

```
[1]: %pylab inline
import sk_dsp_comm.sigsys as ss
import sk_dsp_comm.fir_design_helper as fir_d
import sk_dsp_comm.iir_design_helper as iir_d
import sk_dsp_comm.multirate_helper as mrh
import scipy.signal as signal
from IPython.display import Audio, display
from IPython.display import Image, SVG
```

Populating the interactive namespace from numpy and matplotlib

```
[2]: %config InlineBackend.figure_formats=['svg'] # SVG inline viewing
```

Filter Design Using the Helper Modules

The Scipy package *signal* assists with the design of many digital filter types. As an alternative, here we explore the use of the filter design modules found in `scikit-dsp-comm` (<https://github.com/mwickert/scikit-dsp-comm>).

In this note we briefly explore the use of `sk_dsp_comm.fir_design_helper` and `sk_dsp_comm.iir_design_helper`. In the examples that follow we assume the import of these modules is made as follows:

```
import sk_dsp_comm.fir_design_helper as fir_d
import sk_dsp_comm.iir_design_helper as iir_d
```

The functions in these modules provide an easier and more consistent interface for both finite impulse response (FIR) (linear phase) and infinite impulse response (IIR) classical designs. Functions inside these modules wrap `scipy.signal` functions and also incorporate new functionality.

Design From Amplitude Response Requirements

With both `fir_design_helper` and `iir_design_helper` a design starts with amplitude response requirements, that is the filter passband critical frequencies, stopband critical frequencies, passband ripple, and stopband attenuation. The number of taps/coefficients (FIR case) or the filter order (IIR case) needed to meet these requirements is then determined and the filter coefficients are returned as an ndarray `b` for FIR, and for IIR both `b` and `a` arrays, and a second-order sections `sos` 2D array, with the rows containing the corresponding cascade of second-order sections topology for IIR filters.

For the FIR case we have in the z -domain

with ndarray `b` = $[b_0, b_1, \dots, b_N]$. For the IIR case we have in the z -domain

$$\begin{aligned} H_{\text{IIR}}(z) &= \frac{\sum_{k=0}^M b_k z^{-k}}{\sum_{k=1}^N a_k z^{-k}} \\ &= \prod_{k=0}^{N_s-1} \frac{b_{k0} + b_{k1} z^{-1} + b_{k2} z^{-2}}{1 + a_{k1} z^{-1} + a_{k2} z^{-2}} = \prod_{k=0}^{N_s-1} H_k(z) \end{aligned}$$

where $N_s = \lfloor (N+1)/2 \rfloor$. For the `b/a` form the coefficients are arranged as

`b` = `[b0, b1, ..., bM-1]`, the numerator `filter` coefficients
`a` = `[a0, a1, ..., aN-1]`, the denominator `filter` coefficients

For the `sos` form each row of the 2D `sos` array corresponds to the coefficients of $H_k(z)$, as follows:

```
SOS_mat = [[b00, b01, b02, 1, a01, a02], #biquad 0
            [b10, b11, b12, 1, a11, a12], #biquad 1
            .
            .
            [bNs-10, bNs-11, bNs-12, 1, aNs-11, aNs-12]] #biquad Ns-1
```

Linear Phase FIR Filter Design

The primary focus of this module is adding the ability to design linear phase FIR filters from user friendly amplitude response requirements.

Most digital filter design is motivated by the desire to approach an ideal filter. Recall an ideal filter will pass signals of a certain of frequencies and block others. For both analog and digital filters the designer can choose from a variety of approximation techniques. For digital filters the approximation techniques fall into the categories of IIR or FIR. In the design of FIR filters two popular techniques are truncating the ideal filter impulse response and applying a window, and optimum equiripple approximations [Oppenheim2010](#). Frequency sampling based approaches are also popular, but will not be considered here, even though `scipy.signal` supports all three. Filter design generally begins with a specification of the desired frequency response. The filter frequency response may be stated in several ways, but amplitude response is the most common, e.g., state how $H_c(j\Omega)$ or $H(e^{j\omega}) = H(e^{j2\pi f/f_s})$ should behave. A completed design consists of the number of coefficients (taps) required and the coefficients themselves (double precision float or `float64` in Numpy, and `float64_t` in C). Figure 1, below, shows amplitude response requirements in terms of filter gain and critical frequencies for lowpass, highpass, bandpass, and bandstop filters.

The critical frequencies are given here in terms of analog requirements in Hz. The sampling frequency is assumed to be in Hz. The passband ripple and stopband attenuation values are in dB.

Filter Design in dB terms. Attenuation is the negative of gain, e.g., -60 dB of stopband gain is equivalent to 60 dB of stopband attenuation.

```
[3]: Image('300ppi/FIR_Lowpass_Highpass_Bandpass_Bandstop@300ppi.png',width='90%')
```

[3]:

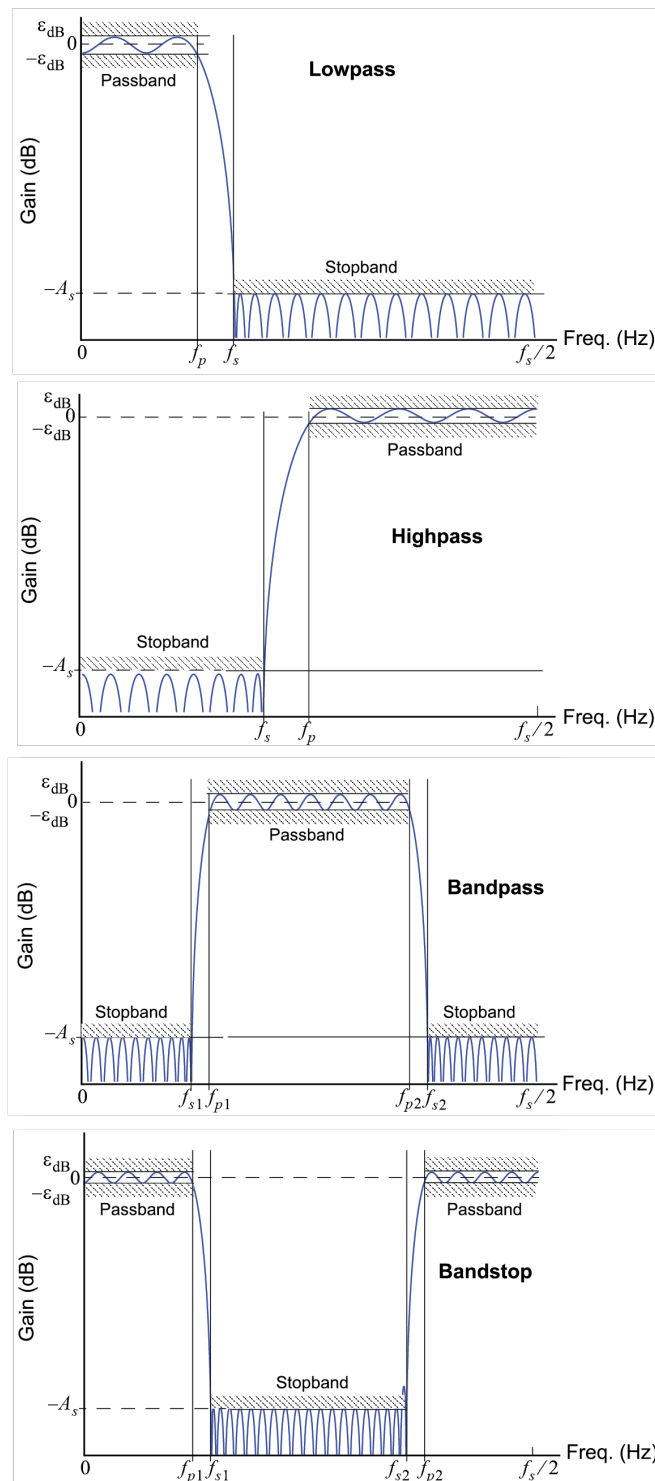


Figure: 1 General amplitude response requirements for the lowpass, highpass, bandpass, and bandstop filter types.

