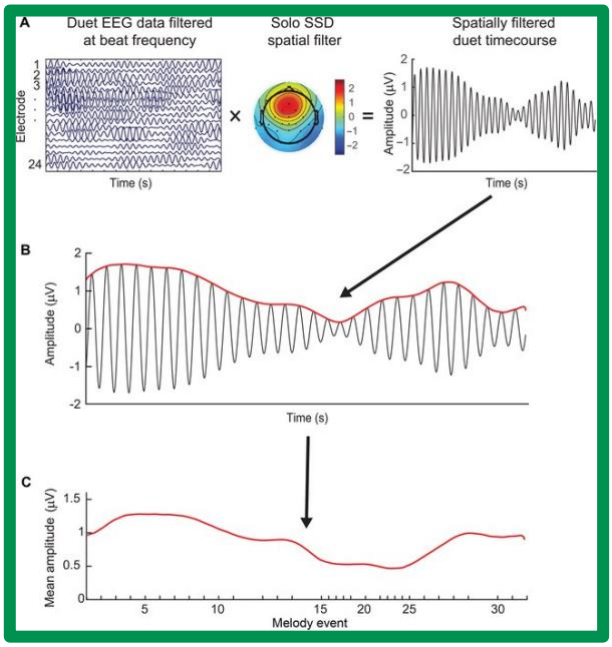
UVM Spring 2024 Brain-Computer Interfaces
Lab 4: **Filtering**

Overview:

One of the advantages of SSVEP is that it can allow **asynchronous** communication. This means that the stimuli can be flashing in the background, and our analyses don't need to know when the flashes started and stopped to infer the user's intent. This makes it different from the synchronous P300 speller, where we need to know when each row or column flashed to find the P300 signal.



In this lab, we'll use **band-pass filtering** to isolate the part of each signal that is oscillating at the same frequencies as each of the stimuli. We'll then use the **Hilbert transform** to get an instantaneous amplitude of the signal at those frequencies. From this, we should ideally be able to see which frequency the user was looking at in each moment.

Submission: Remember to submit all Python files, input files, and output images on Gradescope. Please review the General Lab Instructions before you begin, and refer back to them again at the end to make sure your submission is ready.

You should complete & submit this assignment as a pair. Please work together rather than dividing and conquering, and be sure to each take the lead on some parts so that you are both pulling your weight. You will each complete a **peer evaluation** to document any social loafing.

Part 1: Load the Data

As in previous labs, let's use your past code to get the data loaded into a Python dictionary. You can reuse your previous code for this purpose.

Cell 1: Create 2 new python scripts and save them as filter_ssvep_data.py and test_filter_ssvep_data.py. In your test_ script, use functions written in previous labs to load the data from subject 1 into a python dictionary called data.

Part 2: Design a Filter

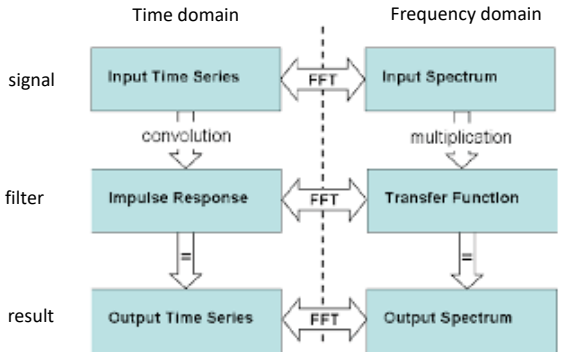
Read about filtering in Kramer & Eden Ch. 6: Filtering Field Data
https://mark-kramer.github.io/Case-Studies-Python/06.html

Read about scipy's `firwin` function:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.firwin.html

UVM Spring 2024 Brain-Computer Interfaces
Lab 4: **Filtering**

Read about scipy's `freqz` function:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.freqz.html

