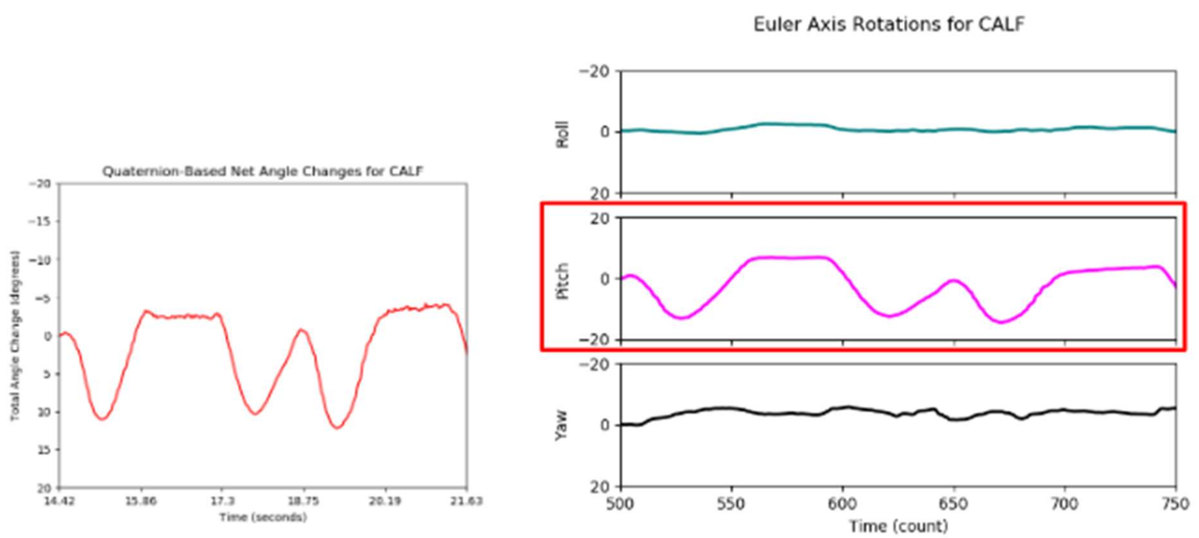

Figure 10: Our quaternion analysis can agree with the angle analysis when only the pitch (magenta) is changing significantly.

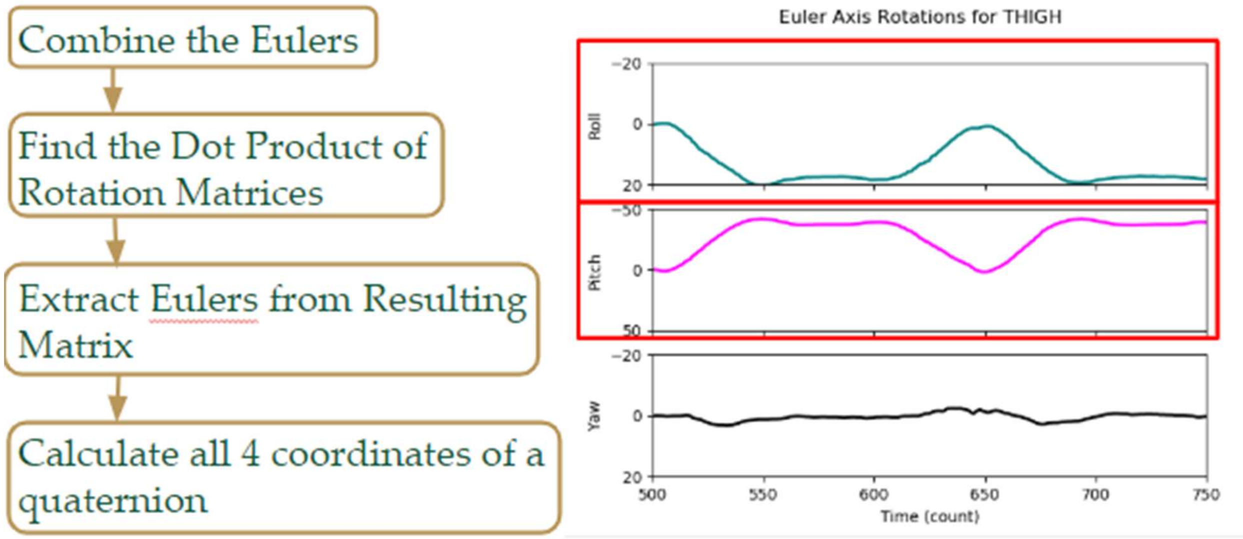

Figure 11: Our quaternion analysis can also agree with the angle analysis when two or more Euler angles changing significantly. A small flowchart is provided to understand how we could plot the total angle changes within a combined Euler angle (see Figure 12).

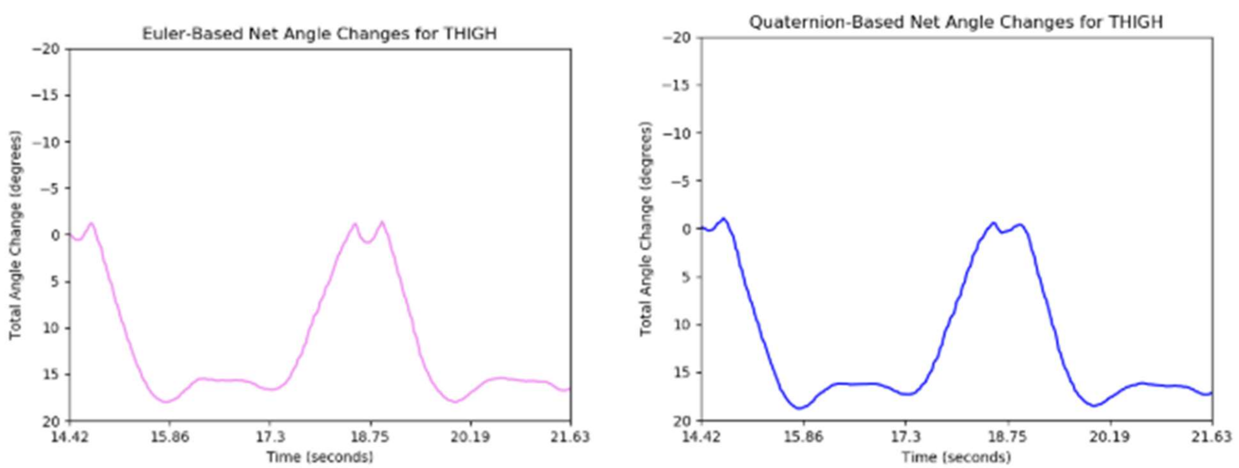

Figure 12: A comparison between the Euler-based (violet) and quaternion-based (blue) net angle changes.