There are 10 filter design functions and one plotting function available in `fir_design_helper.py`.

Four functions for designing Kaiser window based FIR filters and four functions for designing equiripple based FIR filters. Of the eight just described, they all take in amplitude response requirements and return a coefficients array. Two of the 10 filter functions are simply wrappers around the `scipy.signal` function `signal.firwin()` for designing filters of a specific order when one (lowpass) or two (bandpass) critical frequencies are given. The wrapper functions fix the window type to the `firwin` default of hann (hanning). The remaining eight are described below

```
[4]: Image('300ppi/FIR_Kaiser_Equiripple_Table@300ppi.png',width='80%')
```

[4]:

Table 1: FIR filter design functions in `fir_design_helper.py`.

Type	FIR Filter Design Functions
Kaiser Window	
Lowpass	<code>h_FIR = firwin_kaiser_lpf(f_pass, f_stop, d_stop, fs = 1.0, N_bump=0)</code>
Highpass	<code>h_FIR = firwin_kaiser_hpf(f_stop, f_pass, d_stop, fs = 1.0, N_bump=0)</code>
Bandpass	<code>h_FIR = firwin_kaiser_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop, fs = 1.0, N_bump=0)</code>
Bandstop	<code>h_FIR = firwin_kaiser_bsf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop, fs = 1.0, N_bump=0)</code>
Equiripple Approximation	
Lowpass	<code>h_FIR = fir_remez_lpf(f_pass, f_stop, d_pass, d_stop, fs = 1.0, N_bump=5)</code>
Highpass	<code>h_FIR = fir_remez_hpf(f_stop, f_pass, d_pass, d_stop, fs = 1.0, N_bump=5)</code>
Bandpass	<code>h_FIR = fir_remez_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_pass, d_stop, fs = 1.0, N_bump=5)</code>
Bandstop	<code>h_FIR = fir_remez_bsf(f_pass1, f_stop1, f_stop2, f_pass2, d_pass, d_stop, fs = 1.0, N_bump=5)</code>
Support Function: Compare Designs	
Plot a List of Designs	<code>freqz_resp_list([b],a=[1],mode = 'dB',fs=1.0,Npts = 1024,fsize=(6,4))</code> where <code>[b]</code> is a list coefficient arrays and mode can be: 'dB', 'phase' in radians, 'groupdelay_s' in samples or 'groupdelay_t' in seconds

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).

Design Examples

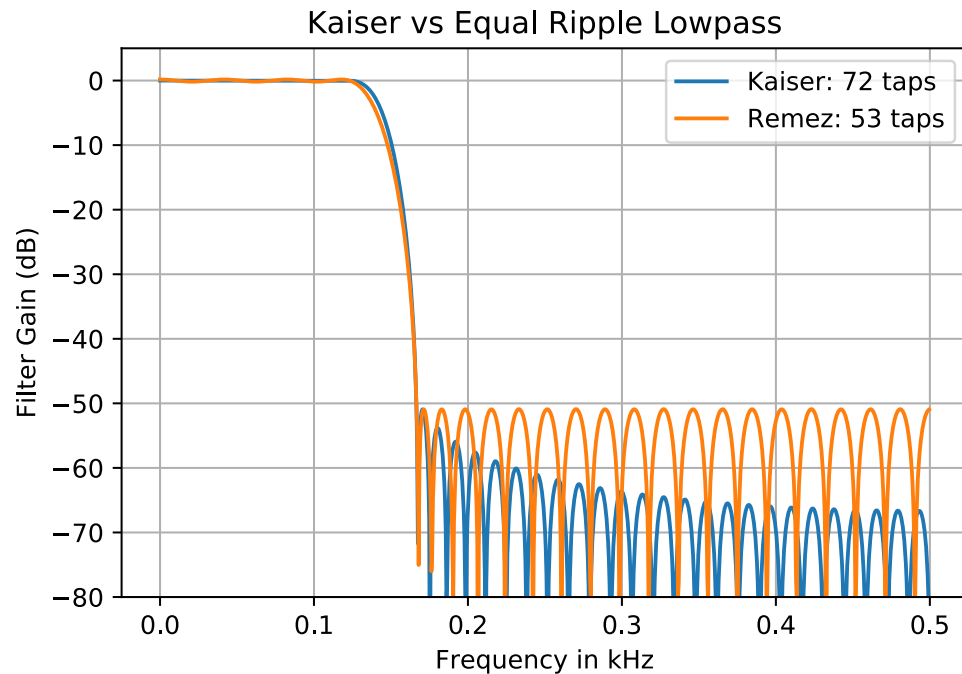
Example 1: Lowpass with $f_s = 1$ Hz

For this 31 tap filter we choose the cutoff frequency to be $F_c = F_s/8$, or in normalized form $f_c = 1/8$.

```
[5]: b_k = fir_d.firwin_kaiser_lpf(1/8,1/6,50,1.0)
      b_r = fir_d.fir_remez_lpf(1/8,1/6,0.2,50,1.0)
```

Kaiser Win filter taps = 72.
Remez filter taps = 53.

```
[6]: fir_d.freqz_resp_list([b_k,b_r],[[1],[1]], 'dB',fs=1)
      ylim([-80,5])
      title(r'Kaiser vs Equal Ripple Lowpass')
      ylabel(r'Filter Gain (dB)')
      xlabel(r'Frequency in kHz')
      legend((r'Kaiser: %d taps' % len(b_k),r'Remez: %d taps' % len(b_r)),loc='best')
      grid();
```

