Transform Analysis of Linear Time-Invariant Systems

In [1]:
```
# %pylab inline
from numpy import *
from matplotlib.pyplot import *
import sk_dsp_comm.sigsys as ss
# import simple_sa_new as sa_new
import scipy.signal as signal
from IPython.display import Audio, display
from IPython.display import Image, SVG

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

Group Delay

In Python group delay can be displayed for any ratio system function using the function

```
def freqz_resp_list(b,a=np.array([1]),mode = 'dB',fs=1.0,Npts
= 1024,fsize=(6,4)):
    """
    A method for displaying digital filter frequency response
magnitude,
    phase, and group delay. A plot is produced using
matplotlib

    freq_resp(self,mode = 'dB',Npts = 1024)

    A method for displaying the filter frequency response
magnitude,
    phase, and group delay. A plot is produced using
matplotlib

    freqz_resp(b,a=[1],mode = 'dB',Npts = 1024,fsize=(6,4))

        b = ndarray of numerator coefficients
        a = ndarray of denominator coefficents
     mode = display mode: 'dB' magnitude, 'phase' in radians,
or
            'groupdelay_s' in samples and 'groupdelay_t' in
sec,
            all versus frequency in Hz
     Npts = number of points to plot; default is 1024
    fsize = figure size; defult is (6,4) inches
```

```
    Mark Wickert, January 2015
    """
```

which is in the module `fir_design_helper` and `iir_design_helper`.

In [2]:
```
import sk_dsp_comm.iir_design_helper as iir_d
import sk_dsp_comm.fir_design_helper as fir_d
```

Equalization

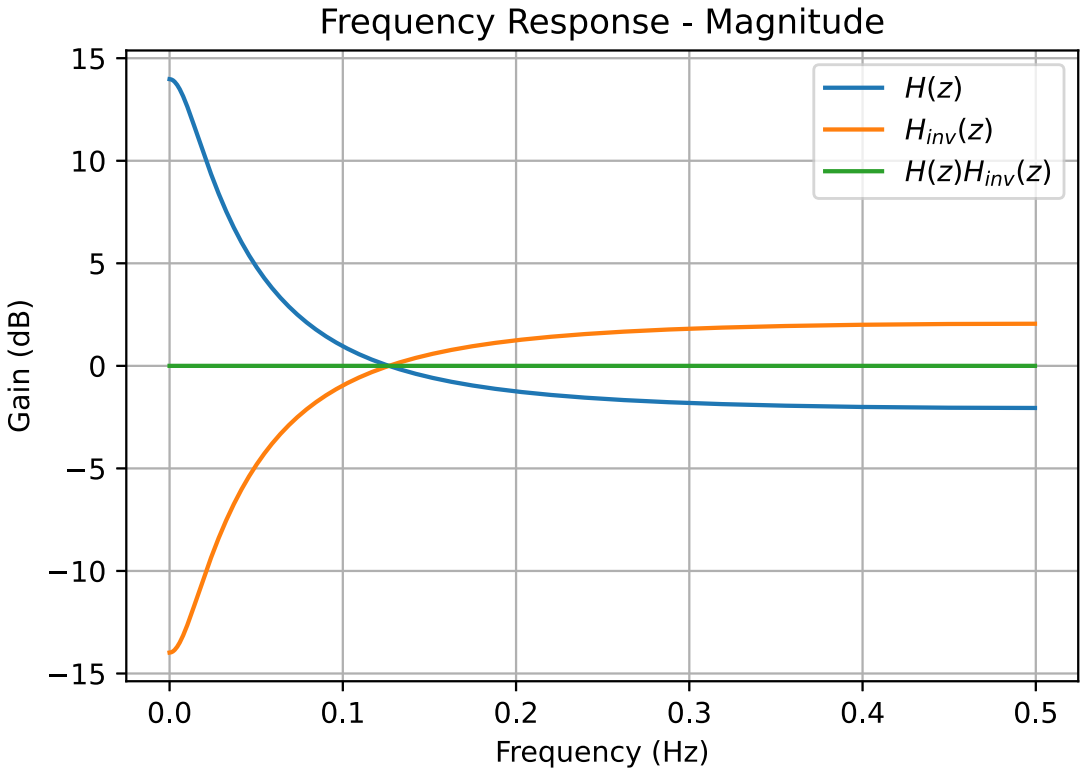
Notes Example 5.4

From the notes example

$$H(z) = \frac{1 - 0.5z^{-1}}{1 - 0.9z^{-1}}, \text{ ROC} = |z| > 0.9$$

In [3]:
```
b1 = [1, -.5]
a1 = [1, -0.9]
b2 = [1,-0.9]
a2 = [1, -0.5]
b3 = [1]
a3 = [1]

iir_d.freqz_resp_list([b1,b2,b3],[a1,a2,a3],'dB',1.0)
# iir_d.freqz_resp_list([b1,b2,b3],[a1,a2,a3],'phase',1.0)
# iir_d.freqz_resp_list([b1,b2],[a1,a2],'groupdelay_s',1.0)
# ylim([-8,10]) # limits for group delay
legend((r'$H(z)$',r'$H_{inv}(z)$', r'$H(z)H_{inv}(z)$'))
grid();
```



- In the above the inverse perfectly corrects for $H(z)$

A System with a Non-Causal Inverse

Suppose the system is modified to be

$$H(z) = \frac{1 - 2z^{-1}}{1 - 0.9z^{-1}}, \text{ ROC} = |z| > 0.9$$

We will soon learn about allpass filters and the ability to modify systems from *non-minimum* phase to *minimum* phase (all poles and zeros inside the unit circle). Here the zero at $z = 2$ can be reflected inside the unit circle. Now a causal and stable equalizer can be made. In the case of the cascade the frequency response will be flat (not unity without a scale factor), but the phase will in general be nonlinear in general. This is a consequence of $H_{\text{inv}}(z) = 1/H_{\text{min}}(z)$ as opposed to using the noncausal $1/H(z)$ in formiing the inverse.

Following the approach taken later in the chapter we will form $H(z)$ as the product of a minimum phase system and a unit gain all-pass system, $H_{\text{ap}}(z)$, ie.,

$$H(z) = \underbrace{(-2)\frac{1 - \frac{1}{2}z^{-1}}{1 - 0.9z^{-1}}}_{H_{\text{min}}(z)} \cdot \underbrace{\frac{z^{-1} - \frac{1}{2}}{1 - \frac{1}{2}z^{-1}}}_{H_{\text{ap}}(z)}$$

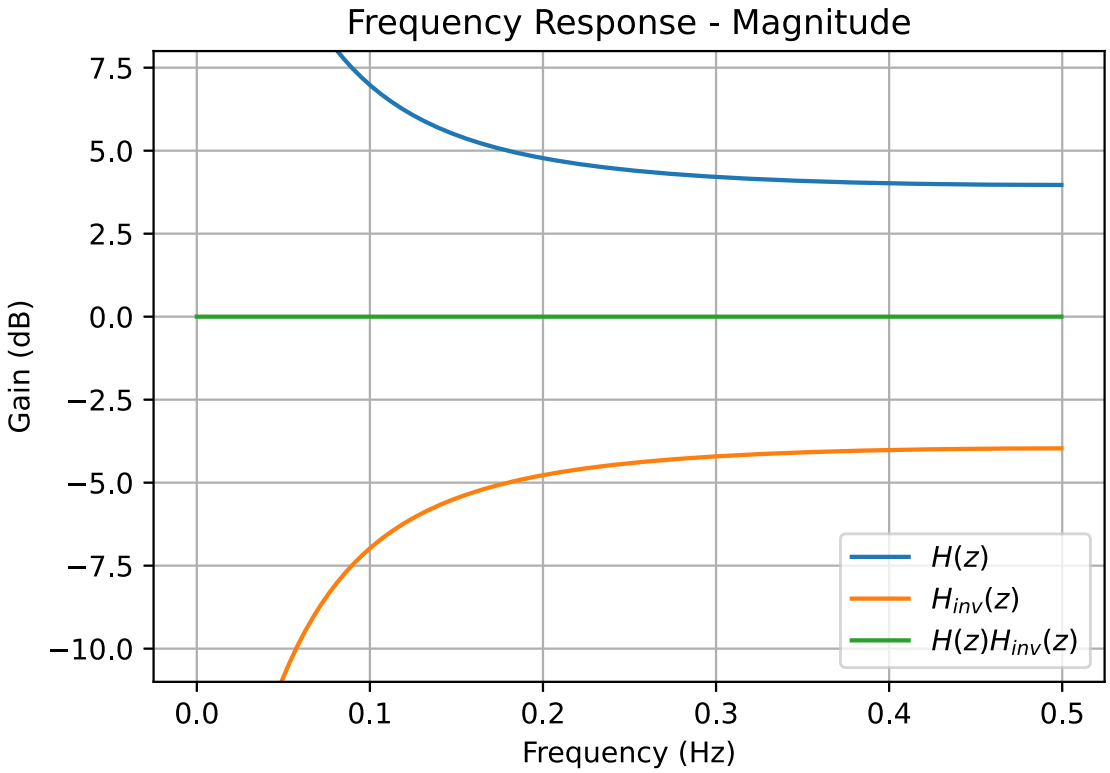
so the minimum phase system can be inverted to serve as the inverse filter

$$H_{\text{inv}}(z) = \frac{1}{H_{\text{min}}(z)} = \frac{1 - 0.9z^{-1}}{(-2)(1 - \frac{1}{2}z^{-1})}$$

The scale factor -2 insures that the gain of the cascade is unity.

