Jerry Chen
A13310365
March 06, 2018

# ECE 16 Lab #3: Data Visualization & Signal Processing

## 1. Introduction

This lab introduced us to data sampling with an EMG-EKG Olimex Shield (specifically electrical impulses), plotting data with python using MatPlotLib, filtering data into a more readable form with pre-made and manual filters, plotting live data, and using BLE communication between two Arduino boards. The first objective showed how to sample data from the EMG-EKG Olimex Shield (in this case the electrical impulses and contractions). Objective 2 taught how to plot that information in python, using MatPlotLib. Objective 3 demonstrated how to filter information with pre-made filters, while objective 4 taught us the inner workings of one of the filters (the Infinite Impulse Response filter). Objective 5 introduced us to plotting data live, and objective 6 introduced us to live data communication between Arduino boards using Bluetooth Low Energy.

## 2. Objective 1: Reading EMG

Objective 1 involves periodically sampling data from an attached Olimex EKG-EMG shield via analog pin on the Arduino board and reading it in two forms: raw data bits and mV using sketches named **ReadEMGRaw.ino** and **ReadEMGScaled.ino** respectively, and then save it using a python script **data_collection.py** to **EMG_raw_data.txt** and **EMG_scaled_data.txt** respectively.

In **ReadEMGRaw.ino**, we will start by including library CurieTimerOne.h defining the global variables, namely int(s) *startTime*, *analogPin*, *lastReading*, *timeSinceStart*, and bool *newRead*. In the setup function, we set up Serial (115200 in this case), set the analogPin we check to 0 and set the pin up for input. Then we set newRead to false, start the timer to sample using a to-be-defined function *sample* at 200Hz or 5000 microseconds per sample, and finally record the time sampling begin with *micros()*. IT IS CRITICAL THAT THIS OCCURS HERE AS EACH PRECEDING STATEMENT TAKES TIME TO EXECUTE. We now define function *sample.* Each time it triggers In loop, it sets a new *lastReading* to represent the changed reading from the EMG shield, sets a new *timeSinceStart* using the difference between *micros()* and *startTime*, and flags newRead as true. Finally, in the loop statement, we check for whether or not a newRead has been produced (holds true). If so, we print out the last raw reading as merely lastReading and the corresponding timeSinceStart in microseconds with accompanying indicating statements. Once all this has been printed, we set newRead to false.

**ReadEMGScaled.ino** is essentially the same code, but rather than printing merely lastReading, it will first convert the raw reading into a scaled reading by multiplying the result by 3.3 dividing by 1024.0, then subtracting 1.5 and finally dividing by 3.600 to the result in mV.

**data_collection.py** begins by importing serial and matching the serial port the serial rate to that set by the Arduino (115200) using two variables and setting a printCount to 0. we then open up the serial connection and read the first line, decode it from ASCII to UTF-8 and check if the data is raw or scaled depending one which is in the text file. We then open the corresponding text file to write it to and write the samples given for 5 seconds. Since the samples occur at 200Hz, that is a total of 200*5 samples. So we will have a while loop that ends when 1000 samples have been written to a text file. At

Now both times we run the Arduino files and the python sketch and attach the board to the computer and the Olimex shield to the board and the muscle to the shield, we will flex the muscle at 1Hz. The attached files are the result of this.

## 3. Objective 2: Plotting

Now it is time to plot the data from the previous objective and Lab 2's objective 5 gyro and accelerometer data using two sketches we write: **plot_EMG.py** and **plot_IMU.py**. Both will require us to *import matplotlib.pyplot*.

In **plot_EMG.py**, we open **EMG_raw_data.txt** for reading and read the entirety of the file into a variable *reading* before closing it. We now start a while loop that searches for every reading until every reading has been found and recorded, find the time at which each sample occurs and the corresponding sample and append it to arrays *xValue* and *yValue* and modify *reading* to remove everything prior to and at the next new line. We then plot the xValue and yValue, label the plot's title ('EMG raw data') and axes ('time (microseconds)' and 'Raw data bits'), set the axes to [0, 5000000, 0, 1024] and then show the plot. Then we repeat the code, only this time it is for **EMG_scaled_data.txt**, and modify the labels as needed.