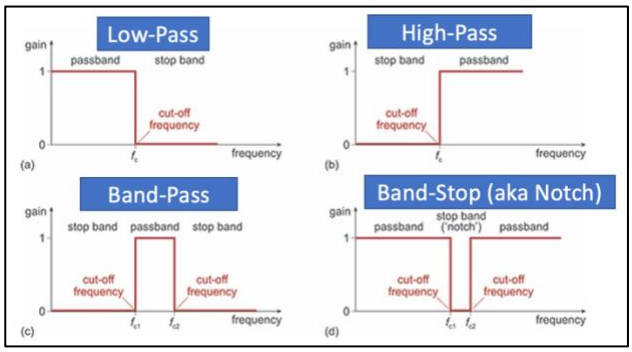
Filters are a widely used way to attenuate some frequencies in a signal while keeping others. A filter's "**frequency response**" or "transfer function" is a measure of the gain applied to each frequency in the signal. You can apply the filter by taking the Fast Fourier Transform (FFT) of a raw signal, multiplying it by the filter's frequency response, and applying the inverse FFT (IFFT). Alternatively, you can convolve the raw signal (in the time domain) with the filter's "**impulse response**," which is what comes out when we filter a delta function (a one-sample impulse of nonzero input). The frequency response is the Fourier Transform of its impulse response, and the impulse response is the inverse Fourier Transform of the frequency response.



We can **apply the filter** to our data in one of two ways: (1) take the signal's FT, multiply by the filter's frequency response, and take the IFFT, or (2) convolve the signal with the filter's impulse response. These are equivalent because convolution in the time domain is the same as multiplication in the frequency domain.

There are 4 types of filters:

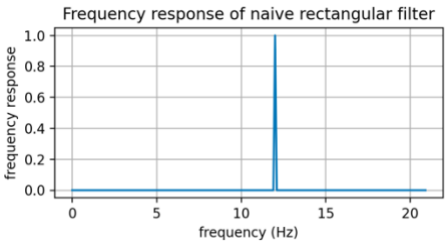
1. low-pass filters that let low frequencies "pass" through it but remove high ones
2. high-pass filters that let high frequencies pass through but remove low ones
3. band-pass filters that let a certain band of frequencies pass through but remove everything above and below it.
4. band-stop or "notch" filters that remove things only in a narrow frequency band and let everything above and below through.



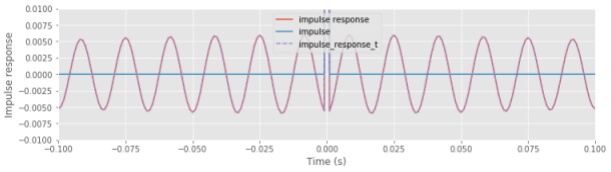
Take a look at the figure above (and the Kramer/Eden text) and familiarize yourself with terms like "passband", "stop band", and "cut-off frequency". These will be used throughout the lab.

To see our SSVEP signal with everything else removed, we'd like to apply a band-pass filter around our 12Hz stimulus frequency. This means that we would like to keep any activity near 12Hz while removing everything else. We can do this for every channel. Then we'll do the same for 15Hz. We'll end up with two matrices the same size as our raw data, one with only ~12Hz oscillations remaining and one with only ~15Hz oscillations.

One way we might think to do this is to convert our signal into the frequency domain using an FFT, then set all the values outside of 12Hz (or 15Hz) to zero, then take the inverse FFT. Put another way, we are multiplying our signal in the frequency domain by the **frequency response** of a filter seen at right:

