HW2

Jump to bottom

Nick Marchuk edited this page 3 days ago · 8 revisions

Digital Signal Processing

Video: https://youtu.be/RAbAyVcjtYw

It is rare that a signal you are interested in is delivered in a form without requiring some kind of post processing. Maybe you want to amplify a specific frequency range in the data, or know the amplitude at a specific frequency, or need to filter noise. When the signal is an analog voltage, you can build a circuit to clean it up. But if the signal is inherently digital, you must implement your "clean up" in code.

Digital signal processing, or DSP, has some benefits over an analog circuit. The code is more flexible, easier to simulate, and is "free", in that you do not have to purchase expensive components like extra-fast op-amps or add negative voltage supplies to your design. The down side is, of course, the complexity of the code.

The most common thing you will want to do is "smooth out" noisy data. If you imagine noisy data in the time domain, it resembles your signal summed with something like white noise. In the frequency domain, you would see peaks at the frequencies you are interested in (your signal), and peaks at the main frequencies of the noise or just a high overall amplitude of all frequencies. Your goal is to increase the "signal to noise ratio", or basically subtract out the frequencies you are not interested in. To do this, you want to reject high frequencies and keep low frequencies, making this an example of a low-pass filter.

The easiest way to low-pass filter a signal is to average a few data points, and use the average as your "smoothed" value. The more data points you average, the smoother your signal, but you will start to notice delay introduced by the filter. To do this in code, you need to implement some kind of buffer to store the signal. As you collect new data, you need to over-write the oldest data in the buffer, and average everything in the buffer. This is a better method then, say, hard coding in exactly 4 variables to store your data, but later deciding you want to average over 8, and having to rewrite all the code. This filter is called a Moving Average filter, or MAF.

One way to evaluate how well this type of method works is to imagine that the buffer values are all 0, and the input is a perfect step function. How many new values are required for the average to equal the amplitude of the step? If the buffer is just one value, then it takes only one step. If the buffer has 10 values, it takes 10 steps (10 steps to over-write all of the zeros in the buffer with the step amplitude.) So this method adds phase delay to the signal, which could be undesirable. But it does give the exact value of the step amplitude after the buffer size of steps.

Perhaps an easier way to implement the low-pass filter is to merely average the current value into a running average of the previous values. No buffer to remember, only the previous value. But how do you merge the two values? The math looks like: new_average = A * previous_value + B * new_value, where A + B = 1. If the data is very noisy, you would choose a large A value, and a small B. If the data is not so noisy, a larger B. Imagine the effect on a perfect step input: with a large B, the new_average gets to the step amplitude relatively quickly. A larger A takes longer. And as the new_value approaches the step amplitude, the rate of approach decreases, and technically never gets there. For this reason, this type of filter is called an infinite impulse response, or IIR filter. The MAF described above takes a finite number of steps to converge, so it is called a finite impulse response filter, or FIR.

There are many tools available to simulate digital filters, to allow you to see the effect of your buffer size or scalar weights like A and B. We will use Python to see how our filters will work, since it is faster to code in than C and has plotting tools. (I guess you can also use MATLAB if you really want to, but the coding experience in Python is a much more valuable skill in the long run).

Make a folder called HW2 in your repo and place the following **deliverables** in it as you go.

1. Download the sample data sets sigA.csv, sigB.csv, sigC.csv, and sigD.csv. A CSV file contains columns of data separated by commas and newlines. It is not a very efficient way to store data, but it is easy to view and manipulate. Look at this sample code and practice getting data out of the CSVs and into lists in Python.

2. Practice plotting the data in Python using MATPLOTLIB. This sample code shows some examples. Be sure to always give your axes labels and the plot a title.

3. The data is collected at a regular rate called the sample rate. We will need to know this value later, so figure out how to calculate it from the data. The first column in these files is time, the second is the value. Hint: sample rate = number of data points / total time of samples. Python has a useful ability here, the [-1] index of a list is the last value in the list.

