Sampling Theory

```
# %pylab inline
from numpy import *
from matplotlib.pyplot import *
%matplotlib inline
import sk_dsp_comm.sigsys as ss
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
```

Using Python it is easy to investigate, various aspects of sampling theory. In particular sampling theory and quantization can be studied.

Starting Point

Uniform sampling a continuous-time signal $x_c(t)$ using sampling period T

$$x[n] = x_c(nT) \tag{1}$$

Lowpass Sampling Theorem

For signal $x_c(t)$ bandlimited such that

$$\text{In rad/s:} \quad X_c(j\Omega) = \begin{cases} \text{nonzero}, & |\Omega| \leq \Omega_N \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

$$\text{In Hz:} \quad X_c(f) = \begin{cases} \text{nonzero}, & |f| \leq f_N \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

perfect reconstruction of the signal from its samples is possible using a lowpass reconstruction filter, provided you choose $f_s > 2f_N$ Hz.

When sampling is viewed in the frequency domain you get a clear view of how aliasing takes place through the superposition of spectral translates:

$$X_s(j\Omega) = \frac{1}{T}\sum_{k=-\infty}^{\infty} X_c(j\Omega - jk\Omega_s) \tag{4}$$

$$X(e^{j\omega}) = \frac{1}{T}\sum_{k=-\infty}^{\infty} X_c\left(j\frac{\omega}{T} - j\frac{2\pi k}{T}\right) \tag{5}$$

Python can be used to visualize sampling theory in action. For starters consider the principle alias frequency.

Principle Alias Frequency

Given the sampling rate the principle alias frequency is given by

$$f_{\text{prin}} = \left|\text{round}\left(\frac{f_{\text{in}}}{f_s}\right) f_s - f_{\text{in}}\right| \tag{6}$$

For example, given an input frequency at 6 Hz and sampling frequency of 10 Hz use the function `f_prin = ss.prin_alias(f_in,fs)`.

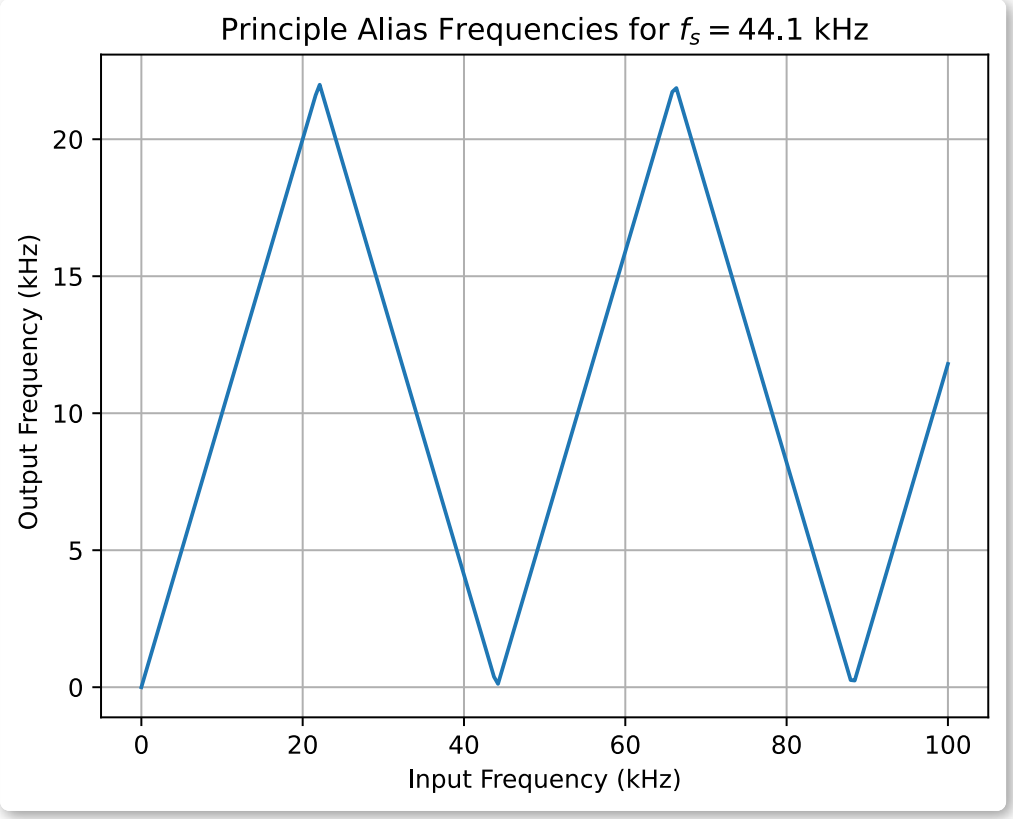
```
ss.prin_alias(6.,10.)
```

```
4.0
```

Example:

Consider a range of f_{in} then you can consider the corresponding principle alias values.

```
fs = 44.1 # kHz
fin = linspace(0,100.,200)
fout = ss.prin_alias(fin,fs)
plot(fin,fout)
xlabel('Input Frequency (kHz)')
ylabel(' Output Frequency (kHz)')
title(r'Principle Alias Frequencies for $f_s = 44.1$ kHz')
grid();
```



Example

Set $f_s = 20$ kHz and consider $x_c(t) = \cos(2\pi f_o t)$ with f_0 equal to 2 kHz, 18 kHz, and 22 kHz. Check the principle alias frequencies:

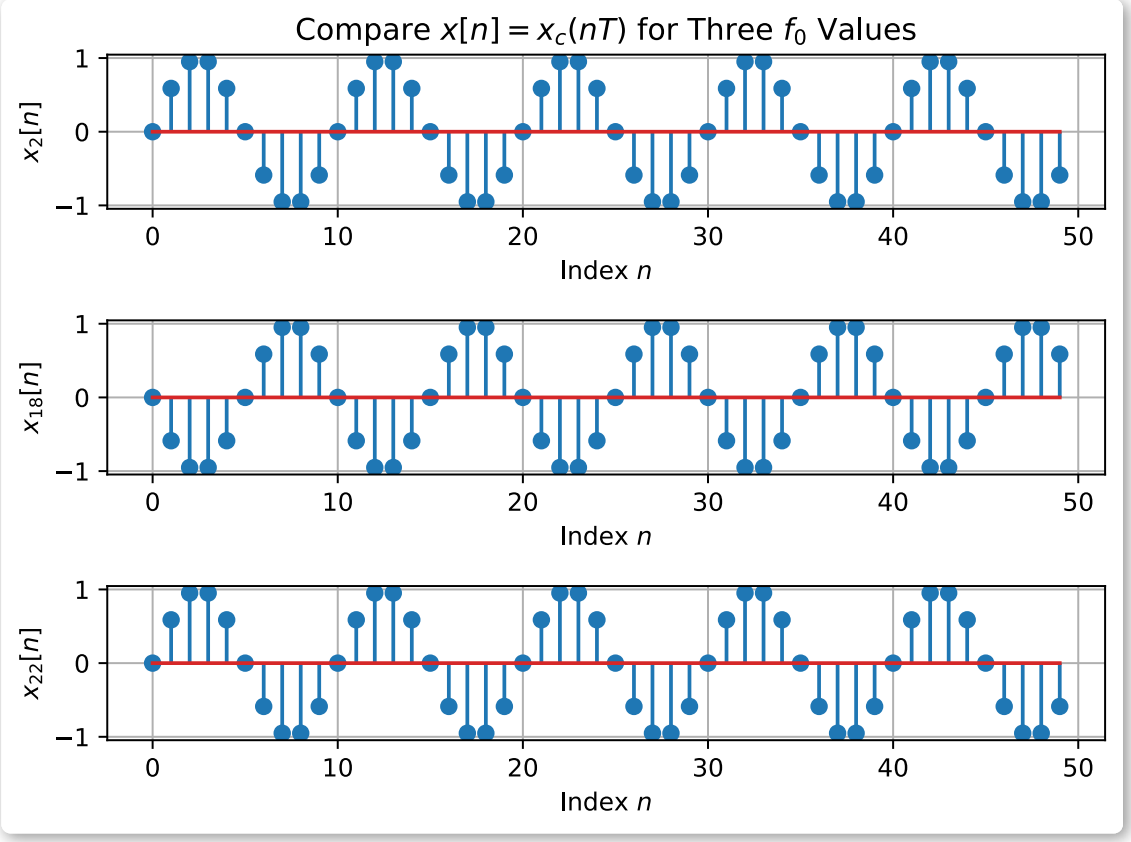
```
ss.prin_alias(array([2,18, 22]),20.0) #need to make sure f_in is an
ndarray
```

```
array([2., 2., 2.])
```

You see that all three f_0 values have the same principle alias. Plot the sampled signals to verify they are the same.

```
# fs = 20 kHz
fs = 20000
n = arange(0,50)
x2 = sin(2*pi*2000/fs*n)
x18 = sin(2*pi*18000/fs*n)
x22 = sin(2*pi*22000/fs*n)
```

```
subplot(311)
stem(n,x2)
xlabel(r'Index $n$')
ylabel(r'$x_2[n]$')
title(r'Compare $x[n]= x_c(nT)$ for Three $f_0$ Values')
grid();
subplot(312)
stem(n,x18)
xlabel(r'Index $n$')
ylabel(r'$x_{18}[n]$')
grid();
subplot(313)
stem(n,x22)
xlabel(r'Index $n$')
ylabel(r'$x_{22}[n]$')
grid();
tight_layout()
```



As expected all three waveforms are identical. See what happens if `cos()` is replaced by `sin()`. Explain what is happening. Also see notes Chapter 4 Example 4.3.

Sampling Bandpass Signals

When the signal being sampled has a bandpass spectrum, that is the spectrum of the analog signal is zero everywhere except over a band of frequencies not centered on zero Hz, i.e.,

$$\text{In rad/s:} \quad X_c(j\Omega) = \begin{cases} \text{nonzero}, & \Omega_l \le |\Omega| \le \Omega_h \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

$$\text{In Hz:} \quad X_c(f) = \begin{cases} \text{nonzero}, & f_l \le |f| \le f_h \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

With the lowpass sampling theorem you have to sample at greater than the highest frequency, i.e., $\Omega_s > 2\Omega_h$ rad/s ($f_s > 2f_H$ Hz). With a bandpass signal it is possible to find lower sampling rates that allow recovery of the original $x_c(t)$ using a bandpass reconstruction filter.

