Text & Notes Chapter 2 Fall 2024

```
# %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
```

Introduction

This is some text with math.

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

Suppose that $x[n] = u[n]$, then

$$y[n] = \sum_{k=0}^{\infty} u[n-k]$$

In the sum above make the variable substitution $m = n - k$. The sum limits for finite n become $k = 0 \Rightarrow m = n - 0 = n$ and $k = \infty \Rightarrow m = n - \infty = -\infty$. Making substitutions into the original sum, we have

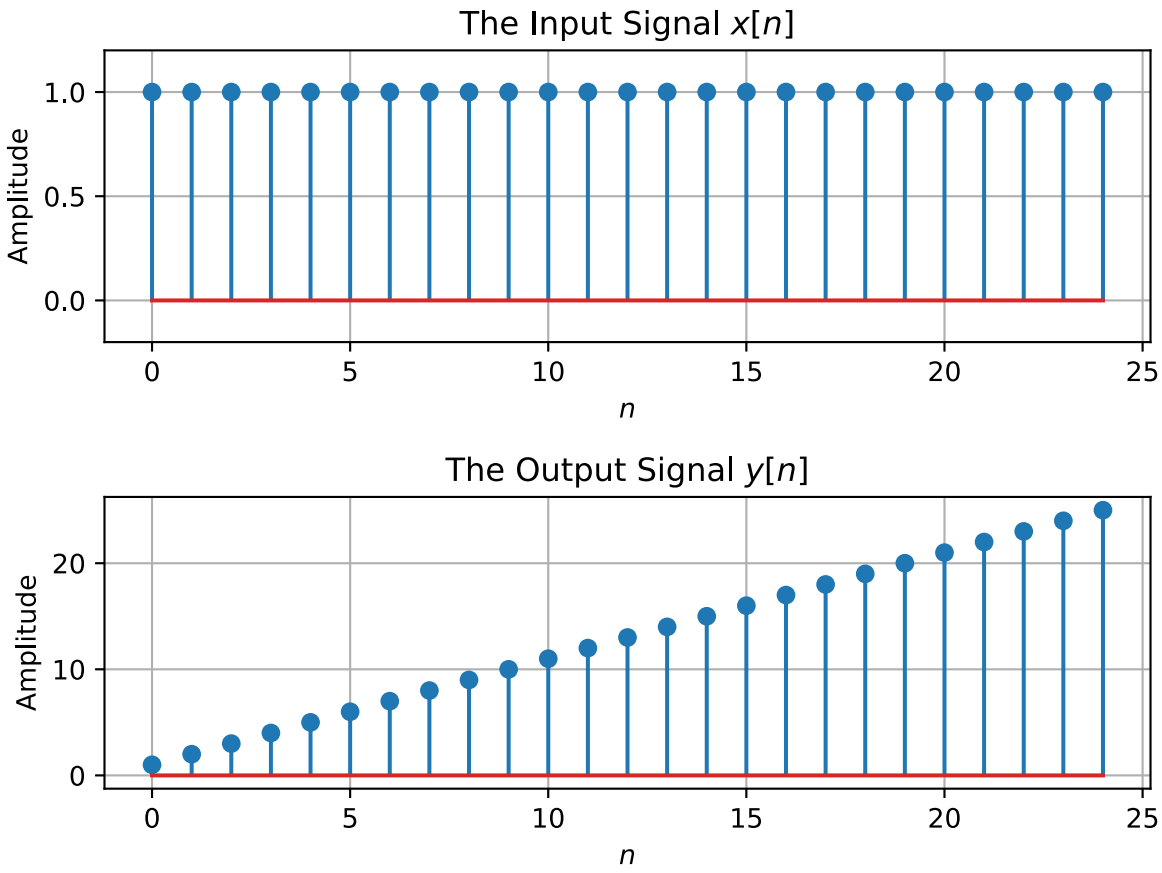
$$y[n] = \sum_{m=n}^{-\infty} u[m] = \sum_{m=-\infty}^{n} u[m] = \begin{cases} 0, & n < 0 \\ n+1, & n \geq 0 \end{cases}$$

We see that as $n \to \infty$, $y[n] \to \infty$, making the system unstable.

```
n = arange(0,25)
x = ss.dstep(n)
y = n+1
```

```
subplot(211)
stem(n,x)
grid()
```