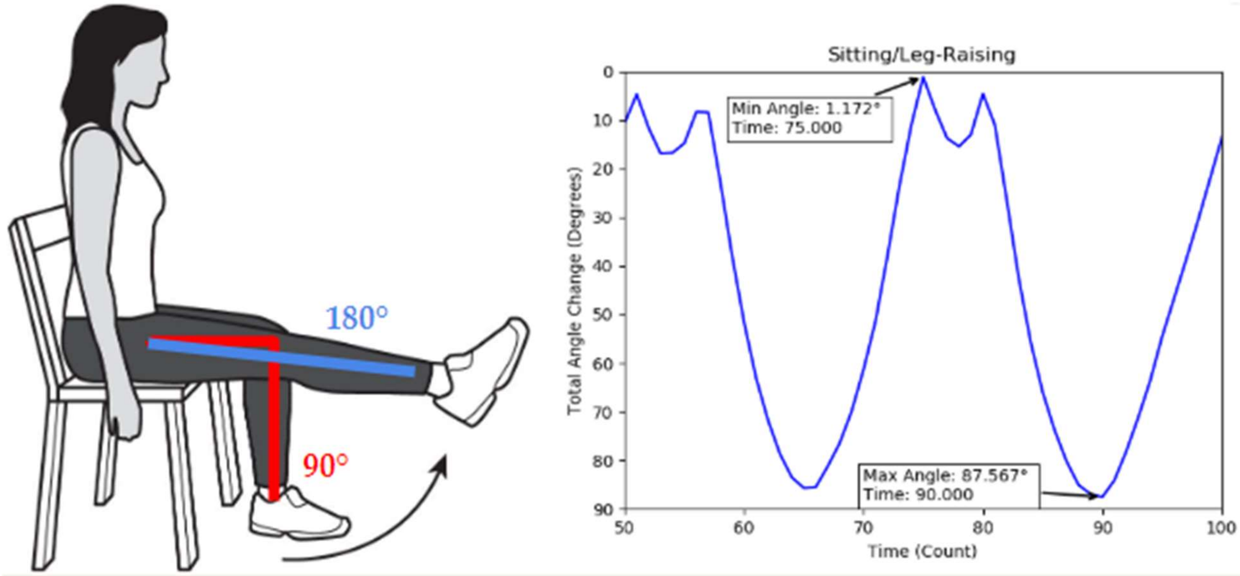

Figure 13: Net angle changes at the raising leg during a sitting session

Figure 14:Lithium polymer battery, part number 1528-1841-ND on Digikey.com

Code A: DebugSubroutinesTeamUS.py

```python
# Lee Bradley, Martha Gizaw, Nate Winneg
# Engineering Physics Capstone Project
# Unstable Seniors: Data Processing
# May 2020

# Import the following libraries.
import matplotlib.pyplot as plt
import numpy as np
import csv
import math

class PlotThigh:
    # PURPOSE: Plot the total angle change about the axis of the thigh.

    # Initialize the Euler and quaternion CSV input variables as empty arrays.
    def __init__(self, xThighRoll = [], yThighRoll = [], xThighPitch = [],
                 yThighPitch = [], xThighYaw = [], yThighYaw = [],
                 xThighW = [], yThighW = [], xThighX = [], yThighX = [],
                 xThighY = [], yThighY = [], xThighZ = [], yThighZ = [],
                 changeThigh = []):
        # Euler angles
        self.xThighRoll = xThighRoll
        self.yThighRoll = yThighRoll
        self.xThighPitch = xThighPitch
        self.yThighPitch = yThighPitch
        self.xThighYaw = xThighYaw
        self.yThighYaw = yThighYaw

        # W, X, Y, and Z coordinates in a quaternion
        self.xThighW = xThighW
        self.yThighW = yThighW
        self.xThighX = xThighX
        self.yThighX = yThighX
        self.xThighY = xThighY
        self.yThighY = yThighY
        self.xThighZ = xThighZ
        self.yThighZ = yThighZ

        # For finding the net angles changes about the limb's axis
        self.changeThigh = changeThigh

    # Execute the CSV readers, and append the data to the appropriate arrays
    # for each Euler angle to be plotted.
    # For presentation purposes, set the Euler angles
    # to zero at the initial time of the user selected interval, where we can describe
    # the events of a single cycle of leg motion (eg, walking, sitting, etc.)
    def euler_angle_thigh(self, xThighRoll, yThighRoll, xThighPitch, yThighPitch,
    xThighYaw, yThighYaw):
        figThigh, axsThigh = plt.subplots(3, sharex = True, sharey = False)
        figThigh.suptitle('Euler Axis Rotations for THIGH')

        with open('angles_thigh_roll2.csv', 'r') as csvfile:
            plots = csv.reader(csvfile, delimiter=',')
            for row in plots:
                xThighRoll.append(float(row[0]))
                yThighRoll.append(float(row[1]))
        setRoll2Zero = []
        for t in range(0, len(xThighRoll)):
            setRoll2Zero.append(yThighRoll[t]-yThighRoll[500])
        axsThigh[0].plot(xThighRoll, setRoll2Zero, linewidth = 2, color='teal')
        axsThigh[0].set(xlabel='', ylabel='Roll')
        axsThigh[0].set_xlim(500, 750)
        axsThigh[0].set_ylim(20, -20)

        with open('angles_thigh_pitch2.csv', 'r') as csvfile:
            plots = csv.reader(csvfile, delimiter=',')
            for row in plots:
```

```
                xThighPitch.append(float(row[0]))
                yThighPitch.append(float(row[1]))
        setPitch2Zero = []
        for t in range(0, len(xThighPitch)):
            setPitch2Zero.append(yThighPitch[t]-yThighPitch[500])
        axsThigh[1].plot(xThighPitch, setPitch2Zero, linewidth = 2, color='magenta')
        axsThigh[1].set(xlabel='', ylabel='Pitch')
        axsThigh[1].set_xlim(500, 750)
        axsThigh[1].set_ylim(50, -50)

        with open('angles_thigh_yaw2.csv', 'r') as csvfile:
            plots = csv.reader(csvfile, delimiter=',')
            for row in plots:
                xThighYaw.append(float(row[0]))
                yThighYaw.append(float(row[1]))
        setYaw2Zero = []
        for t in range(0, len(xThighYaw)):
            setYaw2Zero.append(yThighYaw[t]-yThighYaw[500])
        axsThigh[2].plot(xThighYaw, setYaw2Zero, linewidth = 2, color='black')
        axsThigh[2].set(xlabel='', ylabel='Yaw')
        axsThigh[2].set_xlim(500, 750)
        axsThigh[2].set_ylim(20, -20)

        # Optional!
        # figThigh.show()

    # Execute the CSV readers, and append the data to the appropriate arrays
    # for each quaternion to be plotted.
    def quaternion_thigh(self, xThighW, yThighW, xThighX, yThighX, xThighY,
                         yThighY, xThighZ, yThighZ):
        figQuats, axsQuats = plt.subplots(4, sharex = True, sharey = False)
        figQuats.suptitle('Quaternion Axis Rotations for THIGH')

        with open('angles_thigh_W2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xThighW.append(float(row[0]))
                yThighW.append(float(row[1]))
        axsQuats[0].plot(xThighW, yThighW, color='blue')
        axsQuats[0].set(xlabel='', ylabel='W')
        axsQuats[0].set_xlim(500, 750)

        with open('angles_thigh_X2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xThighX.append(float(row[0]))
                yThighX.append(float(row[1]))
        axsQuats[1].plot(xThighX, yThighX, color='red')
        axsQuats[1].set(xlabel='', ylabel='X')
        axsQuats[1].set_xlim(500, 750)

        with open('angles_thigh_Y2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xThighY.append(float(row[0]))
                yThighY.append(float(row[1]))
        axsQuats[2].plot(xThighY, yThighY, color='green')
        axsQuats[2].set(xlabel='', ylabel='Y')
        axsQuats[2].set_xlim(500, 750)

        with open('angles_thigh_Z2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xThighZ.append(float(row[0]))
                yThighZ.append(float(row[1]))
        axsQuats[3].plot(xThighZ, yThighZ, color='orange')
        axsQuats[3].set(xlabel='Time (count)', ylabel='Z')
```

```
        axsQuats[3].set_xlim(500, 750)

        # Optional!
        # figQuats.show()

    # Calculate 2 times the inverse cosine
    # of the dot product between two quaternions, and convert the net angle
    # change to degrees. Show the plots!
    def dot_product_thigh(self, xThighW, yThighW, xThighX, yThighX, xThighY,
                          yThighY, xThighZ, yThighZ, changeThigh):
        oneRad2Degrees = 57.296
        changeThighFix = []
        for t1 in range(0, len(xThighW)):
            changeThigh.append(np.arccos(np.minimum(1, yThighW[0]*yThighW[t1] +
                                                yThighX[0]*yThighX[t1] +
                                                yThighY[0]*yThighY[t1] +

                                                yThighZ[0]*yThighZ[t1]))*(180/np.pi)-(oneRad2Degrees/2))

        for t2 in range(0, len(xThighW)):
            changeThighFix.append(changeThigh[t2]-changeThigh[500])

        fig, axs = plt.subplots()
        axs.set_title('Quaternion-Based Net Angle Changes for THIGH')
        axs.plot(changeThighFix, color='blue')
        axs.set_xlim(500, 750)
        axs.set_ylim(-20, 20)
        axs.set(xlabel='Time (seconds)', ylabel='Total Angle Change (degrees)')
        axs.invert_yaxis()
        positions = (500, 550, 600, 650, 700, 750)
        labels = (14.42, 15.86, 17.30, 18.75, 20.19, 21.63)
        plt.xticks(positions, labels)
        fig.show()


    # Combine the Euler angles when more than one are changing significantly.
    def euler_combo_thigh(self, xThighRoll, yThighRoll, yThighPitch, yThighYaw,
                          yThighY, yThighZ):

        # Initialize the following variables for Euler-based net angle changes.
        theta_array = []
        R = []

        combinedEulerX = []
        combinedEulerY = []
        combinedEulerZ = []

        combinedNetAngle = []
        undoCombinedCos = []
        undoCombinedSin = []

        combinedQuatW = []
        combinedQuatX = []
        combinedQuatY = []
        combinedQuatZ = []

        rebuildCombined = []
        rebuildCombinedFix = []

        # Convert the original Euler angles into rotation matrices to be all
        # multiplied.
        for t3 in range(0, len(xThighRoll)):
            theta = [yThighRoll[t3] * (np.pi/180), yThighPitch[t3]* (np.pi/180),
            yThighYaw[t3]* (np.pi/180)]
            theta_array.append(theta)

            R_x = np.array([[1,         0,                  0                   ],
```

```
                             [0,          math.cos(theta[0]), -math.sin(theta[0]) ],
                             [0,          math.sin(theta[0]), math.cos(theta[0])  ]
                             ])



            R_y = np.array([[math.cos(theta[1]),      0,       math.sin(theta[1])  ],
                             [0,                       1,       0                   ],
                             [-math.sin(theta[1]),     0,       math.cos(theta[1])  ]
                             ])

            R_z = np.array([[math.cos(theta[2]),      -math.sin(theta[2]),     0],
                             [math.sin(theta[2]),      math.cos(theta[2]),      0],
                             [0,                       0,                       1]
                             ])

            R.append(np.dot(R_x, np.dot(R_y, R_z)))

            # Report the new Euler rotations about their axes from the resultant
            # rotation matrix.
            combinedEulerX.append(math.atan2(R[t3][2,1], R[t3][2,2]))