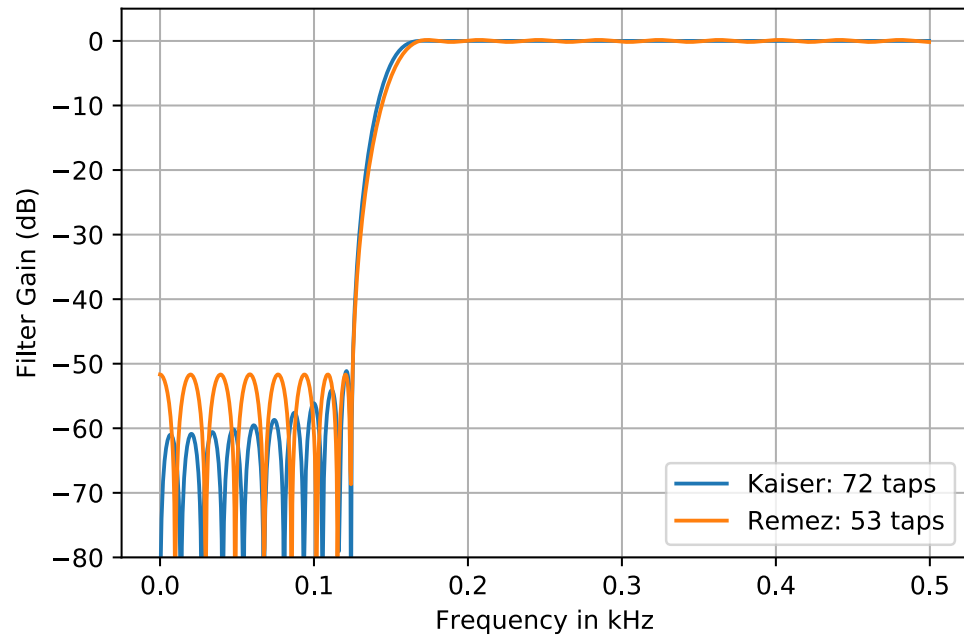


```
[7]: b_k_hp = fir_d.firwin_kaiser_hpf(1/8,1/6,50,1.0)
      b_r_hp = fir_d.fir_remez_hpf(1/8,1/6,0.2,50,1.0)
```

Kaiser Win filter taps = 72.
Remez filter taps = 53.

```
[8]: fir_d.freqz_resp_list([b_k_hp,b_r_hp],[[1],[1]], 'dB',fs=1)
      ylim([-80,5])
      title(r'Kaiser vs Equal Ripple Lowpass')
      ylabel(r'Filter Gain (dB)')
      xlabel(r'Frequency in kHz')
      legend((r'Kaiser: %d taps' % len(b_k),r'Remez: %d taps' % len(b_r)),loc='best')
      grid();
```

Kaiser vs Equal Ripple Lowpass

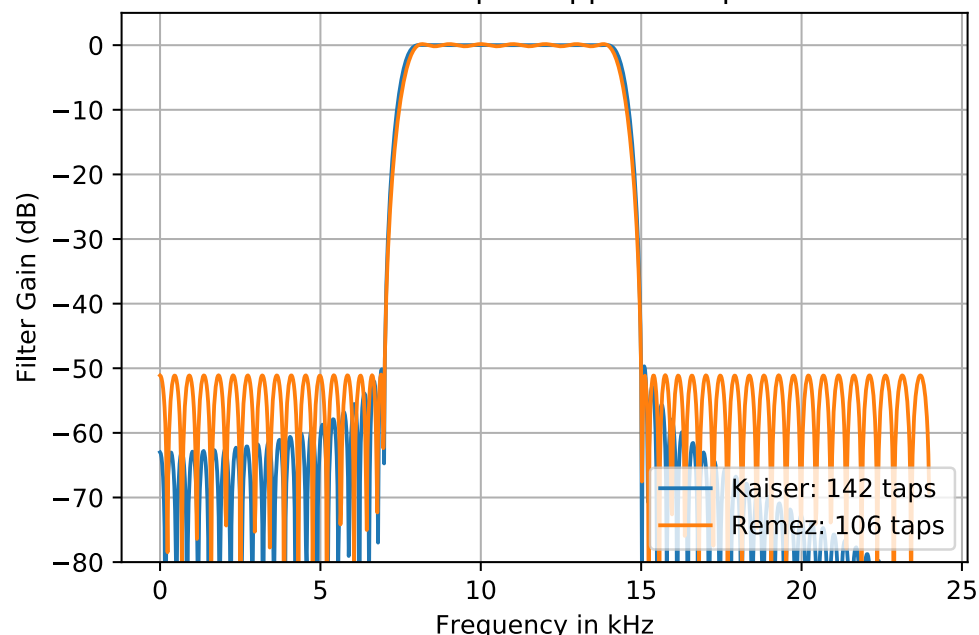


```
[9]: b_k_bp = fir_d.firwin_kaiser_bpf(7000,8000,14000,15000,50,48000)
      b_r_bp = fir_d.fir_remez_bpf(7000,8000,14000,15000,0.2,50,48000)
```

Kaiser Win filter taps = 142.
Remez filter taps = 106.

```
[10]: fir_d.freqz_resp_list([b_k_bp,b_r_bp],[[1],[1]], 'dB', fs=48)
      ylim([-80,5])
      title(r'Kaiser vs Equal Ripple Bandpass')
      ylabel(r'Filter Gain (dB)')
      xlabel(r'Frequency in kHz')
      legend((r'Kaiser: %d taps' % len(b_k_bp),
              r'Remez: %d taps' % len(b_r_bp)),
            loc='lower right')
      grid();
```

Kaiser vs Equal Ripple Bandpass



A Design Example Useful for Interpolation or Decimation

Here we consider a lowpass design that needs to pass frequencies from [0, 4000] Hz with a

the `scikit-dsp-comm` module `multirate_helper.py` to increase the sampling rate from 8000 Hz to 96000 Hz, or an interpolation factor of $L = 12$. Note at the top of this notebook we have also have the import

```
import sk_dsp_comm.multirate_helper as mrh
```

so that some of the functionality can be accessed. For more details on the use of `multirate_helper` see.

Start with an equalripple design having transition band centered on 4000 Hz with passband ripple of 0.5 dB and stopband attenuation of 60 dB.

```
[11]: b_up = fir_d.fir_remez_lpf(3300, 4300, 0.5, 60, 96000)
```

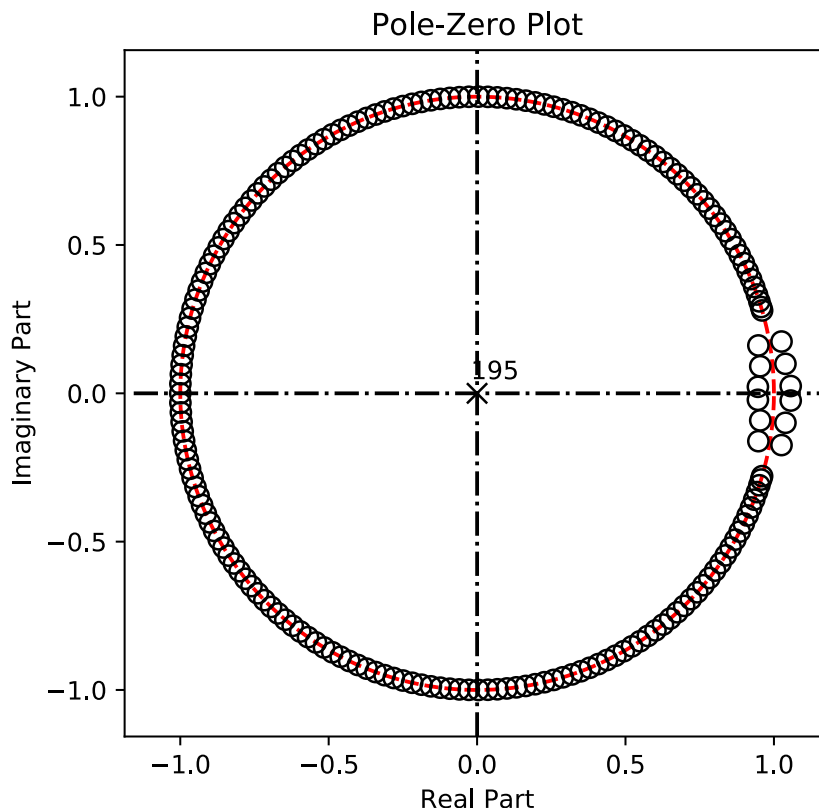
Remez filter taps = 196.

```
[12]: mr_up = mrh.multirate_FIR(b_up)
```