In [4]:
```
b1 = [1, -2]
a1 = [1, -0.9]
b2 = [1,-0.9]
a2 = -2*array([1, -1/2]) # reflect the zero at z = 2 to z = 1/2 and ca
a3 = signal.convolve(a1,a2)
b3 = signal.convolve(b1,b2)


iir_d.freqz_resp_list([b1,b2,b3],[a1,a2,a3],'dB',1.0)
# iir_d.freqz_resp_list([b1,b2,b3],[a1,a2,a3],'phase',1.0)
# iir_d.freqz_resp_list([b1,b2,b3],[a1,a2,(-2)*a3],'groupdelay_s',1.0)
ylim([-11,8])
legend((r'$H(z)$',r'$H_{inv}(z)$', r'$H(z)H_{inv}(z)$'))
grid();
```



- In the above we see that gain is flat (a scale factor is needed to bring the gain back to unity)
- If we look at the phase and/or we will see that the phase is not linear and the group delay is not a constant

IIR and FIR Examples

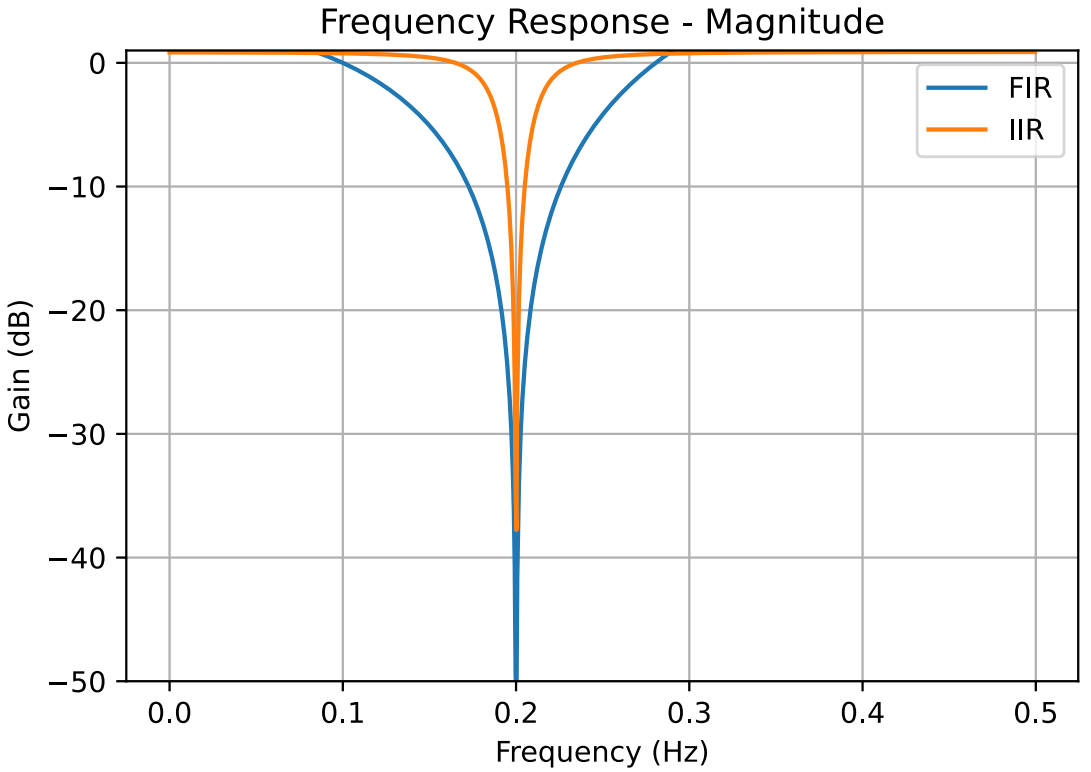
Compare magnitude phase and group delay FIR and IIR notch filters. The simple FIR notch filter takes the form

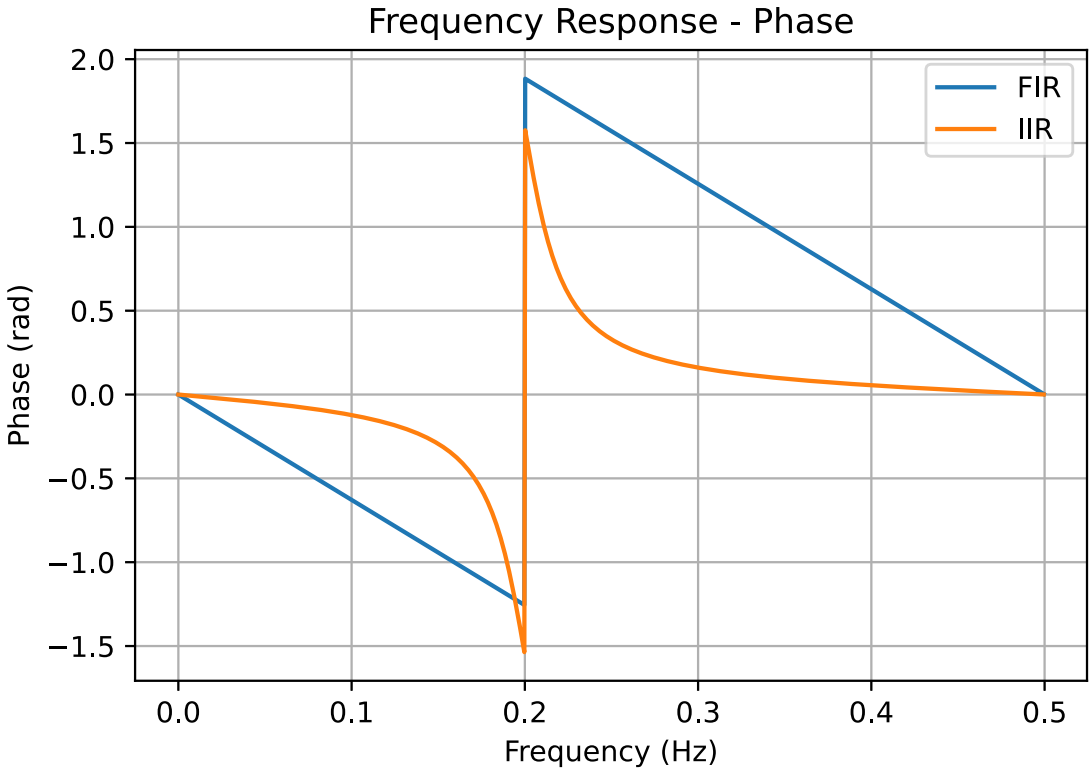
$$H_{\text{FIR}}(z) = 1 - 2\cos(\omega_0)z^{-1} + z^{-2}$$

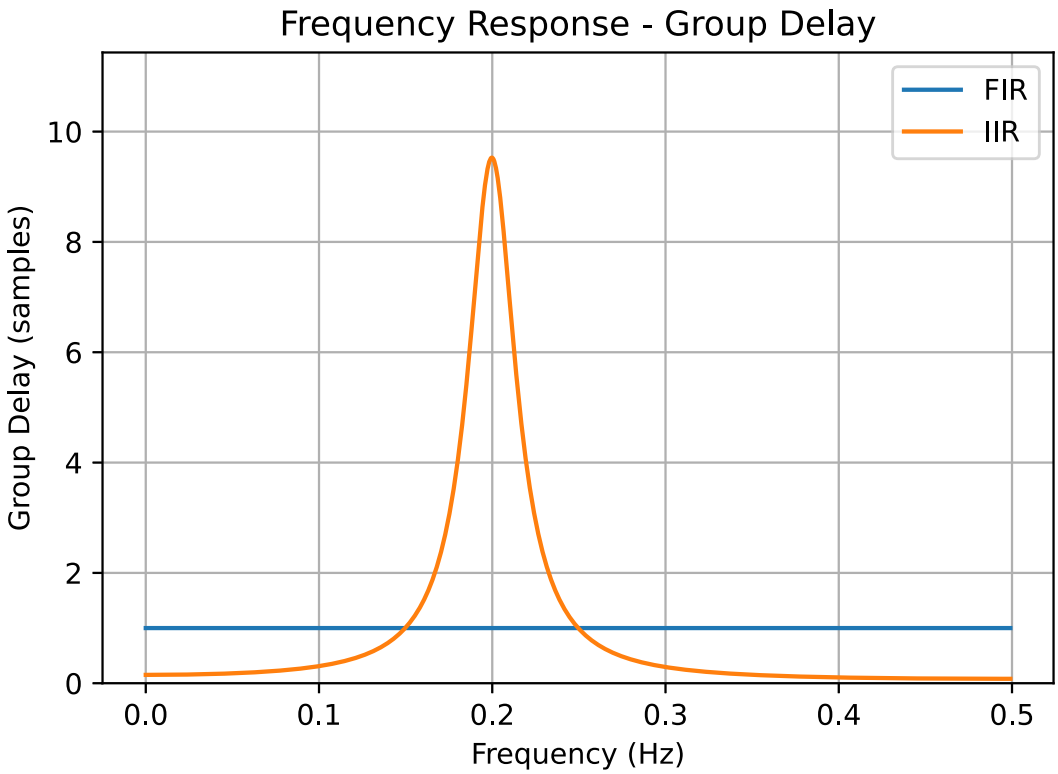
while the IIR notch takes the form

$$H_{\text{IIR}}(z) = \frac{1 - 2\cos(\omega_0)z^{-1} + z^{-2}}{1 - 2r\cos(\omega_0)z^{-1} + r^2z^{-2}}$$

In [5]:
```
w0 = 2*pi*.2
r = 0.9
fs = 1
b_FIR = [1, -2*cos(w0), 1]
a_FIR = [1]
b_IIR = [1, -2*cos(w0), 1]
a_IIR = [1, -2*r*cos(w0), r**2]
# Magnitude in dB
iir_d.freqz_resp_list([b_FIR, b_IIR],[a_FIR,a_IIR],'dB',fs)
ylim([-50,1])
legend(('FIR','IIR'))
grid();
# Phase in radians
iir_d.freqz_resp_list([b_FIR, b_IIR],[a_FIR,a_IIR],'phase',fs)
legend(('FIR','IIR'))
grid();
# Group delay in samples
iir_d.freqz_resp_list([b_FIR, b_IIR],[a_FIR,a_IIR],'groupdelay_s',fs)
legend(('FIR','IIR'))
grid();
```


Frequency Response - Magnitude
FIR
IIR
Gain (dB)
0
−10
−20
−30
−40
−50
0.0
0.1
0.2
0.3
0.4
0.5
Frequency (Hz)


Frequency Response - Phase
FIR
IIR
Phase (rad)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.1
0.2
0.3
0.4
0.5
Frequency (Hz)

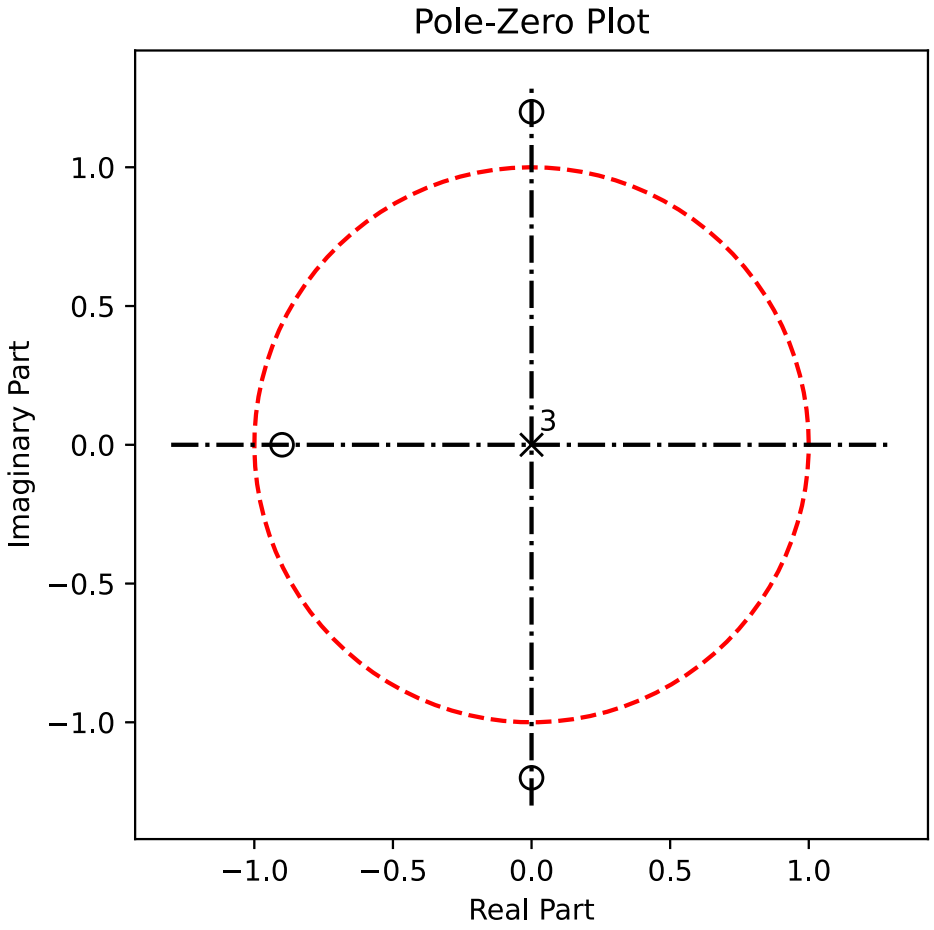


- Note that the IIR design has the best magnitude response, notching just in the vicinity of the center frequency ω_0, but poor group delay. The FIR has linear phase and hence constant group delay, but the magnitude response is very broad.