To explore this using Python the following function(s) allow the spectrum of a sampled bandpass signal to be displayed so you can see what works. Yes, there are papers written on the theory of bandpass sampling. Example 4.12, p. 4--45, of the Chapter 4 notes makes use of bandpass sampling to insure that one of the aliased spectral translates lies at $f_s/4$. The minimum requirement is that the sampling rate must be greater than twice the signal bandwidth. For the bandpass definition above, let $B = f_h - f_l$ then $f_s > 2B$. As you consider f_s over the interval $2B < f_s < 2f_h$ not all values work. There are subintervals do work. The Python code allows you to explore the possibilities experimentally.

```
def BP_spec(f,fs,Ntranslates=10,B=(2,3)):
    """
    Sampled bandpass signal spectrum plot
              f = frequency axis created using arange
             fs = sampling frequency
    Ntranslates = the number of spectrum translates
                  to use in the calculation
              B = (B_lower,B_upper)

    Mark Wickert March 2015
    """
    W = B[1]-B[0]
    X = BP_primP(f-B[0],W)+BP_primN(f+B[0],W)
    for k in range(Ntranslates):
        plot(f,BP_primP(f-B[0]-k*fs,W)+BP_primN(f+B[0]-k*fs,W),'b')
        plot(f,BP_primP(f-B[0]+k*fs,W)+BP_primN(f+B[0]+k*fs,W),'b')
    plot(f,X,'r')
    title(r'Sampled Signal Spectrum for $f_s$ = %2.2f' % fs)
```

```
    ylabel(r'Amplitude')
    xlabel(r'Frequency')
    xlim([-2*B[1],2*B[1]])
    grid();

def BP_primP(f,W=1):
    return ss.tri(f-W,W)*ss.rect(f-W/2,W)

def BP_primN(f,W=1):
    return ss.tri(f+W,W)*ss.rect(f+W/2,W)
```

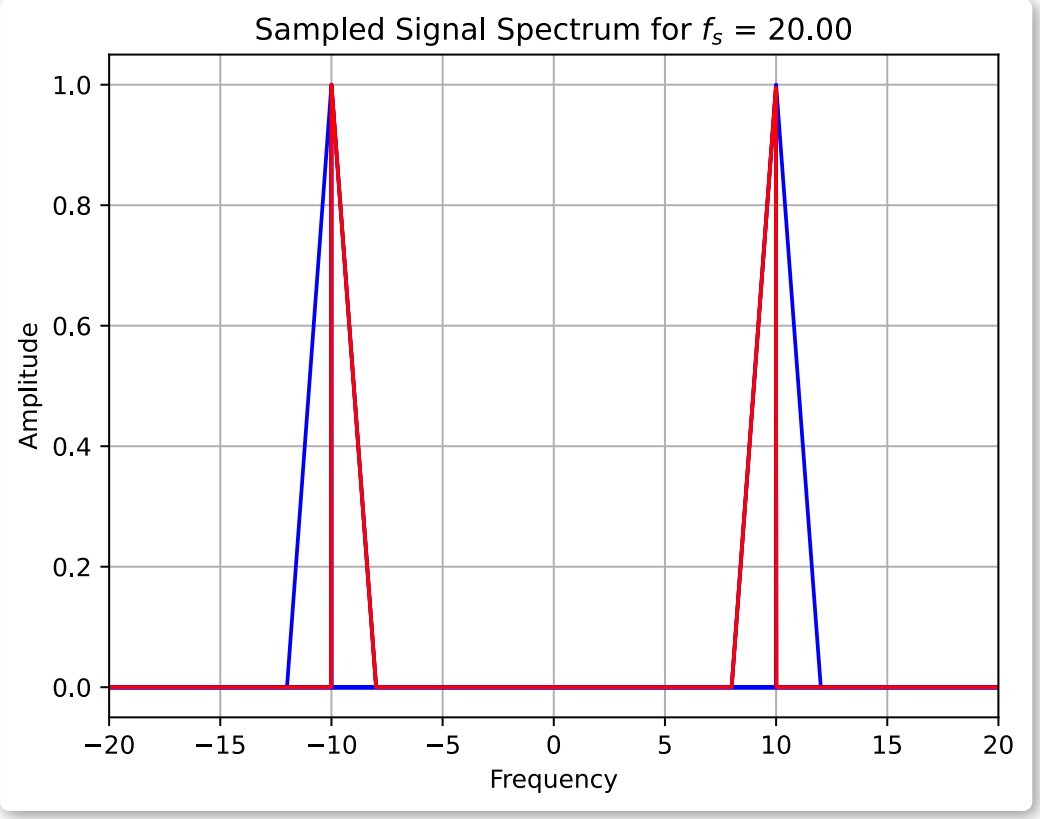
Example:

Consider a bandpass signal nonzero on the interval 8 to 10 Hz. Clearly $f_l = 8$ Hz and $f_h = 10$ Hz. The signal bandwidth is $B = 10 - 8 = 2$ Hz.

The Python function `BP_spec` uses a right triangle spectral shape where you specify the non-zero spectrm as `B = (fl,fh)`, e.g., here `B = (8,10)`. You can then specify the sampling rate `fs` and hown many spectral translates you want the function to plot over the interval that `f` covers by using `f = arange(fmin,fmax,step_size)`.
Note: The number translates needed for an accurate plot increases as the sampling rate becomes small relative to the required lowpass sampling rate. If you are suspicious that a plot is missing some spectral translates, try increasing `Ntranslates`. The *red* spectrum is the original and the *blue* spectra are translates.

Lowpass Sampling

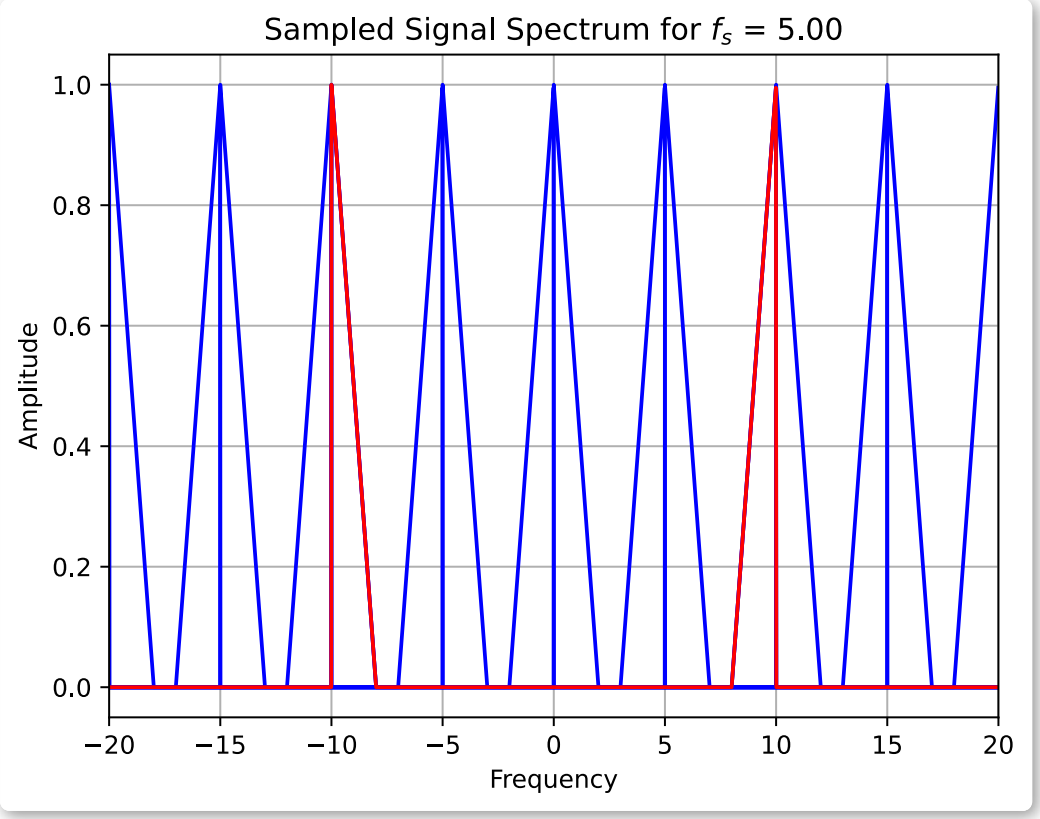
```
f = arange(-20,20,.01)
BP_spec(f,20,8,(8,10))
```



No aliasing of original red signal spectrum when choosing $f_s > 2f_h = 20$ Hz!

Bandpass Sampling or Undersampling

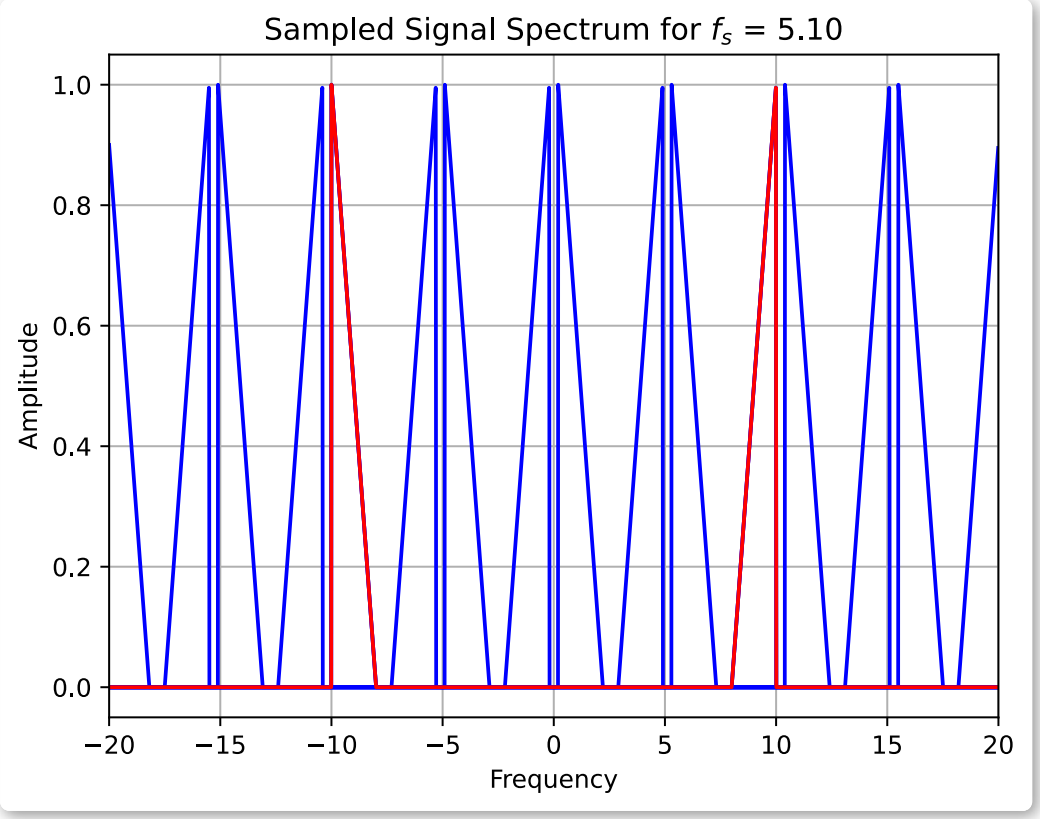
```
f = arange(-20,20,.01)
BP_spec(f,5,8,(8,10))
```



Notice that $f_s = 15 > 2B = 4$ Hz does indeed work.

Try a lower values for f_s:

```
f = arange(-20,20,.01)
BP_spec(f,5.1,8,(8,10))
```



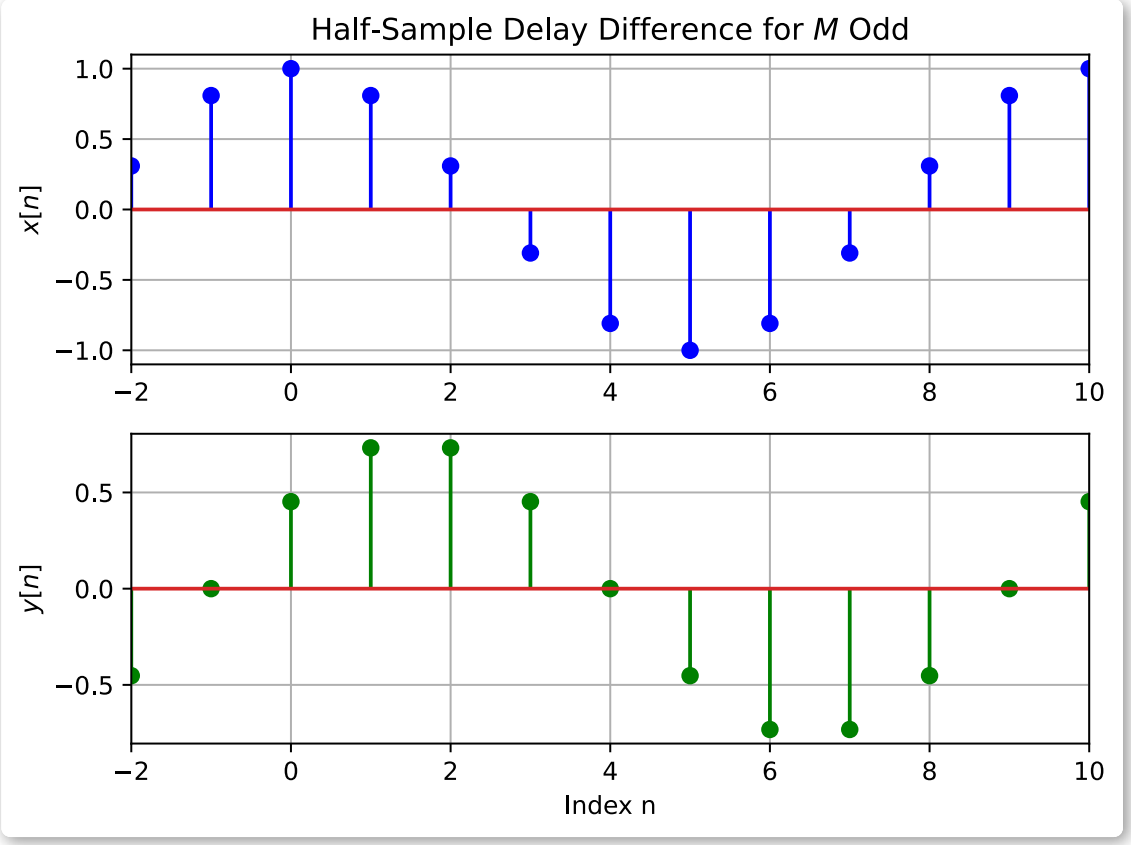
Very nice, choosing $f_s = 5.1$ Hz also works and provides some *guardbands* around the original red spectra for recovery using a bandpass filter!

