University of Washington – ECE Department
EE242 Lab 4 – Digital Filtering
Background Material

In this lab, we will consider different types of digital filters (specifically discrete-time, linear, time-invariant filters) and look at their characterization in both time and frequency. This will give you some insight into how digital filters are implemented and into the properties of different digital filter design algorithms. You'll also learn about some of the signal processing functions available from the signal module in the scipy package which will be useful for designing and implement filters. You will work with examples that show you how filtering can be useful to remove noise and reshape the frequency content of a signal.

1.0 Types of Filters

Filters can be classified by the shape of their frequency responses, which determine their applications.

- A **low-pass filter (LPF)** removes high-frequency components of the data, allowing only frequencies lower than the cut-off frequency to pass through. It is commonly used for smoothing the data. For audio, applying low-pass filter removes high-pitched sound and upper harmonics, which results in a warm and muddy sound. For an image, applying low-pass filter removes sharp details, which results in a blurred image.
- A **high-pass filter (HPF)** removes low-frequency components of the data, allowing only frequencies higher than the cut-off frequency to pass through. It is commonly used for removing DC component from the signal and for extracting details from the data. For audio, applying a high-pass filter removes the bass component, which results in a sharp and crispy sound.
- A **band-pass filter (BPF)** allows only a certain range of frequency to pass through. It is usually used in communication systems, such as radio, to select a channel. Multiple band-pass filters can also form an equalizer which offers detailed control of the level of each band.
- A **band-reject filter** removes a certain range of frequency. It is usually used to remove a specific unwanted frequency from the signal, e.g. removing signaling or 60Hz tones or the feedback of a loudspeaker.
- An **all-pass filter** lets all frequency components equally pass through. However, it may apply different amounts of phase-shift (delay) to different frequencies, which can be useful for some applications.

This terminology applies to both continuous- and discrete-time systems. For discrete-time filters, the frequency response is viewed only over the $(-\pi, \pi]$ or $(-f_s/2, f_s/2]$ range (or only over the positive frequencies), assuming real-valued system coefficients.

Filters can be causal or non-causal. Real-time signal processing requires a causal filter, which is the focus of this lab.

2.0 Discrete-Time LTI Filters

Many continuous-time (physical) LTI systems can be described by linear, constant-coefficient differential equations which relate the output *y(t)* to the input *x(t)*. Similarly, a discrete-time LTI system can be described by a linear, constant-coefficient difference equation (LCCDE). For a **casual LTI digital filter,** the general equation that relates the output of the filter $y[n]$ to the input of the filter $x[n]$:

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

The time-domain difference equation presented above is equivalent to the **frequency-domain transfer function**:

$$H(\omega) = \frac{Y(\omega)}{X(\omega)} = \frac{\sum_{k=0}^{M} b_k e^{-jk\omega}}{\sum_{k=0}^{N} a_k e^{-jk\omega}}$$

In the equation, a_k and b_k are the filter coefficients. a_k is an array of length $N+1$ and b_k is an array of length $M+1$. For implementing on a computer, both *N* and *M* must be finite. The length of the filter, also called the number of taps, is the maximum of the lengths of the two coefficient arrays, i.e. $\max(N, M) + 1$. The order of the filter is the length of the filter minus 1, i.e. $\max(\mathrm{N}, \mathrm{M})$.

By rearranging the equation, we can isolate the current output $y[n]$ to the left-hand side:

$$y[n] = \frac{1}{a_0}\left(\sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]\right)$$

This gives us the formula for calculating the current output of the filter from the current input, past inputs, and past outputs of the filter. The leading denominator coefficient a_0 is usually set to 1, since the effect of a non-unit a_0 is equivalent to dividing all other coefficients by a_0.

There are two broad classes of digital filters, depending on whether or not past outputs are used.

A **finite impulse response** (**FIR**) filter does not use past outputs, so N=0 and the output is given by:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

In other words, the output of a FIR filter is simply a **weighted average** of the current and past inputs, with the weights being the nominator coefficients b_k. For this reason, it is also referred to as a moving average filter. Also notice that the equation exactly describes the convolution operation: $y[n] = b[n] * x[n]$, with the impulse response $b[n]$ corresponding to the coefficients b_k. Because there can only be a finite number of (non-zero) coefficients, the length of the impulse response is also finite; hence the name “finite impulse response”.