Geometric Evaluation of $H(e^{j\omega})$

In [6]:
```
b1 = signal.convolve([1,0.9],signal.convolve([1,-1j*1.2],[1,1j*1.2]))
a1 = [1]
ss.zplane(b1,a1)
```

Out[6]: (3, 0)



- The gain at $\omega = 0, \pi/2, \pi$ can be obtained simply by calculating the complex vector lengths from the three zeros to a point on the unit circle
- We use the function `numpy.linalg.norm` to find the vector lengths

In [7]:
```
linalg.norm(1 + 1j)
```

Out[7]:
```
np.float64(1.4142135623730951)
```

In [8]:
```
# For example a triangle with sides having length 3 and 4
# is known to have hypotenuse of length 5
linalg.norm(3+ 4j)
```

Out[8]:
```
np.float64(5.0)
```

In [9]:
```
H_0 = (1 - (-0.9))*linalg.norm(1**2 - 1j*1.2)*linalg.norm(1 + 1j*1.2)
H_0
```

Out[9]:
```
np.float64(4.635999999999999)
```

In [10]:
```
H_piby2 = linalg.norm(1j - (-0.9))*linalg.norm(1j - 1j*1.2)*linalg.nor
H_piby2
```

Out[10]:
```
np.float64(0.5919594580712433)
```

In [11]:
```
H_pi = linalg.norm((-1) - (-0.9))*linalg.norm((-1) - 1j*1.2)*linalg.no
H_pi
```

Out[11]:
```
np.float64(0.24399999999999994)
```

In [12]:
```
# Check the answers using `signal.freqz()`
w, H = signal.freqz(b1,a1,2*pi*array([0, 0.25, 0.5]))
abs(H)
```

Out[12]:
```
array([4.636     , 0.59195946, 0.244     ])
```

All-Pass Filters

The simple 1-section allpass takes the form given below:

$$H(z) = \frac{z^{-1} - c^*}{1 - cz^{-1}}$$

The higher order design takes the form

$$H_{\text{ap}}(z) = \frac{z^{-N} + a_1^* z^{-N+1} + \cdots + a_N^*}{1 + a_1 z^{-1} + \cdots + a_N z^{-N}} \tag{1}$$

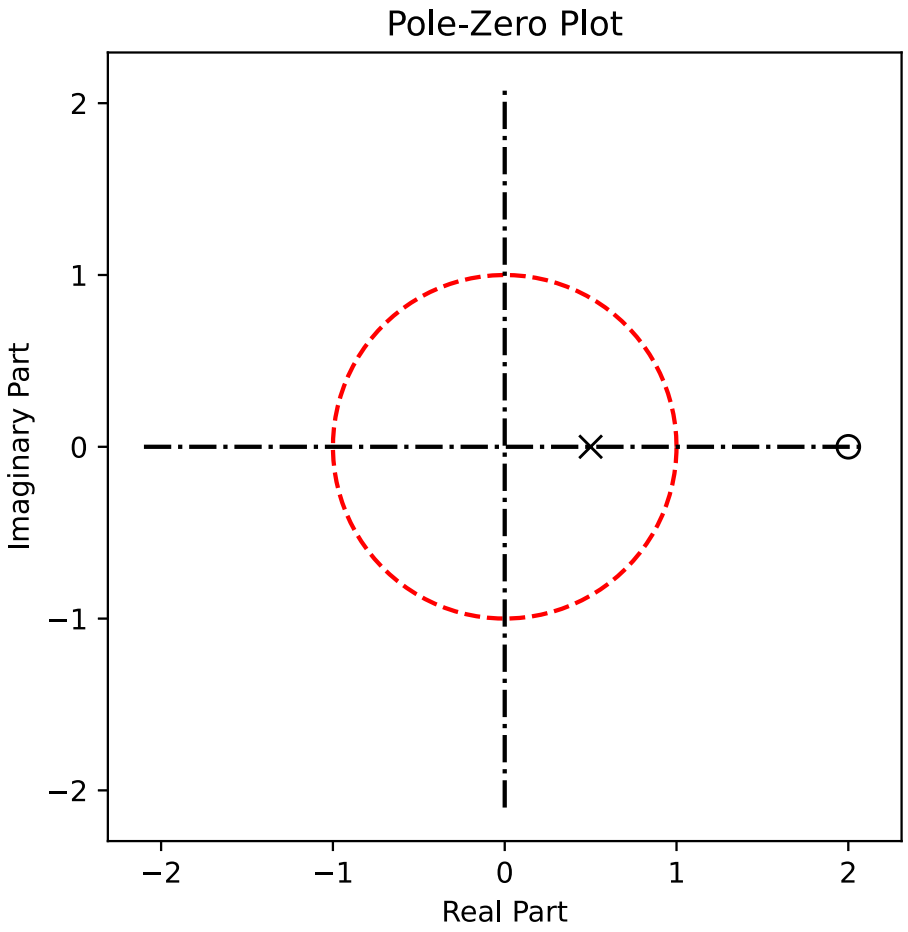
$$= z^{-N} \frac{D^*(z^{-1})}{D(z)} \tag{2}$$