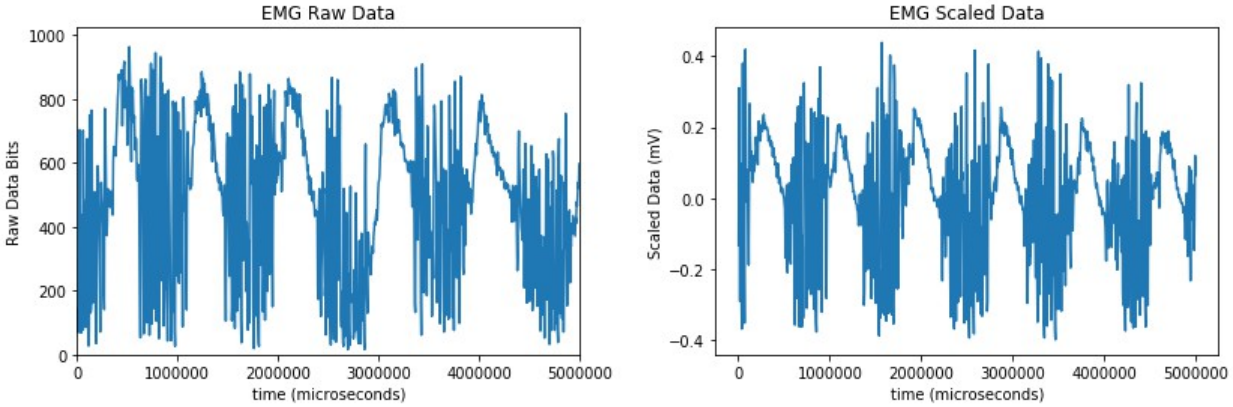


*Figure 1: The results of plot.py*

In **plot_IMU.py**, we open **gyro_output.txt** for reading and read the entirety of

the file into a variable *reading* before closing it. We now start a while loop that searches for every reading until every reading has been found and recorded, find the time at which each sample set occurs and the corresponding samples and append it to arrays *times, xValues, yValues* and *zValues* and modify *reading* to remove everything prior to and at the next new line. We then plot *times* with every other array into a subplot, label each subplot's title ('(insert_name_here) raw data') and axes ('time (ms)' and 'ang. Accel. (deg/s^2)') and then show the plot. Then we repeat the code, only this time it is for **accel_output.txt**, and modify the labels as needed.

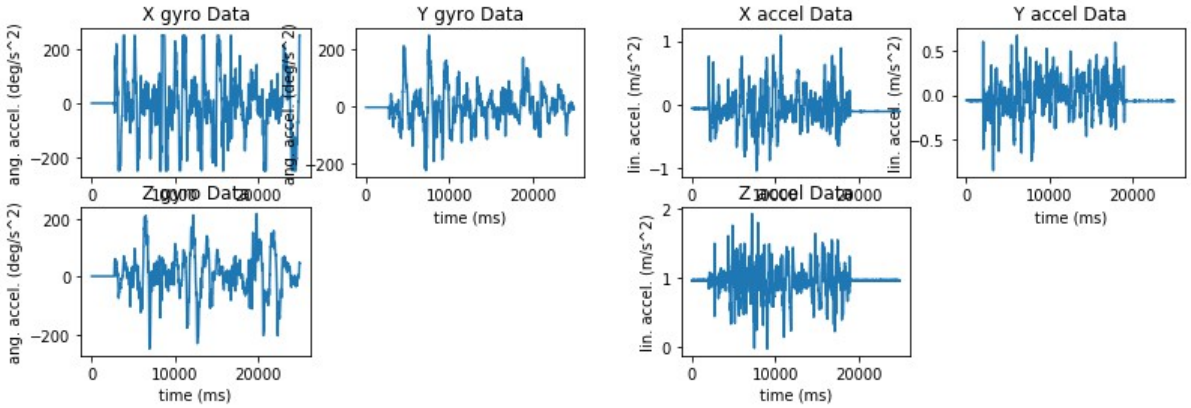


*Figure 2: The results of plot_IMU.py*

4. **Objective 3: Filtering Data**

   In this objective we will begin filtering data into a more readable form with a python script **plot_filters.py**. In this file, for the purpose of convenience, I have written a *plot_signal* function which plots provided data, labels title and the axes using either a default label or an inputted label and shows the resulting plot. After having written the function, we start reading EMG_scaled_data and record the data into two arrays like in the previous objective. We then plot and show the original signal to serve as a benchmark. Then we calculate and record the high pass coefficients and low pass coefficients and use them to calculate and record the high pass, low pass, and high low pass signals with lfilter and plot and show them, in that order. We then rectify the hi-lo pass signal, record and plot the rectified signal, use a boxcar filter with a moving average length of 100, and lfilter to smooth the signal, and plot and show it. Finally, we plot the power spectral density using the hi-lo pass signal and the sampling rate in Hz (200 in this case).

5. **Objective 4: Creating Filters**

   This objective is very similar to the previous objective, with the main difference being that we will only be finding a high pass and low pass resulting signal using our own

filter, defined by a function *my_filter*, a third order filter, based on the IIR filter we have been using for **plot_filters.py**. Essentially it teaches us how the IIR filter we have been using works. In the definition for our function *my_filter*, it contains the parameters for the *b* and *a* coefficients for the filters and the orignal signal. In the function we start an array new_signal which contains the first three samples of the original signal. We then use a for loop with a x variable x that goes from 3 to the length of the original filter and apply the algorithm for the 3rd order IIR filter. In the main code of the script we then repeat the plotting process we did in previous objectives, finding and recording the data from the **EMG_scaled_data.txt** and plotting the data and replotting after filtering using *plot_signal*, which we copy from **plot_filters.py**, only we replace the lfilter with my_filter