FIR filter taps = 196

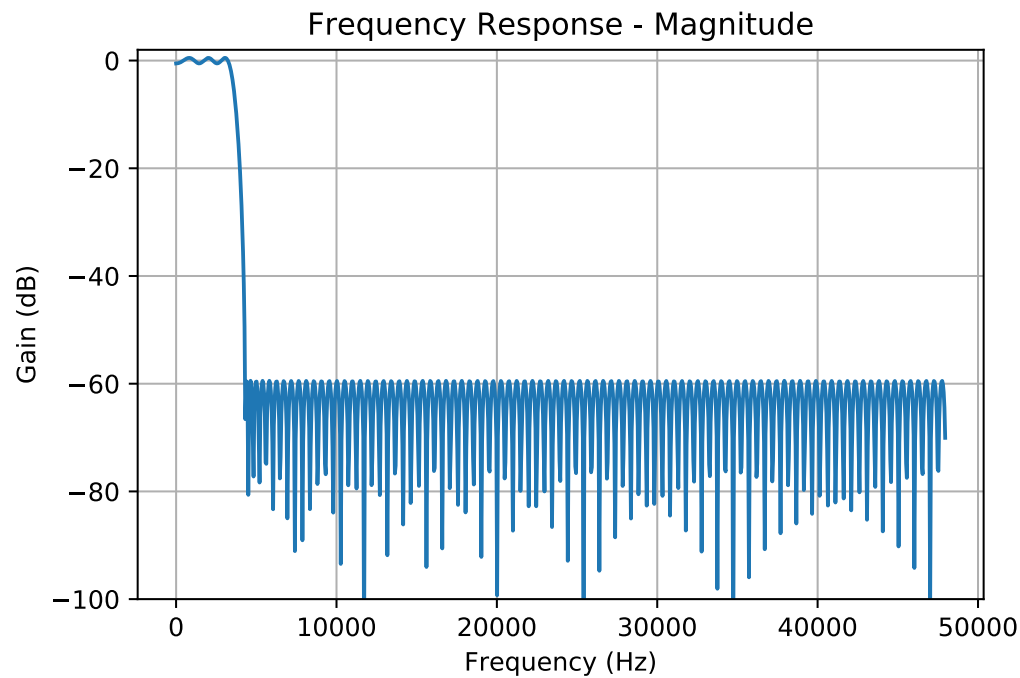
- Consider the pole-zero configuration for this high-order filter

```
[13]: # Take a look at the pole-zero configuration of this very
      # high-order (many taps) linear phase FIR
      mr_up.zplane()
```



- Check out the passband and stopband gains

```
[14]: # Verify the passband and stopband gains are as expected
mr_up.freq_resp('db', 96000)
```

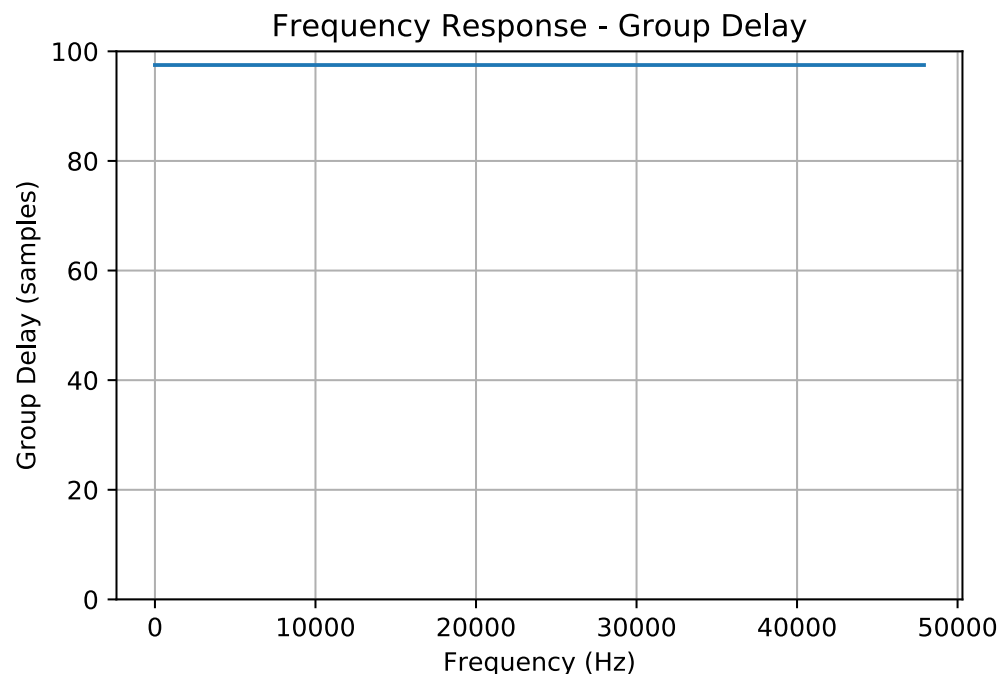


- See that the group delay is the expected value of $(N_{\text{taps}} - 1)/2 = 98$ samples

```
[15]: (len(b_up-1))/2
```

```
[15]: 98.0
```

```
[16]: # Verify that the FIR design has constant group delay (N_taps - 1)/2 samples
mr_up.freq_resp('groupdelay_s', 96000, [0, 100])
```



The object `mr_up` can now be used for interpolation or decimation with a rate change factor of 12.