where in general the coefficients a_k are complex.

Simple First-Order Example

In [8]:
```
ss.zplane([-1/2,1],[1,-1/2])
```

Out[8]:
```
(1, 1)
```



- Verify that the gain is constant at three frequencies

In [9]:
```
f = array([0,pi/2,pi])
w,H = signal.freqz([-1/2,1],[1,-1/2],2*pi*f)
```

In [10]:
```
abs(H)
```

Out[10]:
```
array([1., 1., 1.])
```

A Higher Order Design

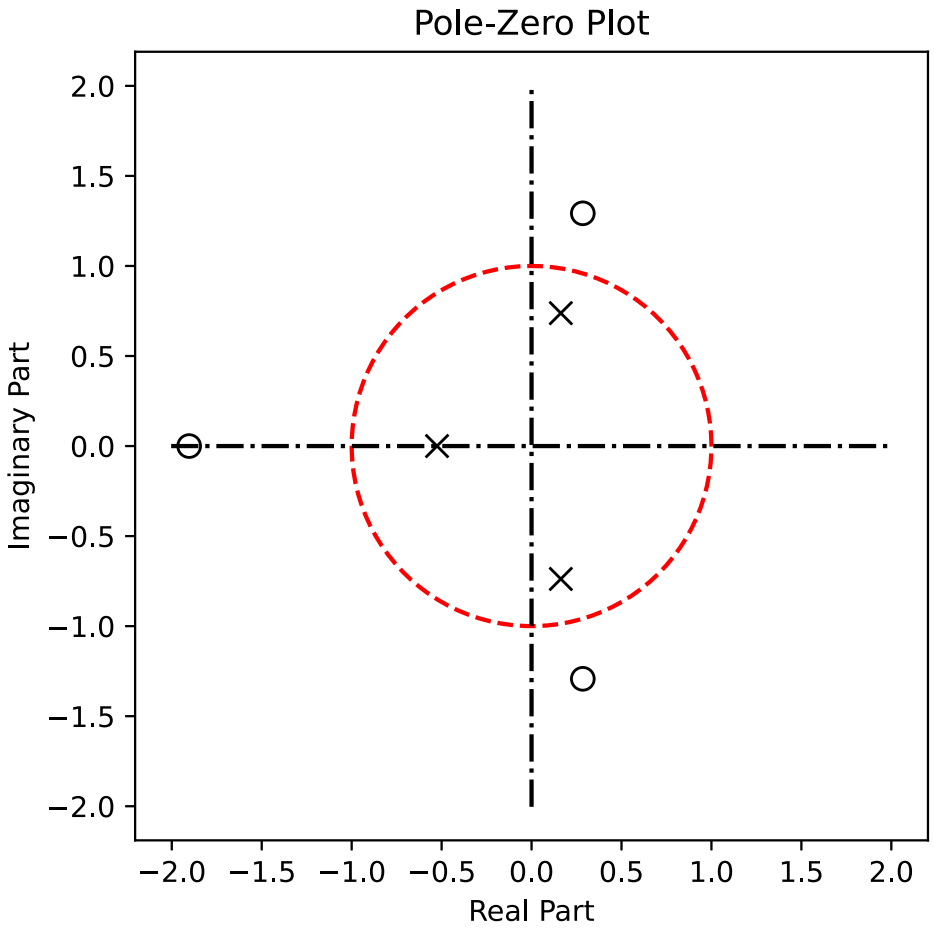
We form this by choosing some coefficients for the denominator polynomial that result in roots inside the unit circle. The numerator coefficients are just the denominator coefficients flipped from left to right:

In [11]:
```
a = [1,.2,.4,.3]
b = a[::-1]
b
```

Out[11]:
```
[0.3, 0.4, 0.2, 1]
```

In [12]:
```
ss.zplane(b,a)
```

Out[12]: (3, 3)

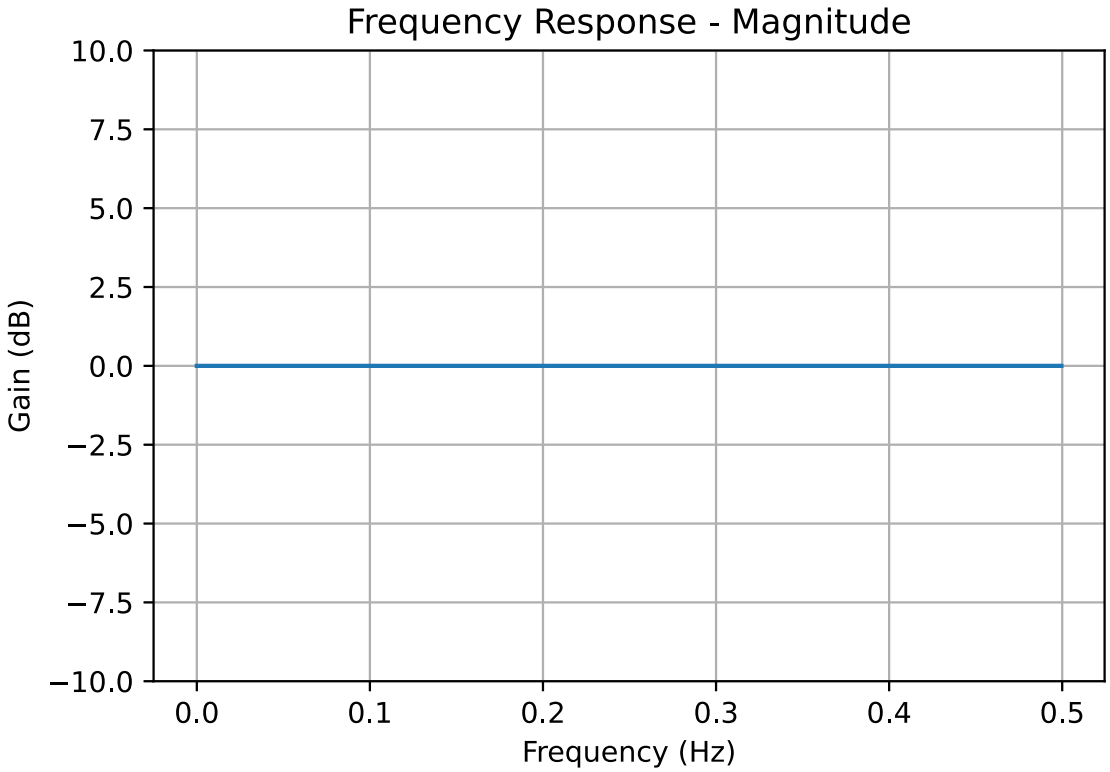


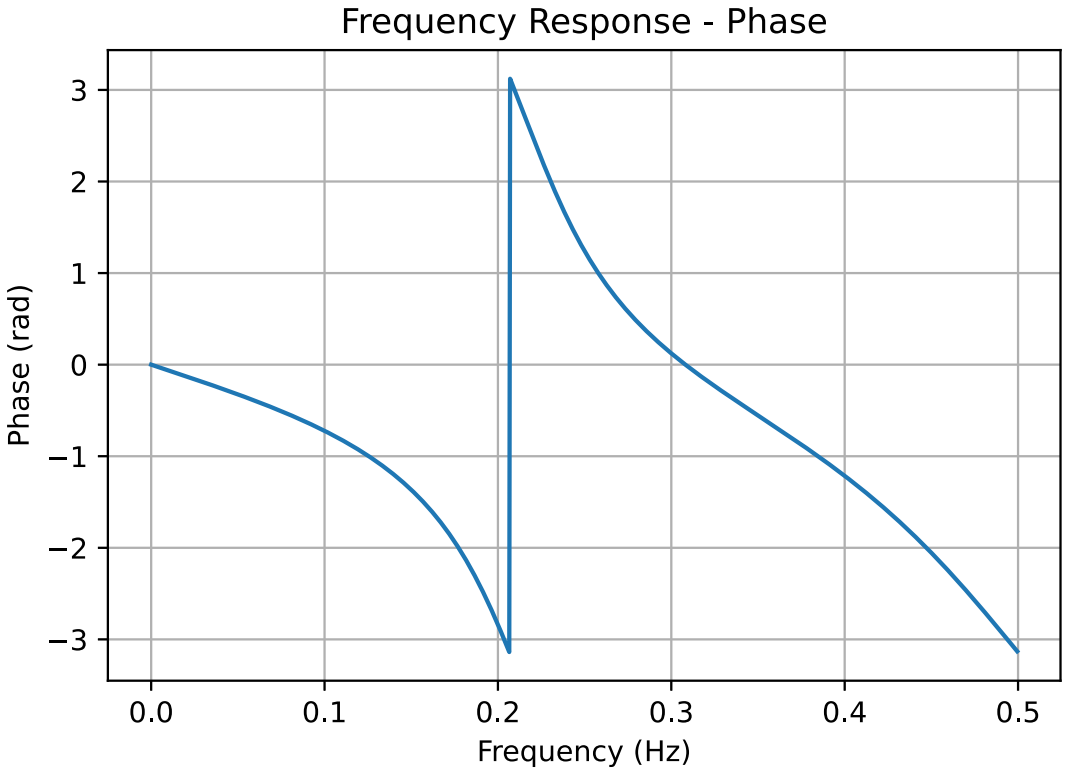
In [13]:
```
f = array([0,pi/2,pi])
w,H = signal.freqz(b,a,2*pi*f)
```

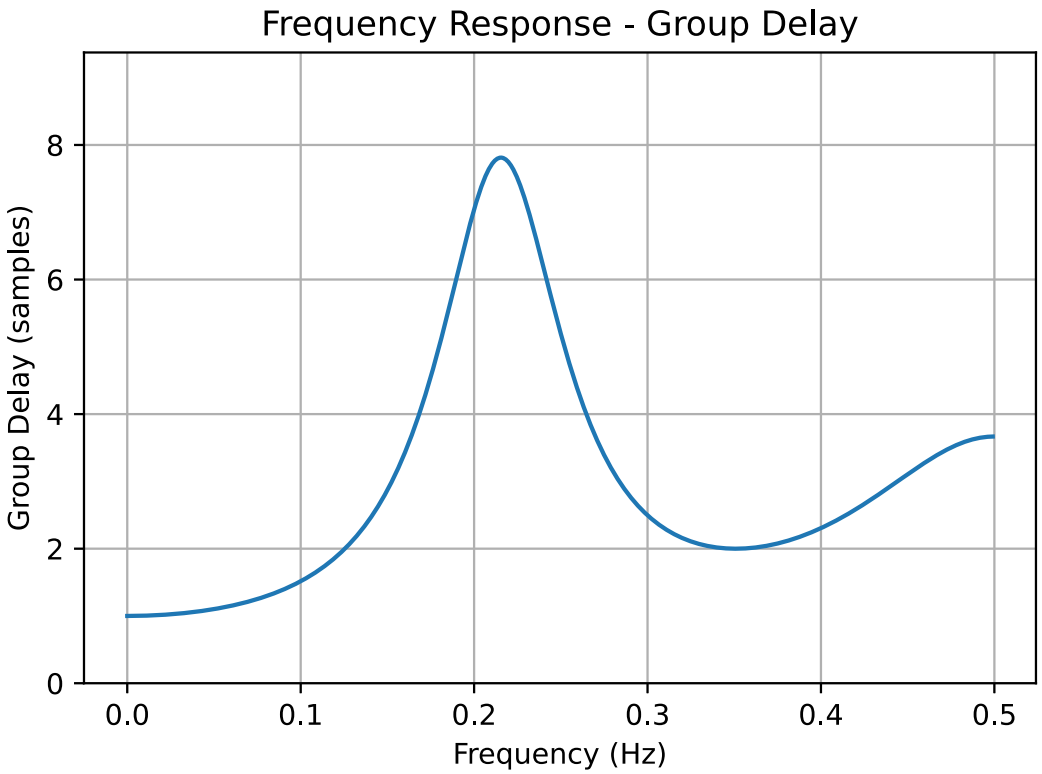
In [19]:
```
abs(H)
```

Out[19]: array([1., 1., 1.])

In [14]:
```
iir_d.freqz_resp_list([b],[a],'dB',1.0)
ylim([-10,10])
grid();
iir_d.freqz_resp_list([b],[a],'phase',1.0)
grid();
iir_d.freqz_resp_list([b],[a],'groupdelay_s',1.0)
grid();
```


Frequency Response - Magnitude
Gain (dB)
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0.0
0.1
0.2
0.3
0.4
0.5
Frequency (Hz)


Frequency Response - Phase
Phase (rad)
3
2
1
0
−1
−2
−3
0.0
0.1
0.2
0.3
0.4
0.5
Frequency (Hz)



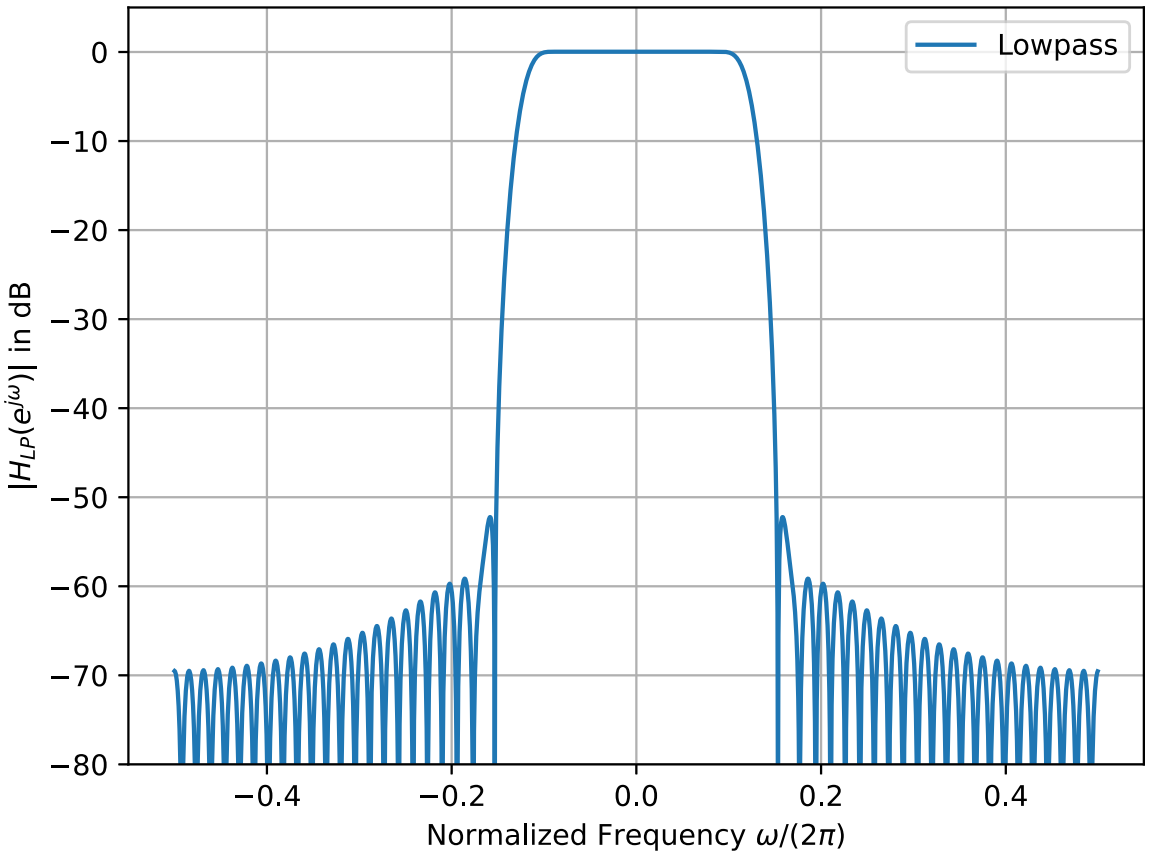
Primitives for Simulations in Text 5.21

To help convince yourself of the filter types in Text Problem 5.21 simple simulation/analysis models can be created in a Jupyter notebook.