One-Half Sample Delay

A simple moving average filter can be used to generate a half-sample delay.

```
n = arange(-20,20)
x = cos(2*pi*0.1*n)
M = 3
y = signal.lfilter(ones(M+1)/(M+1),1,x)
subplot(211)
stem(n,x,linefmt='b',markerfmt='bo')
title(r'Half-Sample Delay Difference for $M$ Odd')
ylabel(r'$x[n]$')
xlim([-2,10])
grid();
subplot(212)
stem(n,y,linefmt='g',markerfmt='go')
xlabel('Index n')
ylabel(r'$y[n]$')
xlim([-2,10])
grid();
```

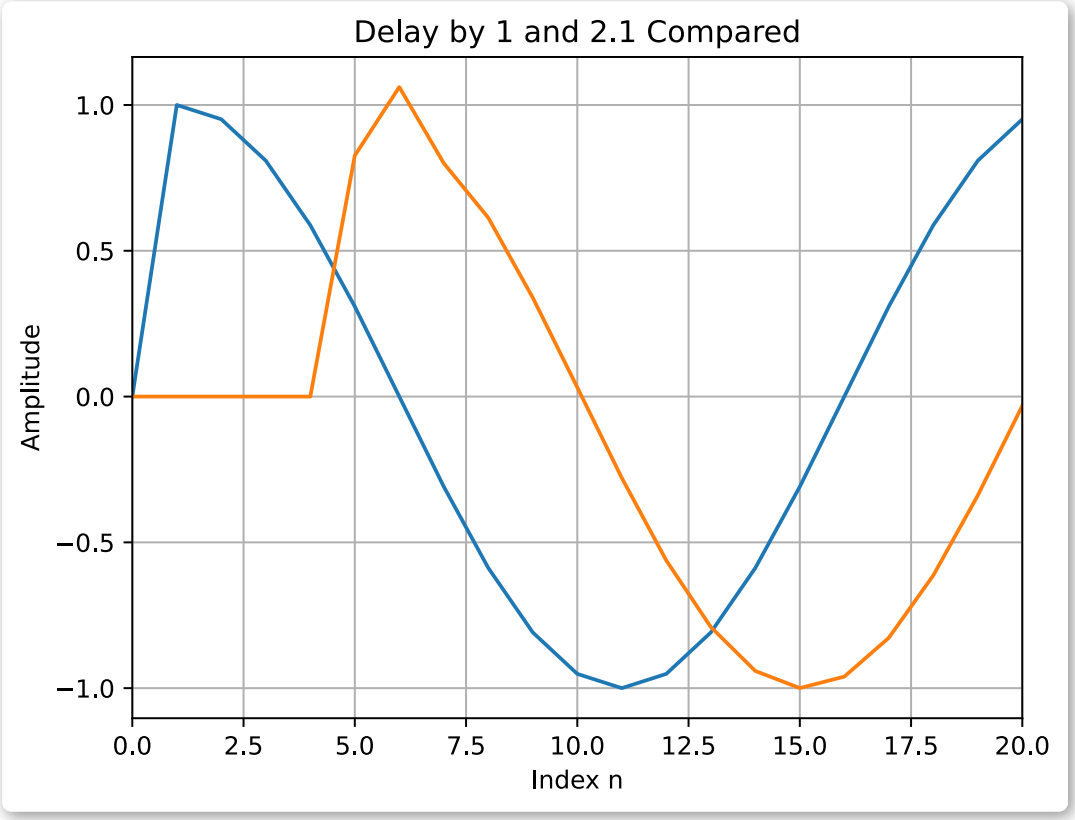
```
tight_layout()
#savefig('ndsp4_ex2_1_toframe.pdf')
```



Fractional Delay

```
import sk_dsp_comm.digitalcom as dc
```

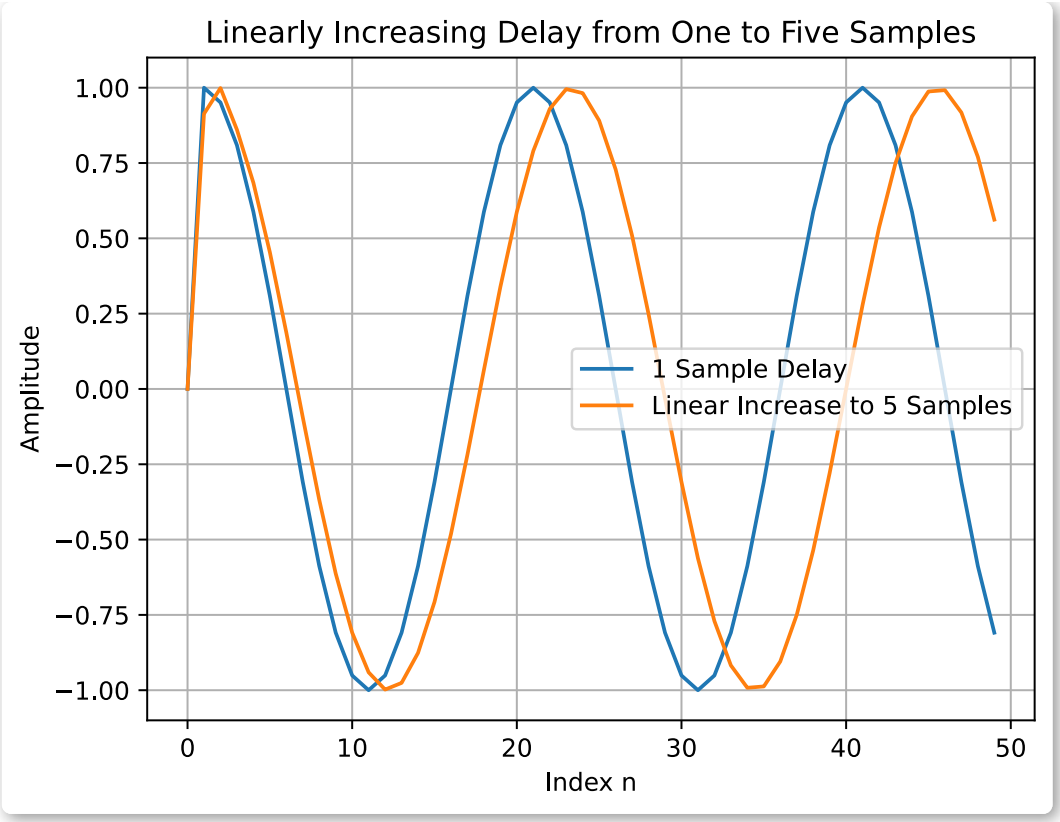
```
n = arange(0,50)
x = cos(2*pi*n/20)
y0 = dc.time_delay(x,1,n=4)
D = 5.1
y1 = dc.time_delay(x,D,n=10)# stem(n,y0)
# stem(n,y1,linefmt='r',markerfmt='rd')
plot(n,y0)
plot(n,y1)
xlabel('Index n')
ylabel(r'Amplitude')
title(r'Delay by 1 and 2.1 Compared')
xlim([0,20])
grid();
```



Introduce a Time Varying Delay

Here we ramp the time delay from one to five samples over the duration of the waveform simulation.

```
n = arange(0,50)
x = cos(2*pi*n/20)
y0 = dc.time_delay(x,1,n=4)
D = arange(len(x))/len(x)*5+1
y1 = dc.time_delay(x,D,n=10)# stem(n,y0)
plot(n,y0)
plot(n,y1)
xlabel('Index n')
ylabel(r'Amplitude')
title(r'Linearly Increasing Delay from One to Five Samples')
legend(('1 Sample Delay','Linear Increase to 5 Samples'))
grid();
```



Multirate Sampling

Start with a signal having a bandlimited spectrum. Here I have chosen a triangle shape, which is a $\text{sinc}^2()$ in the time domain.

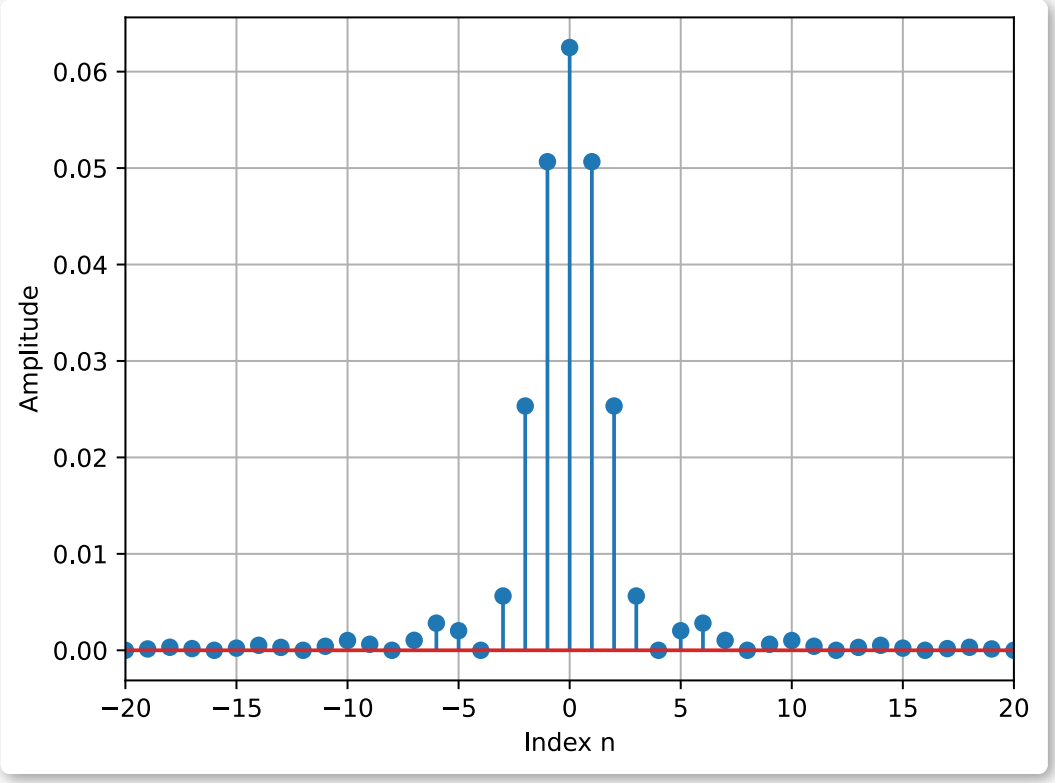
This can be justified from the Fourier transform pairs table entry

$$\frac{\sin((\omega_c/2)n)}{\pi n} \overset{\mathcal{F}}{\longleftrightarrow} \Pi\left(\frac{\omega}{\omega_c}\right) \tag{9}$$

Squaring the signal in the time domain will result in a convolution in the frequency domain. When a rectangle is convolved with a like rectangle it produces a triangle with twice the base width of the width of the rectangle. To properly set the argument of `sinc()` we connect $\text{sinc}(x) = \sin(\pi x)/(\pi x)$ to $\sin(\omega_c/2)n)/(\pi n)$ by writing