            combinedEulerY.append(math.asin(R[t3][0,2]))
            combinedEulerZ.append(math.atan2(R[t3][1,0], R[t3][0,0]))

            # Obtain the cosine and sin of half of one of the new Euler rotations.
            combinedNetAngle.append(combinedEulerY[t3] * (180/np.pi))
            undoCombinedCos.append(np.cos(combinedNetAngle[t3] * (np.pi/360)))
            undoCombinedSin.append(np.sin(combinedNetAngle[t3] * (np.pi/360)))

            # Compute all 4 quaternion coordinates.
            combinedQuatW.append(undoCombinedCos[t3])
            combinedQuatX.append(undoCombinedSin[t3] * np.sin(0.5*np.pi) * np.cos(np.pi))
            combinedQuatY.append(0.01 * (yThighY[t3] / 2) *
            np.cos(yThighY[t3]/(undoCombinedSin[t3])))
            combinedQuatZ.append(0.01 * (yThighZ[t3] / 2) *
            np.cos(yThighZ[t3]/(undoCombinedSin[t3])))

            # Use the coordinates above to find the combined-Euler based net angle
            change.
            rebuildCombined.append(np.arccos(np.minimum(1,
                                      combinedQuatW[0]*combinedQuatW[t3] +
                                      combinedQuatX[0]*combinedQuatX[t3] +
                                      combinedQuatY[0]*combinedQuatY[t3] +
                                      combinedQuatZ[0]*combinedQuatZ[t3])) *(180/np.pi))

        # Set the net angle change to zero at the beginning of the plot interval.
        for t4 in range(0, len(xThighRoll)):
            rebuildCombinedFix.append(rebuildCombined[t4]-rebuildCombined[500])

        # Show the plots!
        fig, axs = plt.subplots()
        axs.set_title('Euler-Based Net Angle Changes for THIGH')
        axs.plot(rebuildCombinedFix, color='violet')
        axs.set_xlim(500, 750)
        axs.set_ylim(-20, 20)
        axs.set(xlabel='Time (seconds)', ylabel='Total Angle Change (degrees)')
        axs.invert_yaxis()
        positions = (500, 550, 600, 650, 700, 750)
        labels = (14.42, 15.86, 17.30, 18.75, 20.19, 21.63)
        plt.xticks(positions, labels)
        fig.show()

class PlotCalf:
    # PURPOSE: Plot the total angle change about the axis of the calf.

    def __init__(self, xCalfRoll = [], yCalfRoll = [], xCalfPitch = [],
                  yCalfPitch = [], xCalfYaw = [], yCalfYaw = [],
```

```
                 xCalfW = [], yCalfW = [], xCalfX = [], yCalfX = [],
                 xCalfY = [], yCalfY = [], xCalfZ = [], yCalfZ = [],
                 changeCalf = []):
        self.xCalfRoll = xCalfRoll
        self.yCalfRoll = yCalfRoll
        self.xCalfPitch = xCalfPitch
        self.yCalfPitch = yCalfPitch
        self.xCalfYaw = xCalfYaw
        self.yCalfYaw = yCalfYaw

        self.xCalfW = xCalfW
        self.yCalfW = yCalfW
        self.xCalfX = xCalfX
        self.yCalfX = yCalfX
        self.xCalfY = xCalfY
        self.yCalfY = yCalfY
        self.xCalfZ = xCalfZ
        self.yCalfZ = yCalfZ

        self.changeCalf = changeCalf

    def euler_angle_calf(self, xCalfRoll, yCalfRoll, xCalfPitch, yCalfPitch, xCalfYaw,
    yCalfYaw):
        figCalf, axsCalf = plt.subplots(3, sharex = True, sharey = False)
        figCalf.suptitle('Euler Axis Rotations for CALF')

        with open('angles_calf_roll2.csv', 'r') as csvfile:
            plots = csv.reader(csvfile, delimiter=',')
            for row in plots:
                xCalfRoll.append(float(row[0]))
                yCalfRoll.append(float(row[1]))
        setRoll2Zero = []
        for t in range(0, len(xCalfRoll)):
            setRoll2Zero.append(yCalfRoll[t]-yCalfRoll[500])
        axsCalf[0].plot(xCalfRoll, setRoll2Zero, linewidth = 2, color='teal')
        axsCalf[0].set(xlabel='', ylabel='Roll')
        axsCalf[0].set_xlim(500, 750)
        axsCalf[0].set_ylim(20, -20)

        with open('angles_calf_pitch2.csv', 'r') as csvfile:
            plots = csv.reader(csvfile, delimiter=',')
            for row in plots:
                xCalfPitch.append(float(row[0]))
                yCalfPitch.append(float(row[1]))
        setPitch2Zero = []
        for t in range(0, len(xCalfPitch)):
            setPitch2Zero.append(yCalfPitch[t]-yCalfPitch[500])
        axsCalf[1].plot(xCalfPitch, setPitch2Zero, linewidth = 2, color='magenta')
        axsCalf[1].set(xlabel='', ylabel='Pitch')
        axsCalf[1].set_xlim(500, 750)
        axsCalf[1].set_ylim(50, -50)

        with open('angles_calf_yaw2.csv', 'r') as csvfile:
            plots = csv.reader(csvfile, delimiter=',')
            for row in plots:
                xCalfYaw.append(float(row[0]))
                yCalfYaw.append(float(row[1]))
        setYaw2Zero = []
        for t in range(0, len(xCalfYaw)):
            setYaw2Zero.append(yCalfYaw[t]-yCalfYaw[500])
        axsCalf[2].plot(xCalfYaw, setYaw2Zero, linewidth = 2, color='black')
        axsCalf[2].set(xlabel='', ylabel='Yaw')
        axsCalf[2].set_xlim(500, 750)
        axsCalf[2].set_ylim(20, -20)

        # Optional!
        # figCalf.show()
```

```python

    def quaternion_calf(self, xCalfW, yCalfW, xCalfX, yCalfX, xCalfY,
                        yCalfY, xCalfZ, yCalfZ):
        figQuats, axsQuats = plt.subplots(4, sharex = True, sharey = False)
        figQuats.suptitle('Quaternion Axis Rotations for CALF')

        with open('angles_calf_W2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xCalfW.append(float(row[0]))
                yCalfW.append(float(row[1]))
        axsQuats[0].plot(xCalfW, yCalfW, color='blue')
        axsQuats[0].set(xlabel='', ylabel='W')
        axsQuats[0].set_xlim(500, 750)

        with open('angles_calf_X2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xCalfX.append(float(row[0]))
                yCalfX.append(float(row[1]))
        axsQuats[1].plot(xCalfX, yCalfX, color='red')
        axsQuats[1].set(xlabel='', ylabel='X')
        axsQuats[1].set_xlim(500, 750)

        with open('angles_calf_Y2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xCalfY.append(float(row[0]))
                yCalfY.append(float(row[1]))
        axsQuats[2].plot(xCalfY, yCalfY, color='green')
        axsQuats[2].set(xlabel='', ylabel='Y')
        axsQuats[2].set_xlim(500, 750)

        with open('angles_calf_Z2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xCalfZ.append(float(row[0]))
                yCalfZ.append(float(row[1]))
        axsQuats[3].plot(xCalfZ, yCalfZ, color='orange')
        axsQuats[3].set(xlabel='Time (count)', ylabel='Z')
        axsQuats[3].set_xlim(500, 750)

        # Optional!
        # figQuats.show()

    def dot_product_calf(self, xCalfW, yCalfW, xCalfX, yCalfX, xCalfY,
                         yCalfY, xCalfZ, yCalfZ, changeCalf):
        oneRad2Degrees = 57.296
        changeCalfFix = []
        for t1 in range(0, len(xCalfW)):
            changeCalf.append(np.arccos(np.minimum(1, yCalfW[0]*yCalfW[t1] +
                                        yCalfX[0]*yCalfX[t1] +
                                        yCalfY[0]*yCalfY[t1] +

                                        yCalfZ[0]*yCalfZ[t1]))*(180/np.pi)-(oneRad2Deg
                                        rees/2))

        for t2 in range(0, len(xCalfW)):
            changeCalfFix.append(changeCalf[t2]-changeCalf[500])

        fig, axs = plt.subplots()
        axs.set_title('Quaternion-Based Net Angle Changes for CALF')
        axs.plot(changeCalfFix, color='blue')
        axs.set_xlim(500, 750)
        axs.set_ylim(-20, 20)
        axs.set(xlabel='Time (seconds)', ylabel='Total Angle Change (degrees)')
        axs.invert_yaxis()
```

```
            positions = (500, 550, 600, 650, 700, 750)
            labels = (14.42, 15.86, 17.30, 18.75, 20.19, 21.63)
            plt.xticks(positions, labels)
            fig.show()

        def euler_combo_calf(self, xCalfRoll, yCalfRoll, yCalfPitch, yCalfYaw,
                             yCalfY, yCalfZ):
            theta_array = []
            R = []

            combinedEulerX = []
            combinedEulerY = []
            combinedEulerZ = []

            combinedNetAngle = []
            undoCombinedCos = []
            undoCombinedSin = []

            combinedQuatW = []
            combinedQuatX = []
            combinedQuatY = []
            combinedQuatZ = []

            rebuildCombined = []
            rebuildCombinedFix = []

            for t3 in range(0, len(xCalfRoll)):
                theta = [yCalfRoll[t3] * (np.pi/180), yCalfPitch[t3]* (np.pi/180),
                yCalfYaw[t3]* (np.pi/180)]
                theta_array.append(theta)

                R_x = np.array([[1,         0,                    0                    ],
                                [0,         math.cos(theta[0]), -math.sin(theta[0]) ],
                                [0,         math.sin(theta[0]), math.cos(theta[0])  ]
                               ])



               R_y = np.array([[math.cos(theta[1]),    0,      math.sin(theta[1])  ],
                               [0,                     1,      0                   ],
                               [-math.sin(theta[1]),   0,      math.cos(theta[1])  ]
                               ])