A Simple Fixed Length FIR Filter

A simple fixed length FIR lowpass can be designed using filter design function in `scipy.signal`. In this case we use a windowed since function design:

In [15]:
```
h_LP = signal.firwin(63,1/4) # set wc = pi*1/4 = 2*pi*1/8
f = arange(-1/2,1/2,1/2048) # range w over -pi to pi
w,H_LP = signal.freqz(h_LP,1,2*pi*f)
plot(w/2/pi,20*log10(abs(H_LP)))
ylabel(r'$|H_{LP}(e^{j\omega})|$ in dB')
legend((('Lowpass',)),facecolor='white',labelspacing=0.0)
xlabel(r'Normalized Frequency $\omega/(2\pi)$')
ylim([-80,5])
grid();
```



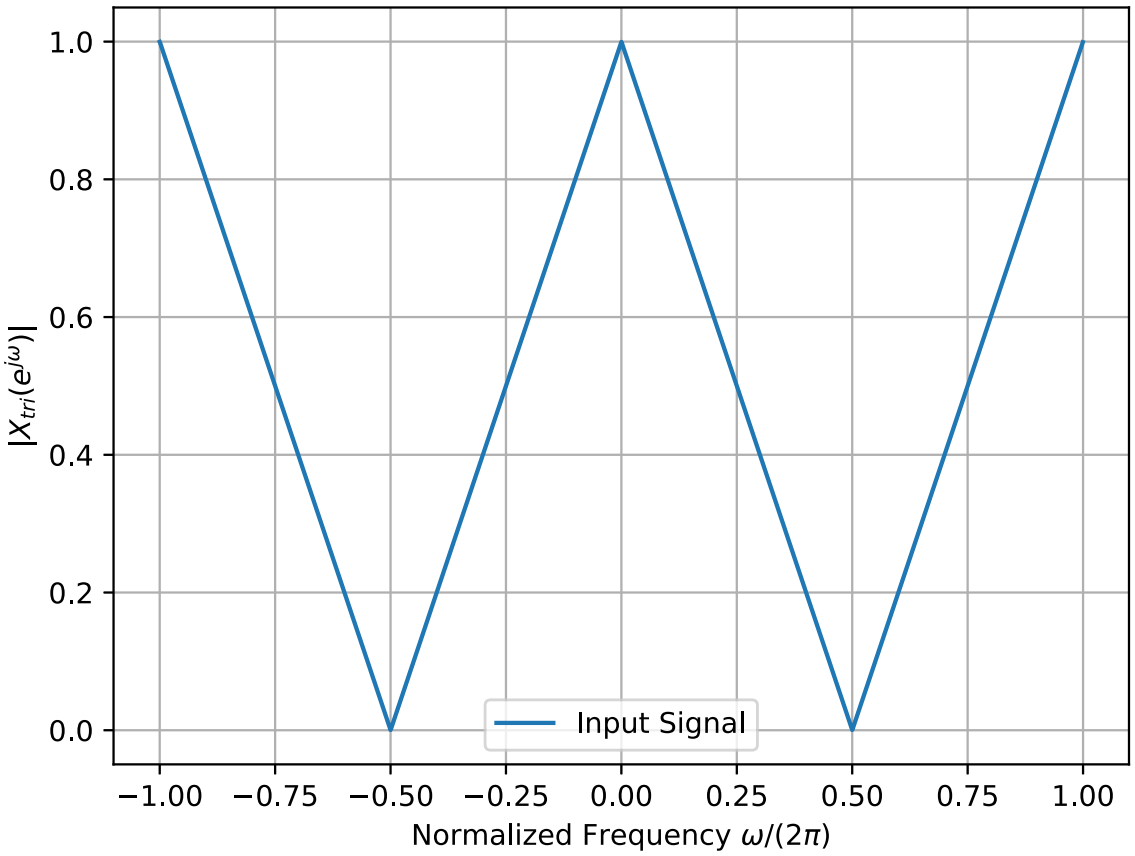
In part (a) for example you need to process a direct through signal with a filtered signal. An ideal LPF has a mean delay of zero samples. The realizable lowpass filter above has mean delay of $(63-1)/2 = 31$ samples. So a direct through signal will need a pure delay of 31 samples.

A Test Signal with a Triangular Shaped Spectrum

For some of the cases you need to see the spectral shaping impart by actually filtering a signal and then noting the output spectrum. Here it is helpful to have a signal having a simple triangular shaped spectrum. Using $\text{sinc}^2()$ we can create such a signal:

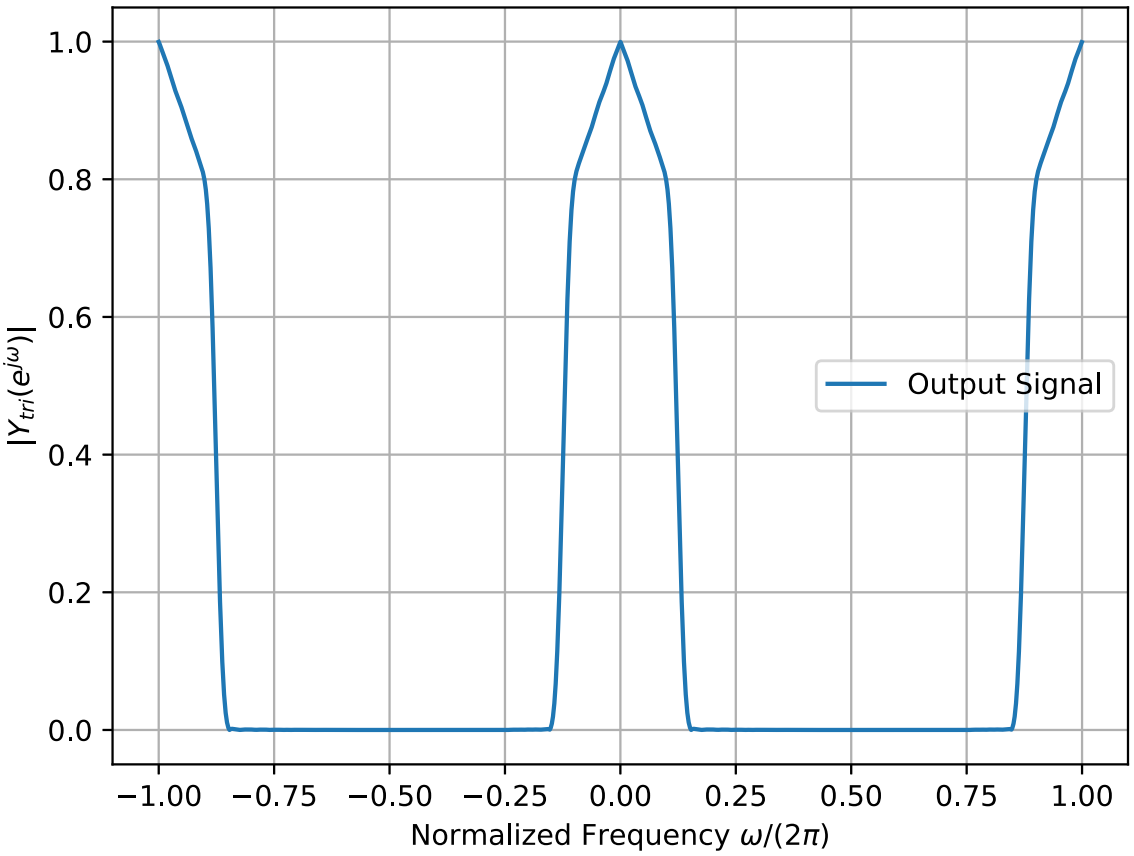
In [16]:
```
n = arange(-500,500)
x_tri = sinc(50*n/100)**2/(100/50) # sinc**2 signal has tri spec

f = arange(-1,1,1/2048) # range w over -2pi to 2pi
w,Xtri = signal.freqz(x_tri,1,2*pi*f)
plot(f,abs(Xtri));
ylabel(r'$|X_{tri}(e^{j\omega})|$')
legend((('Input Signal',)),facecolor='white',labelspacing=0.0)
xlabel(r'Normalized Frequency $\omega/(2\pi)$')
grid();
```



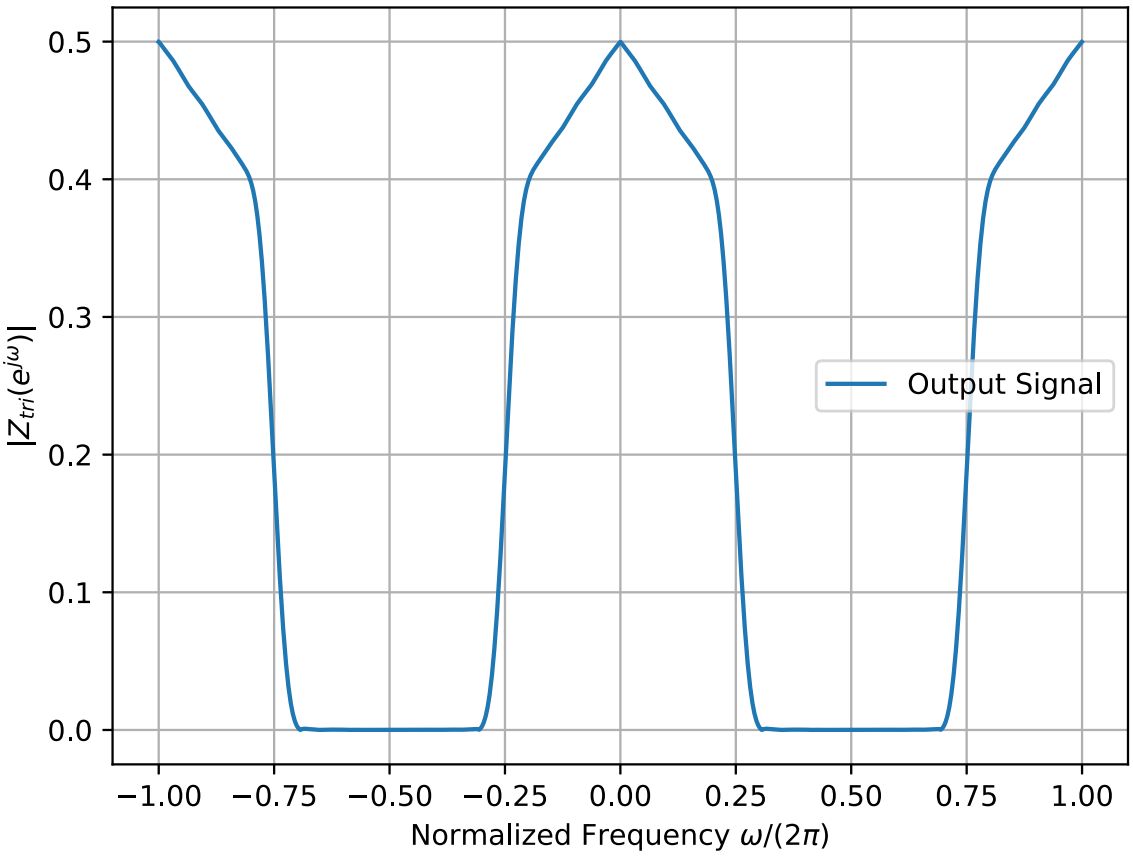
Suppose we want to filter $x_{\text{tri}}[n]$ with the LPF and then plot the resulting spectrum:

In [17]:
```
y_tri = signal.lfilter(h_LP,1,x_tri)
f = arange(-1,1,1/2048) # range w over -2pi to 2pi
w,Ytri = signal.freqz(y_tri,1,2*pi*f)
plot(f,abs(Ytri));
ylabel(r'$|Y_{tri}(e^{j\omega})|$')
legend((('Output Signal',)),facecolor='white',labelspacing=0.0)
xlabel(r'Normalized Frequency $\omega/(2\pi)$')
grid();
```



You can now downsample or upsample $y_{\text{tri}}[n]$, what ever is needed. Below I show downsampling by two:

In [18]:
```
z_tri = ss.downsample(y_tri,2)
f = arange(-1,1,1/2048) # range w over -2pi to 2pi
w,Ztri = signal.freqz(z_tri,1,2*pi*f)
plot(f,abs(Ztri));
ylabel(r'$|Z_{tri}(e^{j\omega})|$')
legend((('Output Signal',)),facecolor='white',labelspacing=0.0)
xlabel(r'Normalized Frequency $\omega/(2\pi)$')
grid();
```



The above is not too exciting as the filtering action is lowpass.