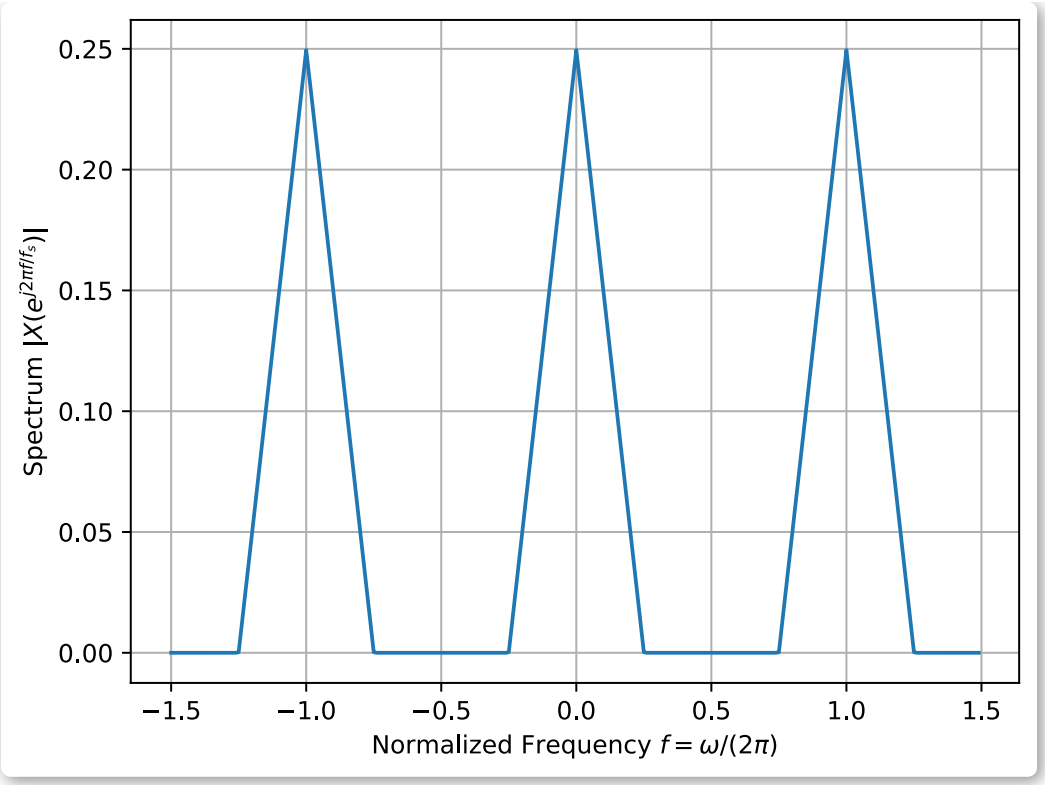
$$\frac{\sin((\omega_c/2)n)}{\pi n} = \frac{\sin((2\pi f_c/2)n)}{\pi n} = f_c \frac{\sin((\pi \cdot f_c n)}{\pi f_c n} = f_c \cdot \text{sinc}(f_c n). \tag{10}$$

```
n = arange(-200,200)
fc = 0.25
x = (fc*sinc(fc*n))**2 # create a signal with a triangle shaped
spectrum
stem(n,x)
grid()
xlim([-20,20])
xlabel('Index n')
ylabel('Amplitude');
```



I now approximate the DTFT using `freqz()` and plot the magnitude spectrum.

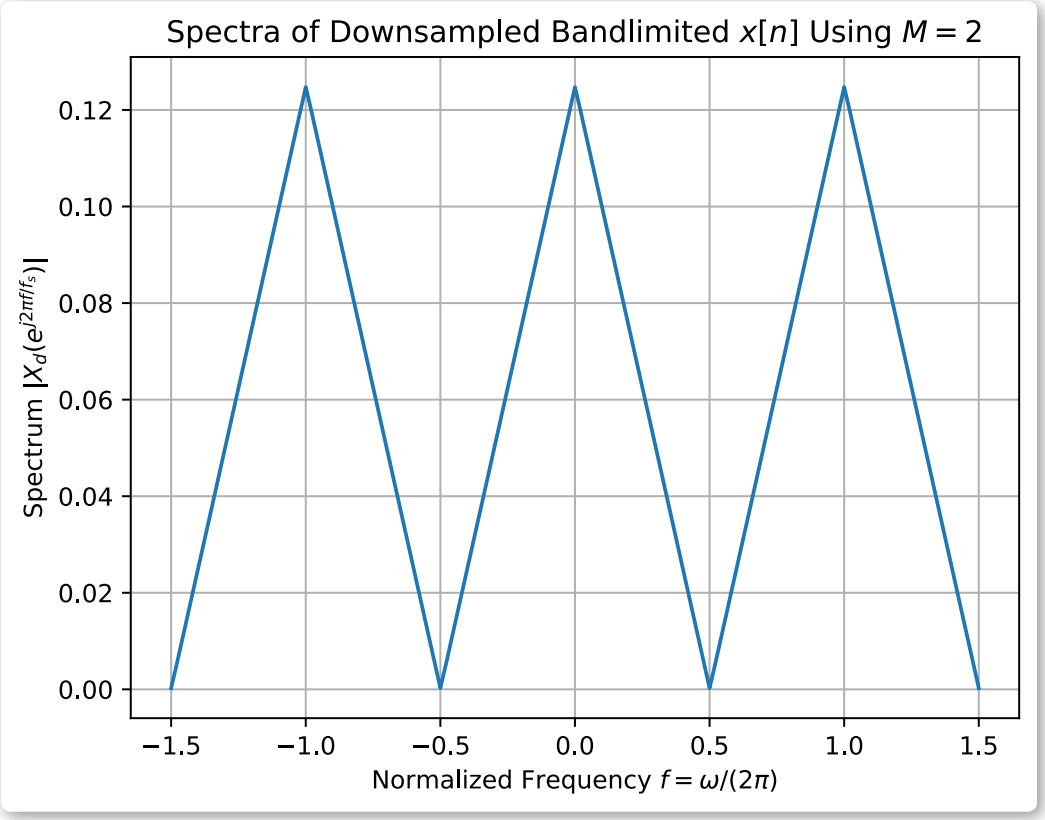
```
f = arange(-1.5,1.5,.01)
w,X = signal.freqz(x,1,2*pi*f)
plot(f,abs(X))
xlabel(r'Normalized Frequency $f = \omega/(2\pi)$')
ylabel(r'Spectrum $|X(e^{j 2\pi f/f_s})$|')
grid();
```



Downsample

Use the `downsample()` function in `sigsys` on the sinc^2() signal:

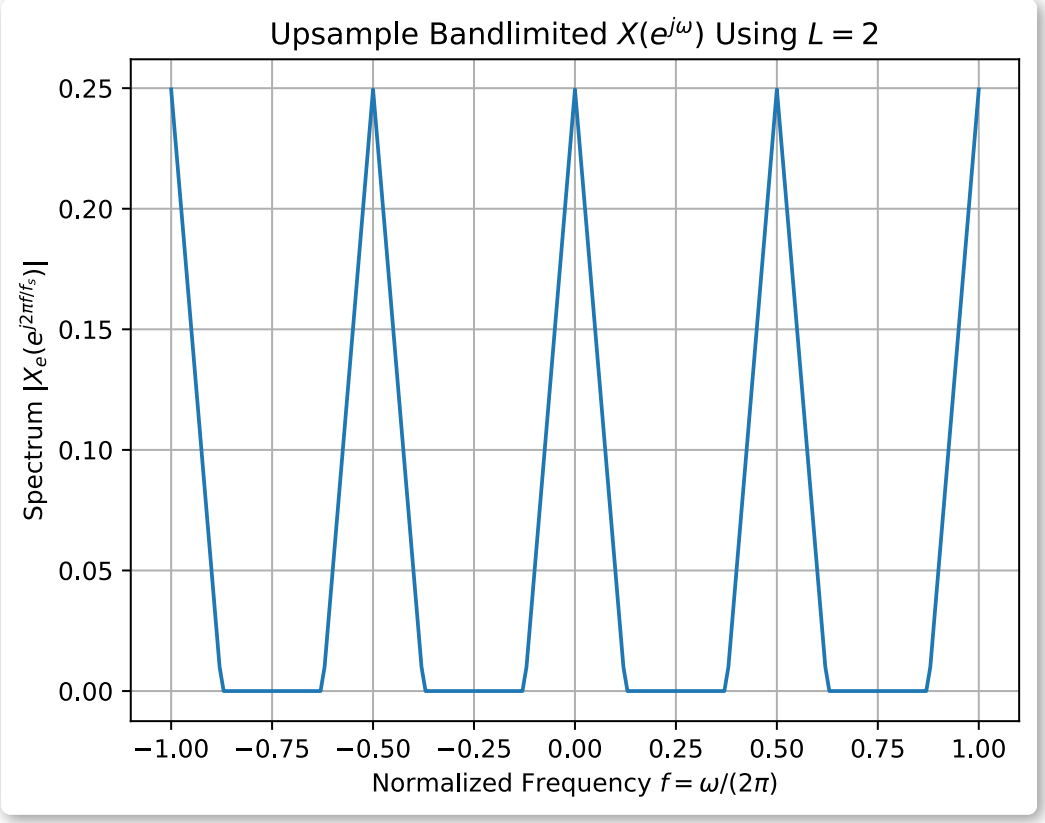
```
# Downsample the triangle spectrum input by 2
n = arange(-200,200)
fc = 0.25
x = (fc*sinc(fc*n))**2 # create a signal with a triangle shaped
spectrum
M = 2
xd = ss.downsample(x,M,0)
# Create a downsampled time (sequence) axis for xd
nd = ss.downsample(n,2)

# Plot the spectrum
f = arange(-1.5,1.5+0.01,.01)
w,Xd = signal.freqz(xd,1,2*pi*f)
plot(f,abs(Xd))
title(r'Spectra of Downsampled Bandlimited $x[n]$ Using $M=$%d' %
(M,))
xlabel(r'Normalized Frequency $f = \omega/(2\pi)$')
ylabel(r'Spectrum $|X_d(e^{j 2\pi f/f_s})$|')
grid()
```



Upsample

Use the upsample function in `sigsys`:

```
# Upsample the triangle spectrum input by 2
n = arange(-200,200)
fc = 0.25
x = (fc*sinc(fc*n))**2 # create a signal with a triangle shaped
spectrum
L = 2
xe = ss.upsample(x,L)

# Plot the spectrum
f = arange(-1.0,1.0+0.01,.01)
w,Xe = signal.freqz(xe,1,2*pi*f)
plot(f,abs(Xe))
xlabel(r'Normalized Frequency $f = \omega/(2\pi)$')
ylabel(r'Spectrum $|X_e(e^{j 2\pi f/f_s})$|')
title(r'Upsample Bandlimited $X(e^{j\omega})$ Using $L=$%d' % (L,))
grid()
```



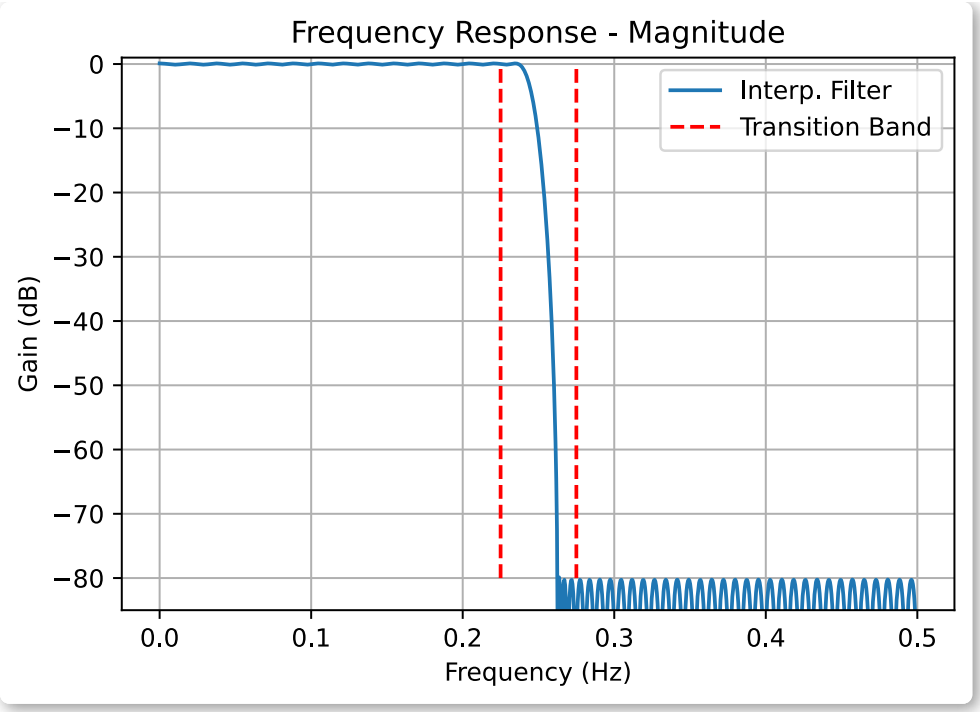
Design an Interpolation Filter to Remove the Image Spectrum