               R_z = np.array([[math.cos(theta[2]),    -math.sin(theta[2]),    0],
                               [math.sin(theta[2]),    math.cos(theta[2]),     0],
                               [0,                     0,                      1]
                               ])

               R.append(np.dot(R_x, np.dot(R_y, R_z)))

               combinedEulerX.append(math.atan2(R[t3][2,1], R[t3][2,2]))
               combinedEulerY.append(math.asin(R[t3][0,2]))
               combinedEulerZ.append(math.atan2(R[t3][1,0], R[t3][0,0]))

               combinedNetAngle.append(combinedEulerY[t3] * (180/np.pi))
               undoCombinedCos.append(np.cos(combinedNetAngle[t3] * (np.pi/360)))
               undoCombinedSin.append(np.sin(combinedNetAngle[t3] * (np.pi/360)))

               combinedQuatW.append(undoCombinedCos[t3])
               combinedQuatX.append(undoCombinedSin[t3] * np.sin(0.5*np.pi) * np.cos(np.pi))
               combinedQuatY.append(0.01 * (yCalfY[t3] / 2) *
               np.cos(yCalfY[t3]/(undoCombinedSin[t3])))
               combinedQuatZ.append(0.01 * (yCalfZ[t3] / 2) *
               np.cos(yCalfZ[t3]/(undoCombinedSin[t3])))

               rebuildCombined.append(np.arccos(np.minimum(1,
                                       combinedQuatW[0]*combinedQuatW[t3] +
```

```python
                                    combinedQuatX[0]*combinedQuatX[t3] +
                                    combinedQuatY[0]*combinedQuatY[t3] +
                                    combinedQuatZ[0]*combinedQuatZ[t3])) *(180/np.pi))

        for t4 in range(0, len(xCalfRoll)):
            rebuildCombinedFix.append(rebuildCombined[t4]-rebuildCombined[500])

        fig, axs = plt.subplots()
        axs.set_title('Euler-Based Net Angle Changes for CALF')
        axs.plot(rebuildCombinedFix, color='violet')
        axs.set_xlim(500, 750)
        axs.set_ylim(-20, 20)
        axs.set(xlabel='Time (seconds)', ylabel='Total Angle Change (degrees)')
        axs.invert_yaxis()
        positions = (500, 550, 600, 650, 700, 750)
        labels = (14.42, 15.86, 17.30, 18.75, 20.19, 21.63)
        plt.xticks(positions, labels)
        fig.show()

class PlotFoot:
    # PURPOSE: Plot the total angle change about the axis of the foot.

    def __init__(self, xFootRoll = [], yFootRoll = [], xFootPitch = [],
                 yFootPitch = [], xFootYaw = [], yFootYaw = [],
                 xFootW = [], yFootW = [], xFootX = [], yFootX = [],
                 xFootY = [], yFootY = [], xFootZ = [], yFootZ = [],
                 changeFoot = []):
        self.xFootRoll = xFootRoll
        self.yFootRoll = yFootRoll
        self.xFootPitch = xFootPitch
        self.yFootPitch = yFootPitch
        self.xFootYaw = xFootYaw
        self.yFootYaw = yFootYaw

        self.xFootW = xFootW
        self.yFootW = yFootW
        self.xFootX = xFootX
        self.yFootX = yFootX
        self.xFootY = xFootY
        self.yFootY = yFootY
        self.xFootZ = xFootZ
        self.yFootZ = yFootZ

        self.changeFoot = changeFoot

    def euler_angle_foot(self, xFootRoll, yFootRoll, xFootPitch, yFootPitch, xFootYaw,
    yFootYaw):
        figFoot, axsFoot = plt.subplots(3, sharex = True, sharey = False)
        figFoot.suptitle('Euler Axis Rotations for FOOT')

        with open('angles_foot_roll2.csv', 'r') as csvfile:
            plots = csv.reader(csvfile, delimiter=',')
            for row in plots:
                xFootRoll.append(float(row[0]))
                yFootRoll.append(float(row[1]))
        setRoll2Zero = []
        for t in range(0, len(xFootRoll)):
            setRoll2Zero.append(yFootRoll[t]-yFootRoll[500])
        axsFoot[0].plot(xFootRoll, setRoll2Zero, linewidth = 2, color='teal')
        axsFoot[0].set(xlabel='', ylabel='Roll')
        axsFoot[0].set_xlim(500, 750)
        axsFoot[0].set_ylim(20, -20)

        with open('angles_foot_pitch2.csv', 'r') as csvfile:
            plots = csv.reader(csvfile, delimiter=',')
            for row in plots:
                xFootPitch.append(float(row[0]))
```

```
                yFootPitch.append(float(row[1]))
        setPitch2Zero = []
        for t in range(0, len(xFootPitch)):
            setPitch2Zero.append(yFootPitch[t]-yFootPitch[500])
        axsFoot[1].plot(xFootPitch, setPitch2Zero, linewidth = 2, color='magenta')
        axsFoot[1].set(xlabel='', ylabel='Pitch')
        axsFoot[1].set_xlim(500, 750)
        axsFoot[1].set_ylim(50, -50)

        with open('angles_foot_yaw2.csv', 'r') as csvfile:
            plots = csv.reader(csvfile, delimiter=',')
            for row in plots:
                xFootYaw.append(float(row[0]))
                yFootYaw.append(float(row[1]))
        setYaw2Zero = []
        for t in range(0, len(xFootYaw)):
            setYaw2Zero.append(yFootYaw[t]-yFootYaw[500])
        axsFoot[2].plot(xFootYaw, setYaw2Zero, linewidth = 2, color='black')
        axsFoot[2].set(xlabel='', ylabel='Yaw')
        axsFoot[2].set_xlim(500, 750)
        axsFoot[2].set_ylim(20, -20)

        # Optional!
        # figFoot.show()

    def quaternion_foot(self, xFootW, yFootW, xFootX, yFootX, xFootY,
                        yFootY, xFootZ, yFootZ):
        figQuats, axsQuats = plt.subplots(4, sharex = True, sharey = False)
        figQuats.suptitle('Quaternion Axis Rotations for FOOT')

        with open('angles_foot_W2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xFootW.append(float(row[0]))
                yFootW.append(float(row[1]))
        axsQuats[0].plot(xFootW, yFootW, color='blue')
        axsQuats[0].set(xlabel='', ylabel='W')
        axsQuats[0].set_xlim(500, 750)

        with open('angles_foot_X2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xFootX.append(float(row[0]))
                yFootX.append(float(row[1]))
        axsQuats[1].plot(xFootX, yFootX, color='red')
        axsQuats[1].set(xlabel='', ylabel='X')
        axsQuats[1].set_xlim(500, 750)

        with open('angles_foot_Y2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xFootY.append(float(row[0]))
                yFootY.append(float(row[1]))
        axsQuats[2].plot(xFootY, yFootY, color='green')
        axsQuats[2].set(xlabel='', ylabel='Y')
        axsQuats[2].set_xlim(500, 750)

        with open('angles_foot_Z2.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xFootZ.append(float(row[0]))
                yFootZ.append(float(row[1]))
        axsQuats[3].plot(xFootZ, yFootZ, color='orange')
        axsQuats[3].set(xlabel='Time (count)', ylabel='Z')
        axsQuats[3].set_xlim(500, 750)

        # Optional!
```

```python
        # figQuats.show()

    def dot_product_foot(self, xFootW, yFootW, xFootX, yFootX, xFootY,
                         yFootY, xFootZ, yFootZ, changeFoot):
        oneRad2Degrees = 57.296
        changeFootFix = []
        for t1 in range(0, len(xFootW)):
            changeFoot.append(np.arccos(np.minimum(1, yFootW[0]*yFootW[t1] +
                                               yFootX[0]*yFootX[t1] +
                                               yFootY[0]*yFootY[t1] +

                                               yFootZ[0]*yFootZ[t1]))*(180/np.pi)-(oneRad2Deg
                                               rees/2))

        for t2 in range(0, len(xFootW)):
            changeFootFix.append(changeFoot[t2]-changeFoot[500])

        fig, axs = plt.subplots()
        axs.set_title('Quaternion-Based Net Angle Changes for FOOT')
        axs.plot(changeFootFix, color='blue')
        axs.set_xlim(500, 750)
        axs.set_ylim(-20, 20)
        axs.set(xlabel='Time (seconds)', ylabel='Total Angle Change (degrees)')
        axs.invert_yaxis()
        positions = (500, 550, 600, 650, 700, 750)
        labels = (14.42, 15.86, 17.30, 18.75, 20.19, 21.63)
        plt.xticks(positions, labels)
        fig.show()

    def euler_combo_foot(self, xFootRoll, yFootRoll, yFootPitch, yFootYaw,
                          yFootY, yFootZ):
        theta_array = []
        R = []

        combinedEulerX = []
        combinedEulerY = []
        combinedEulerZ = []

        combinedNetAngle = []
        undoCombinedCos = []
        undoCombinedSin = []

        combinedQuatW = []
        combinedQuatX = []
        combinedQuatY = []
        combinedQuatZ = []

        rebuildCombined = []
        rebuildCombinedFix = []

        for t3 in range(0, len(xFootRoll)):
            theta = [yFootRoll[t3] * (np.pi/180), yFootPitch[t3]* (np.pi/180),
            yFootYaw[t3]* (np.pi/180)]
            theta_array.append(theta)

            R_x = np.array([[1,         0,                    0                  ],
                            [0,         math.cos(theta[0]), -math.sin(theta[0]) ],
                            [0,         math.sin(theta[0]), math.cos(theta[0])  ]
                            ])



            R_y = np.array([[math.cos(theta[1]),    0,      math.sin(theta[1])  ],
                            [0,                     1,      0                   ],
                            [-math.sin(theta[1]),   0,      math.cos(theta[1])  ]
                            ])