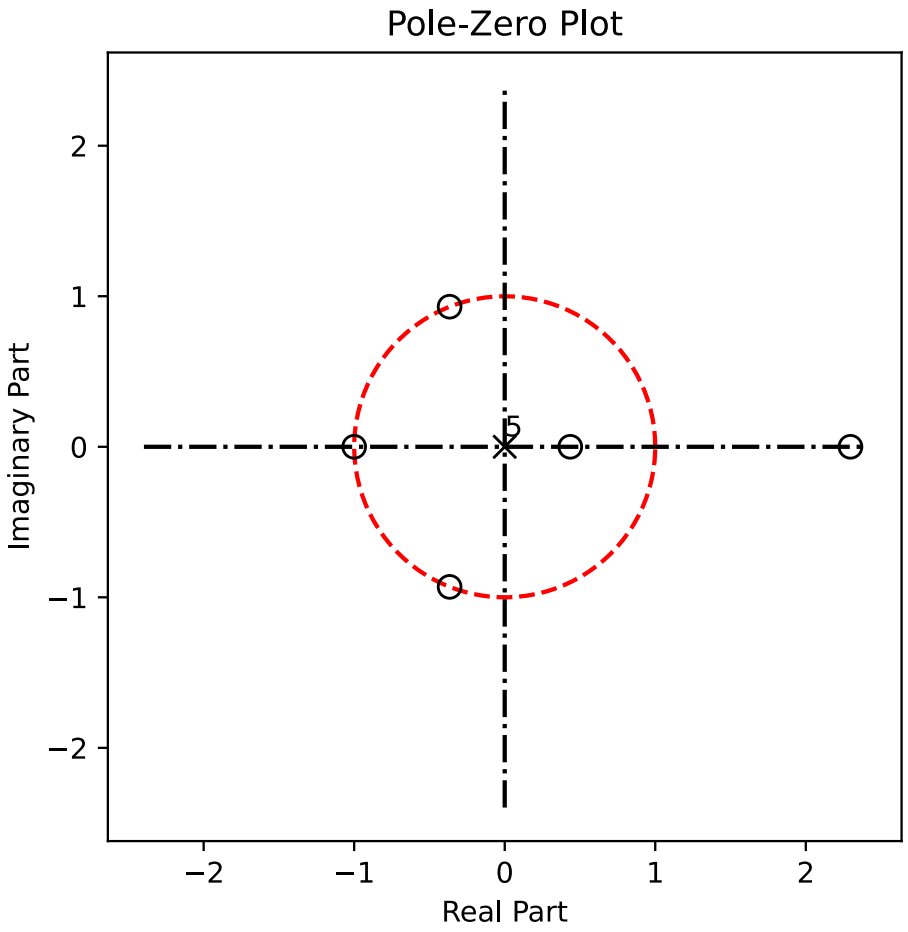
Linear Phase FIR Filters

Pick some simple examples and then a highpass design using `fir_design_helper`.

Simple Examples

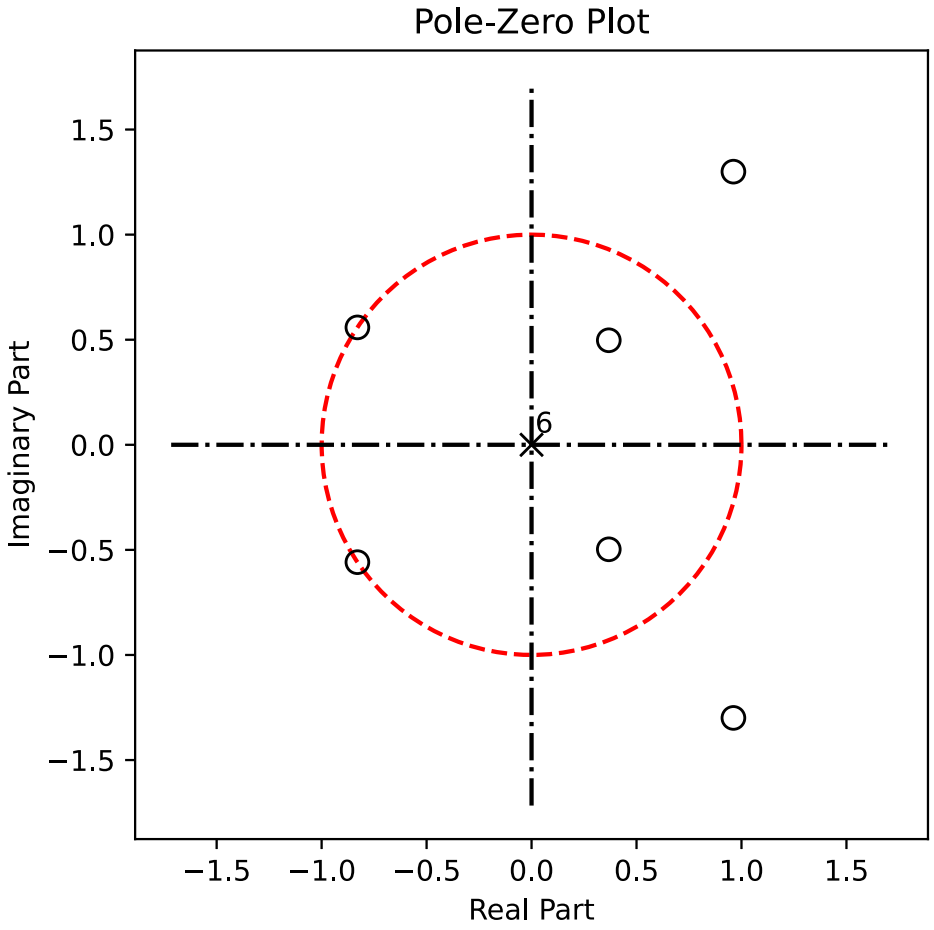
In [19]:
```
ss.zplane([-1,1,2,2,1,-1],1)
```

Out[19]:
```
(5, 0)
```



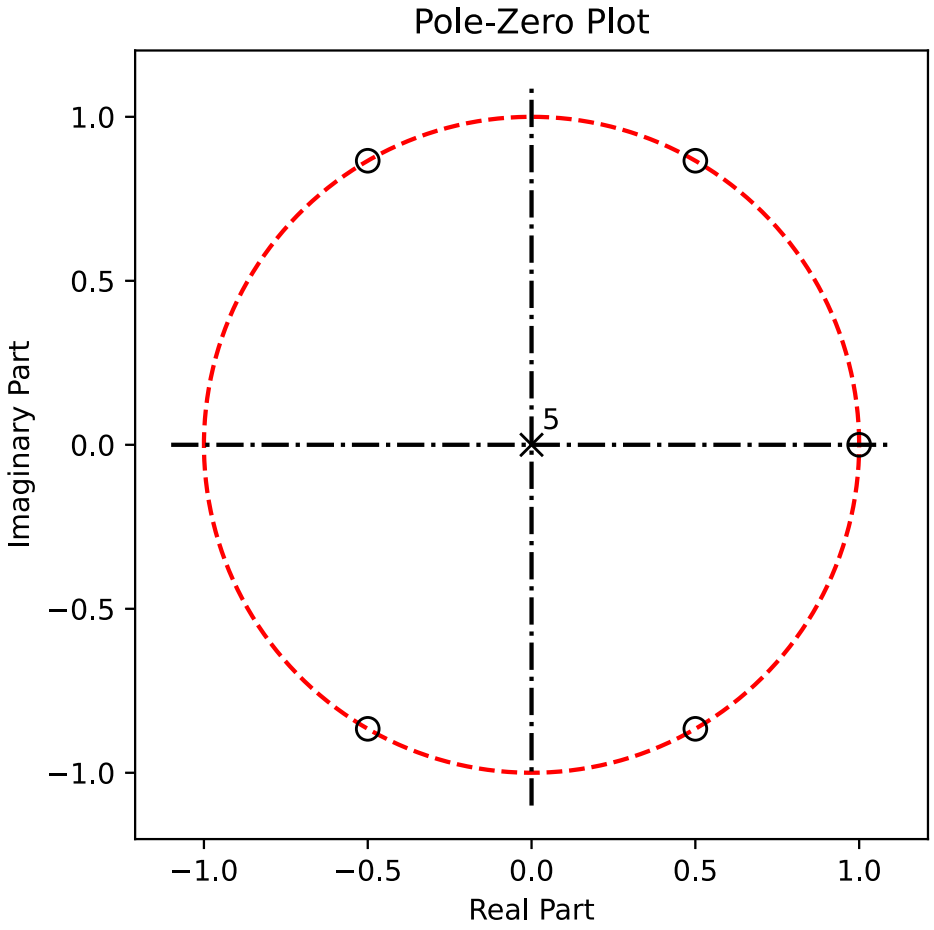
In [20]:
```
ss.zplane([1,-1,1,2,1,-1,1],1)
```

Out[20]: (6, 0)



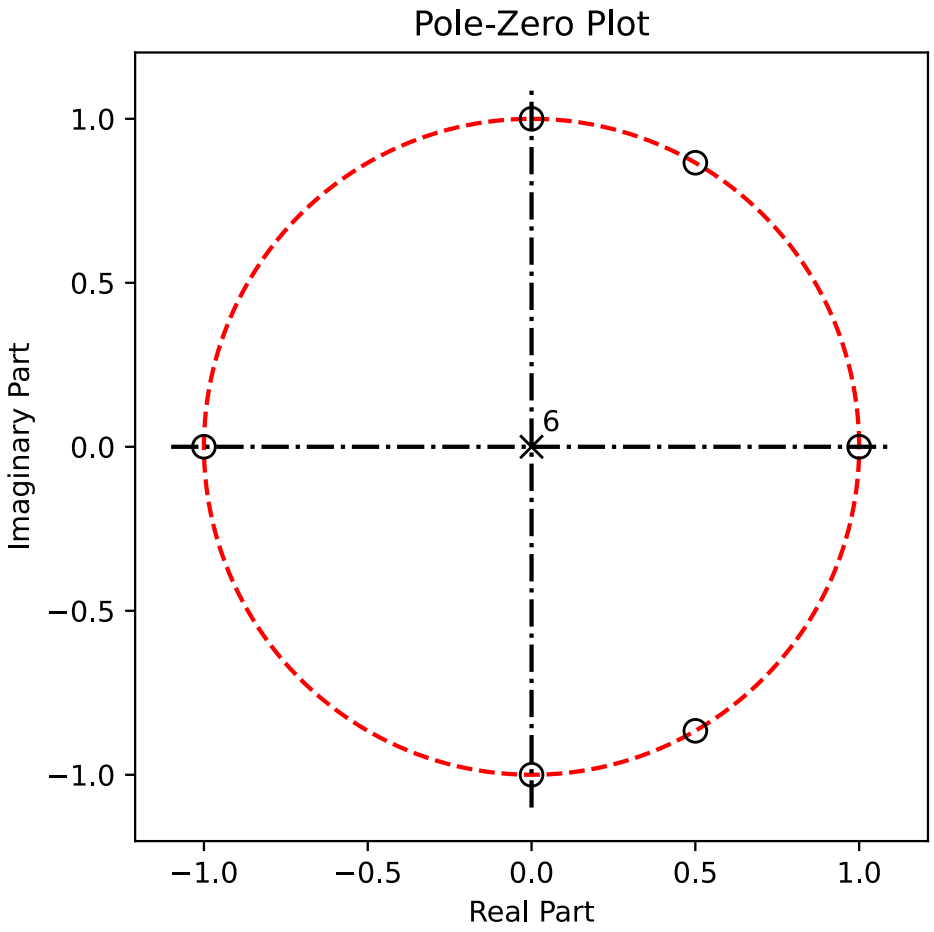
In [21]:
```
ss.zplane([1,-1,1,-1,1,-1],1)
```

Out[21]: (5, 0)



In [22]:
```
ss.zplane([1,-1,1,0,-1,1,-1],1)
```

Out[22]: (6, 0)



Design Using FIR Design Helper

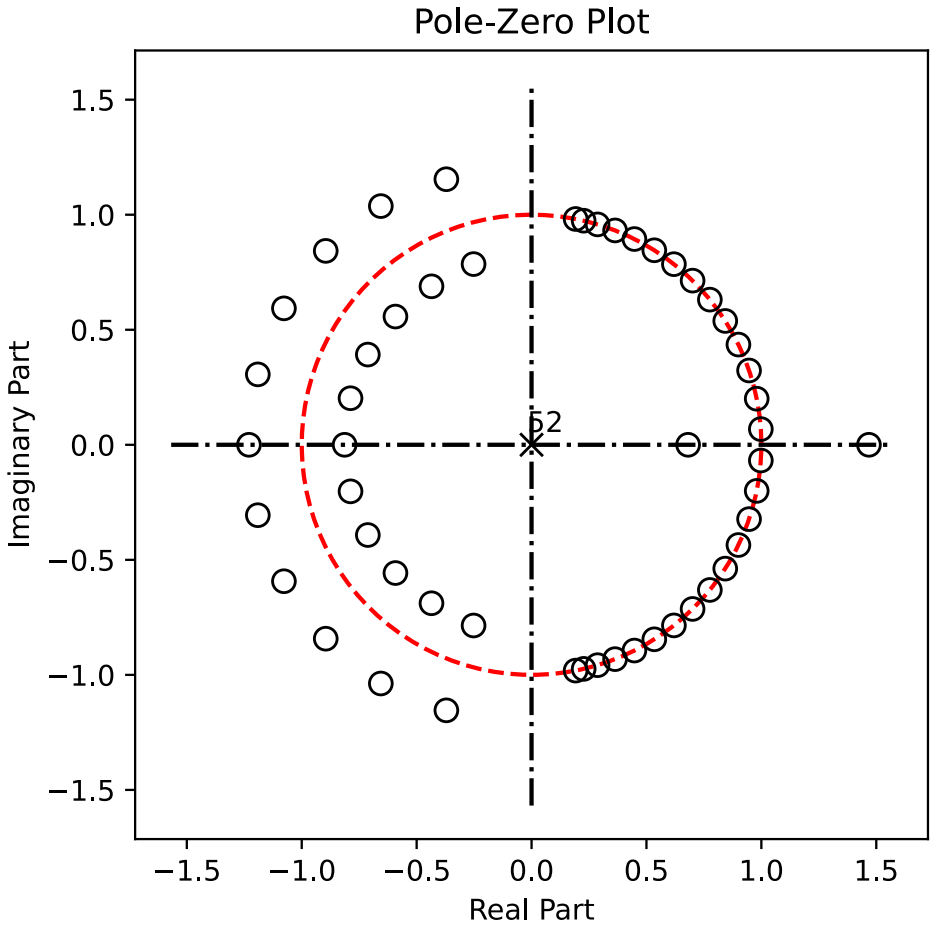
In [ ]:
```
fir_d.
```

In [23]:
```
b_hpf = fir_d.fir_remez_hpf(.22,.28,0.1,80,n_bump=2)