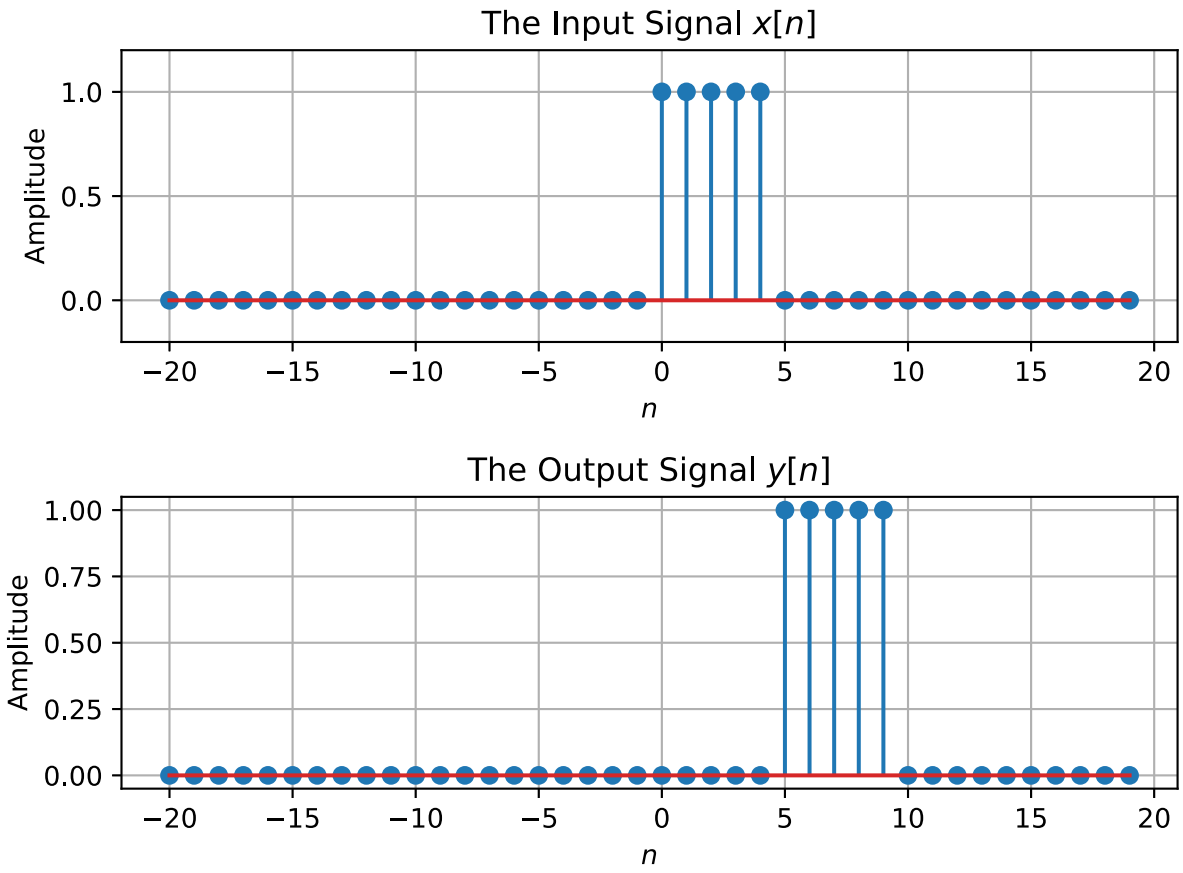
```
xlabel(r'$n$')
ylabel(r'Amplitude')
title(r'The Input Signal $x[n]$')
ylim([-.2,1.2])
subplot(212)
stem(n,y)
grid()
xlabel(r'$n$')
ylabel(r'Amplitude')
title(r'The Output Signal $y[n]$')
tight_layout()
```



Time Shift a `drect` Pulse

```
n = arange(-20,20)
x = ss.drect(n,5)
subplot(211)
stem(n,x)
grid()
xlabel(r'$n$')
ylabel(r'Amplitude')
title(r'The Input Signal $x[n]$')
ylim([-.2,1.2])
```

```
subplot(212)
y = ss.drect(n-5,5)
stem(n,y)
grid()
xlabel(r'$n$')
ylabel(r'Amplitude')
title(r'The Output Signal $y[n]$')
tight_layout()
```



