ECE16: Rapid hardware and software design for interfacing with the world
Lab 3: Data Visualization & Signal Processing

The goal of this lab is to introduce you to methods for plotting sampled data from sensors and to basic signal processing methods that you can apply to clean up your EMG signal and extract signal features of interest. **Please read through the entire lab before starting as you'll likely reuse code from earlier objectives in later objectives and having the big picture in mind could allow you to plan more effectively and save time implementing and testing!**

Objective One - Plot Raw & Scaled EMG Samples

In this first objective, you will first modify your code from Lab 2 to collect EMG data from the electrodes connected to the forearm, as you did for the Lab 1 checkoff. **NOTE: The target sample rate is 200 Hz, you will only be collecting data from one analog channel (use interrupt based sampling). You should write the raw ADC values and the time at which the values were sampled to a file. Time should be recorded with microsecond precision.** We will process the data in this file. Recall that the Arduino ADC has 10-bit precision, so this value is between 0 and 1023 (1024 distinct values, $2^{10} = 1024$).

The datafile should contain **at least 5 seconds of data**, during which you should aim to contract and release your forearm muscles at approximately 1 Hz. **NOTE: excessive movement of the EMG cabling can introduce noise artifacts; thus, you should try to set up data collection in a manner that minimizes extraneous movement, we suggest squeezing an object with your hand to generate good muscle activation with minimal movement.**

From these data, generate a plot with two subplots as follows:

Subplot 1: Read these data into Python and graph the raw EMG samples as a function of time in microseconds (time on the X-axis and EMG sample value on the Y-axis). **Only plot the first 5 seconds of data.** Be sure to label all axes so that they are descriptive and include the proper units. Also, give the plot a descriptive title.

Subplot 2: For this subplot, take the same EMG output and scale it as voltages. Again, **only plot the first 5 seconds of data.** Recall that the Arduino 101 ADC values represent an input voltage range of 0-3.3 Volts and that the EMG amplifier we are using has a gain factor of approximately 3600. Also, recall that there is a bias in the values recorded by the Arduino ADC, an ADC voltage of 1.5 volts corresponds to a 0 Volt input to the EMG amplifier.

Also, rescale the x-axes values so that time is in seconds (not microseconds). Be sure to label all axes so that they are descriptive and include the proper units. Also, give the plot a descriptive title.

Determine an offset and scale term that you can use to convert the ADC values to an estimate of the recorded EMG signal voltages using the following equation:

$$EMG\_Voltage = Scale\_Term * (ADC\_Value - Offset)$$

In the writeup for this objective provide the offset and scale term. Describe how you arrived at these values. Include your Arduino and Python code in the appendix. Include the plot (with both subplots described) in the Objective section. Make sure that your plot is human readable.

Objective Two - Signal Filtering, Rectification, and Power Estimation

Part 1: High Pass & Low Pass Filtering

As we discussed in class, the EMG signal can be affected by external noise sources (like 60 Hz interference). We may also see a slow bias in the signal and noise introduced by cable movement. Although the EMG shield hardware includes both high pass and low pass filtering circuits (analog filters), we can apply digital filters to further clean-up the signal. As shown in class, Scipy provides convenient methods for filtering signals.

To reduce the effect of bias we will use a simple high-pass filter. The following code calculates the coefficients of a high pass filter (first line) and then applies that filter to an input signal (second line):

```
b_high, a_high = scipy.signal.butter(3, 0.1, 'high', analog=False)
signal_out = scipy.signal.lfilter(b_high, a_high, signal_in)
```

To avoid higher-frequency noise, we will also use a simple low-pass filter. The following code calculates the coefficients of a low pass filter and then applies that filter to an input signal:

```
b_low, a_low = scipy.signal.butter(3, .5, 'low', analog=False)
signal_out = scipy.signal.lfilter(b_low, a_low, signal_in)
```

Let's see how these filters affect the EMG signal. Create a plot in which you show the unfiltered EMG signal (y-axis scaled to estimate EMG voltage and x-axis scaled to time, as in subplot two of Objective One), the effect of only the high-pass filter, the effect of only the low-pass filter, and the effect of both the high-pass and low-pass filter. As a result, this plot should have four subplots. For each subplot, **only plot the first 5 seconds of data from the data file you generated in Objective One.**

Include your Python code in the appendix. Include this plot in the Objective section and describe your approach and the effect of filtering. Make sure that your plot is human readable.