Here we design an equal-ripple lowpass to cutoff at $\omega_c = \pi/L = \pi/2$

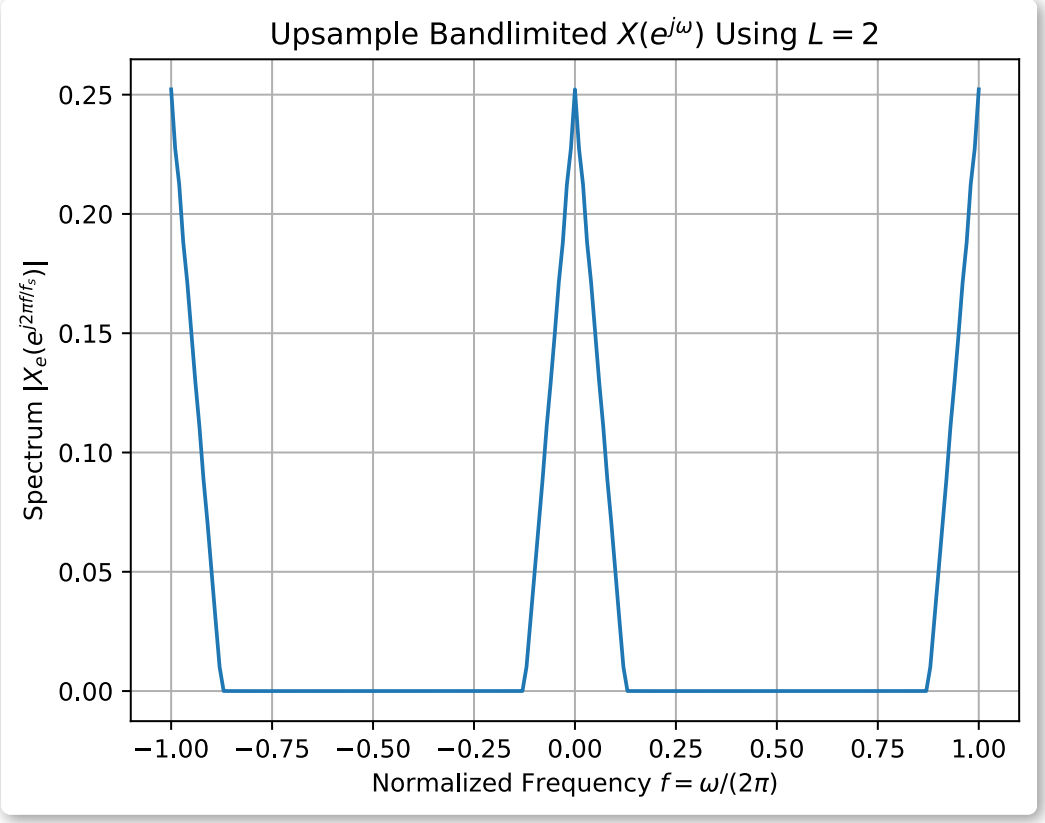
```
import sk_dsp_comm.fir_design_helper as fir_h
```

```
# The filter transition band is centered on f = 1/4
b = fir_h.fir_remez_lpf(0.95/(2*2),1.05/(2*2),0.1,80,n_bump=2)
fir_h.freqz_resp_list([b],[1],'dB')
plot(.9*.25*ones(2),[-80,0],'r--')
plot(1.1*.25*ones(2),[-80,0],'r--')
legend((r'Interp. Filter',r'Transition Band'),loc='best')
ylim([-85,1])
grid();
print('The filter order is large: N = {:3d}'.format(len(b)-1))
```

```
The filter order is large: N = 125
```




```
# Apply the filter and plot the time and frequency domain waveforms
xi = signal.lfilter(b,1,xe)

# Plot the spectrum
f = arange(-1.0,1.0+0.01,.01)
w,Xi = signal.freqz(xi,1,2*pi*f)
plot(f,abs(Xi))
xlabel(r'Normalized Frequency $f = \omega/(2\pi)$')
ylabel(r'Spectrum $|X_e(e^{j 2\pi f/f_s})$|')
title(r'Upsample Bandlimited $X(e^{j\omega})$ Using $L=2$')
grid()
```



- Note in the time domain there is delay due to the lowpass filter
- The image spectrum is gone!
- Making the filter passband ripple 0.1 dB preserves the passband spectrum (lowpass) nicely

---

Project 1 Comments

Problem 1

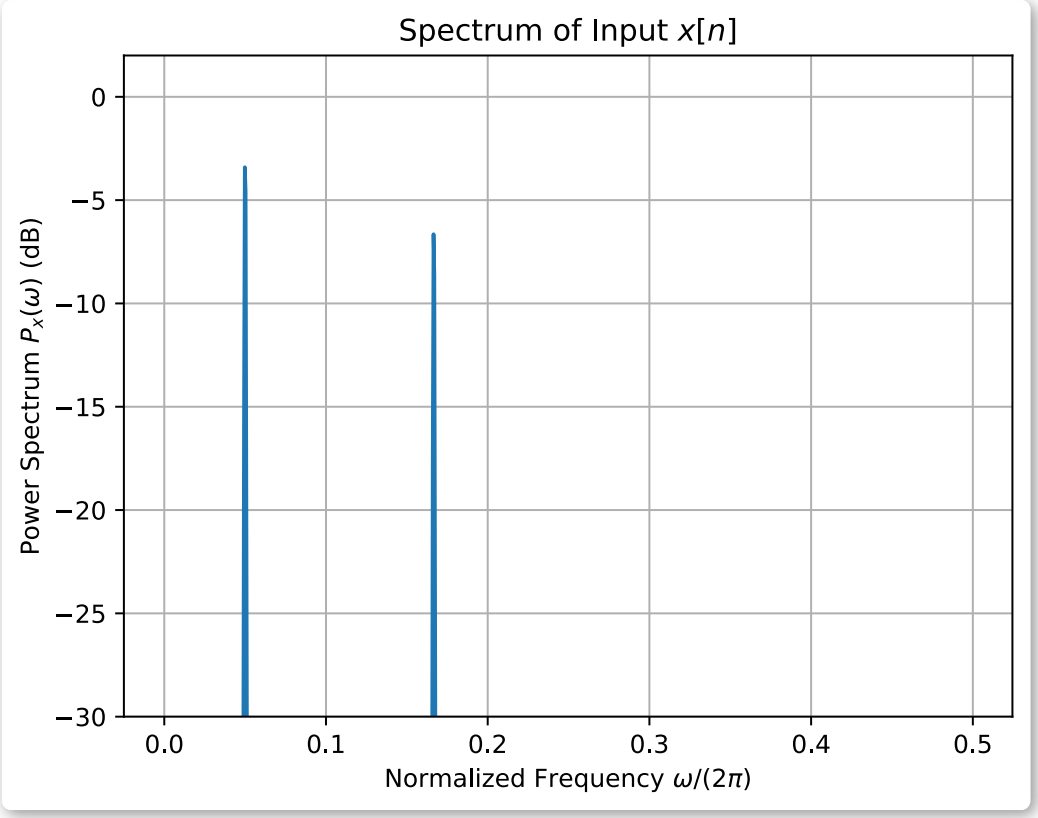
```
def LCCDE(b,a,x,yi=[0],xi=[0]):
    """
    y = LCCDE(b,a,x,yi,xi)
    Causal difference equation solver which
    includes initial conditions/states on
    x[n] and y[n] for n < 0, but consistent
    with the length of the b and a coefficient arrays.

     b = feedforward coefficient array
     a = feedback coefficient array
```

```
     x = input signal array
    yi = output state initial conditions, if needed
    xi = input state initial conditions, if needed
     y = output/returned array corresponding to x

    Examples
    =======
    >> n = arange(20)
    >> x = ss.dimpulse(n)
    >> #x = ss.dstep(n)
    >> y = LCCDE([0,1/3],[1,-5/6,1/6],x,[1],[0])
    >> stem(n,y)
    >> grid();
    >> print(y)

    >> n = arange(20)
    >> x = ss.dimpulse(n)
    >> #x = ss.dstep(n)
    >> y = LCCDE(ones(5)/5,[1,-1],x,[5],[-1,-1])
    >> stem(n,y)
    >> grid();
    >> print(y)
    """
    # Make sure the input list is converted to an ndarray
    a = array(a)
    b = array(b)
    # Initialize the output array
    y = zeros(len(x))
    # Initialize the input/output state arrays to zero
    x_state = zeros(len(b))
    y_state = zeros(len(a)-1)
    # Load the input initial conditions into the
    # input/output state arrays
    if len(a) > 1:
        for k in range(min(len(yi),len(a)-1)):
            y_state[k] = yi[k]
    if len(b) > 1:
        for k in range(min(len(xi),len(b)-1)):
            x_state[k+1] = xi[k]
    # Process sample-by-sample in a loop
    for n, x_n in enumerate(x):
        # Write code here!

    return y
```

Problem 2

```
n = arange(10000)
x = 3*cos(2*pi*n/20) + 2*cos(pi*n/3)
```

```
f,Sx = ss.simple_sa(x,2048,2048,1,window='hann')
plot(f,10*log10(Sx))
title(r'Spectrum of Input $x[n]$')
xlabel(r'Normalized Frequency $\omega/(2\pi)$')
ylim([-30,2])
ylabel(r'Power Spectrum $P_{x}(\omega)$ (dB)');
grid();
```

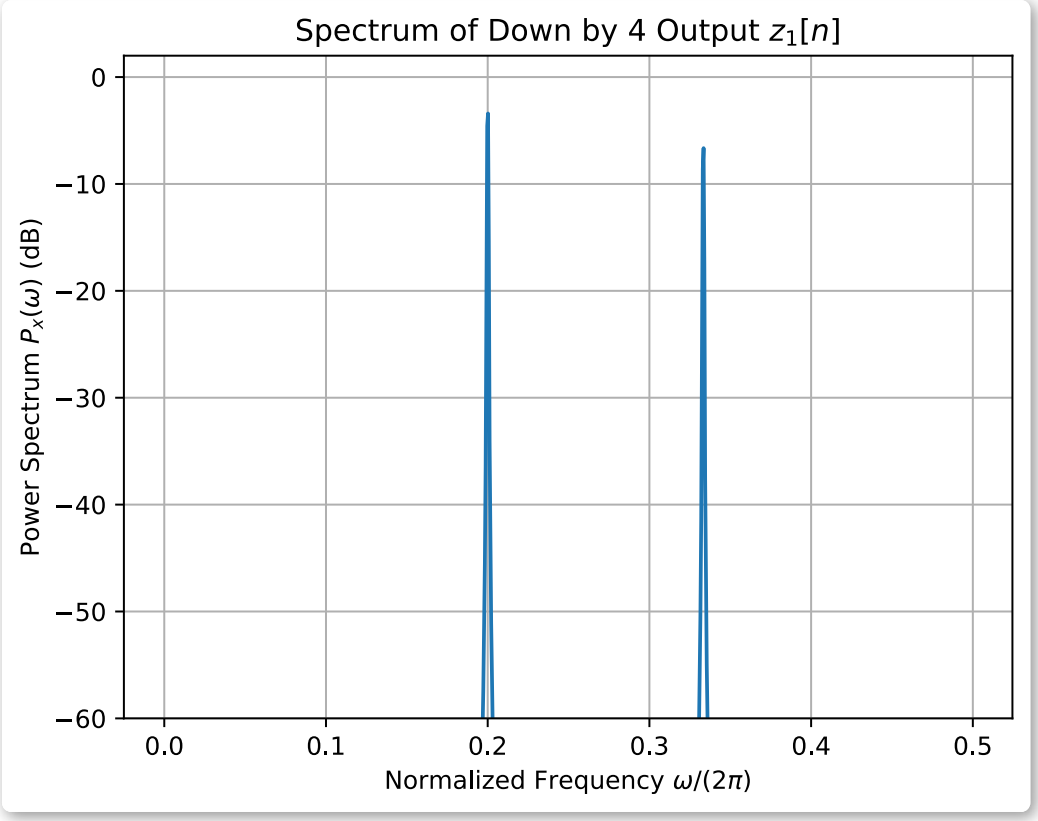


- For details on float formatting see this.

```
idx_peaks,pk_prop = signal.find_peaks(10*log10(Sx),height=-15)
for k in idx_peaks:
    print('Spectrum peak of {:4.2f} Hz at {:8.4f}
dB.'.format(f[k],10*log10(Sx[k])))
```

```
Spectrum peak of 0.05 Hz at  -3.4078 dB.
Spectrum peak of 0.17 Hz at  -6.6511 dB.
```

```
z1 = ss.downsample(x,4)
f,Sz1 = ss.simple_sa(z1,2048,2048,1,window='hann')
plot(f,10*log10(Sz1))
title(r'Spectrum of Down by 4 Output $z_1[n]$')
xlabel(r'Normalized Frequency $\omega/(2\pi)$')
ylim([-60,2])
ylabel(r'Power Spectrum $P_{x}(\omega)$ (dB)');
grid();
```