Difference Equations

Example 2.4

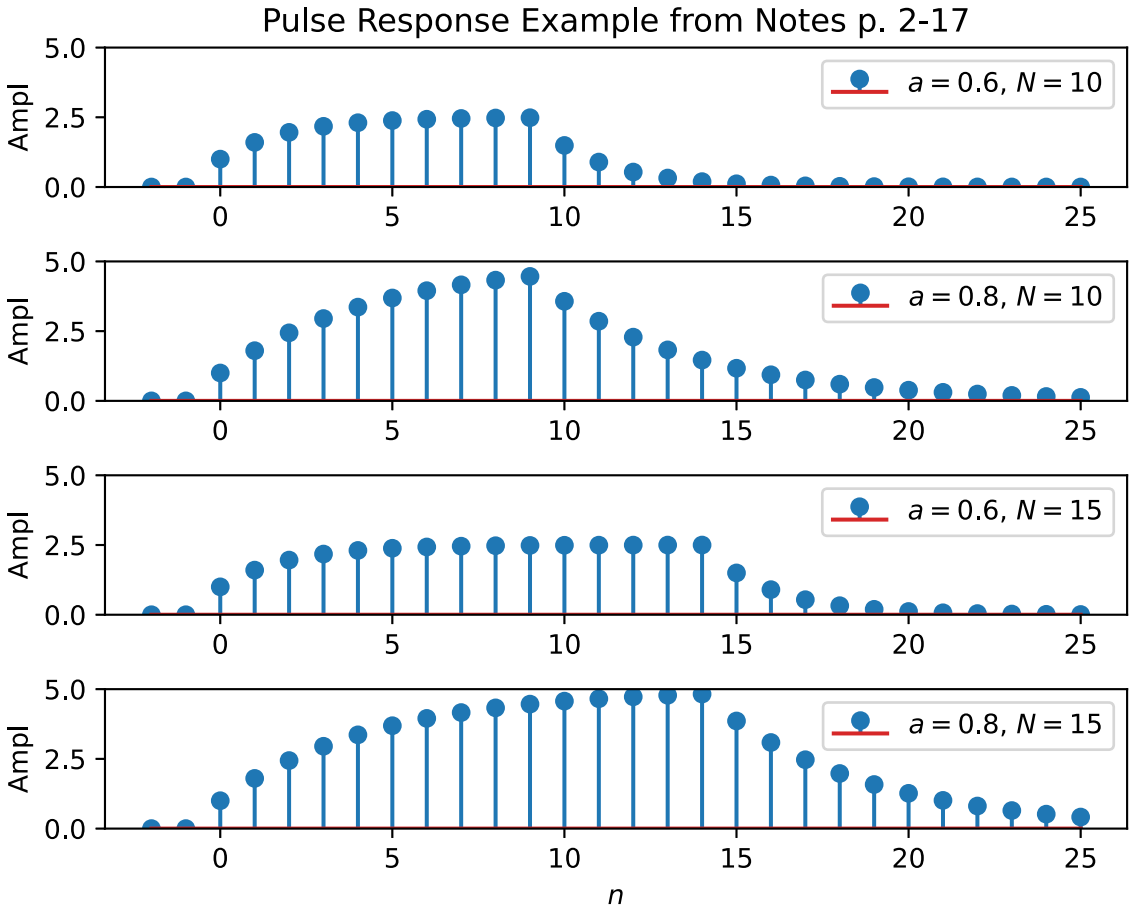
Write a simple function to compute the *piecewise expression* found in the Chapter 2 notes on page 2-17. The function is parameterized in terms of the exponential decay constant a and the pulse width N:

```
def pulse_resp(n,a,N):
    """
    A function to find the pulse response of a 1st-order system
    y = pulse_resp(n,a,N)
    n = an array of sample values
    a = is a parameter
    N = the pulse duration in samples
```

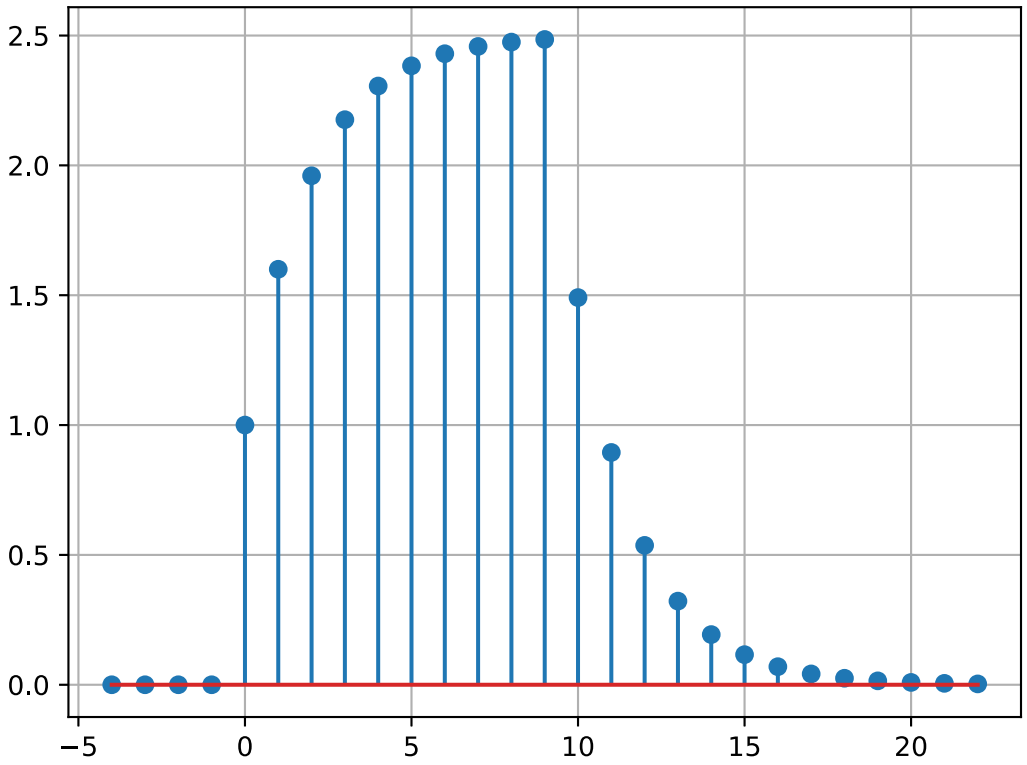
```
    """
    y = zeros(len(n))
    for k,nk in enumerate(n): # range(len(n))
        if nk < 0:
            y[k] = 0
        # elif nk >= 0 and nk <= N-1:
        elif nk <= N-1:
            y[k] = (1 - a**(nk+1))/(1 - a)
        else:
            y[k] = a**(nk-N+1)*(1 - a**N)/(1 - a)
    return y
```