Transform

The `scipy.signal` package fully supports the design of IIR digital filters from analog prototypes. IIR filters like FIR filters, are typically designed with amplitude response requirements in mind. A collection of design functions are available directly from `scipy.signal` for this purpose, in particular the function `scipy.signal.iirdesign()`. To make the design of lowpass, highpass, bandpass, and bandstop filters consistent with the module `fir_design_helper.py` the module `iir_design_helper.py` was written. Figure 2, below, details how the amplitude response parameters are defined graphically.

```
[17]: Image('300ppi/IIR_Lowpass_Highpass_Bandpass_Bandstop@300ppi.png',width='90%')
```

```
[17]:
```

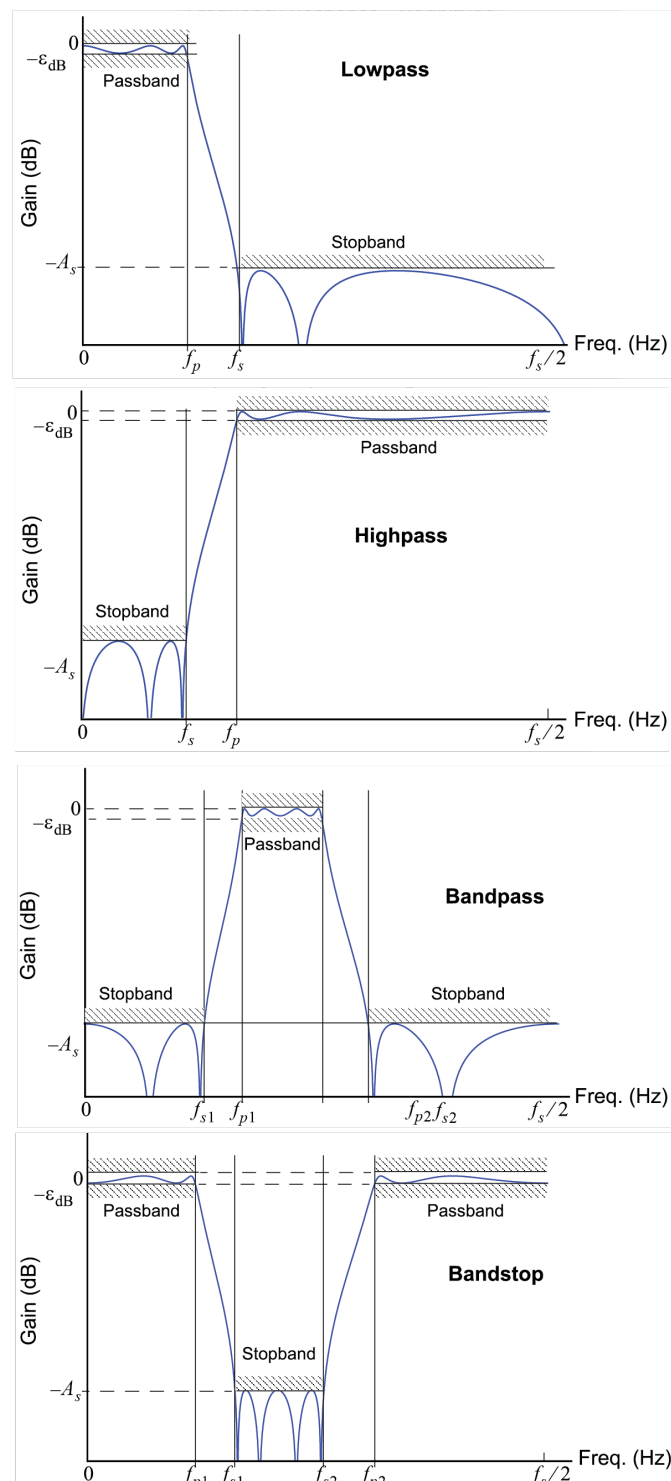


Figure 2: General amplitude response requirements for the lowpass, highpass, bandpass, and bandstop IIR filter types.

functions available. The four filter design functions are used for designing lowpass, highpass, bandpass, and bandstop filters, utilizing Butterworth, Chebyshev type 1, Chebyshev type 2, and elliptical filter prototypes. See [Oppenheim2010](#) and [ECE 5650 notes Chapter 9](#) for detailed design information. The function interfaces are described in Table 2.

```
[18]: Image('300ppi/IIR_Table@300ppi.png',width='80%')
```

[18]: **Table 2:** 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)	<code>b, a, sos = IIR_lpf(f_pass, f_stop, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')</code> ftype may be 'butter', 'butter'cheby1', 'cheby2', or 'elliptic'
Highpass (bilinear)	<code>b, a, sos = IIR_hpf(f_stop, f_pass, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')</code> ftype may be 'butter', 'butter'cheby1', 'cheby2', or 'elliptic'
Bandpass (bilinear)	<code>b, a, sos = IIR_bpf(f_stop1, f_pass1, f_pass2, f_stop2, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')</code> ftype may be 'butter', 'butter'cheby1', 'cheby2', or 'elliptic'
Bandstop (bilinear)	<code>b, a, sos = IIR_bsf(f_pass1, f_stop1, f_stop2, f_pass2, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')</code> ftype may be 'butter', 'butter'cheby1', 'cheby2', or 'elliptic'
Support Functions	
SOS list plot	<code>freqz_resp_cas_list([sos],mode = 'dB',fs=1.0,Npts = 1024,fsize=(6,4))</code> where: [sos] is a list coefficient arrays and mode can be: 'dB', 'phase' in radians, 'groupdelay_s' in samples or 'groupdelay_t' in seconds
SOS freqz	<code>w, Hcas = freqz_cas(sos,w)</code> freqz for a single sos section
SOS plot pole-zero	<code>sos_zplane(sos,auto_scale=True,size=2,tol = 0.001)</code> More accurate root factoring results in a more accurate pole-zero plot.
Cascade SOS	<code>sos = sos_cascade(sos1,sos2)</code> Create a new sos by cascading sos1 with 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.

The filter functions return the filter coefficients in two formats:

1. Traditional transfer function form as numerator coefficients `b` and denominator `a` coefficients arrays, and
2. Cascade of biquadratic sections form using the previously introduced sos 2D array or matrix.

Both are provided to allow further analysis with either a direct form topology or the sos form. The underlying `signal.iirdesign()` function also provides a third option: a list of poles and zeros. The `sos` form desirable for high precision filters, as it is more robust to coefficient quantization, in spite using double precision coefficients in the `b` and `a` arrays.

Of the remaining support functions four are also described in Table 2, above. The most significant functions are `freqz_resp_cas_list`, available for graphically comparing the frequency response over several designs, and `sos_zplane` a function for plotting the pole-zero pattern. Both operate using the `sos` matrix. A transfer function form (`b/a`) for frequency response plotting,

Filter Design Using the Filter Designer in the module. This function was first introduced in the 5.0.3 release...

section. The frequency response function plotting offers modes for gain in dB, phase in radians, group delay in samples, and group delay in seconds, all for a given sampling rate in Hz. The pole-zero plotting function locates pole and zeros more accurately than `sk_dsp_commsigsys.zplane`, as the numpy function `roots()` is only solving quadratic polynomials. Also, repeated roots can be displayed as theoretically expected, and also so noted in the graphical display by superscripts next to the pole and zero markers.

IIR Design Based on the Bilinear Transformation

There are multiple ways of designing IIR filters based on amplitude response requirements. When the desire is to have the filter approximation follow an analog prototype such as Butterworth, Chebychev, etc., is using the bilinear transformation. The function `signal.iirdesign()` described above does exactly this.

In the example below we consider lowpass amplitude response requirements and see how the filter order changes when we choose different analog prototypes.

