Signal Processing and Communications Hands-On Using scikit-dsp-comm Part 1

Signal and System Modeling and Simulation

- Moving ahead from the introductions of Part 0, now its time to introduce/review some of the foundations DSP-Comm
- DSP and Communications sit on top of the electrical engineering discipline known as *signals and systems*
    - If you have an electrical engineering background you have studied this subject one upoin a time
    - For others this may be totally new, but the concepts has similarities to other disciplines

Signals

- If you have an electrical engineering background the term *signal and systems* will be familiar and maybe brings back bad memories
-

Basic Signals

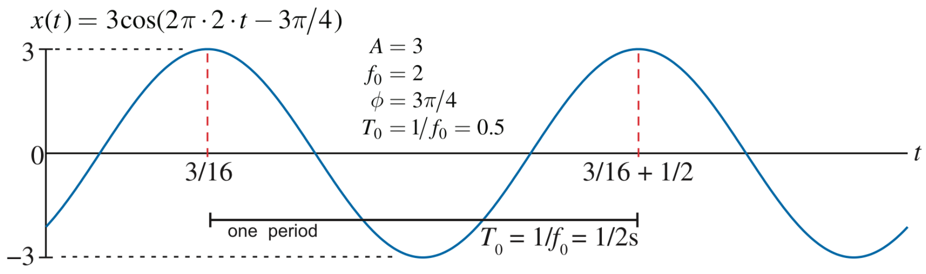
- What are signals anyway?
    - From a physics perspective they may occur naturally, e.g., a beating heart or wind velocity can both be measured by a *transducer* and converted to an electrical signal which can then be *digitized* by an analog-to-digital converter (DAC);
    - The output of the DAC is a signal in sampled form, or for the purposes of this tutorial *time* sampled form; abstractly a time series
    - Signals frequently are man-made, especially in the realm of DSP for communications

Time Domain

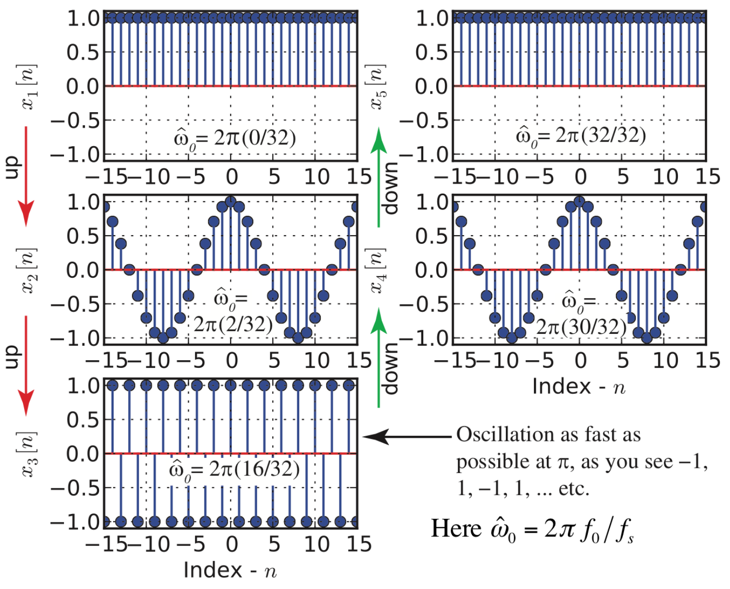
- A fundamental signal is the sinusoid, that is a function of independent variable t in seconds, having mathematical form

$$x_a(t) = A\cos(2\pi f_0 t + \phi) \tag{1}$$
$$x[n] = A\cos(2\pi f_0/f_s n + \phi) \tag{2}$$

  where the parameter A is known as the *amplitude*, f_0 is known as the *frequency* in cycles per second or Hz, and ϕ is the *phase shift* of the waveform relative to $t = 0$ or $n = 0$