```

```python
            R_z = np.array([[math.cos(theta[2]),    -math.sin(theta[2]),     0],
                            [math.sin(theta[2]),    math.cos(theta[2]),      0],
                            [0,                     0,                       1]
                            ])

            R.append(np.dot(R_x, np.dot(R_y, R_z)))

            combinedEulerX.append(math.atan2(R[t3][2,1], R[t3][2,2]))
            combinedEulerY.append(math.asin(R[t3][0,2]))
            combinedEulerZ.append(math.atan2(R[t3][1,0], R[t3][0,0]))

            combinedNetAngle.append(combinedEulerY[t3] * (180/np.pi))
            undoCombinedCos.append(np.cos(combinedNetAngle[t3] * (np.pi/360)))
            undoCombinedSin.append(np.sin(combinedNetAngle[t3] * (np.pi/360)))

            combinedQuatW.append(undoCombinedCos[t3])
            combinedQuatX.append(undoCombinedSin[t3] * np.sin(0.5*np.pi) * np.cos(np.pi))
            combinedQuatY.append(0.01 * (yFootY[t3] / 2) * np.cos(yFootY[t3]/(undoCombinedSin[t3])))
            combinedQuatZ.append(0.01 * (yFootZ[t3] / 2) * np.cos(yFootZ[t3]/(undoCombinedSin[t3])))

            rebuildCombined.append(np.arccos(np.minimum(1,
                                    combinedQuatW[0]*combinedQuatW[t3] +
                                    combinedQuatX[0]*combinedQuatX[t3] +
                                    combinedQuatY[0]*combinedQuatY[t3] +
                                    combinedQuatZ[0]*combinedQuatZ[t3])) *(180/np.pi))

        for t4 in range(0, len(xFootRoll)):
            rebuildCombinedFix.append(rebuildCombined[t4]-rebuildCombined[500])

        fig, axs = plt.subplots()
        axs.set_title('Euler-Based Net Angle Changes for FOOT')
        axs.plot(rebuildCombinedFix, color='violet')
        axs.set_xlim(500, 750)
        axs.set_ylim(-20, 20)
        axs.set(xlabel='Time (seconds)', ylabel='Total Angle Change (degrees)')
        axs.invert_yaxis()
        positions = (500, 550, 600, 650, 700, 750)
        labels = (14.42, 15.86, 17.30, 18.75, 20.19, 21.63)
        plt.xticks(positions, labels)
        fig.show()

class PlotLegRaising:
    # PURPOSE: Plot the total angle change for when a person is sitting and
    # raising a leg by up to 90 degrees.

    def __init__(self, xLegW = [], yLegW = [], xLegX = [], yLegX = [],
                 xLegY = [], yLegY = [], xLegZ = [], yLegZ = [],
                 changeLeg = [], pointsMinMax = []):
        self.xLegW = xLegW
        self.yLegW = yLegW
        self.xLegX = xLegX
        self.yLegX = yLegX
        self.xLegY = xLegY
        self.yLegY = yLegY
        self.xLegZ = xLegZ
        self.yLegZ = yLegZ

        self.changeLeg = changeLeg
        self.pointsMinMax = pointsMinMax

    def leg_quat_analysis(self, xLegW, yLegW, xLegX, yLegX, xLegY,
                          yLegY, xLegZ, yLegZ):
        figLeg, axsLeg = plt.subplots(4, sharex = True, sharey = False)
        figLeg.suptitle('Sitting/Leg Raising Quaternions')