Probability, Random Variables, & Random Sequences

Familiarity with discrete-time random processes is fundamental to understanding quantization noise as well as a few topics from statistical signal processing. The starting point is:

- A brief review of probability
- A brief review of random variables
- An introduction to discrete-time random (stochastic) processes

Random Variables

Working with random variables is easy when using `pylab`.

- We will use the basic stats capability of `numpy`, but `scipy.stats` is also nearby

```
import scipy.stats as stats
```

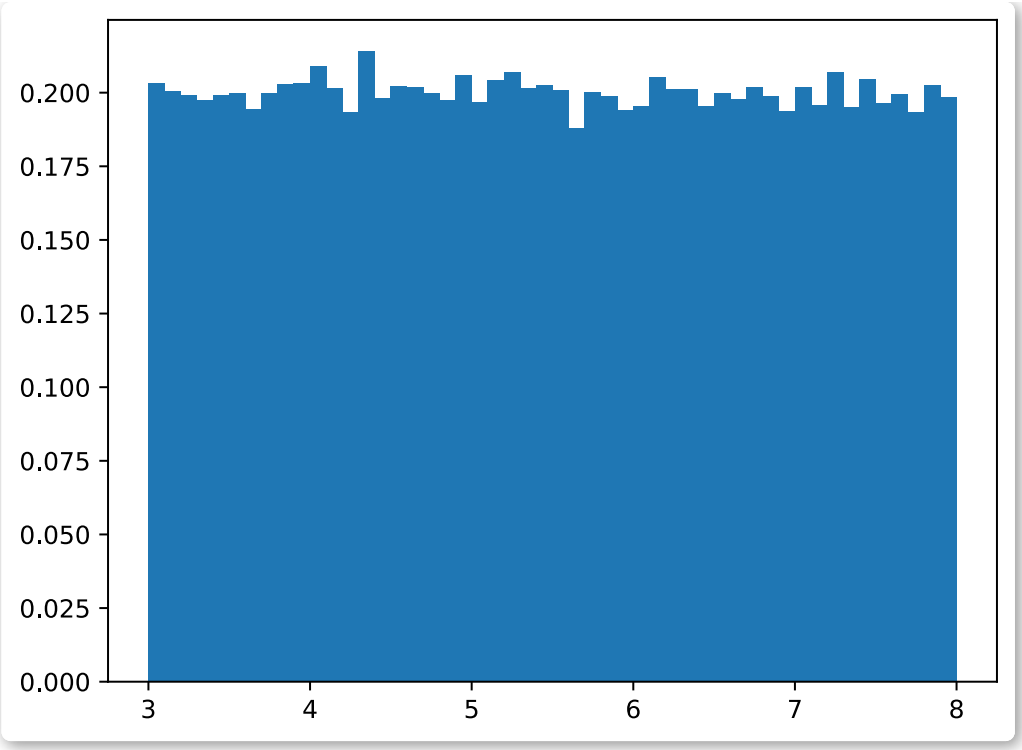
```
stats.
```

- From `numpy` we have already loaded the sub module `numpy.random`

```
random.
```

```
x1 = random.rand(100000)-1/2
x1a = random.rand(100000)-1/2
x3 = random.rand(100000)
x2 = 5*random.rand(100000)+3
```

```
hist(x2,50,density=True);
```



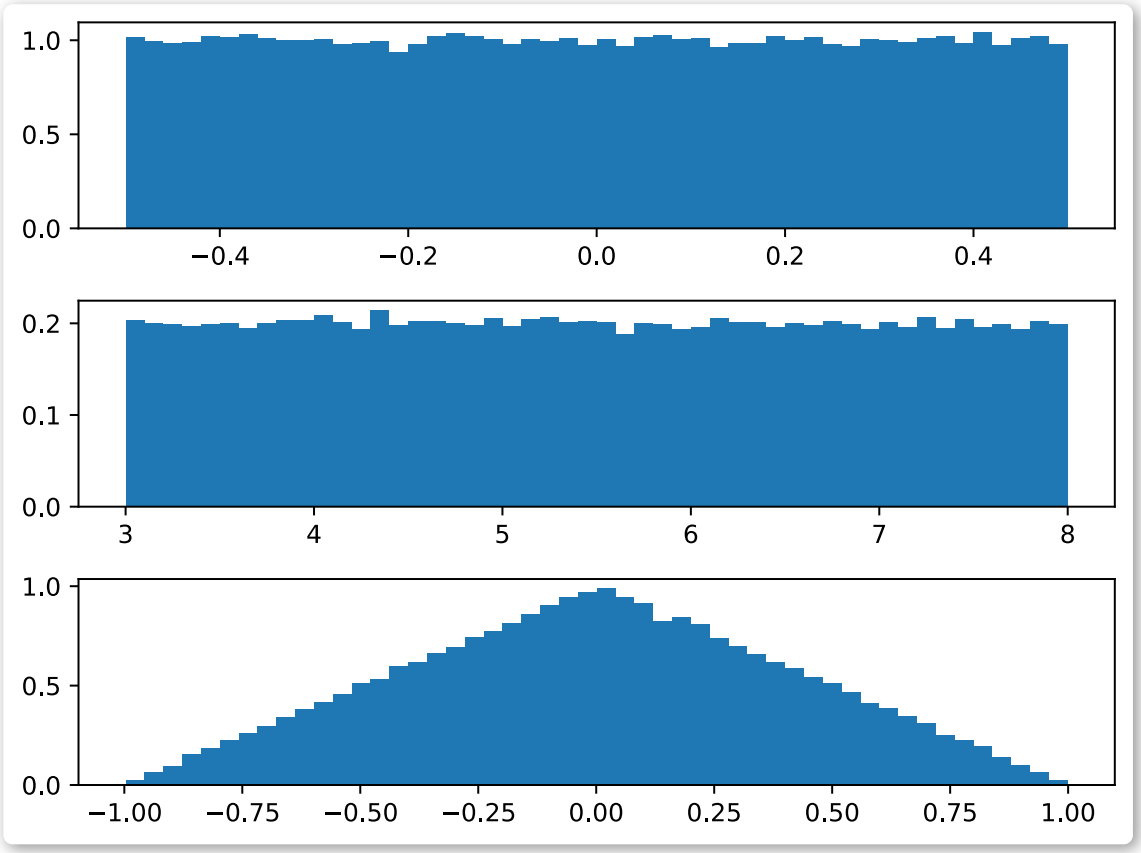

```
(var(x1),1/12)
```

```
(0.08350837306613831, 0.08333333333333333)
```

```
cov(x1,x2)
```

```
array([[8.35092082e-02, 9.46066214e-04],
       [9.46066214e-04, 2.08065681e+00]])
```

```
subplot(311)
hist(x1,50,density=True);
subplot(312)
hist(x2,50,density=True);
subplot(313)
hist(x1+x1a,50,density=True);
tight_layout()
```



Uniform and Gaussian Noise

There are many pdfs available in Python. To some options type:

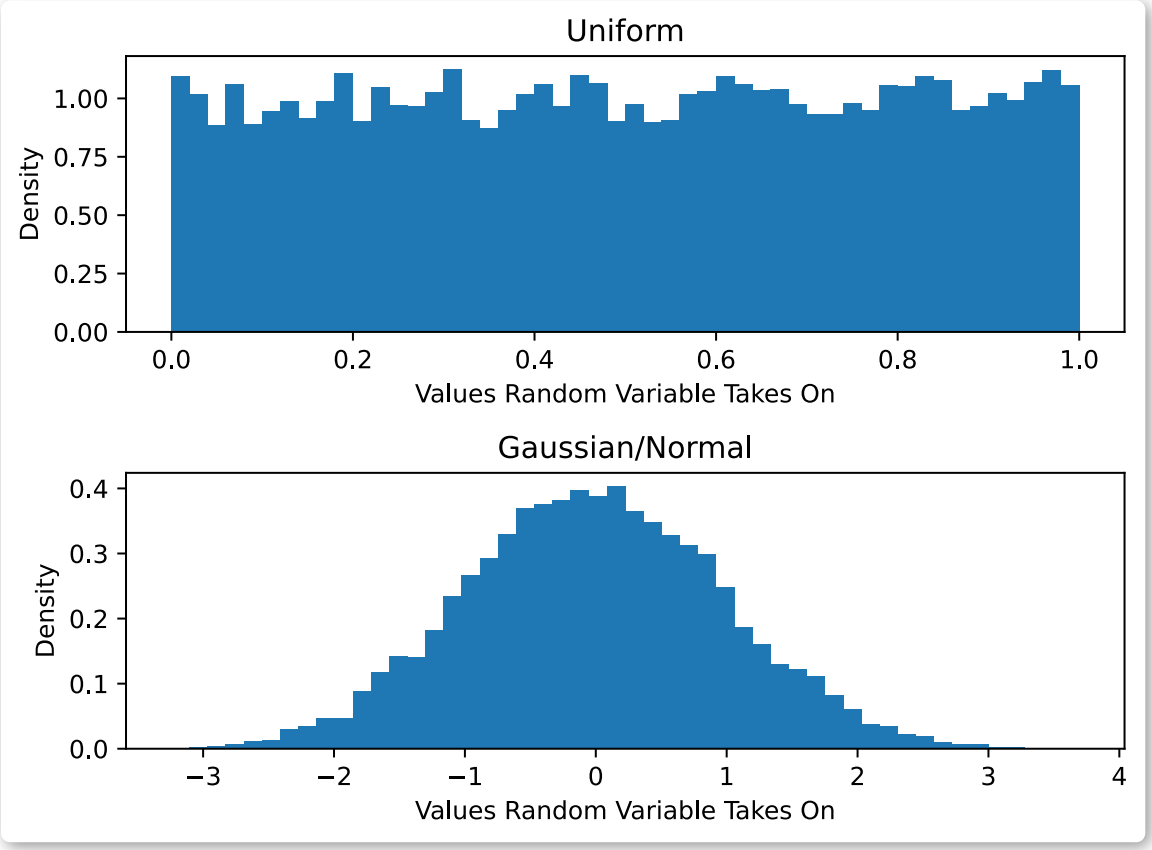
```
random?
```

in a notebook code cell.

```
random?
```

```
w_u = random.rand(10000)
w_g = random.randn(10000)
```

```
subplot(211)
hist(w_u,bins=50,density=True);
title(r'Uniform')
xlabel(r'Values Random Variable Takes On')
ylabel(r'Density')
subplot(212)
hist(w_g,bins=50,density=True);
title('Gaussian/Normal')
xlabel(r'Values Random Variable Takes On')
ylabel(r'Density')
tight_layout()
```



The variance of each of the above types is not the same:

```
(var(w_u),var(w_g))
```

```
(0.08400225634848207, 0.9767823122176156)
```

```
signal.correlate()
```

Auto and Cross Correlation

When dealing with random processes (RPs) you frequently deal with the auto- and cross-correlatiuon between RPs. In `digital_comm.py` there is a function `xcorr()` for estimating the time averaged correlation between two waveforms.

```
"""
Signature: dc.xcorr(x1, x2, Nlags)
Docstring:
r12, k = xcorr(x1,x2,Nlags), r12 and k are ndarray's
Compute the energy normalized cross correlation between the
sequences
x1 and x2. If x1 = x2 the cross correlation is the autocorrelation.
The number of lags sets how many lags to return centered about zero
"""
```

```
import sk_dsp_comm.digitalcom as dc
```

Generate some test vectors of white (uncorrelated zero mean) noise and correlated (colored zero mean) noise.