Example: Lowpass Design Comparison

The lowpass amplitude response requirements given $f_s = 48$ kHz are: 1. $f_{\text{pass}} = 5$ kHz 2. $f_{\text{stop}} = 8$ kHz 3. Passband ripple of 0.5 dB 4. Stopband attenuation of 60 dB

Design four filters to meet the same requirements: `butter`, `cheby1`, `cheby2`, and `ellip`:

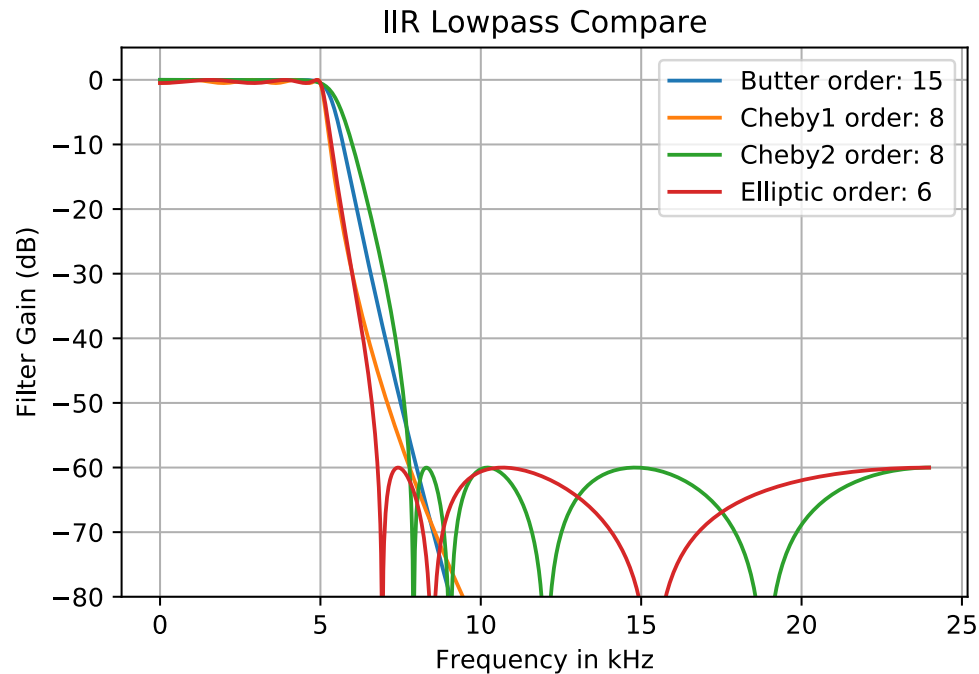
```
[19]: fs = 48000
      f_pass = 5000
      f_stop = 8000
      b_but,a_but,sos_but = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs, 'butter')
      b_cheb1,a_cheb1,sos_cheb1 = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs, 'cheby1')
      b_cheb2,a_cheb2,sos_cheb2 = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs, 'cheby2')
      b_elli,a_elli,sos_elli = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs, 'ellip')
```

```
IIR butter order = 15.  
IIR cheby1 order = 8.  
IIR cheby2 order = 8.  
IIR ellip order = 6.
```

Frequency Response Comparison

Here we compare the magnitude response in dB using the `sos` form of each filter as the input. The elliptic is the most efficient, and actually over achieves by reaching the stopband requirement at less than 8 kHz.

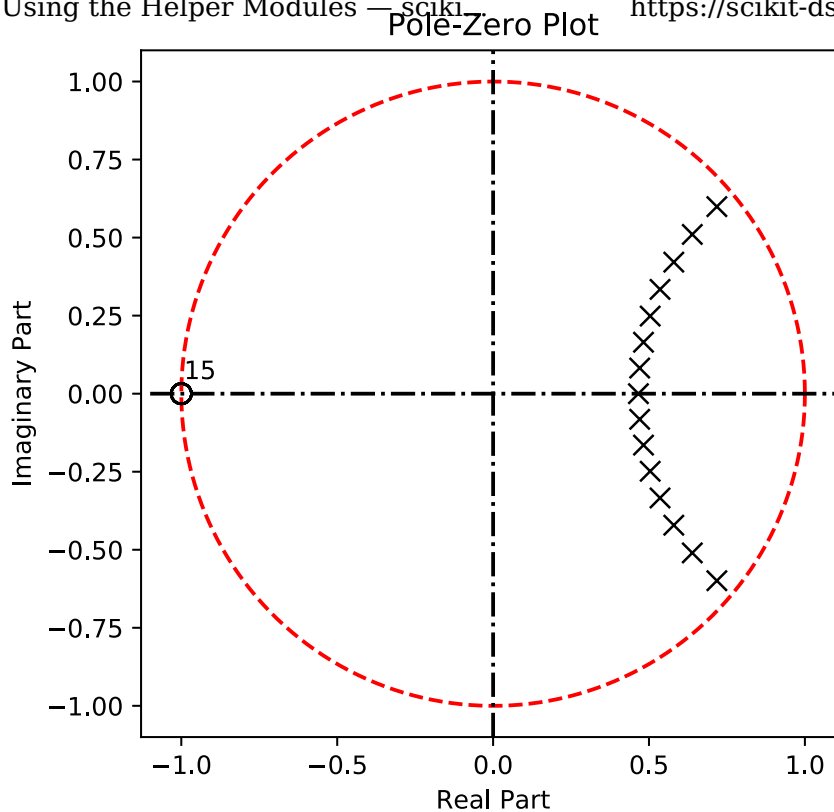
```
[20]: iir_d.freqz_resp_cas_list([sos_but, sos_cheb1, sos_cheb2, sos_elli], 'dB', fs=48)
ylim([-80, 5])
title(r'IIR Lowpass Compare')
ylabel(r'Filter Gain (dB)')
xlabel(r'Frequency in kHz')
legend((r'Butter order: %d' % (len(a_but)-1),
        r'Cheby1 order: %d' % (len(a_cheb1)-1),
        r'Cheby2 order: %d' % (len(a_cheb2)-1),
        r'Elliptic order: %d' % (len(a_elli)-1)), loc='best')
grid();
```



Next plot the pole-zero configuration of just the butterworth design. Here we use the a special version of `ss.zplane` that works with the `sos` 2D array.

```
[21]: iir_d.sos_zplane(sos_but)
```

```
[21]: (15, 15)
```



Note the two plots above can also be obtained using the transfer function form via

`iir_d.freqz_resp_list([b],[a], 'dB', fs=48)` and `ss.zplane(b,a)`, respectively. The `sos` form will yield more accurate results, as it is less sensitive to coefficient quantization. This is particularly true for the pole-zero plot, as rooting a 15th degree polynomial is far more subject to errors than rooting a simple quadratic.

For the 15th-order Butterworth the bilinear transformation maps the expected 15 s-domain zeros at infinity to $z = -1$. If you use `sk_dsp_comm.sigsys.zplane()` you will find that the 15 zeros are in a tight circle around $z = -1$, indicating polynomial rooting errors. Likewise the frequency response will be more accurate.