# b_hpf = fir_d.firwin_kaiser_hpf(.22,.28,80,n_bump=2)
len(b_hpf)
```

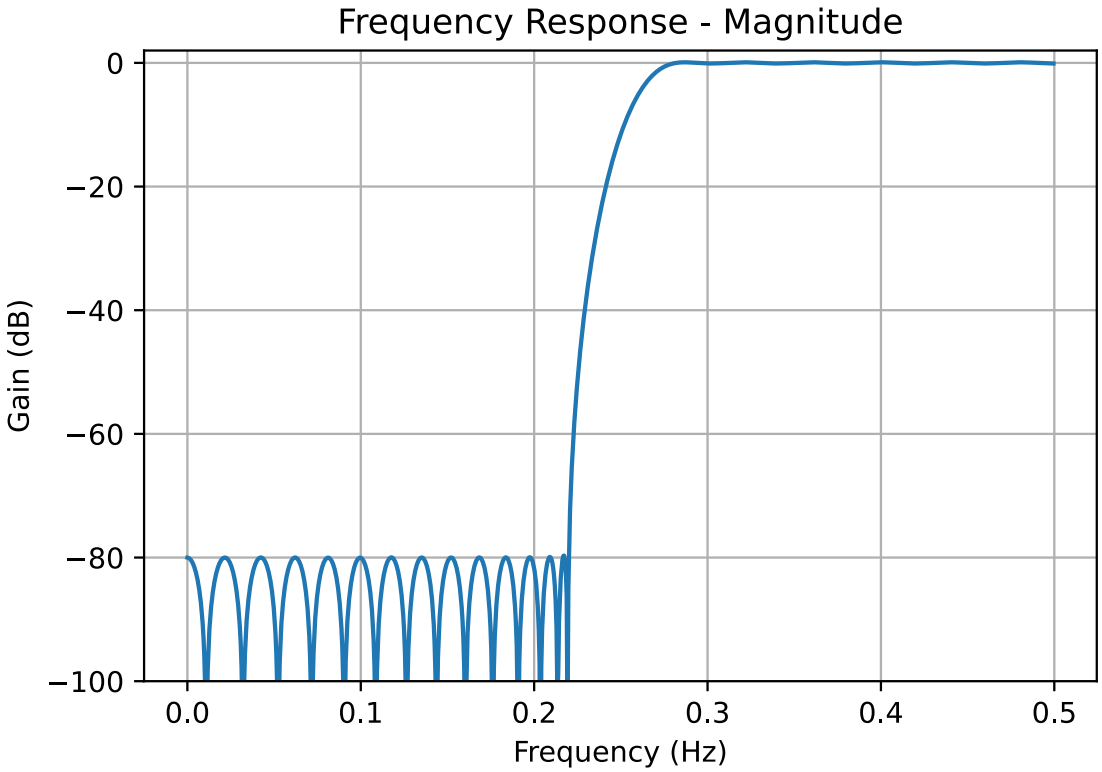
Out[23]: 53

In [24]:
```
ss.zplane(b_hpf,1)
```

Out[24]: (52, 0)



In [25]:
```
fir_d.freqz_resp_list([b_hpf],[1],'dB',1)
ylim([-100,2])
grid()
```

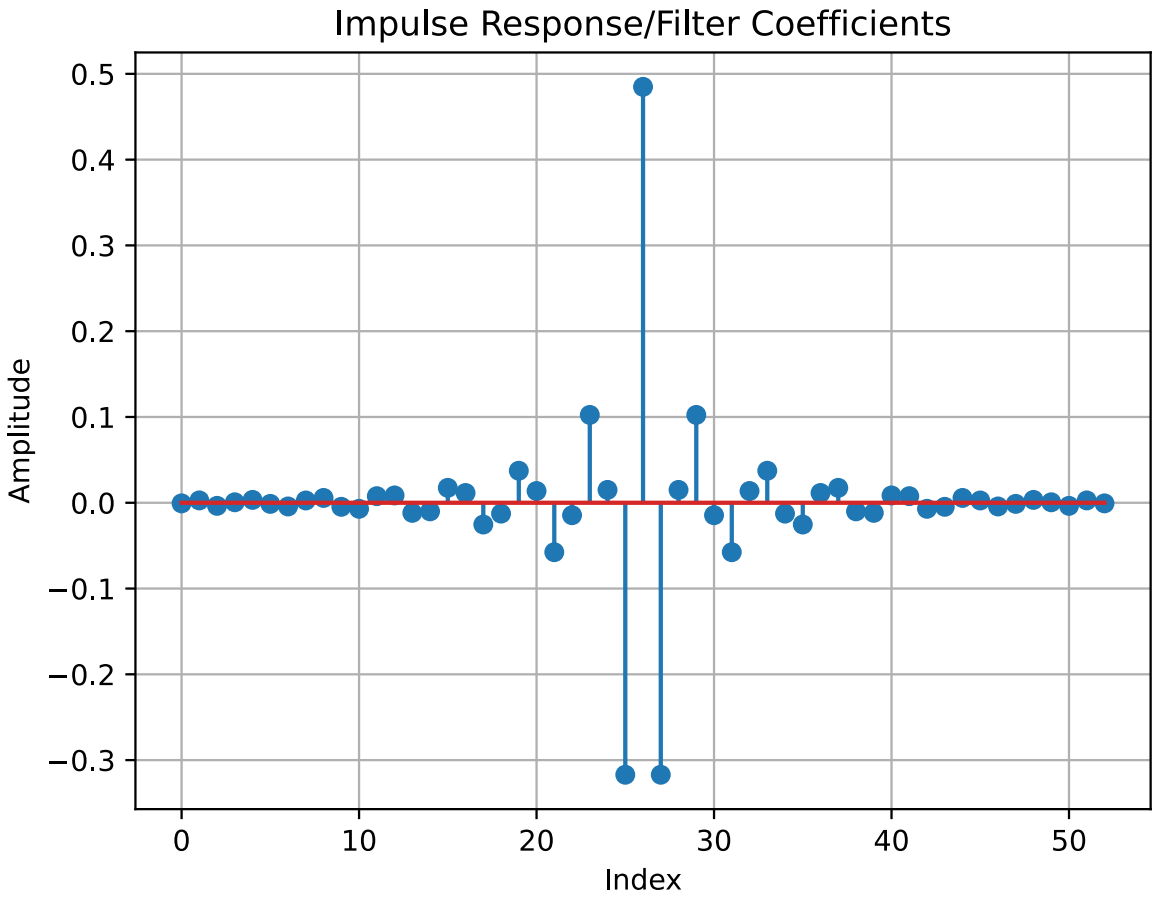


- Check that the DC gain of the filter is indeed down -80 dB

In [27]:
```
20*log10(sum(b_hpf))
```

Out[27]:
```
np.float64(-80.01229092826497)
```

In [28]:
```
stem(b_hpf)
title(r'Impulse Response/Filter Coefficients')
xlabel(r'Index')
ylabel(r'Amplitude')
grid();
```



A Real-Time Lowpass Design

Consider a lowpass filter running in `C` on a microcontroller using $f_s = 48$ ksps.