- $x_1[n]$ is white noise of variance 1
- $x_2[n]$ is white noise of variance 2 (independent, so also uncorrelated) from $x_1[n]$
- $y_1[n]$ is a filtered version of $x_1[n]$ so it is correlated with $x_1[n]$

The filter is defined as
\begin{align}
  H(z) = \frac{1-a}{1 - az^{-1}},\ a = 0.8
\end{align}
Note: The filter DC gain is one since $H(z=1) = H(e^{j0}) = 1$.

```
x1 = random.randn(100000)
x2 = random.randn(100000)
y1 = signal.lfilter([1-.8],[1,-.8],x1)
```

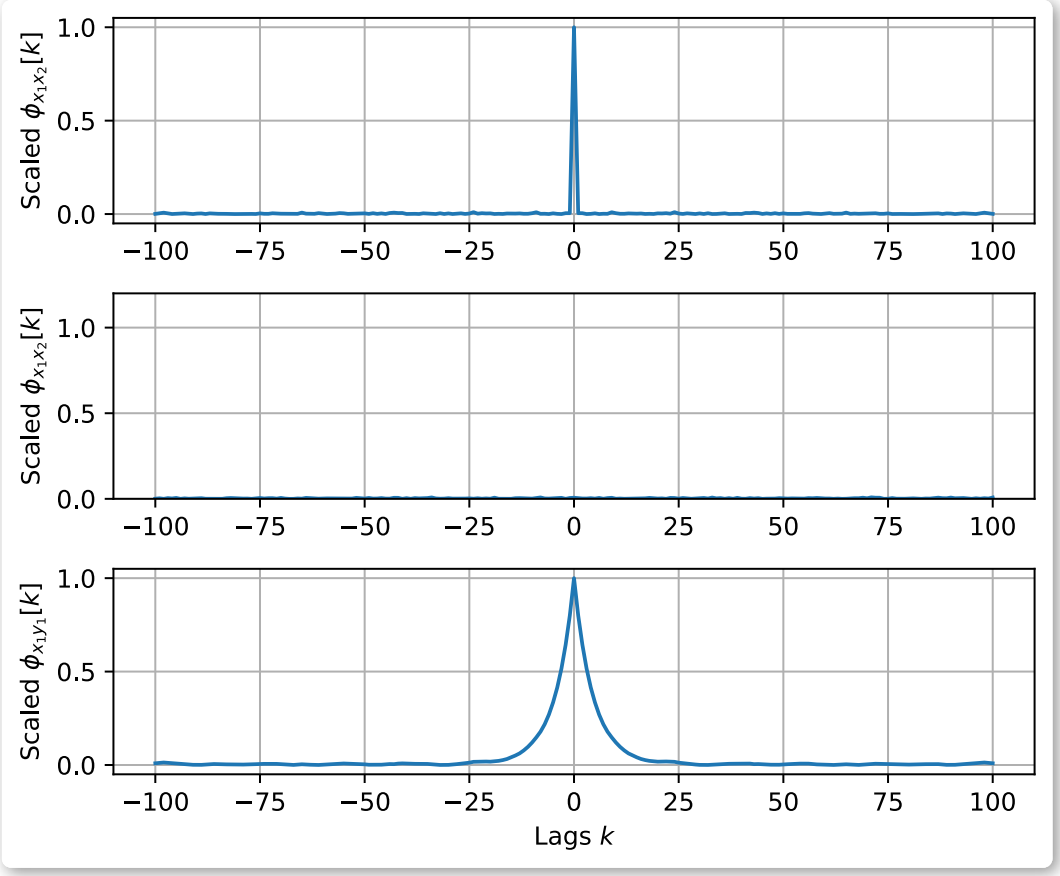
The `cov(x1,x2)` function allows you to obtain the variance and covariance of the inputs as a 2D array or matrix
\begin{align}
  \mathbf{C}*{x_1 x_2} = |begin{bmatrix}*
    |sigma*{x_1}^2 & C*{x_1x_2}* |
    C*{x_1x_2} & \sigma*{x_2}^2*
   |end*{bmatrix}*
|end*{align}*
Note: *$C*{x_1x_2} = E*{(x_1[n]-m*{x_1})(x_2[n]-m*{x_2})}$* $which is equivalent to$ *\gamma_{x_1x_2}[k]* at *k=0$.*

```
print('C_x1x2 = ')
print(cov(x1,x2))
print('C_x1y1 = ')
print(cov(x1,y1))
```

```
C_x1x2 =
[[0.99717337 0.00649304]
 [0.00649304 1.00475595]]
C_x1y1 =
[[0.99717337 0.19970741]
 [0.19970741 0.11109963]]
```

```
figure(figsize=(6,5))
r_x1x1,k = dc.xcorr(x1,x1,100)
subplot(311)
plot(k,abs(r_x1x1))
ylabel(r'Scaled $\phi_{x_1 x_2}[k]$')
grid()
r_x1x2,k = dc.xcorr(x1,x2,100)
subplot(312)
plot(k,abs(r_x1x2))
ylabel(r'Scaled $\phi_{x_1 x_2}[k]$')
ylim([0,1.2])
grid()
r_x1y1,k = dc.xcorr(y1,y1,100)
subplot(313)
plot(k,abs(r_x1y1))
```

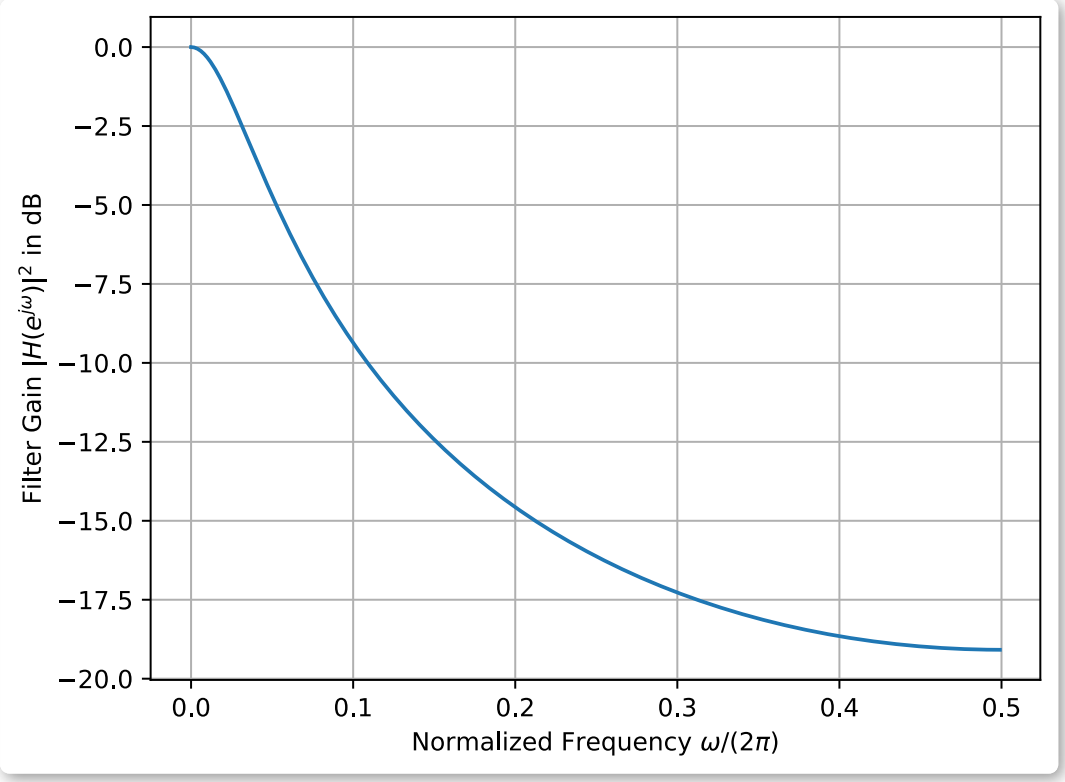
```
ylabel(r'Scaled $\phi_{x_1 y_1}[k]$')
xlabel(r'Lags $k$')
grid()
tight_layout()
```



Note: In the cross correlation of plot 3 above, you expect the plot to represent a scaled version of the time reverse of the system impulse response, $h[-k]$. If the cross-correlation order is changed to `r_x1y1,k = dc.xcorr(y1,x1,100)`, then you expect a scaled estimate of $h[n]$ to be returned. From notes page 4-85 it must be that the function `dc.xcorr()` has the correlation order reversed internally.

The Filter Frequency Response

```
f = arange(0,.5,.001)
w,H = signal.freqz([1-.8],[1,-.8],2*pi*f)
plot(f,20*log10(abs(H)))
xlabel(r'Normalized Frequency $\omega/(2\pi)$')
ylabel(r'Filter Gain $|H(e^{j\omega{}})|^2$ in dB')
grid();
```



Quantization

To model quantization there is a function in `sk_dsp_comm.sigsys` called `simpleQuant()`

```
"""
Signature: ss.simpleQuant(x, Btot, Xmax, Limit)
Docstring:
A simple rounding quantizer for bipolar signals having Btot = B + 1
bits.

This function models a quantizer that employs Btot bits that has
one of
three selectable limiting types: saturation, overflow, and none.
The quantizer is bipolar and implements rounding.

Parameters
----------
x : input signal ndarray to be quantized
Btot : total number of bits in the quantizer, e.g. 16
Xmax : quantizer full-scale dynamic range is [-Xmax, Xmax]
Limit = Limiting of the form 'sat', 'over', 'none'

Returns
```

```
-------
xq : quantized output ndarray

Notes
-----
The quantization can be formed as e = xq - x

Examples
--------
>>> n = arange(0,10000)
>>> x = cos(2*pi*0.211*n)
>>> y = sinusoidAWGN(x,90)
>>> yq = simpleQuant(y,12,1,'sat')
>>> psd(y,2**10,Fs=1);
>>> psd(yq,2**10,Fs=1)
"""
```

Using this function we can visualize the quantization noise on signals in the time domain and the frequency domain.