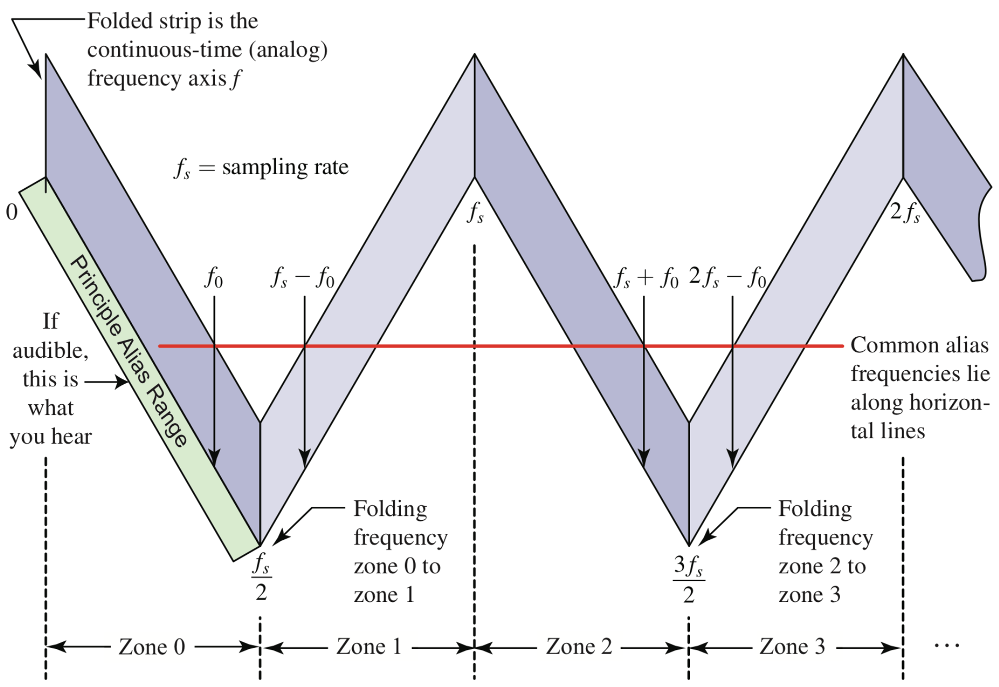
- The first form is a *continuous-time* signal while the second form is *discrete-time*
  - The second form can be obtained from the first by sampling, such as with an ADC (having infinite precision, that is), e.g., $x[n] = x_a(t)|_{t\to nT}$, where the subscript a emphasizes the fact that the signal is analog in nature and $T = 1/f_s$ is the sampling period
  - The sampling operation has some quirks that show up the frequency f_0 in increased (see the figure below):



- The above figure shows us that the frequency of a discrete-time sinusoid is not unique, since as f_0 is increased above $f_s/2$, know as the folding frequency, the signal oscillation rate comes back down, e.g. attempting to set $f_0 = 3f_s/4$ produces the same signal (to within a phase shift) as setting $f_0 = f_s/4$

- > This behavior is known as aliasing, and can be managed so long as the sampling rate, f_s is greater than the highest frequency present in the sampled analog signal $x_a(t)$

- If you extend the notion of aliasing the sampled sinusoid frequency from $f_s/2 <= f_0 <= f_s$ to $0 <= f_0 <= f_s/2$ to higher frequency bands, you end up with the diagram shown below:


Folded strip is the continuous-time (analog) frequency axis f
f_s = sampling rate
0
f_s
$2f_s$
f_0
$f_s - f_0$
$f_s + f_0$
$2f_s - f_0$
Principle Alias Range
If audible, this is what you hear
Common alias frequencies lie along horizontal lines
Folding frequency zone 0 to zone 1
Folding frequency zone 2 to zone 3
$\frac{f_s}{2}$
$\frac{3f_s}{2}$
Zone 0
Zone 1
Zone 2
Zone 3
…

- In **Part2** a cell phone signal generator will be used to study real-time DSP and in particular real-time filtering
  - The phone app signal is generated in discrete-time form using DSP algorithms and then output to a DAC

