Lab #4 – analyzing an sEMG on the host PC

Overview:

This lab involves minor coding, lots of playing with filter parameters and thinking about the questions, and writing a report. As usual, you only need to turn in one report per group.

In this lab, we'll

- use the host PCs to examine the sEMG signals that you collected last week
- explore different algorithms to clean up the noisy signals
- build an algorithm to turn the signals into a simple "muscle on" and "muscle off" output

This will prepare us for next week's lab, where we let the PyBoard do the muscle on-vs.-off calculatons itself

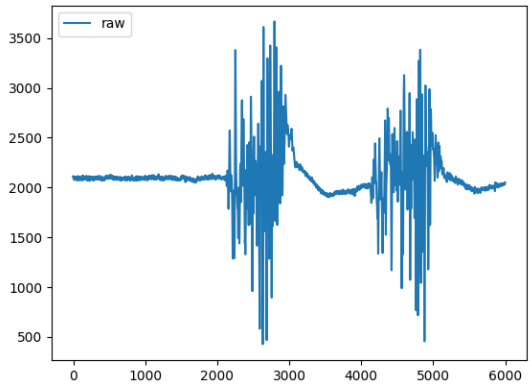
Legal reminder:

This lab involves analyzing your sEMG. Legally, this signal does represent medical data. As such, federal law says that you have the right to keep it private, and we will of course respect that law. Please refer to our separate legal sheet (which you should have read and signed).

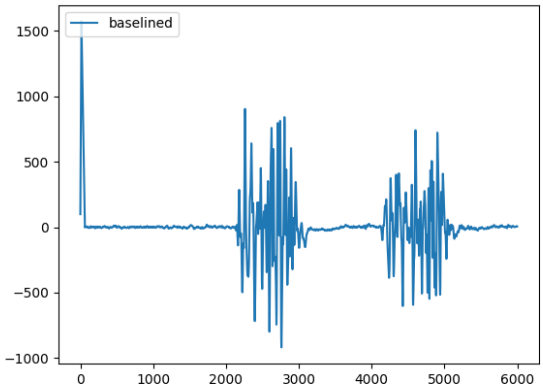
Background

We talked in class about most sEMG signals being quite noisy, and how we can use *moving-average* filters to clean them up. We talked about using one filter to remove the quickly-changing (i.e., high-frequency) noise, and another filter to remove *baseline wander* (which is a much more slowly-changing, low-frequency noise).

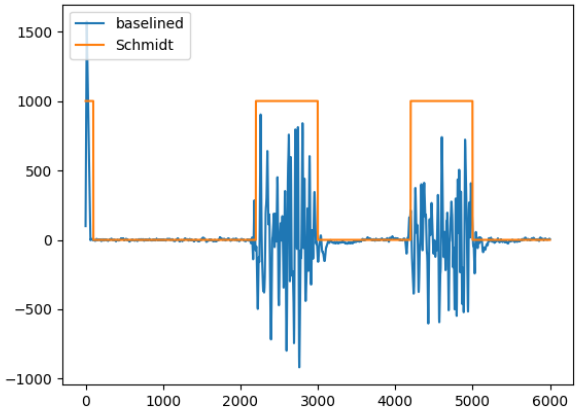
Our raw data is *lots* of numbers – our sEMG signal got amplified, sampled 1000 times/second; then each sample was converted to a number in the range [0,4096). It probably looked something like this (where the *x* axis is sample number; it shows about six seconds of data, or 6000 samples).



After both of our filters, we'll hopefully have a signal that looks something like this:

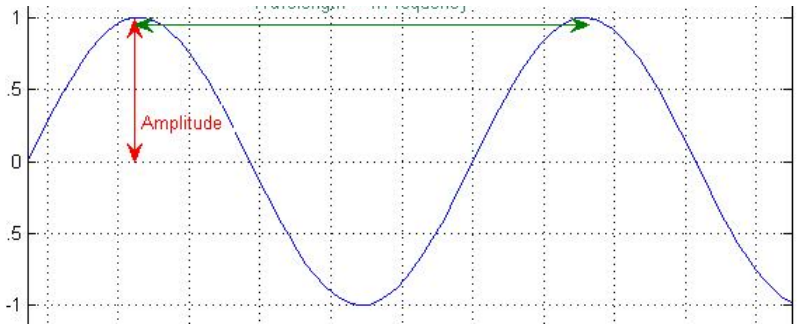


But our end goal for the moment is simply to turn on an LED whenever you use your muscle. We don't need all of the detail in the signal above – we would greatly prefer a signal that looks more like this (shown in orange along with the filtered signal):



This signal is clearly much easier to use – it's just on when the muscle is on and off when the muscle is off. So now we have an interesting filtering problem; how do we get from the original signal to the second simpler one?