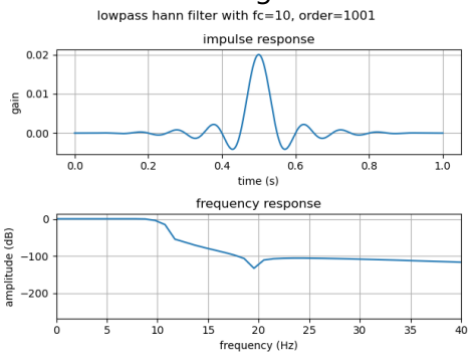


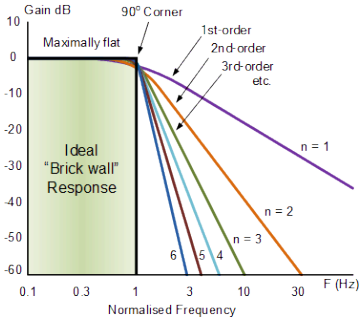
Unfortunately, this approach has a problem: it gives brief activity (or artifacts) long-lasting repercussions in the filtered signal. We can see this in the **impulse response** described by Kramer & Eden: when they try to remove 60Hz noise using this method, it means that the filter spreads every impulse into a small 60Hz sine wave that lasts for the whole recording.



We could chop up the data into shorter periods and then apply this approach, but this would limit our frequency resolution (which, remember, is 1/T), and thus our ability to distinguish the stimulus frequency from those around it. It could also create unexpected artifacts at the edge of the window.

Finite Impulse Response (FIR) filters offer a more balanced approach. These filter the data by setting the filtered data at each time equal to a weighted combination of the raw data at various delays. We can make a filter that has a brief impulse response to limit how long a brief burst of activity will impact the signal. There are several popular kinds of FIR filters, but we'll use the **Hann** (aka Hanning) filter here. Hann filters are popular because they have a smooth change in amplitudes over time and a reasonably sharp frequency response cutoff. Here is the frequency response and impulse response of an example low-pass Hann filter.





Once we've decided on the kind of filter, we must choose the filter's **order**, or the number of coefficients in the filter (for FIR filters, this is the length of the impulse response in samples). The higher the order, the steeper the cutoff will be in the frequency response (good, because it keeps only what we want), but the longer the impulse response will be in the time domain (bad, because brief inputs can have long-lasting outputs).

To get a sense of this tradeoff, we will have our code plot both the frequency response and the impulse response of our filter. To do this in Python, we're going to make use of the following functions from the scipy.signal package:

1. `firwin`: create an FIR filter according to common windows like Hann. Returns the filter coefficients, often called **b**.
2. `freqz`: given the filter coefficients and sampling frequency, returns the frequency response.

Be sure to read the documentation for these functions to learn the inputs and outputs and their units. The **sampling frequency inputs** are particularly important here.

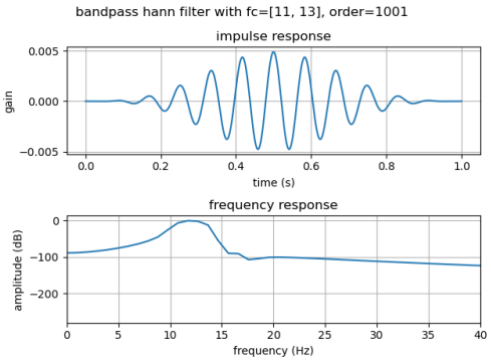


Figure 1. Sample output for this part.

*Cell 2: **In your filter_ module**, write a function called make_bandpass_filter() that takes the following inputs:*

- *low_cutoff, the lower cutoff frequency (in Hz),*
- *high_cutoff, the higher cutoff frequency (in Hz),*
- *filter_type, the filter type, (this will be passed to the "window" input of scipy.signal.firwin and should be "hann" by default),*
- *filter_order, the filter order, and*
- *fs, the sampling frequency (in Hz).*

It should then create a filter matching these specifications and return the filter coefficients in an array called filter_coefficients (of length filter_order+1). Your code should plot the impulse response and frequency response of your filter and save the plot as a .png file with this naming convention: the figure for a Hann filter with pass-band 10-20 Hz and order 100 should be saved as hann_filter_10-20Hz_order100.png. The function should then return the filter coefficients b.

***In your test_ script**, call this function twice to get 2 filters that keep the data around 12Hz and around 15Hz. Use a 1000-order Hann filter and a 2Hz bandwidth for each. In a comment, address the following questions:*

A) How much will 12Hz oscillations be attenuated by the 15Hz filter? How much will 15Hz oscillations be attenuated by the 12Hz filter?
B) Experiment with higher and lower order filters. Describe how changing the order changes the frequency and impulse response of the filter.