## 6. Objective 5: Streaming/Real-Time

This objective introduces how to plot data real time rather than plotting data once. Using a script we call **stream_filters.py**, we will plot the original signal, a high-low pass signal passed through the custom IIR filter in the previous objective, a rectified signal, a smoothed signal and the power spectral density onto 5 subplots live. To do this, we need to import *matplotlib.pyplot, matplotlib.animation,* and *scipy.signal*. We will also define each as *plt*, *animation* and *signal* respectively in this code. Once these imports have been completed, we will need to define a figure fig = plt.figure and define the 5 subplots. Then we write the function for the filter from objective 4, write the plot_signal function, which involves clearing the subplot defined by the function parameters and plotting the new data set in its place, and define the *main(i)* function for plotting each signal using plot_signal. Taking information from serial, we determine the next reading of the original signal, plot it, filter it through the high and low pass filters we created in objective 4 then plot that, plot the rectified and smoothed signals, and plot the power spectral frequency and end the function. Then in the actual main code of stream_filters (after the functions have been defined), we write the animation *ani = animation.FuncAnimation(fig, main, interval = 1000)* and then show the final plot. The result should be a plot with the five intended subplots.

## 7. Objective 6: Bluetooth Integration

Here we go to one of the most difficult objectives of any lab we perform: implementing a Bluetooth communication channel between two Arduino 101s: a Peripheral and a Central board. The Peripheral Arduino board will run a script called **Peripheral.ino** while the Central board will run a script called **Central.ino**. Ultimately the Peripheral board will pass IMU and analog data from itself to the Central board, which passes it to a second computer, which, with a python script, **BLE_plot.py**, will

plot the data.

In the **Peripheral.ino** script, we will start by including the *CurieBLE.h*, *CurieTimerOne.h*, and *CurieIMU.h* libraries. Next, we name the *BLEservice* we wish to advertise as *dataService*, as it transfers data from the Peripheral board to the Central board and define its characteristics and notification string. Then we need to initialize the variables to record the data we will pass to the Central board (*ax, ay, az, gx, gy, gz, pinVal1,* and *pinVal2*. In addition we will need a *newRead* boolean to trigger the *loop()* function's contents which transmits the data in the form of Strings to the Central board, and a char array to record the information to be transmitted before it is transmitted. In the setup() function we will set up Serial with a bit rate of 9600 and set up the accelerometer and gyro with ranges 5 and 250. Then we set up the BLE by applying the *begin()* function, set the advertised UUID, add the characteristics to *dataService*, set the initial values for the characteristics, set newRead to false, start the CurieTimerOne to trigger a function *samplingISR* every 5000 microseconds (5 milliseconds), and begin advertising the BLE. In function samplingISR, we sample from the accelerometer and save the values to *ax, ay, az*, gyroscope to *gx, gy, gz*, and analog pins 1 and 2 to *pinVal1* and *pinVal2* and set the *newRead* flag to true. In *loop()*, we now start by searching for BLE centrals to connect to with *BLEDevice central = BLE.central();*. Once it is found, we go into a *while* loop that will transmit the gathered data every time the *samplingISR* is triggered (*newRead == true*), making sure *newRead* is set to false every time, only exiting the while loop once the connection to central ends, starting the main *loop()* function again. Each time a connection is made or broken it should be printed out to make it clear to the user.

In the **Central.ino** script we will need to include CurieBLE.h and define the setup() function as such: begin Serial with a bit rate of 9600, print out the purpose of the central as *"BLE Central - Data Plotting"*, begin BLE and start scanning for the UUID defined in **Peripheral.ino**. In *loop()* we now check for a peripheral using *BLEDevice peripheral = BLE.available();* and print out if a peripheral has been found and critical information about the peripheral (*address, localName, and advertisedServiceUuid*) and end the scanning initiated by either *setup()* or last iteration of *loop()* and trigger to-be-defined function *print_data*, which will contain a *while()* loop that operates so long as the peripheral connection is maintained. Once that function stops running, restart the *scanForUuid* function and initiate the next iteration of loop. In the last function print_data, with parameter *BLEDevice peripheral*, it begins by printing an indication that it is trying to connect to the peripheral device and printing whether or not the connection was successful. If not, the function terminates with a return statement. If so, it checks if the peripheral has attributes and continues if so, and terminates if not. The *notifyCharacteristic* is retrieved and it is determined if it exists. If it is not present, the function terminates. If it is, then the function goes into a while loop that prints data from *notifyCharacteristic* so long as the peripheral connection is maintained. If the connection is broken the function ends upon printing out that the peripheral has been disconnected.

## 8. Problems and Issues

In Objective 5, some issues may occur when it comes to data retrieval.

In Objective 6, there will also be some issues with ble_plot.py (incomplete).

## 9. Conclusion

This lab ultimately introduced us to how to use data sampling and using CurieTimerOne over polling and CurieIMU to sample data from the in built accelerometer and gyroscope at regular intervals.