Part 2: Rectification & Smoothing

As described in class, a simple strategy for EMG power estimation is full-wave rectification and smoothing. Let's start with the output of the filtered EMG signal (the signal after applying both the high-pass and low-pass filters sequentially from Part 1 of this Objective). Apply rectification and smoothing to this signal and plot each step. **The plot should be composed of 3 sub-plots:**

1) The filtered EMG output (high-pass and low-pass filters applied from Part 1)
2) The rectified signal

3) Power estimation by smoothing. You can smooth the output using a moving average of 100 samples of data. This will estimate signal power as a 500 ms moving average of the full-wave rectified signal. You can achieve this by creating a "box car filter":

```
box = scipy.signal.boxcar(100)
signal_out = scipy.signal.lfilter(box, 1, signal_in)
```

For each subplot, **only plot the first 5 seconds of data from the data file you generated in Objective One.**

Include your Python code in the appendix. Include this plot in the Objective section and describe your approach and the effect of these processing steps. Make sure that your plot is human readable.

Objective Three - Real-time Signal Processing on the Arduino!

In this objective, you'll implement the same signal processing methods you tested and plotted in Python on the Arduino! The low-pass and high-pass filters that you applied in Objective Two are called Infinite Impulse Response (IIR) filters. An IIR filter has a simple structure and is defined by two lists of coefficients (*a* & *b* coefficients); this simple structure allows for filtering with low computational complexity, allowing filters to be run efficiently in Real-time as data is collected. The structure of the filter forms a recursion over previous values of the input signal AND the output of the filter:

$$y[n] = \frac{1}{a_0}\left(\sum_{i=0}^{P} b_i x[n-i] - \sum_{j=1}^{Q} a_j y[n-j]\right)$$

P is the length of list *a*
Q is the length of list *b*
x[n] is the input signal (i.e. the original signal)
y[n] is the output signal (i.e. the filtered signal)

You can get these numbers from the `b_high, a_high, b_low,` and `a_low` variables returned by the `scipy.signal.butter()` function calls in Objective Two. Note, you do not need to communicate these values to the running Arduino code for this objective, you copy these values directly into your Arduino code as hard-coded array initializations. Both the high pass and low pass filters we created in Objective Two are third order filters. P & Q (the lengths of *a* and *b*) are both equal to 4 for third order IIR filters. We've provided you with template code for third order IIR filtering.

The boxcar filter can also be written in the same form as low pass and high pass filters (in this case the *a = [1/Q]* and *b* is a vector of ones with a length *Q* that is the width of the boxcar filter. If you've taken a signal processing course, you might recognize that this has the form of a Finite Impulse Response (FIR) filter, since the length of *a* is 1 (i.e. the filter output does not depend on previous values of *y*, it only depends on previous values of *x*). However, given how simple the shape of the boxcar filter is, there is a much more computationally efficient way to compute the output of a boxcar filter. We've provided you with this code.

In Objective Two, you used Python to process the signal and plotted the output at every stage of processing. For this Objective, repeat the same signal processing process on an EMG signal that you are processing live as it is sampled and for every sample, print out every stage of signal processing. You can modify the Arduino code you wrote for Objective One to output:

1) The Arduino time (return value from micros()) at which the sample was taken
2) The raw integer sample
3) The sample scaled to represent EMG voltage in millivolts
4) The output of the high pass filter
5) The output of the low pass filter
6) The rectified signal
7) The output of the boxcar filter
8) The time spent filtering

Note: Again, sample the amplified EMG signal at 200 Hz. You'll be transmitting much more data than in previous Objectives (sending out 8 different values at 200 Hz). We suggest that you use a high baud rate (e.g. 250000 baud).