One very standard tool in an engineer's toolbox is something called the *root mean square* (RMS) value of a signal. If you wind up taking circuits or signals courses you'll learn a *lot* about RMS values. An RMS value can take a nice sine wave like this



and change it into a simple number that is roughly how "big" the sine wave is. How does it do that? First, it *squares* each sample. x^2 is a nice function in that it takes negative numbers and turns them into positive ones. So the fact that the sine wave is negative half the time immediately becomes irrelevant. Nice! Then it averages together all the x^2 values, and finally takes the square root of that average (to get back to a number of roughly the right magnitude again).

So after our filtering, we'll compute the RMS value of our filtered signal during various segments and use a Schmidt trigger (which we discussed in class) to turn the RMS output into the nice, clean final on/off output. By the end of the lab, we will hopefully have a good way to turn your sEMG waveform into a simple "muscle-on / muscle-off" signal.

Lab setup – hardware

This lab uses only the PC host computer. It does not use the PyBoard, the scope or any other hardware.

Lab setup – software

The setup for this lab involves first copying a Python program from the class shared folder to your own folder (where you can modify your copy).

The file locations are the same as usual; the class folder is Q → en1EIY → 2022f → public_html → labs → code. Use the standard Windows copy/paste to grab the file *4_emg_process_host.py* and paste it into your own Z-drive folder (i.e., your own home directory).

Step 1: Make sure Thonny is computing on the host and not on the PyBoard

In lab #1, we ensured that Thonny did its computation on the PyBoard. This time, we want the computing to happen on the host. Again, the key is the lower right of the Thonny window – it should *not* say "MicroPython generic," but instead should say than "Python 3.7.9" (or some similar version). If not, click on that lower-right corner and pick "the same interpreter which runs Thonny."

If you forget this step, then nothing much can happen – since the PyBoard isn't even connected in this lab, you cannot compute on it!

Step 2: reading your sEMG file and plotting it

Using Thonny to edit *4_emg_process_host.py*, change the code at roughly line #115 to read your own file instead (whatever you named the file):

```
with open('Z:\\semg_lab3.csv') as f:
```

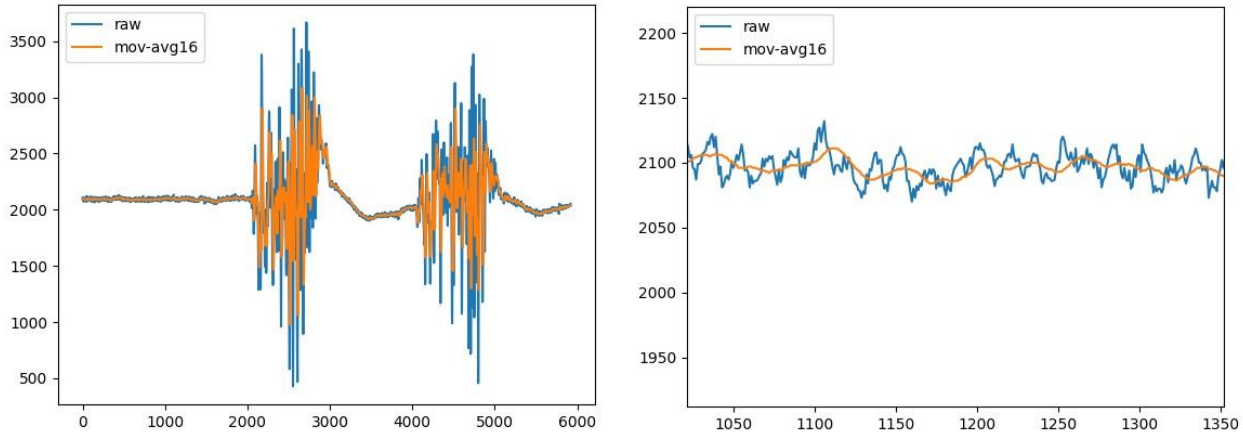
When you run the program in Thonny, you will get a graph of your signal. Hopefully it matches what you saw last week! If not, something has gone wrong. Note that file names on Windows are *not* case sensitive; it won't matter whether you use upper case or lower in the file name (but getting it correct is a good habit to get into, since other operating systems do enforce case sensitivity).