```

```
        with open('test_quat_wholeleg_w.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xLegW.append(float(row[0]))
                yLegW.append(float(row[1]))
        axsLeg[0].plot(xLegW,yLegW,linewidth=2, color='teal')
        axsLeg[0].set(xlabel='', ylabel="Quat'n (W)")
        axsLeg[0].set_xlim(50, 100)

        with open('test_quat_wholeleg_x.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xLegX.append(float(row[0]))
                yLegX.append(float(row[1]))
        axsLeg[1].plot(xLegX,yLegX,linewidth=2, color='red')
        axsLeg[1].set(xlabel='', ylabel="Quat'n (X)")
        axsLeg[1].set_xlim(50, 100)

        with open('test_quat_wholeleg_y.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xLegY.append(float(row[0]))
                yLegY.append(float(row[1]))
        axsLeg[2].plot(xLegY,yLegY,linewidth=2, color='green')
        axsLeg[2].set(xlabel='', ylabel="Quat'n (Y)")
        axsLeg[2].set_xlim(50, 100)

        with open('test_quat_wholeleg_z.csv', 'r') as csvfile:
            plots= csv.reader(csvfile, delimiter=',')
            for row in plots:
                xLegZ.append(float(row[0]))
                yLegZ.append(float(row[1]))
        axsLeg[3].plot(xLegZ,yLegZ,linewidth=2, color='orange')
        axsLeg[3].set(xlabel='Time (Count)', ylabel="Quat'n (Z)")
        axsLeg[3].set_xlim(50, 100)
        # Display the matplotlab figure showing quaternion behaviors in the
        # raising leg (optional).
        # figLeg.show()

    def leg_net_angles(self, xLegW, yLegW, xLegX, yLegX, xLegY,
                       yLegY, xLegZ, yLegZ, changeLeg, pointsMinMax):
        # Append changeLeg with the total angle change, which equates to
        # the inverse cosine of the dot product for each quaternion at the
        # initial and final time points, all multiplied by 360 degrees over
        # pi (for converting from radians to degrees).
        for t1 in range(0, len(xLegW)):
            changeLeg.append(np.arccos(np.minimum(1, yLegW[0] * yLegW[t1] +
                           yLegX[0] * yLegX[t1] +
                           yLegY[0] * yLegY[t1] +
                           yLegZ[0] * yLegZ[t1]))*(360/np.pi))

        # Plot the total angle change for the raising leg based on the
        # quaternions using the changeLeg array. Limit the x-axis to
        # 50-100, and invert and limit the y-axis to 90-0.
        figAngleLeg, axsAngleLeg = plt.subplots()
        axsAngleLeg.set_title('Sitting/Leg-Raising')
        axsAngleLeg.set_ylabel("Total Angle Change (Degrees)")
        axsAngleLeg.set_xlabel('Time (Count)')
        axsAngleLeg.plot(changeLeg, color='blue')
        axsAngleLeg.set_xlim(50, 100)
        axsAngleLeg.set_ylim(90, 0)

        # Narrow down the time interval to 50-100, and append pointMinMax with
        # the y-values occuring within that interval.
        for t2 in range(50, 101):
            pointsMinMax.append(changeLeg[t2])
```

```
        # Determine the highest and lowest values of pointMinMax, and find their
        # locations within the x-axis.
        xmax = pointsMinMax.index(max(pointsMinMax))+50
        ymax = max(pointsMinMax)
        xmin = pointsMinMax.index(min(pointsMinMax))+50
        ymin = min(pointsMinMax)

        # Annotate the highest point in the plot within the selected interval.
        text1= "Max Angle: {:.3f}° \nTime: {:.3f}".format(ymax, xmax)
        bbox_props1 = dict(boxstyle="square,pad=0.3", fc="w", ec="k", lw=0.72)
        arrowprops1=dict(arrowstyle="->", lw=1.5)
        kw1 = dict(xycoords='data',textcoords="axes fraction",
                  arrowprops=arrowprops1, bbox=bbox_props1, ha="left", va="top")
        axsAngleLeg.annotate(text1, xytext=(0.4, 0.0925), xy=(xmax, ymax), **kw1)

        # Annotate the lowest point in the plot within the selected interval.
        text2= "Min Angle: {:.3f}° \nTime: {:.3f}".format(ymin, xmin)
        bbox_props2 = dict(boxstyle="square,pad=0.3", fc="w", ec="k", lw=0.72)
        arrowprops2=dict(arrowstyle="->", lw=1.5)
        kw2 = dict(xycoords='data',textcoords="axes fraction",
                  arrowprops=arrowprops2, bbox=bbox_props2, ha="left", va="bottom")
        axsAngleLeg.annotate(text2, xytext=(0.18, 0.85), xy=(xmin, ymin), **kw2)

        # Display the matplotlab figure showing the total angle change in the
        # raising leg.
        figAngleLeg.show()
```