When M>0, in contrast, the filter has some non-zero denominator coefficients a_k for $k \geq 1$. With the convention that $a_0 = 1$, the output of the filter can be expressed as:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]$$

which is the weighted combination of not only the current and past inputs, but also past outputs. Since the current value of y depends on the previous values of y, this filter is also known as a recursive filter. Because of the recursion, the impulse response of a stable filter decays over time but never dies off, making it infinite length, and hence the name "**infinite impulse response**" **(IIR).**

3.0 Functions in Python for Filtering & Filter Design

LTI systems in general are often referred to as filters. As we saw in lab 2, the system output can be computed using convolution. Another solution is to implement the LCCDE directly. For an FIR filter, there is little difference between the two options in terms of computational cost if you use the equations directly. (A common implementation of the convolve function is actually more efficient for reasons that you'll learn in EE342.) However, for an IIR filter, the computation for each time step of a convolution involves an infinite sum, so using the finite coefficient equation above is more practical. In Python, LTI digital filters can be implemented using the lfilter function from the signal module:

```
y = signal.lfilter(b, a, x)
```

In the example code above, x is the input signal, y is the output signal, and a and b are the filter coefficients associated with the LCCDE in its initial form (all y terms on the left hand side and x terms on the right hand side). For example, the system described by

$$y[n] - 0.8y[n-1] = 0.2x[n]$$

would have a = [1 -0.8] and b = [0.2]. **Both the signals and the filter coefficients are stored as 1D ndarrays.**

The SciPy signal package provides multiple for options for designing filters using well-established algorithms. For an IIR filter, we can use the Butterworth design, which is a design that has no ripple in the frequency response. To create a Butterworth filter, use the signal.butter function:[1]

b, a = signal.butter(order, W, type)

where b and a are the filter coefficients, order is the order of the filter, W is the cutoff frequency, and type can be either 'lowpass', 'highpass', 'bandpass', or 'bandstop'. For LPF and HPF filters, W is a scalar cut-off frequency. For the BPF, W is a length-2 sequence with the low and high cut-off frequencies. The unit of W is the normalized frequency, which has the range $\mathbf{0 \leq W \leq 1}$ where 1 corresponds to the Nyquist frequency (half of the sampling rate), which also maps to the DTFT range: $0 \leq f \leq 0.5$ or $0 \leq \omega \leq \pi$.

For an FIR filter, we can use the window-method design, which approximates the impulse

[1] The Butterworth filter is actually an analog (continuous-time) filter, so the digital IIR version is only approximated.

response of an ideal filter. To create a window-method filter, use the signal.firwin function:

Lowpass:

b = signal.firwin(order, W)

Highpass:

b = signal.firwin(order, W, pass_zero = False)

where b is the numerator coefficients or impulse response, order is the length of the filter and W is the 3dB cutoff frequency which has the unit of the normalized frequency.

Another useful function is

```
w, hf = signal.freqz(b, a)
```

where w is a vector of frequency samples over the range $[0, \pi)$ and h is a vector with the complex frequency response of the filter sampled at the frequencies in w. When plotting the frequency response, it is typical to use two plots, one for magnitude and one for phase. The magnitude is usually plotted on a dB (log) scale, as in:

plot(w, 20 * np.log10(np.abs(hf)))

A gain of 1 in magnitude corresponds to 0dB; a gain of 2 corresponds to a 6dB increase; and a gain of 0.5 (a factor of 2 attenuation) corresponds to -6dB). The dB scale is useful for considering the impact of multiple systems in series, and it aligns better with how we hear. The phase is often plotted in the the [-180,180] range, using:

plot(w, np.rad2deg(np.unwrap(np.angle(hf))))

Using w, the frequency axis will be given in radians. If you want to plot the frequency scale in Hz, you can use the sampling frequency fs and compute

f = w * fs/(2*np.pi)

If you want to find the impulse response of a system specified by (b, a), then you can compute

ht = signal.lfilter(b,a,x)

where x corresponds to the unit impulse function: a vector that starts with a 1 and is followed by a sequence of zeroes.