Signal filtering of ndarray `x` is done using the filter designs is done using functions from

`scipy.signal`:

1. For transfer function form `y = signal.lfilter(b,a,x)`
2. For sos form `y = signal.sosfilt(sos,x)`

A Half-Band Filter Design to Pass up to $W/2$ when $f_s = 8$ kHz

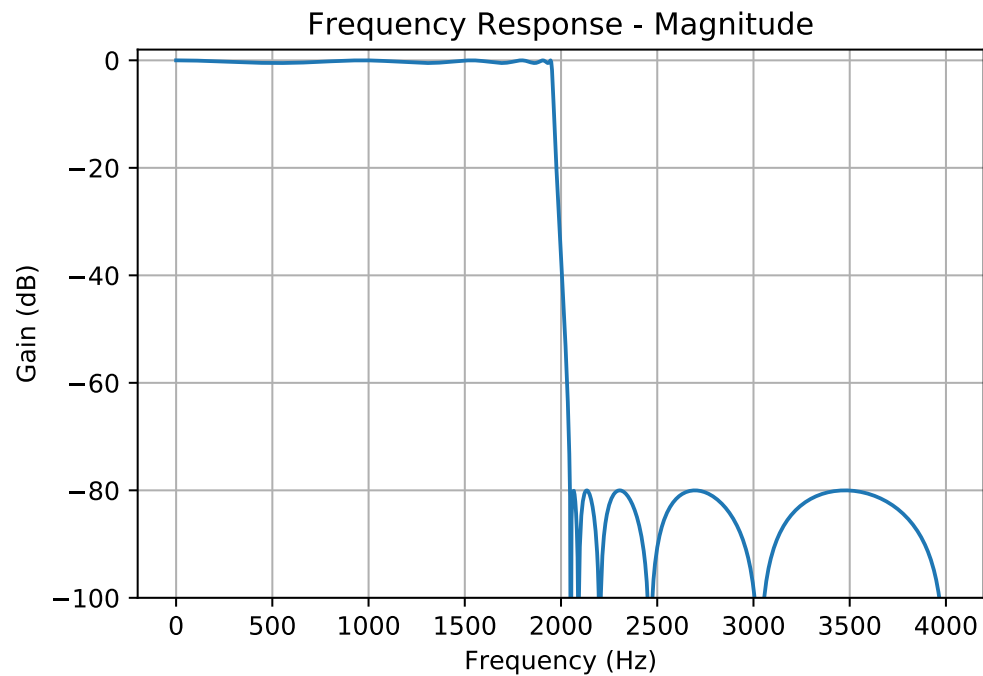
Here we consider a lowpass design that needs to pass frequencies up to $f_s/4$. Specifically when $f_s = 8000$ Hz, the filter passband becomes $[0, 2000]$ Hz. Once the coefficients are found a `mrh.multirate` object is created to allow further study of the filter, and ultimately implement filtering of a white noise signal.

Filter Design and stopband attenuation of 80 dB. The transition bandwidth is set to 100 Hz with 50 Hz on either side of 2000 Hz.

```
[22]: # Elliptic IIR Lowpass
b_lp,a_lp,sos_lp = iir_d.IIR_lpf(1950,2050,0.5,80,8000., 'ellip')
mr_lp = mrh.multirate_IIR(sos_lp)

IIR ellip order = 11.
IIR filter order = 11
```

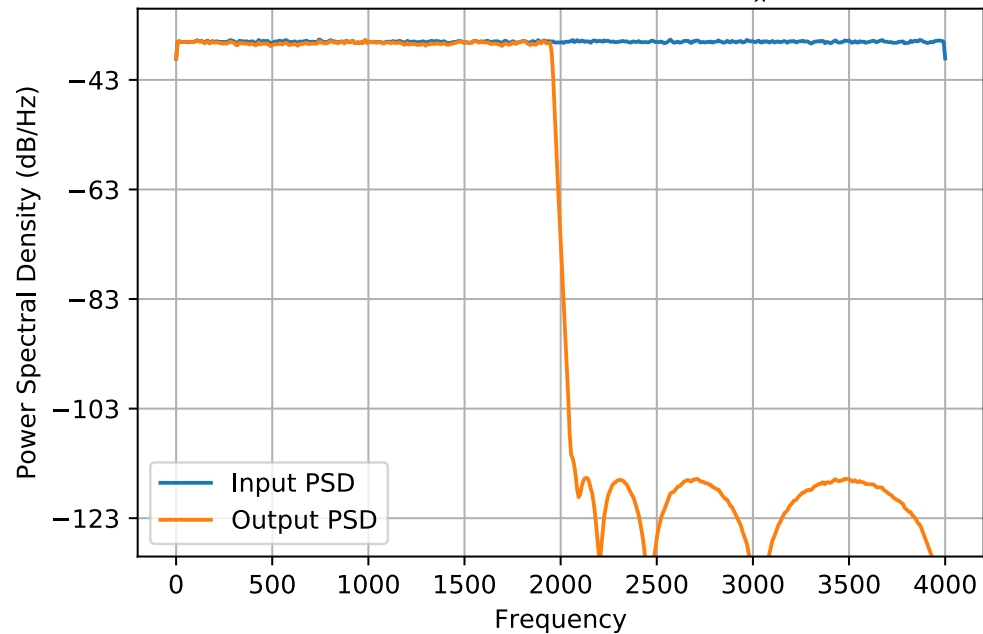
```
[23]: mr_lp.freq_resp('db',8000)
```



Pass Gaussian white noise of variance $\sigma_x^2 = 1$ through the filter. Use a lot of samples so the spectral estimate can accurately form $S_y(f) = \sigma_x^2 \cdot |H(e^{j2\pi f/f_s})|^2 = |H(e^{j2\pi f/f_s})|^2$.

```
[24]: x = randn(1000000)
y = mr_lp.filter(x)
psd(x,2**10,8000);
psd(y,2**10,8000);
title(r'Filtering White Noise Having $\sigma_x^2 = 1$')
legend(('Input PSD','Output PSD'),loc='best')
ylim([-130,-30])
```

```
[24]: (-130, -30)
```

Filtering White Noise Having $\sigma_x^2 = 1$ 

```
[25]: fs = 8000
print('Expected PSD of %2.3f dB/Hz' % (0-10*log10(fs),))
```