Code B: TeamUSDataProcessingFinal2020.py

```python
# Lee Bradley, Martha Gizaw, Nate Winneg
# Engineering Physics Capstone Project
# Unstable Seniors: Data Processing
# May 2020

# Import the libraries below
from DebugSubroutinesTeamUS import PlotThigh, PlotCalf, PlotFoot, PlotLegRaising

# Turn on/off the following debug variables to control which human feature to
# look at.
debugThigh = False # Change to True if you wish to visualize the thigh data.
debugCalf = False # Change to True if you wish to visualize the calf data.
debugFoot = False # Change to True if you wish to visualize the foot data.
debugLeg = False # Change to True if you wish to visualize the raising leg data.

# Initialize the following variables as empty arrays.
xRoll = []
yRoll = []

xPitch = []
yPitch = []

xYaw = []
yYaw = []

xQuatW = []
yQuatW = []

xQuatX = []
yQuatX = []

xQuatY = []
yQuatY = []

xQuatZ = []
yQuatZ = []

netAngleChange = []
pointsMinMax = []

# Call the subroutines by turning on only one debug value for any human feature.
if (debugThigh == True) and (debugCalf == False) and (debugFoot == False) and (debugLeg == False):
    thigh = PlotThigh()
    thigh.euler_angle_thigh(xRoll, yRoll, xPitch, yPitch, xYaw, yYaw)
    thigh.quaternion_thigh(xQuatW, yQuatW, xQuatX, yQuatX, xQuatY, yQuatY, xQuatZ, yQuatZ)
    thigh.dot_product_thigh(xQuatW, yQuatW, xQuatX, yQuatX, xQuatY, yQuatY, xQuatZ, yQuatZ, netAngleChange)
    thigh.euler_combo_thigh(xRoll, yRoll, yPitch, yYaw, yQuatY, yQuatZ)

elif (debugThigh == False) and (debugCalf == True) and (debugFoot == False) and (debugLeg == False):
    calf = PlotCalf()
    calf.euler_angle_calf(xRoll, yRoll, xPitch, yPitch, xYaw, yYaw)
    calf.quaternion_calf(xQuatW, yQuatW, xQuatX, yQuatX, xQuatY, yQuatY, xQuatZ, yQuatZ)
    calf.dot_product_calf(xQuatW, yQuatW, xQuatX, yQuatX, xQuatY, yQuatY, xQuatZ, yQuatZ, netAngleChange)
    calf.euler_combo_calf(xRoll, yRoll, yPitch, yYaw, yQuatY, yQuatZ)

elif (debugThigh == False) and (debugCalf == False) and (debugFoot == True) and (debugLeg == False):
    foot = PlotFoot()
    foot.euler_angle_foot(xRoll, yRoll, xPitch, yPitch, xYaw, yYaw)
    foot.quaternion_foot(xQuatW, yQuatW, xQuatX, yQuatX, xQuatY, yQuatY, xQuatZ, yQuatZ)
    foot.dot_product_foot(xQuatW, yQuatW, xQuatX, yQuatX, xQuatY, yQuatY, xQuatZ, yQuatZ, netAngleChange)
```

```
        foot.euler_combo_foot(xRoll, yRoll, yPitch, yYaw, yQuatY, yQuatZ)

    elif (debugThigh == False) and (debugCalf == False) and (debugFoot == False) and
    (debugLeg == True):
        leg = PlotLegRaising()
        leg.leg_quat_analysis(xQuatW, yQuatW, xQuatX, yQuatX, xQuatY, yQuatY, xQuatZ, yQuatZ)
        leg.leg_net_angles(xQuatW, yQuatW, xQuatX, yQuatX, xQuatY, yQuatY, xQuatZ, yQuatZ,
        netAngleChange, pointsMinMax)

    else:
        print("Sorry, but you would rather want to look at the plots one human" +
              " feature at a time and explain them before moving on. Please turn" +
              " off or turn on any of the debug variables provided to you, and" +
              " have only one of them turned on to plot the desired data.")