Step 3:16-sample moving-average filter

If you look near the end of the file at roughly line #124, you'll see the code

```
#plt.plot (filter1, label=f"mov-avg{f1}")
```

Uncomment this line. Rerun the program and you'll now see two signals on your graph – not only the raw data as before, but also the results of a 16-cycle *moving-average filter*. This is basically a filter that starts with our 10000 input samples, produces 10000 output samples, and follows the rule that any output sample is just the average of the previous (in our case) 16 input samples. A moving-average filter is very good at filtering out quickly-changing noise. Here's what my picture looks like:



The picture on the left is the entire graph. It's kind of hard to see what the moving-average filter is doing, so I've zoomed in on the right. You can see that the high-frequency wiggling (which presumably is noise) has been mostly averaged away.

Try changing the *f1* parameter on roughly line #8 from 16 to a smaller number (e.g. 8) or a larger one (e.g., 32). How does that affect your graph (both in the muscle-quiet parts and the active parts)?

Step 4: fixing the baseline

Your plot may still have a "wandering baseline." In the graph above, you can see the graph falling and rising during roughly samples 3000-4000 and again from samples 5100-5900. Other graphs may show this even more dramatically; baseline wander can come from, e.g., the electrodes moving slowly on your arm if you move your body.

Our code tries to fix this by tracking the baseline as it wanders and then subtracting off the baseline. To see this, uncomment the following line (at roughly line #126)

```
#plt.plot (f2_baselined, label=f"baselined{f2}")
```

Again, try playing with the parameter that controls this – change the `f2 = 64` at roughly line line 9 to smaller or larger numbers and see what happens. Just as we used a 16-sample moving-average filter to remove high-frequency noise earlier, we're using a 64-sample moving-average filter to compute (and then flatten) the baseline.

Step 5: root-mean-square calculation

At this point, we're starting to hopefully have a signal that is near zero when your muscle is quiet, and oscillates up and down when you activate your muscle. However, oscillating up and down is not easy to deal with, so we try to turn it into a simple number. We'll do that by

computing a quantity called the "RMS" (**r**oot **m**ean **s**quare) value, which is one way that engineers compute how much power is in a signal.

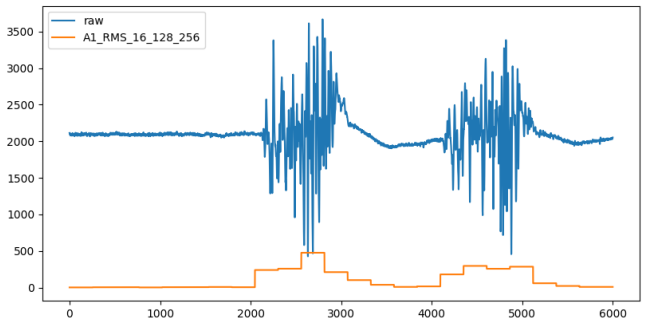
To do this, the algorithm breaks the waveform into segments of **`f3`** (by default 100) samples each and analyzes each segment in turn. Inside of a segment, we first square each of the 100 samples (hence turning negative numbers into positive ones), then take the average of these 100 positive numbers, and finally takes the square root. This turns the 100 samples from our waveform into a single number representing how strong the signal is in that segment.

To see the result, you can uncomment the line

```
plt.plot (f2_RMS, label=f"RMS{f3}")
```

You thus have another number to play with – the **`100`** controls the length of each time segment.

By this point, you hopefully have a signal that looks like the bottom signal line in this figure:



It should be near zero when your muscle is quiet, and a large value when your muscle is on. We're ready for our final step – converting this to a simple 1 (for ON) and 0 (for OFF).

Step 6: Schmidt trigger

We might consider converting our signal to a 1 or 0 via a simple threshold. E.g., in the figure above, we might say that any value greater than 100 is a 1 and any value less than that is a zero. But our signal may still have some noise that causes it to bounce around as it crosses 100; so we instead employ a *Schmidt trigger*, which works by using *hysteresis*.

What does this mean? A Schmidt trigger converts a signal to 0 and 1 by using *two* comparison points. You'll notice on roughly lines 15-16 two parameters to control our Schmidt trigger – **`44`** and **`68`**. This means that when the output is labeled as "0", it stays 0 until we get a segment whose value is at least 68 and then turns to 1. At that point, it stays at 1 until we get a segment whose value is 44 or lower. This "hysteresis" can produce a much cleaner conversion to 0 and 1 values.

Again, feel free to play with these two numbers. What happens? You can see your results by uncommenting the line

```
#plt.plot (f2_schmidt, label=f"A1_{f1}_{f2}_{f3}")
```

Questions:

- You've played with the various algorithm parameters. Explain roughly what they do and why they work.
- For each parameter, explain what happens if you set it too low or too high.

- Do the same parameter settings work equally well across sEMG signal samples for multiple people?

Feel free to ask any of us to review your answers to the questions during the lab.

What to turn in:

A lab report that contains your pictures and the answers to the questions above. As usual, you need only turn in one report for each group.