- Plot a few response curves

```
n = arange(-2,25+1)
y1 = pulse_resp(n,0.6,10)
y2 = pulse_resp(n,0.8,10)
y3 = pulse_resp(n,0.6,15)
y4 = pulse_resp(n,0.8,15)
figure(figsize=(6,5)) # 6 x 5. inches
subplot(411)
stem(n,y1,label=r'$a=0.6$, $N=10$')
ylim([0,5])
legend()
title(r'Pulse Response Example from Notes p. 2-17')
ylabel(r'Ampl')
subplot(412)
stem(n,y2,label=r'$a=0.8$, $N=10$')
ylim([0,5])
legend()
ylabel(r'Ampl')
subplot(413)
stem(n,y3,label=r'$a=0.6$, $N=15$')
ylim([0,5])
legend()
ylabel(r'Ampl')
subplot(414)
stem(n,y4,label=r'$a=0.8$, $N=15$')
ylim([0,5])
legend()
xlabel(r'$n$')
ylabel(r'Ampl')
tight_layout()
```

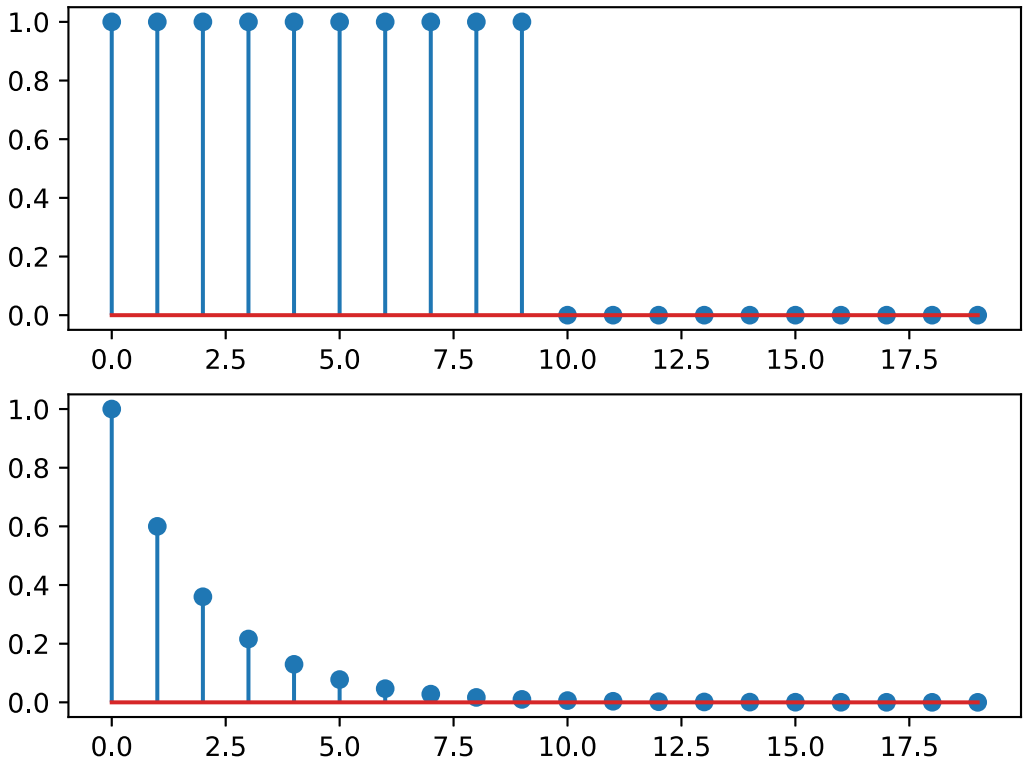



```
# Direct Convolution
nx = arange(-2,25)
nh = arange(-2,25)
x = 0.6**nx*ss.dstep(nx)
h = ss.dstep(nh) - ss.dstep(nh-10)
y,ny = ss.conv_sum(x,nx,h,nh,extent=('r', 'r'))
stem(ny,y)
grid();
```