Part 3: Filter the EEG Signals

Read about scipy's `filtfilt` function:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.filtfilt.html

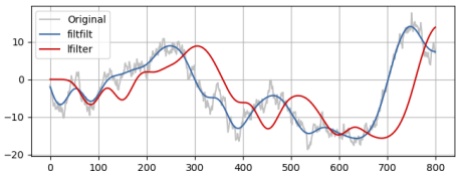


Figure 2. An arbitrary signal (not your EEG data) filtered with lfilter and filtfilt.

Now we're going to apply our filter to our EEG signals. This would typically produce a delay that might be different for signals of each different frequency (this can be plotted as the phase response). To get around this, we can apply the **filter both forwards and backwards** in time using `scipy.filter.filtfilt()`. This has the disadvantage of breaking causality (i.e., the data at time t can be affected by something that happened after time t), but the lack of distortion you get with `filtfilt` is usually considered worth it.

When we filter with an IIR filter, the `filtfilt` function takes inputs **b** (the coefficients for the raw data) and **a** (the coefficients for the filtered data) as inputs. With an FIR filter, we only have **b**, and we set input **a**=1. Of course, both versions also take the raw data as input.

*Cell 3: In your **filter_ module**, write a function called filter_data() that takes 2 inputs:*

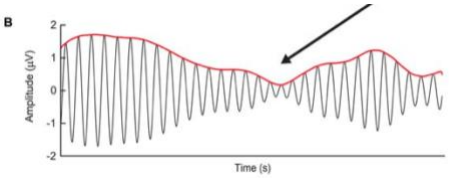
- *data, the raw data dictionary*
- *b, the filter coefficients that you produced in the last part.*

This function should apply the filter forwards and backwards in time to each channel in the raw data. It should return the filtered data (in uV) in a variable called filtered_data.

*In your **test_ script**, call filter_data() twice to filter the data with each of your two band-pass filters (the ones designed to capture 12Hz and 15Hz oscillations) and store the results in separate arrays.*

Part 4: Calculate the Envelope

Read about the Hilbert Transform:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.hilbert.html



We care about the amplitude of the oscillations, not the phase or instantaneous voltage. For that reason, we would like to extract the **envelope** surrounding our wave. This envelope tends to surf along the top of our wave, connecting all the peaks. But because it stays high between peaks, it reflects the wave's amplitude even when the wave is at a trough and the voltage is low.

To calculate this envelope, we use the **Hilbert Transform**, which can provide the instantaneous amplitude, frequency, and phase of our signal. To get the amplitude, you can use `np.abs(scipy.signal.hilbert(signal))` . When you've done this, you should be able to see the amplitude of the oscillations that made it through your band-pass filter at any given moment. This is particularly helpful for our

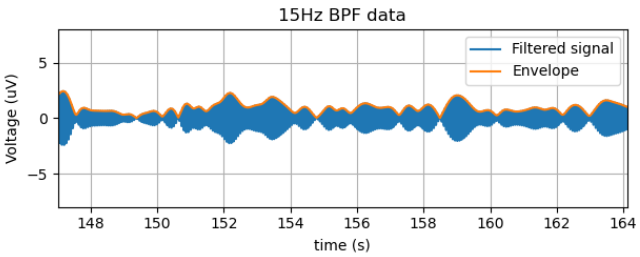


Figure 3. Sample output for this part, zoomed in arbitrarily.

SSVEPs, where the momentary amplitude of, say, 12Hz oscillations is thought to reflect the presence of 12Hz SSVEPs.

*Cell 4: **In your filter_ module**, write a function called get_envelope() that takes the following inputs:*

- *data, the raw data dictionary,*
- *filtered_data, the filtered data (one of the outputs from the last part),*
- *channel_to_plot, an optional string indicating which channel you'd like to plot*
- *ssvep_frequency, the SSVEP frequency being isolated (this is for the title).*

If channel_to_plot is not None (which should be the default), the function should create a new figure and plot the band-pass filtered data on the given channel with its envelope on top. (If your computer slows or freezes when you try to do this, it's ok to plot every 10th sample instead of every sample.) The function should return the amplitude of oscillations (on every channel at every time point) in an array called envelope.