Importing Network Analyzer Data

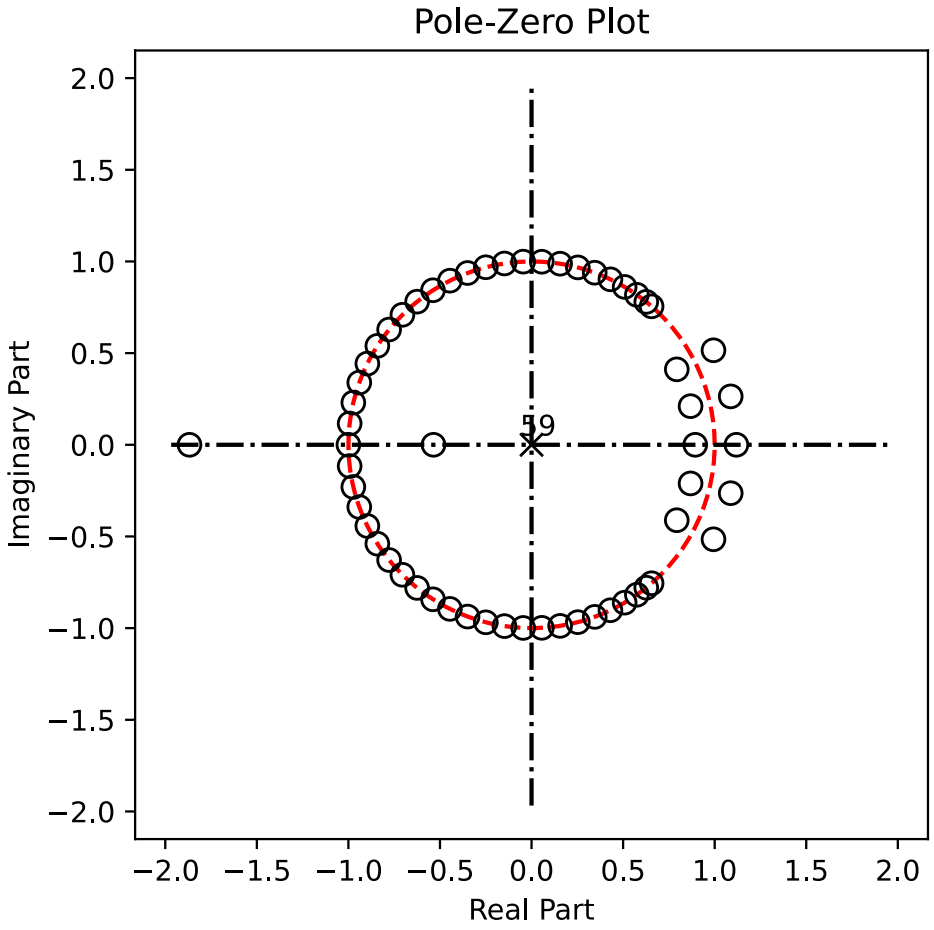
Import the text file using the `numpy` function `loadtxt`. Then gain shift the passband to center on 0 dB. We are not trying to measure insertion loss, just see if the shape matched the theoretical expectations.

In [29]:
```
b_L4_2A = fir_d.fir_remez_lpf(5,6.5,1,60,48,n_bump=4)
print(f'N_taps = {len(b_L4_2A):d}')
f_L4 = arange(0,24,.1)
w_L4,H_L4 = signal.freqz(b_L4_2A,1,2*pi*f_L4/48)
```

```
N_taps = 60
```

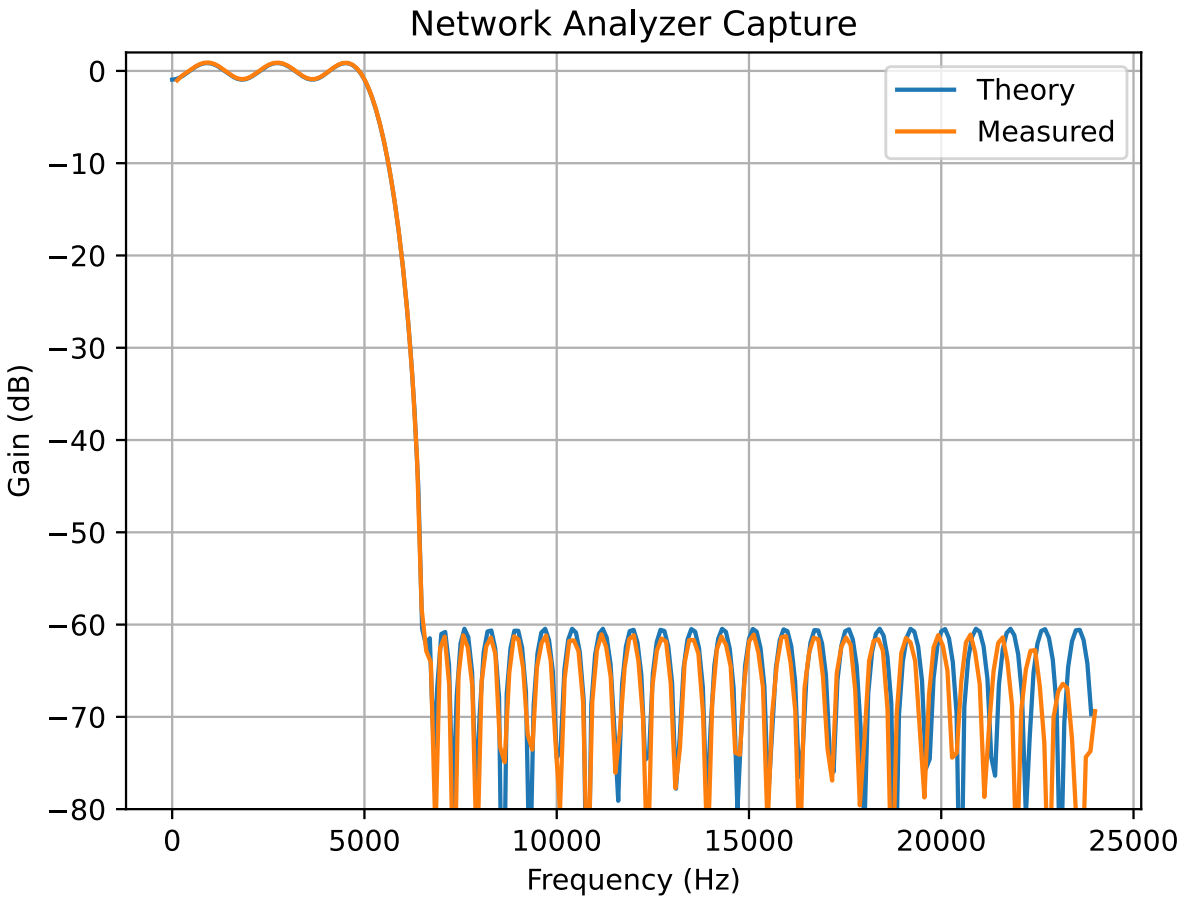
In [30]:
```
ss.zplane(b_L4_2A,1,auto_scale=True,size=1.2)
```

Out[30]: (59, 0)



In [31]:
```
f, H = loadtxt('L4_2A.TXT',skiprows=15,usecols=(0,1),unpack=True)
```

In [32]:
```
# Plot the theory
plot(f_L4*1e3,20*log10(abs(H_L4)),label='Theory')
# Normalize the passband around 0 dB
peak_ripple = 0.9 # trial and error fit
gain_shift = max(H) - peak_ripple
# Plot the data
plot(f[1:],H[1:] - gain_shift,label='Measured')
title(r'Network Analyzer Capture')
ylabel(r'Gain (dB)')
xlabel(r'Frequency (Hz)')
# ylim([-2,2])
ylim([-80,2])
legend()
grid();
```


Network Analyzer Capture
Theory
Measured
Gain (dB)
0
−10
−20
−30
−40
−50
−60
−70
−80
0
5000
10000
15000
20000
25000
Frequency (Hz)