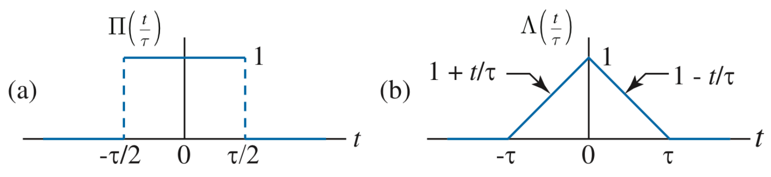
Pulse Type Signals

- Signals that have finite time duration, or at least have finite energy

$$E = \int_{-\infty}^{\infty} |x_a(t)|^2 \, dt \tag{3}$$

are also of interest

- These signal types are useful in encoding digital information, such as data transmissions wired and wirelessly

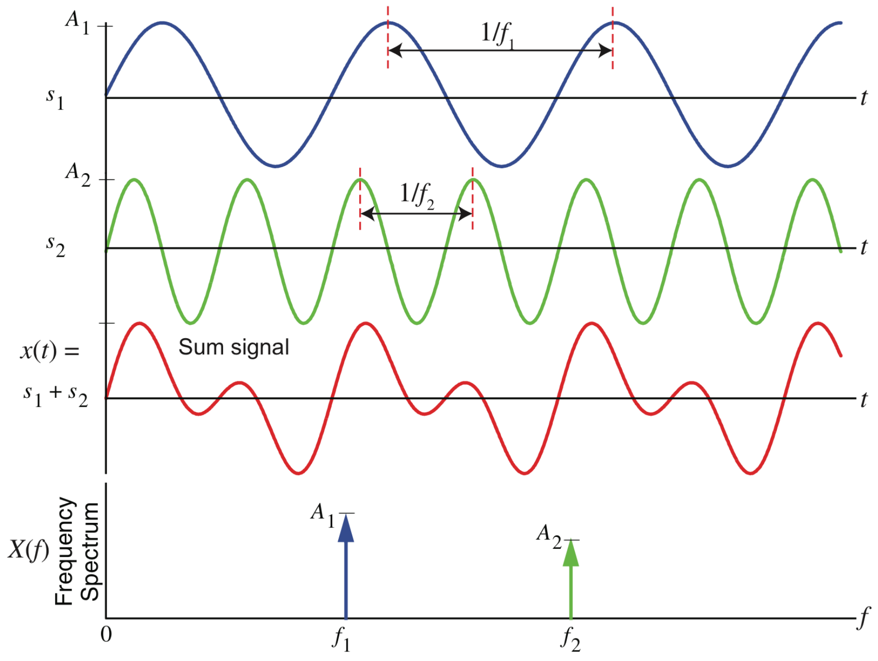


(a)
$\Pi\left(\frac{t}{\tau}\right)$
1
-τ/2
0
τ/2
t
(b)
$\Lambda\left(\frac{t}{\tau}\right)$
1 + t/τ
1
1 - t/τ
-τ
0
τ
t

- The rectangular pulse shape was utilized extensively is earlier generations of digital communications, today pulse shapes with a more compact spectrum, such as the *raised cosine* and *square-root cosine* are popular in most all wireless data
  - See spectral view of a signal is discussed in the next subsection
  - `sigsys` contains the functions `rc_imp(Ns,alpha)` for rasied cosine and `sqrt_rc_imp(Ns,alpha)`, where N_s is the number of samples per bit and α is the excess bandwidth attribute

Frequency Domain

- Signals *exist* in the *time-domain*, but it is convenient to view their frequency domain representation
- The spectrum is a function of frequency, f, and for a sinusoidal signal the notion all of the spectral content lies along the frequency axis at the frequency of the sinusoid (see the figure bvelow)



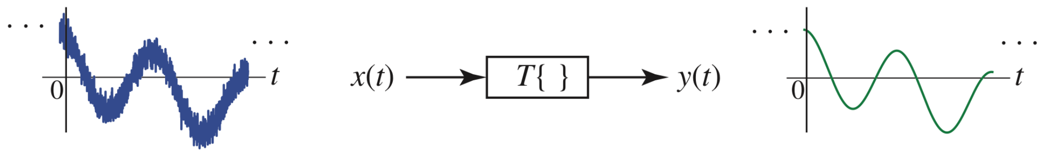
- For pulse type signals the spectrum, $X(f)$, is defined as the *Fourier transform* of the time domain signal

$$X(f) = \int_{-\infty}^{\infty} x(t)\, e^{-j2\pi f t}\, dt \tag{4}$$

- The integral can be approximate numerically, and in particular via the *fast Fourier transform* (FFT)
  - This is investigated briefly in Lab2a using the function `F, X = FT_approx(x,t,Nfft)` found in the notebook `Signals and Systems.ipynb`