***In your test_ script**, call this function twice to get the 12Hz and 15Hz envelopes. In each case, choose electrode Oz to plot.*

Part 5: Plot the Amplitudes

Next, let's plot these 12Hz and 15Hz envelopes alongside the task events to get a better sense of whether they're responding to task events. This will use elements of code you've already completed to answer a new question.

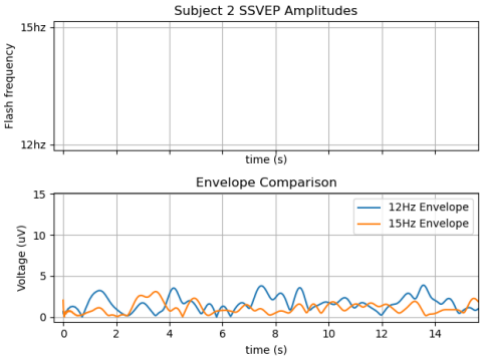


Figure 4. Sample output for this part, zoomed in arbitrarily.

*Cell 5: **In your filter_ module**, create a function called plot_ssvep_amplitudes() that takes the following inputs:*

- *data, the raw data dictionary,*
- *envelope_a, the envelope of oscillations at the first SSVEP frequency,*
- *envelope_b, the envelope of oscillations at the second frequency,*
- *channel_to_plot, an optional string indicating which channel you'd like to plot,*
- *ssvep_freq_a, the SSVEP frequency being isolated in the first envelope (12 in our case),*
- *ssvep_freq_b, the SSVEP frequency being isolated in the second envelope (15 in our case),*
- *subject, the subject number.*

The last 3 inputs are for the legend and plot title. The function should plot two things in two subplots (in a single column): 1) the event start & end times, as you did in the previous lab, and (2) the envelopes of the two filtered signals. Be sure to link your x axes so you can zoom around and investigate different times in the task.

In your test_ script*, call this function to produce the plot for channel Oz. In a comment in the test_ script, describe what you see. What do the two envelopes do when the stimulation frequency changes? How large and consistent are those changes? Are the brain signals responding to the events in the way you'd expect? Check some other electrodes – which electrodes respond in the same way and why?*

Part 6: Examine the Spectra

Finally, import and/or adapt your code from Lab 3 to plot the average power spectra across epochs (as defined in Lab 3) on electrodes Fz and Oz at 3 stages of our analysis:

1. The raw data
2. The filtered data (with the filter centered at 15Hz)
3. The envelope (Hilbert transform) of this filtered data

*Cell 5: **In your filter_ module**, create a function called plot_filtered_spectra() that takes the following inputs:*

- *data, the raw data dictionary,*
- *filtered_data, the filtered data*
- *envelope, the envelope of oscillations at the first SSVEP frequency,*

It should produce and save a 2x3 set of subplots where each row is a channel and each column is a stage of analysis (raw, filtered, envelope). The power spectra should be normalized and converted to dB as in your previous lab.

In your test_ script*, call this function. In a comment, describe how the spectra change at each stage and why. Changes you should address include (but are not limited to) the following:*

1. *Why does the overall shape of the spectrum change after filtering?*
2. *In the filtered data on Oz, why do 15Hz trials appear to have less power than 12Hz trials at most frequencies?*
3. *In the envelope on Oz, why do we no longer see any peaks at 15Hz?*

Part Last: Submit your Code

Update your headers to describe the full assignment. Refer back to the General Lab Instructions to make sure your submission is ready. Save your python scripts as described above and submit them on GradeScope. Also **submit any helper scripts (especially those from earlier labs)** required for them to work, and any data files, images, or movies you produced in this lab. Because the files are large, **you do not need to submit the dataset.**