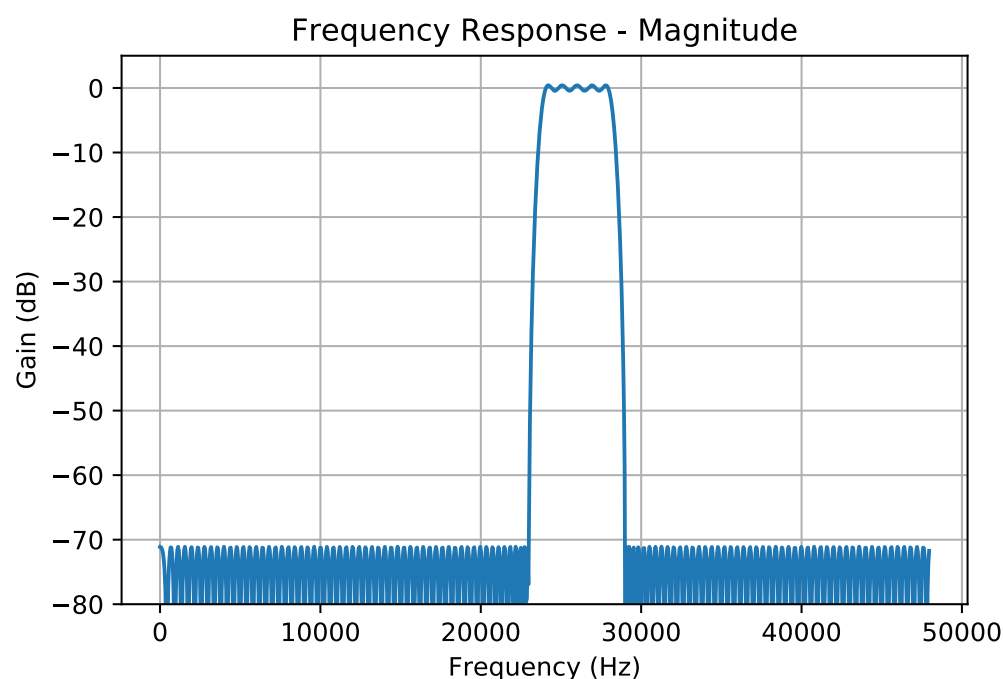
Expected PSD of -39.031 dB/Hz

Amplitude Response Bandpass Design

Here we consider FIR and IIR bandpass designs for use in an SSB demodulator to remove potential adjacent channel signals sitting either side of a frequency band running from 23 kHz to 24 kHz.

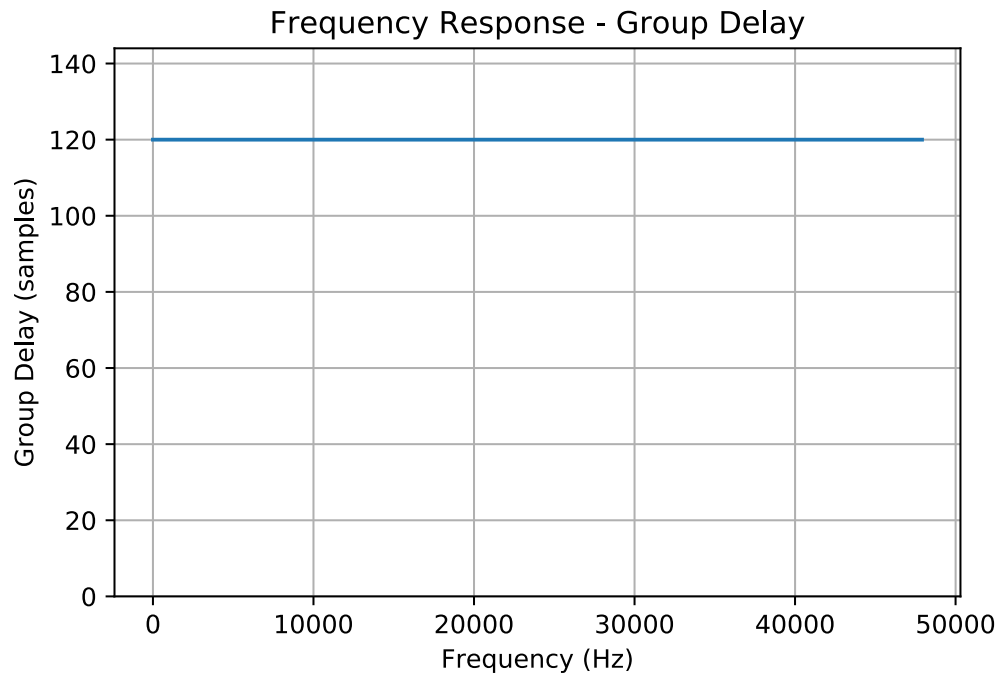
```
[26]: b_rec_bpf1 = fir_d.fir_remez_bpf(23000,24000,28000,29000,0.5,70,96000,8)
fir_d.freqz_resp_list([b_rec_bpf1],[1],mode='dB',fs=96000)
ylim([-80, 5])
grid();
```

Remez filter taps = 241.



```
[27]: b_rec_bpf1 = fir_d.fir_remez_bpf(23000,24000,28000,29000,0.5,70,96000,8)
      fir_d.freqz_resp_list([b_rec_bpf1],[1],mode='groupdelay_s',fs=96000)
      grid();
```

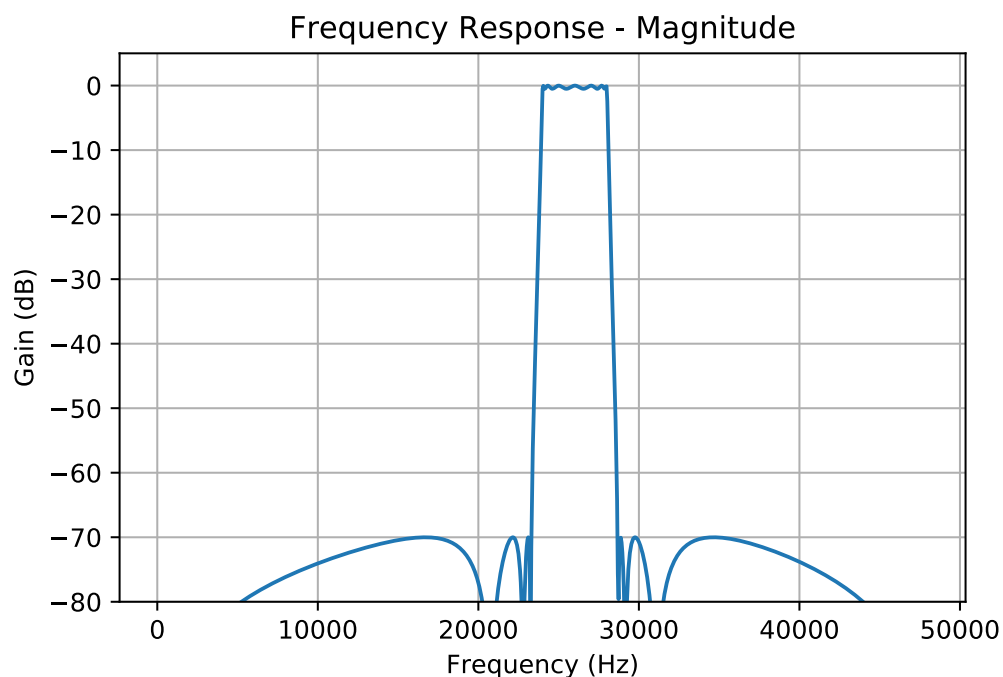
Remez filter taps = 241.



Compare the FIR design with an elliptical design:

```
[28]: b_rec_bpf2,a_rec_bpf2,sos_rec_bpf2 = iir_d.IIR_bpf(23000,24000,28000,29000,
                                                         0.5,70,96000,'ellip')
      with np.errstate(divide='ignore'):
          iir_d.freqz_resp_cas_list([sos_rec_bpf2],mode='dB',fs=96000)
      ylim([-80, 5])
      grid();
```

IIR ellip order = 14.



Filter Design on the lips has a nice tight amplitude response for minimal coefficients, but the group delay is terrible:

```
[29]: with np.errstate(divide='ignore', invalid='ignore'): #manage singularity warnings
      iir_d.freqz_resp_cas_list([sos_rec_bpf2], mode='groupdelay_s', fs=96000)
      #ylim([-80, 5])
      grid();
```