4. sigA, sigB, and sigD contain data with noise (sigC is a square wave). We can filter out the noise, but we need to figure out what cutoff frequencies to use. Sometimes you know the frequencies to keep and get rid of based on what the project is, but a lot of times you don't. To figure them out, you need to be able to see the frequency content of the signal, to basically turn a signal vs time into a signal vs frequency plot. This is where the FFT comes in. The Fast Fourier Transform is an algorithm that identifies the magnitude of frequencies in a signal, from 0Hz to half of the sampling frequency, called the Nyquist Frequency. The Python package NUMPY has an FFT library. Note that an FFT is not the exact representation of the frequency spectrum of a signal. Ideally you would have a lot of data sampled very rapidly, but inevitably you won't. Also, the number of datapoints must be a factor of 2, if it is not, the algorithm may insert 0s to make it a factor of 2, but what does this do to you data? Also, the FFT returns the magnitude at specific frequencies, but you cannot interpolate the magnitude between frequencies, because there is no guarantee that the spectrum is smooth like that. Great care should be taken to understand how the program you are using has implemented the FFT algorithm, because you may misinterpret the output. So, after saying that, we will skip all of the details, there are whole courses you can take on DSP, data analysis, and spectral analysis. This sample code will take a data list and generate the FFT and plot it. **Use this code to generate a figure with a subplot of the signal vs time and a subplot of the FFT of each CSV.**

5. Let's start by low-pass filtering the data with a moving average filter. Write a loop that averages the last X number of data points and saves the result in a new list. The new list will have X fewer data points, or you can assume the X data points before you started were 0. For each CSV, try a few different X values and choose the "best" by eye, some value that smooths the value without delaying it too much. Generate the FFT of the filtered data and plot it on the same plot as the unfiltered FFT. Can you see the low-pass effect of the moving average filter based on the difference in the FFTs? **Upload an image of a plot of each CSV with the unfiltered data in black, the filtered in red, and the number of data points averaged in the title, as well as your code.**

6. Now try low-pass filtering with an IIR. Write a loop that adds a value into the average with the two weights A and B, so that new_average[i] = A * new_average[i-1] + B * signal[i]. Note that A+B=1, otherwise the signal will get bigger with time (A+B>1) or go to 0 (A+B<1). Try a few different values of A and B for each CSV, save the "best", and make a plot comparing the before and after FFTs. **Upload an image of a plot of each CSV with the unfiltered data in black, the filtered in red, and the weights A and B in the title, as well as your code.**

7. So far the process has been rather guess-and-check. You can usually get away with a MAF or IIR filter, just like you can with a simple RC filter on an analog circuit. But when the signal is small and the noise is big, you need to break out the FIR. In an FIR filter you remember that last X number of samples, and sum them with unique weights to create the filtered output, like a combination of the MAF and IIR. The interesting part is the selection of the values of the weights, and how many samples to use (X). To do this, we need a simulator to show the frequency response, or how much each frequency will be attenuated, kind of like a frequency dependent gain. This website generates weights and shows the frequency response. Try the MAF to see the shape of the frequency response. Note that there is no clear cutoff frequency, and some lower frequencies are attenuated more than some higher frequencies. Based on the FFTs you made in Part 4, choose a cutoff frequency and use the Low-pass sinc, and try a few different cutoff frequencies, bandwidths, and window types. Write a loop to apply the weights, compare the filtered and unfiltered signals, and compare the FFTs. **Upload an image of a plot of each CSV with the unfiltered data in black, the filtered in red, and some information about weights used in the title, like how many, what type of filter, the cutoff frequency and bandwidth, as well as your code.**

That's probably enough coding for now. There are lots of other types of filters to try, and implementing them in C is a little different than Python, but hopefully you've picked up some intuition on how to make and evaluate a filter.

To the Schedule

Pages 9

Find a Page...

Home
Github
HW1
HW2
HW3
HW4
Installations
Schedule
Syllabus

Clone this wiki locally

https://github.com/ndm736/ME433_2021.wiki.git