```

References:

- [1] "Falls." *World Health Organization*, World Health Organization, www.who.int/news-room/fact-sheets/detail/falls.
- [2] *An Automated Gait Feature Extraction Method for Identifying Gait Asymmetry Using Wearable Sensors.* Arif Reza Anwary[1], Hongnian Yu[1], Michael Vassallo[2].
- [3] *The effect of hip abductor fatigue on static balance and gait parameters*. Wonjeong Hwang[a], Jun Ha Jang[b], Minjin Huh[b], Yeon Ju Kim[b], Sang Won Kim[b], In Ui Hong[b], and Mi Young Lee[b]
- [4] CR2032 Battery. https://www.batteryjunction.com/panasonic-cr2032-bulk.html?gclid=Cj0KCQjw7qn1BRDqARIsAKMbHDZVC0noGta2xe8_qEa5NcyxfxJN2AHfen8-AlZARjZY6FtguoGNSlwaAqf_EALw_wcB
- [5] ARM Mbed LPC1768 Microcontroller. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc1700-cortex-m3/arm-mbed-lpc1768-board:OM11043
- [6] Bosch BNO055 IMU. https://learn.adafruit.com/adafruit-bno055-absolute-orientation-sensor
- [7] mbed MAX32630FTHR. https://os.mbed.com/platforms/MAX32630FTHR/
- [8] NodeMCU ESP8266 12-E. https://nodemcu.readthedocs.io/en/master/
- [9] ESP32 Dev Kit 3C. https://www.espressif.com/en/products/devkits/esp32-devkitc/overview
- [10] 500mAh LiPo Battery. https://www.adafruit.com/product/1578
- [11] Adafruit. "Adafruit Unified Sensor Library." *GitHub*, 4 Feb. 2020, github.com/adafruit/Adafruit_BNO055.
- [12] "Paho-Mqtt." *PyPI*, pypi.org/project/paho-mqtt/.
- [13] "Rotations, Orientation, and Quaternions." *Rotations, Orientation, and Quaternions - MATLAB & Simulink*, www.mathworks.com/help/fusion/examples/rotations-orientation-and-quaternions.html.