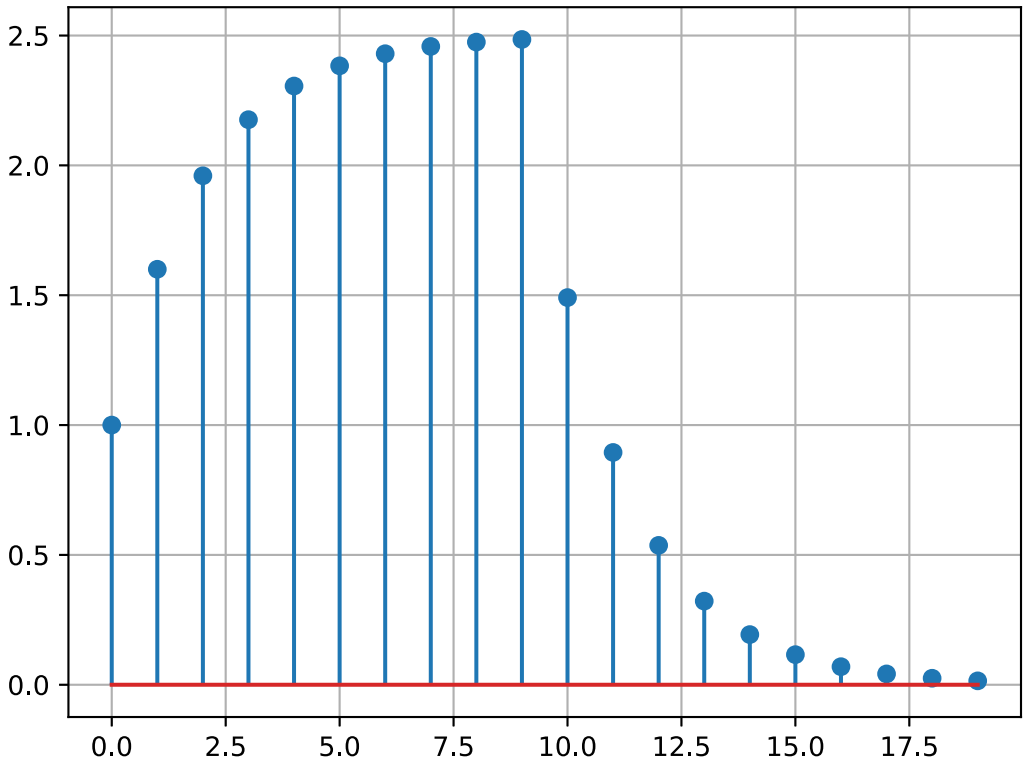

```
n = arange(0,20)
x = ss.dstep(n) - ss.dstep(n-10)
h = 0.6**n*ss.dstep(n)
subplot(211)
stem(n,x)
subplot(212)
stem(n,h)
```

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




```
y_cs,ny = ss.conv_sum(x,n,h,n,extent=('r','r'))
```

```
stem(ny,y_cs)
grid();
```




```
# y = signal.lfilter([1],[1, -.8],ss.dimpulse(n))
x = ss.drect(n,10)
y = signal.lfilter([1],[1, -.6],x)
```

```
stem(n,y)
grid();
```