Basic Systems

- A system, $T\{\ \}$, operates on a signal, $x(t)$, to make some change, perhaps to remove noise in a *minimum mean-squared error* (MMSE) sense



> A system for which the present output depends only upon the present and past inputs is said to be *causal* or nonanticipatory; **we want this!**

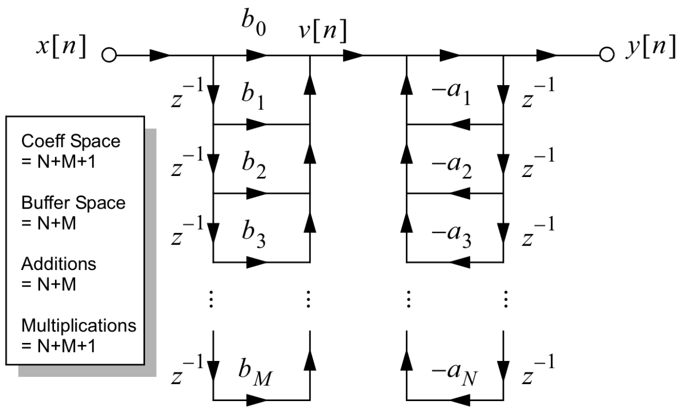
> In the remainder of this section I focus on discrete-time systems, as discrete-time systems is at the core of the tutorial.

- Linear constant coefficient difference equations (LCCDE), e.g., for *causal* systems, as describer in **Part0** is the ticket:

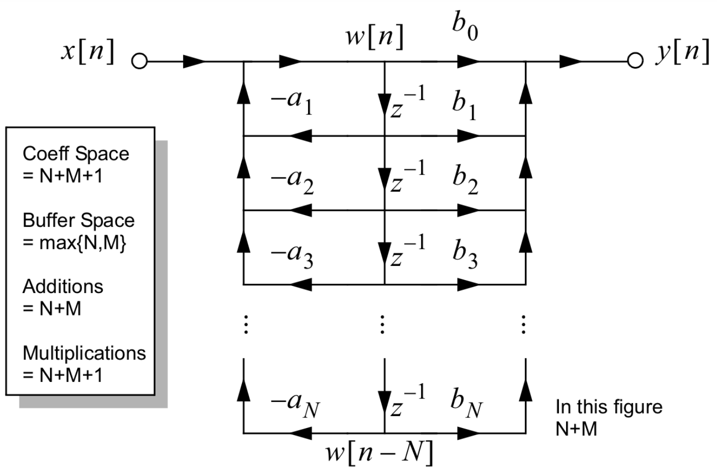
$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k] \tag{5}$$

where $x[n]$ is the input, $y[n]$ is the output, $N\ = 0$ is the system order and $M >= 0$

  - That (5) actually describes an algorithm for finding $y[n]$; turns out more than one exists, but `y = scipy.signal.lfilter(b,a,x)` performs something very similar on arrays of data
  - The array `b` contains the b_k coefficients and the array `a` contains $[1, a_1, a_2, \ldots.]$



Direct Form I Structure



Direct Form II Structure

- The analysis of LCCDE systems relies the z*-transform* representation of the LCCDE, known as the *system function*

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{\sum_{k=0}^{N} a_k z^{-k}} = G\frac{\prod_{k=1}^{M}(1-z_k z^{-1})}{\prod_{k=1}^{N}(1-p_k z^{-1})} \tag{6}$$

  where G is a gain scaling factor

> Simply put the LCCDE description becomes a ratio of polynomials in the complex variable z. Filter design tools found in `scipy.signal` and enhanced by the modules `fir_design_helper` and `iir_design_helper` are important pieces in all that is to come/remains in this tutorial