In the case of the additive noise model for quantization error you have

$$\hat{x}[n] = x[n] + e[n] \tag{11}$$

so the additive error with *rounding* is $e[n]$, where

$$e[n] \simeq U\left[\frac{-\Delta}{2}, \frac{\Delta}{2}\right] \tag{12}$$

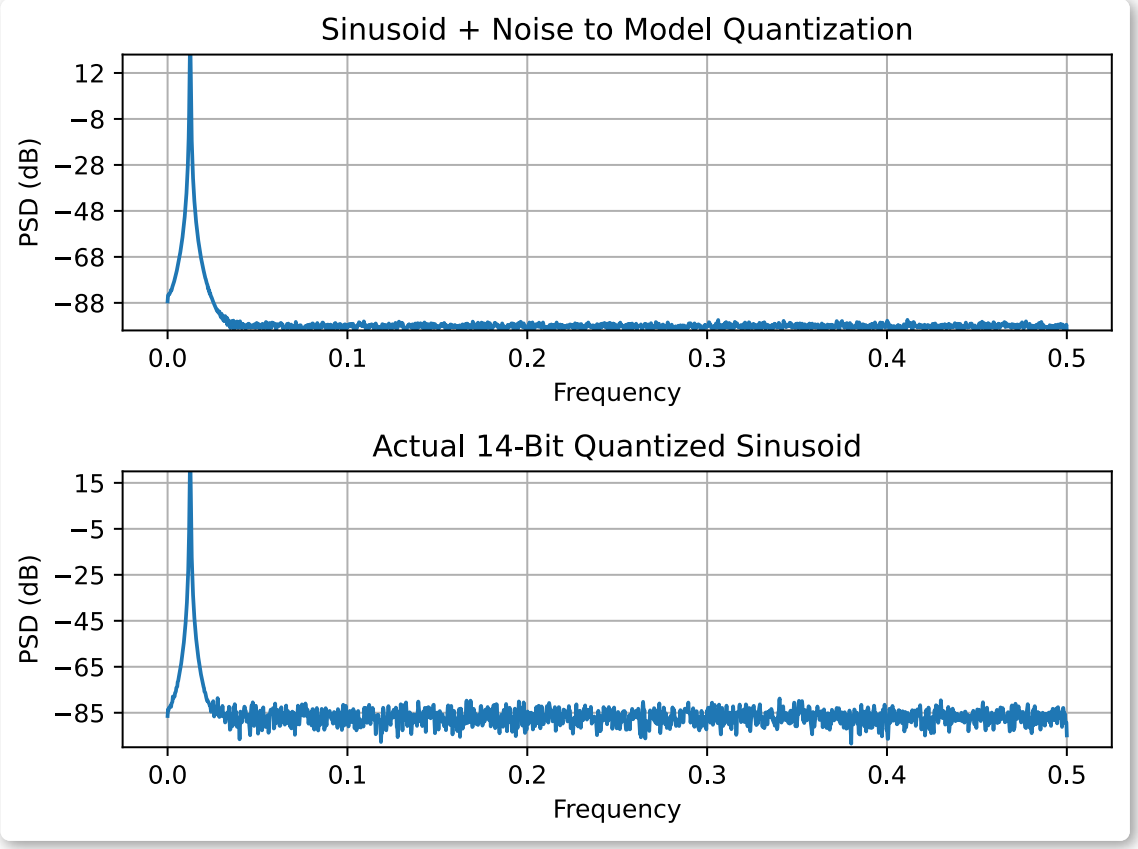
and $\Delta = X_m/2^B$. The quantizer dynamic range is $\pm X_m$ and the total number of bits is $B+1$ with B being effectively the magnitude bits and one sign bit. The assumption is $e[n]$ is a white RP with variance $\sigma_e^2 = \Delta^2/12$ and autocorrelation sequence $\phi_{ee}[k] = \gamma_{ee}[k] = \sigma_e^2\delta[k]$.

Suppose that $X_m = 1$ and $B+1 = 14$ bits. Let the test signal be a sinusoid with amplitude $A < 0$ to avoid saturation in the quantizer. The theoretical SNR_Q is given by
\begin{align}
  \text{SNR}_Q = \frac{\sigma_X^2}{\sigma_e^2}
\end{align}
Here $\sigma_x^2 = A^2/2$ and $\sigma_e^2 = (X_m/2^B)^2/12$.

```
A = 0.5
B = 16-1
Xm = 1
SNR_Q_dB = 10*log10(A**2/2/((Xm/2**B)**2/12))
print('SNR_Q = %6.2f dB' % SNR_Q_dB)
```

```
SNR_Q =  92.07 dB
```

```
fs = 96000 # Sampling frequency (Hz)
f0 = 1205 # Sinusoid frequency (Hz)
A = 0.5 # Sinusoid amplitude
n = arange(0,100000)
x = A*cos(2*pi*f0/fs*n)
y = ss.sinusoid_awgn(x,SNR_Q_dB) # set SNR to SNR_Q
yq = ss.simple_quant(x,14,1,'sat')
subplot(211)
psd(y,2**12,Fs=1); # FFT length is 4096
title(r'Sinusoid + Noise to Model Quantization')
ylabel(r'PSD (dB)')
ylim([-100,20])
subplot(212)
psd(yq,2**12,Fs=1); # FFT length is 4096
title(r'Actual 14-Bit Quantized Sinusoid')
ylabel(r'PSD (dB)')
ylim([-100,20])
tight_layout()
```



We see from the above plots that the two plots are very similar. Not also that the difference in dB between the spectral peak and the *noise floor* is greater than SNR$_Q$. This is because the spectrum resolution, which is inversely proportional to the FFT length, factors into the calculation.

```
import sk_dsp_comm.fir_design_helper as fir_h
```

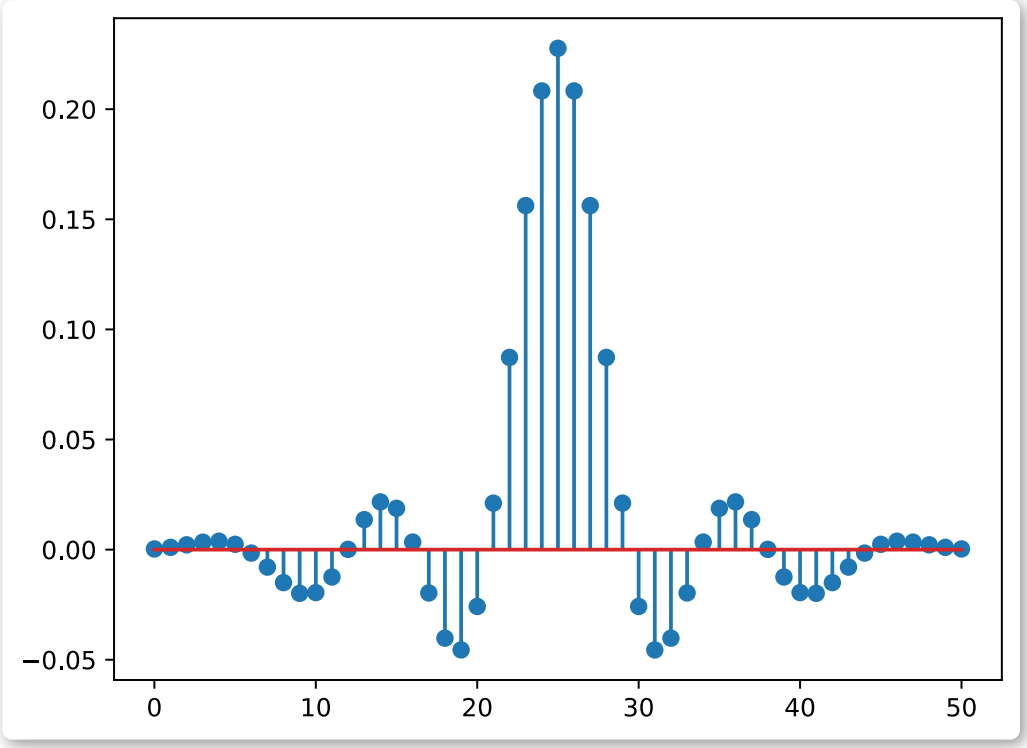
```
b = fir_h.fir_remez_lpf(.1,.15,1,80,1)
```

```
(51-1)//2
```

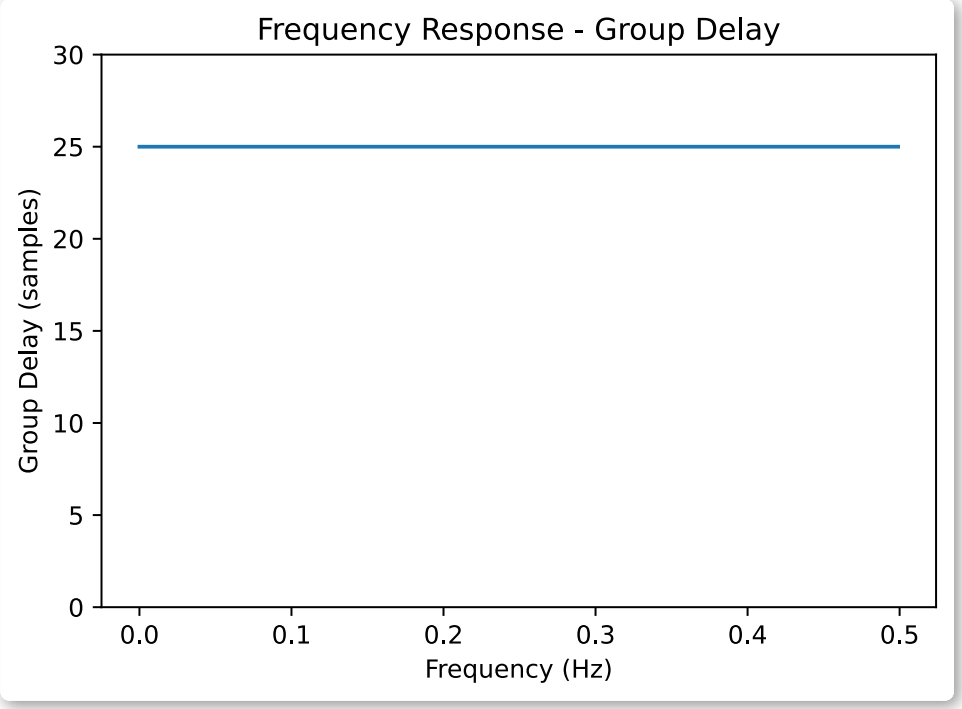
```
25
```

```
stem(b)
```

```
<StemContainer object of 3 artists>
```

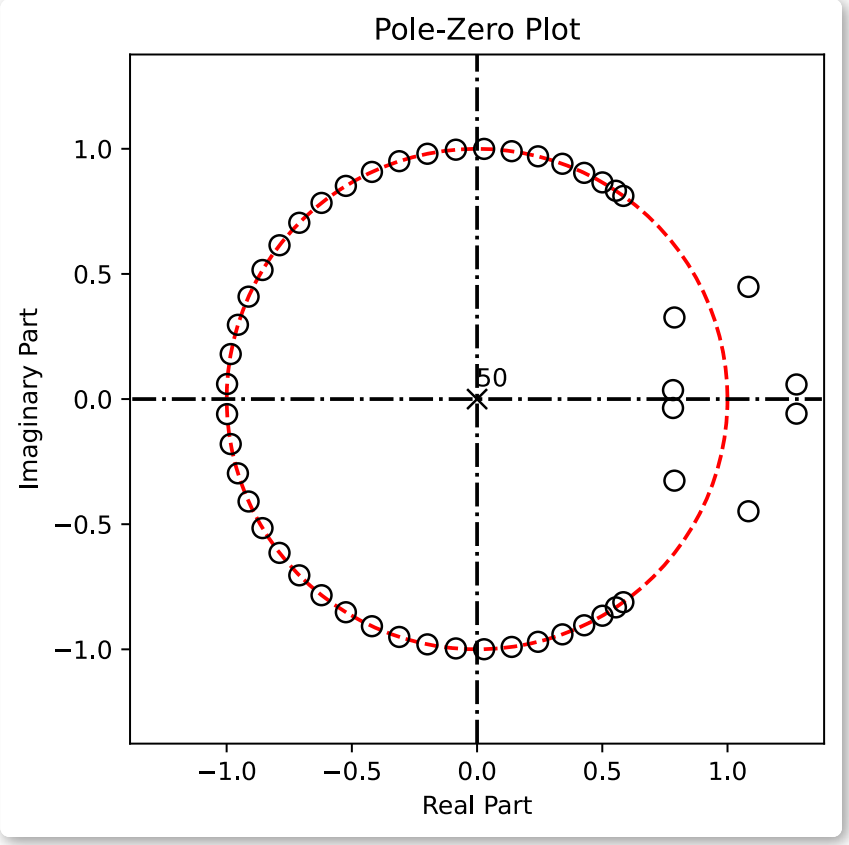



```
fir_h.freqz_resp_list([b],[1],'groupdelay_s',1)
```




```
ss.zplane(b,[1])
```

```
(50, 0)
```


Pole-Zero Plot
Imaginary Part
Real Part
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
50