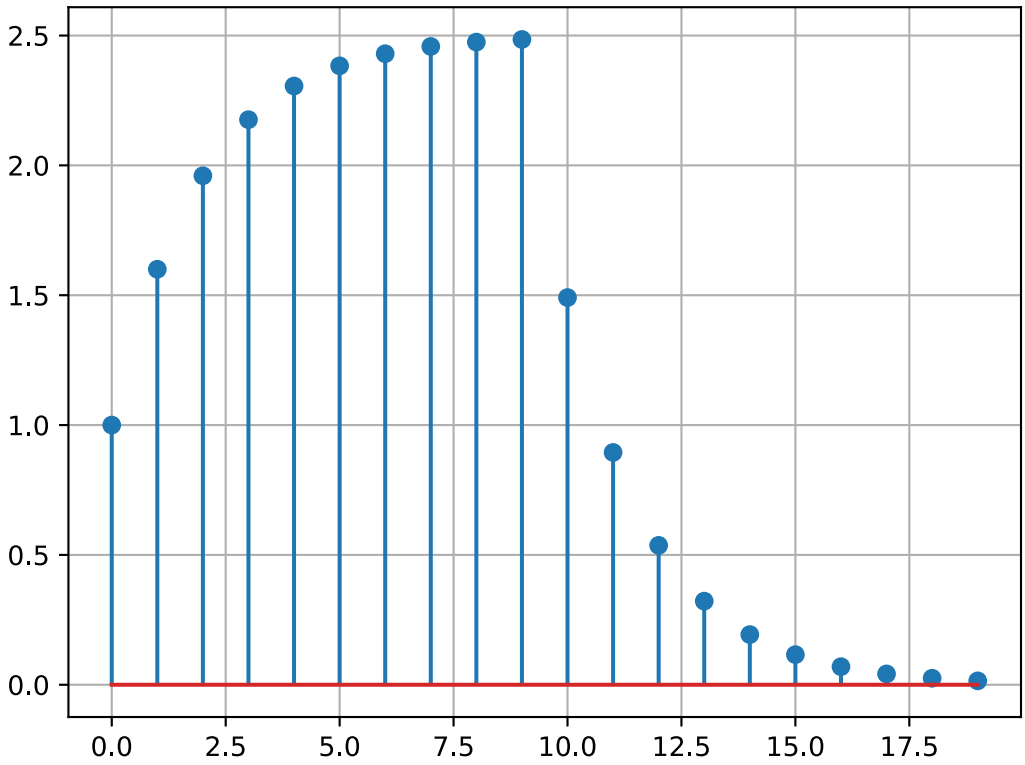


Table Compare Solutions Using `pandas`

Difference Equations

Example 2.11 Part a

$$y[n] - 3y[n-1] - 4y[n-2] = x[n] + 2x[n-1]$$

A problem to watch out for with `numpy` ndarray operations is managing data types. In the calculation of the closed-form impulse response, we need to compute $6/5 \cdot 4^n$. The narray `n` is an integer type and yes Python does suppose integer operation with infinite precision. Since we are raising a scalar to a power that is an array unexpected things can happen when the numbers become large. In this case it is better to make 4 a float, i.e., 4.0, or make `n` and float64 array, i.e., `float64(n)`.

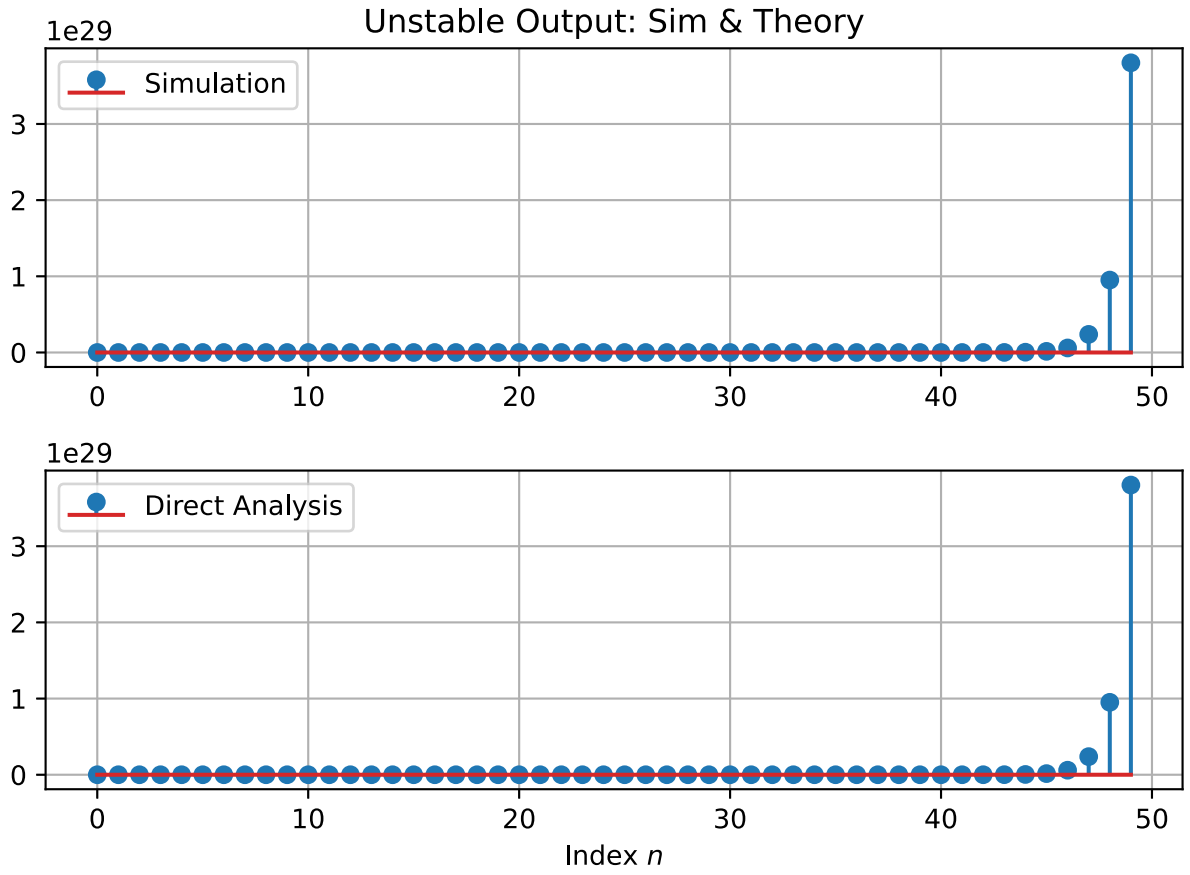
```
n = arange(50)
x_d = ss.dimpulse(n)
y_d = signal.lfilter([1,2],[1,-3,-4],x_d)
y_d_cf = -1/5*(-1)**n + 6/5*(4)**float64(n) # closed-form solution
```

```
n.dtype
```

```
dtype('int32')
```