- This above is fundamentally otained by taking the z-transform of both sides of the LCCDE and then forming the ratio of $Y(z) = \mathcal{Z}\{y[n]\}$ over $X(z) = \mathcal{Z}\{y[x]\}$, where

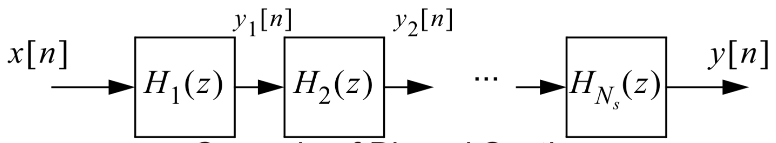
$$X(z) = \sum_{n=-\infty}^{\infty} x[n]z^{-n} \tag{7}$$

  where z is a complex variable

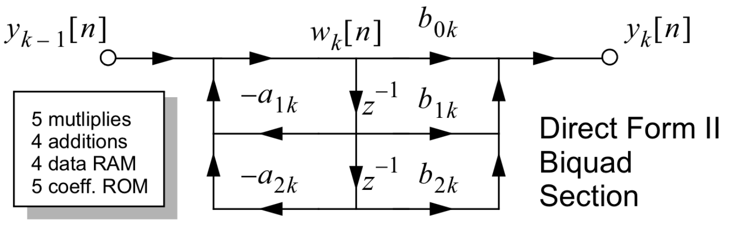
- For higher order IIR filters where numerical stability is critical `spicy.signal` and `fir_design_helper` support *cascade of biquads* form, where the numerator and denominator polynomials are each factored into products of second-order polynomials, e.g.,

$$H(z) = \prod_{k=1}^{N_s} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^2}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}} = \prod_{k=1}^{N_s} H_k(z) \tag{8}$$

  where $N_s = \lfloor (N+1)/2 \rfloor$ is the largest integer in $(N+1)/2$ and the $H_k(z)$ are the biquad sections



Cascade of Biquad Sections



- Filters that fit the LCCDE form fall into two camps: (1) finite impulse response (FIR) for the case $a_k = 0$ for $k > 0$ (through pole-zero cancellation it is possible otherwise too, but unusual) and (2) infinite impulse response (IIR) where in general $a_k \neq 0$ for $k > 0$
  - The notion *impulse response* comes from taking a system at rest and inputting an *impulse*, which in the discrete-time domain is the signal

$$\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

> Without feedback terms in the LCCD, the impulse response of an FIR filter is zero outside some finite interval

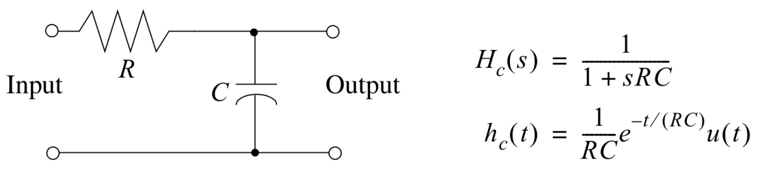
Example: One-pole Averaging Filter

- A simple, yet popular, IIR filter is the one-pole smoothing filter:

$$y[n] = \alpha y[n-1] + (1-\alpha)x[n] \tag{10}$$
$$H(z) = \frac{1-\alpha}{1-\alpha z^{-1}} \tag{11}$$

where $0 < \alpha < 1$ is known as the *forgetting factor*

- The frequency response of this filter, obtained by setting $z = e^{j2\pi f/f_s}$ is related to similar to the RC lowpass studied in basic circuit theory:

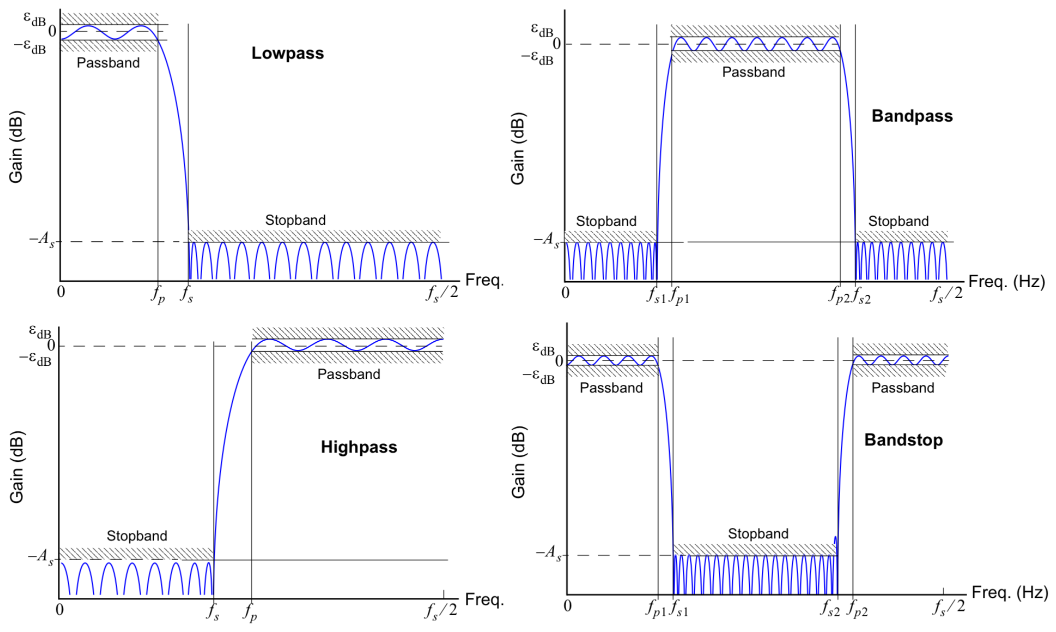

R
Input
C
Output
$H_c(s) = \frac{1}{1+sRC}$
$h_c(t) = \frac{1}{RC}e^{-t/(RC)}u(t)$

- The filter parameter α is related to the RC time constant via

$$\alpha = e^{-T/(RC)} = e^{-1/(f_s RC)} \tag{12}$$

- For filter $H(z)$ has a zero at $z = 0$ and a pole at $z = \alpha$

FIR Filter Design From Amplitude Response Requirements

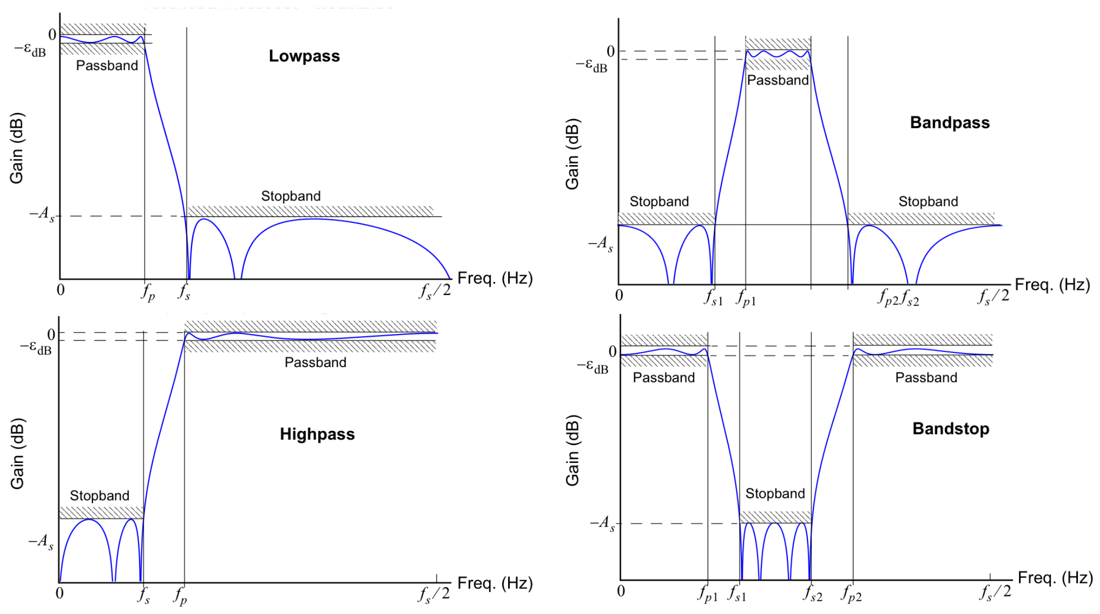


FIR filter design functions in `fir_design_helper.py`.