Collect a new set of EMG data by repeating the task from Objective One. The datafile should contain **at least 5 seconds of data**, during which you should aim to contract and release your forearm muscles at approximately 1 Hz. **NOTE: excessive movement of the EMG cabling can introduce noise artifacts; thus, you should try to set up data collection in a manner that minimizes extraneous movement.**

To verify the accuracy of your Arduino implementation, use Python to plot the data collected to generate graphs with each signal processing step (i.e. replicate many of the graphs from Objective Two). There will be a total of six graphs (signals 2-7 in the list above).

Include your Arduino and Python code in the appendix. Include these plots in the Objective section and describe your approach to implementing real-time signal processing on the Arduino. Make sure that your plots are human readable.

Objective Four - Streaming Arduino Data!

Part 1: Streaming from a File

As we move towards working with live EMG (and accelerometer/gyroscope) signals and designing controllers, it will be helpful to be able to visualize signals in real-time. Let's use the datafile to help design Python plotting code that will be able to handle data streaming over the serial port.

We want this code to read a set of samples from the serial port, process the samples, and update the display. Assume that we want to update the display every 100 milliseconds and that we want to show up to two seconds of data on screen. We will assume that the hardware is providing us with samples at the correct sampling rate (precisely 200 Hz). Let's break this process down into the following algorithm:

1. Initialize a set of plotting axes
2. Read 20 samples of the **processed signal** (100 ms of data that has been high pass filtered, low pass filtered, rectified, and boxcar filtered) from the the serial port
3. Add the samples to the end of an array of data
4. If the array of data contains more than two seconds of data, remove the extra samples from the beginning of the array
5. Plot the data on the axes initialized in step 1
6. Repeat steps 2-5

Although you'll be writing this code to eventually stream data from the serial port, in this objective your goal is to simulate the real-time plotting process by reading data from a file. This will allow you to work on the code logic without having to setup hardware, like the sensors. This is often a handy approach to writing code that will eventually interact with real world data, as it simplifies the debugging and testing process. When you plot data with pyplot in a loop, you need to ask pyplot to pause for a moment and update the screen. You can accomplish this by calling: `matplotlib.pyplot.pause(duration_in_seconds)`. This input argument is accepted as a float type, allowing fractional second pauses. You may want to use interactive plotting mode!

Note 1: In general, file reads will occur more quickly than the serial reads for this application (since the serial reads will have to wait for real data to be produced!). Thus, you will need to delay iterations of the loop so that the plot moves at a reasonable rate. In case you need to test timing for debugging, you can import the Python module `time`. The function `time.time()` returns the time in seconds since the beginning of a specific epoch (on most operating systems this is January 1st, 1970, at 0 hours). The precision varies on different operating systems (~1 microsecond on my Macbook, ~1 millisecond on a tested Windows 8 machine).

Tip: To easily break out of a running infinite loop in Python press ctrl+c.

For this Objective, include your code in the Appendix and describe your approach and any challenges you ran into during implementation in the Objective section. Provide a screenshot of the figure window in the Objective section and make sure that it is human readable in your report.

Part 2: Streaming Live Data

Now, let's edit your code from Part 1 to stream live data. Copy your code into a new file and modify it to read from the serial port. Make sure to remove the extra delays!

For this Objective, include the new version of the code (that now reads from the serial port) in the Appendix. Describe your approach and any challenges you ran into during implementation in the Objective section. Be sure to stop by TA lab hours to complete steps 2 and 3 of this Objective, show a TA the system up and running to receive credit for completing the Objective!

Objective Five - Multiple Data Streams! (Bonus: up to 10% credit for this lab)

Write a Python program that plots two streaming signals as separate sub-plots. Each sub-plot should be a row of the figure window and aligned so that the time axes match in range. This plot should allow the user to compare the two signals directly. Note that each signal will have a separate x-axis. Before plotting begins, have the program prompt the user to select which two signals to plot. These signals should be selected from the six signals output by the Arduino code wrote for Objective Three (raw integer sample, scaled sample, high pass filter output, low pass filter output, rectified signal, boxcar filter output).