Do Some Plotting

```
subplot(211)
stem(n,y_d,label=r'Simulation')
title(r'Unstable Output: Sim & Theory')
legend()
grid();
subplot(212)
stem(n,y_d_cf,label=r'Direct Analysis')
xlabel(r'Index $n$')
legend()
grid();
tight_layout()
```



Example 2.11 Part b

Recall the LCCDE

$$y[n] - y[n-1] = x[n] + x[n-1]$$

We can simulate this system using `signal.lfilter()`

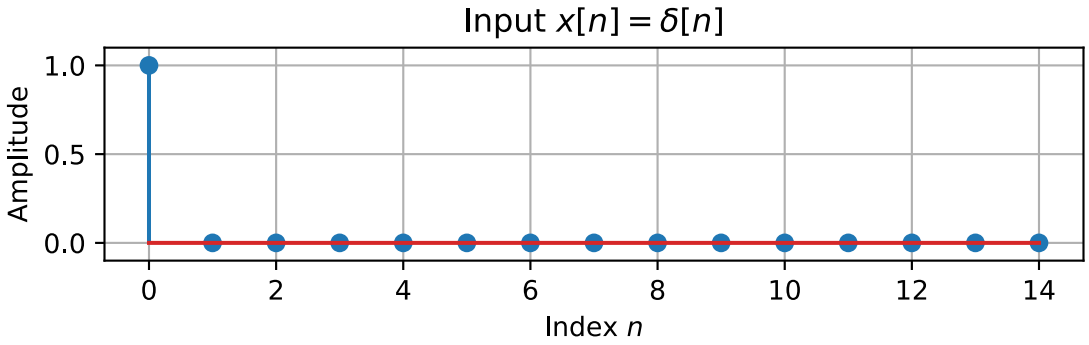
1. Create an interger axis `n` using `arange`
2. Using `ss.dimpulse()` and `n` create an impulse sequence `x`
3. Load `x` and the LCCDE coefficient arrays into `signal.filter()`

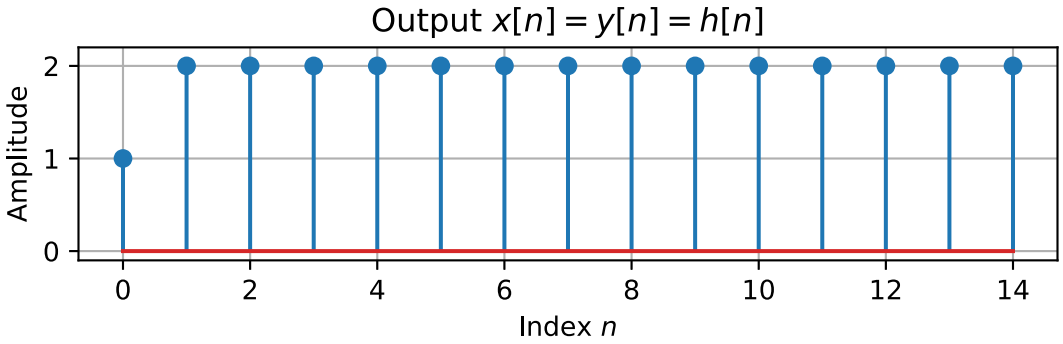
```
n = arange(0,15)
x = ss.dimpulse(n)
h = signal.lfilter([1,1],[1,-1],x)
```

```
# Take a look at the values in the output h[n]
h
```

```
array([1., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.])
```

```
#create a plot of the input and output sequences
figure(figsize=(6,4))
subplot(211)
stem(n,x)
ylim([-.1,1.1])
xlabel(r'Index $n$')
ylabel(r'Amplitude')
title(r'Input $x[n] = \delta[n]$')
grid();
subplot(212)
stem(n,h)
ylim([-.1,2.2])
xlabel(r'Index $n$')
ylabel(r'Amplitude')
title(r'Output $x[n] = y[n] = h[n]$')
grid();
tight_layout()
```