Type	FIR Filter Design Functions
Kasier Window	
Lowpass	`h_FIR = firwin_kaiser_lpf(f_pass, f_stop, d_stop, fs = 1.0, N_bump=0)`
Highpass	`h_FIR = firwin_kaiser_hpf(f_stop, f_pass, d_stop, fs = 1.0, N_bump=0)`
Bandpass	`h_FIR = firwin_kaiser_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop, fs = 1.0, N_bump=0)`
Bandstop	`h_FIR = firwin_kaiser_bsf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop, fs = 1.0, N_bump=0)`
Equiripple Approximation	
Lowpass	`h_FIR = fir_remez_lpf(f_pass, f_stop, d_pass, d_stop, fs = 1.0, N_bump=5)`
Highpass	`h_FIR = fir_remez_hpf(f_stop, f_pass, d_pass, d_stop, fs = 1.0, N_bump=5)`
Bandpass	`h_FIR = fir_remez_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_pass, d_stop, fs = 1.0, N_bump=5)`
Bandstop	`h_FIR = fir_remez_bsf(f_pass1, f_stop1, f_stop2, f_pass2, d_pass, d_stop, fs = 1.0, N_bump=5)`

The optional `N_bump` argument allows the filter order to be bumped up or down by an integer value in order to fine tune the design. Making changes to the stopband gain main also be helpful in fine tuning. Note also that the Kaiser bandstop filter order is constrained to be even (an odd number of taps).

IIR Filter Design from Amplitude Response Requirements



IIR filter design functions in `iir_design_helper.py` and key support functions.

Type	IIR Filter Design Functions[*]
Transfer Function (b,a) and SOS	

Lowpass (bilinear)	`b, a, sos = IIR_lpf(f_pass, f_stop, Ripple_pass, Atten_stop,`
	`fs = 1.00, ftype = 'butter')`
	`ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or bessel`

Highpass (bilinear)	`b, a, sos = IIR_hpf(f_stop, f_pass, Ripple_pass, Atten_stop,`
	`fs = 1.00, ftype = 'butter')`
	`ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or bessel`

Bandpass (bilinear)	`b, a, sos = IIR_bpf(f_stop1, f_pass1, f_pass2, f_stop2, Ripple_pass,`
	`Atten_stop, fs = 1.00, ftype = 'butter')`
	`ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or bessel`

Bandstop (bilinear)	`b, a, sos = IIR_bsf(f_pass1, f_stop1, f_stop2, f_pass2, Ripple_pass,`
	`Atten_stop, fs = 1.00, ftype = 'butter')`
	`ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or bessel`

Support Functions	
SOS list plot	`freqz_resp_cas_list(sos,mode = 'dB',fs=1.0,Npts = 1024,fsize=(6,4))`
SOS `freqz`	`w, Hcas = freqz_cas(sos,w)`
SOS plot pole-zero	`sos_zplane(sos,auto_scale=True,size=2,tol = 0.001)` More accurate root factoring results in more accurate pole-zero plot.
Cascade SOS	`sos = sos_cascade(sos1,sos2)`

[*]These functions wrap `scipy.signal.iirdesign()` to provide an interface more consistent with the FIR design functions found in the module `fir_design_helper.py`. The function `unique_cpx_roots()` is used to mark repeated poles and zeros in `sos_zplane`. Note: All critical frequencies given in increasing order.

- At this point we are ready to move into Lab2b `FIR Filter Design and C Headers.ipynb` and Lab2c `IIR Filter Design and C Headers.ipynb` which will involve designing FIR and IIR filters, respectively
- Its time to open some Jupyter notebooks