4.0 Audio Equalizers

An audio equalizer works by first breaking down the audio signal into multiple frequency bands using **band-bass filters**. Each band is then amplified by a **different gain**. Finally, all bands are **summed together** to produce the output signal. Audio mixing boards do exactly this, though with many more bands than the three bands we have here. The term “equalizer” comes from the fact that people often want to boost the gain of low-energy frequency ranges of the sound or to reduce the gain of high-energy frequency ranges of the sound. By doing so, each frequency range of the sound will have about equal loudness, allowing them to mix together nicely.

Shown below in Figure 1 is the signal flow diagram of the 3-band equalizer you will implement. G_1, G_2 and G_3 are correspondingly the **gains** of the **bass**, **mid** and **treble** frequency ranges. The magnitude response of each filter is shown in Figure 2.

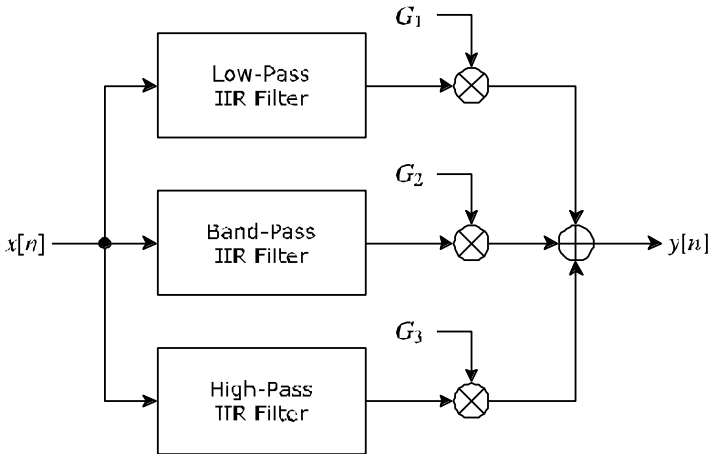

Figure 1: Signal flow of the equalizer

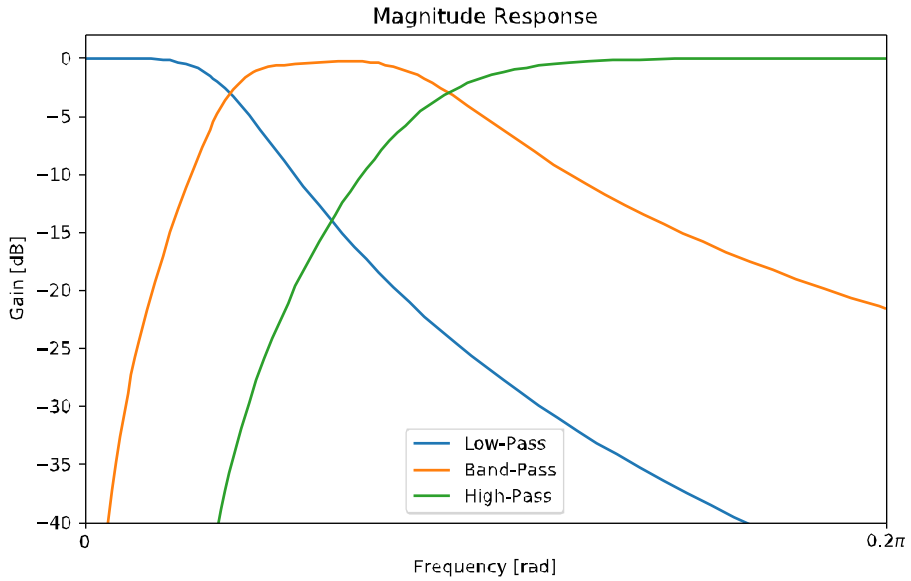

Figure 2: Magnitude response of the equalizer