The results are as expected from the lecture notes example, that is

$$h[n] = 2u[n] - \delta[n]$$

Frequency Response

The first-order Difference Equation

The first-order difference equation of Example 2.10 in notes Chapter 2 is a good starting point:

$$y[n] = ay[n-1] + x[n],\ |a| < 1 \text{ for stability}$$

Using classical solution techniques or via simple recursion, we can find the impulse response to be

$$h[n] = a^n u[n]$$

The frequency response of this system can be obtained using the definition and the infinite geometric series

$$H(e^{j\omega}) = \sum_{n=0}^{\infty} a^n \cdot e^{-j\omega n} = \sum_{n=0}^{\infty} \left(ae^{-j\omega}\right)^n = \frac{1}{1 - ae^{j\omega}}$$

Note in particular that the DC gain of this system is

$$H(e^{j0}) = H(1) = \frac{1}{1-a}$$

From Example 2.19 in notes Chapter 2 we can write $H(e^{j\omega})$ is polar form (mag/phase) as

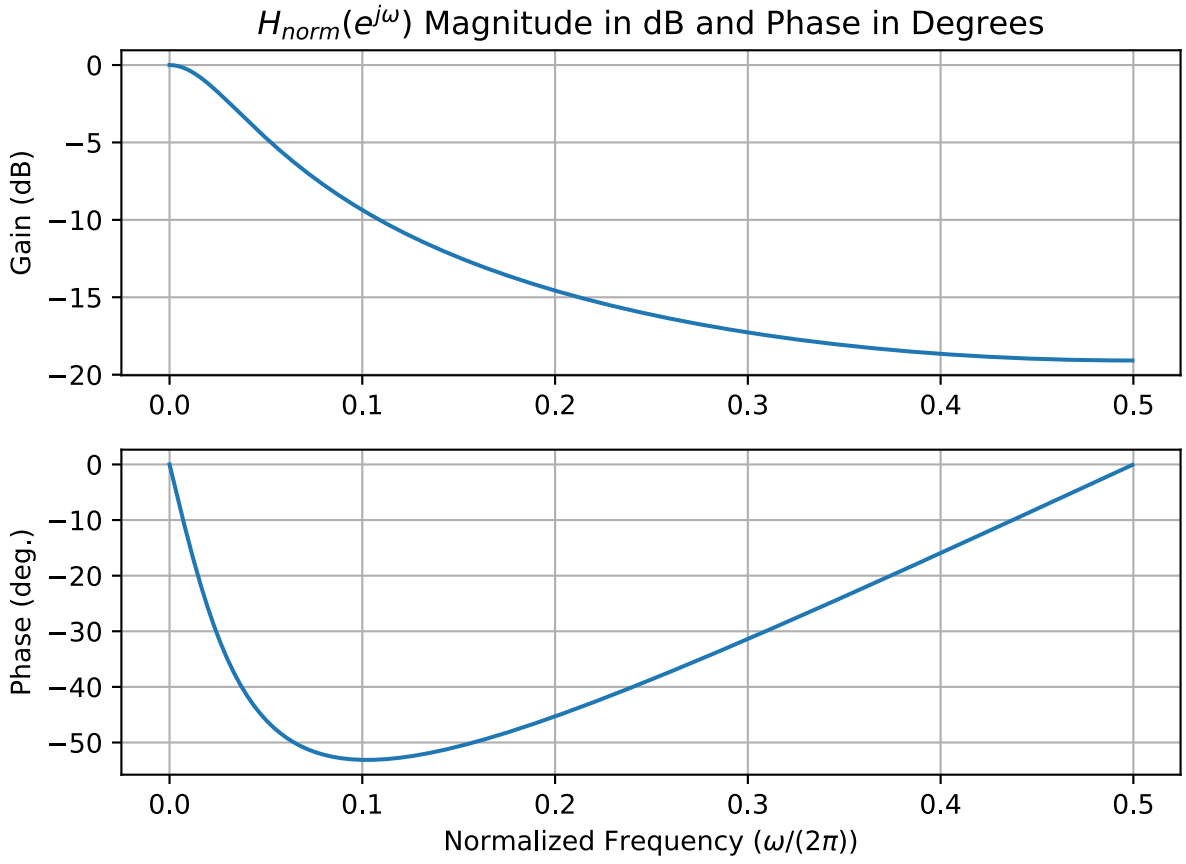
$$H(e^{j\omega}) = \frac{1}{\sqrt{1 + a^2 - 2a\cos\omega}}, \exp\left[j\tan^{-1}\left(\frac{a\sin\omega}{1 - a\cos\omega}\right)\right]$$

To make the system gain unity at DC we can multiply by $1-a$ and define $H_{\text{norm}}(e^{j\omega}) = (1-a)H(e^{j\omega})$

Plot the Frequency Response

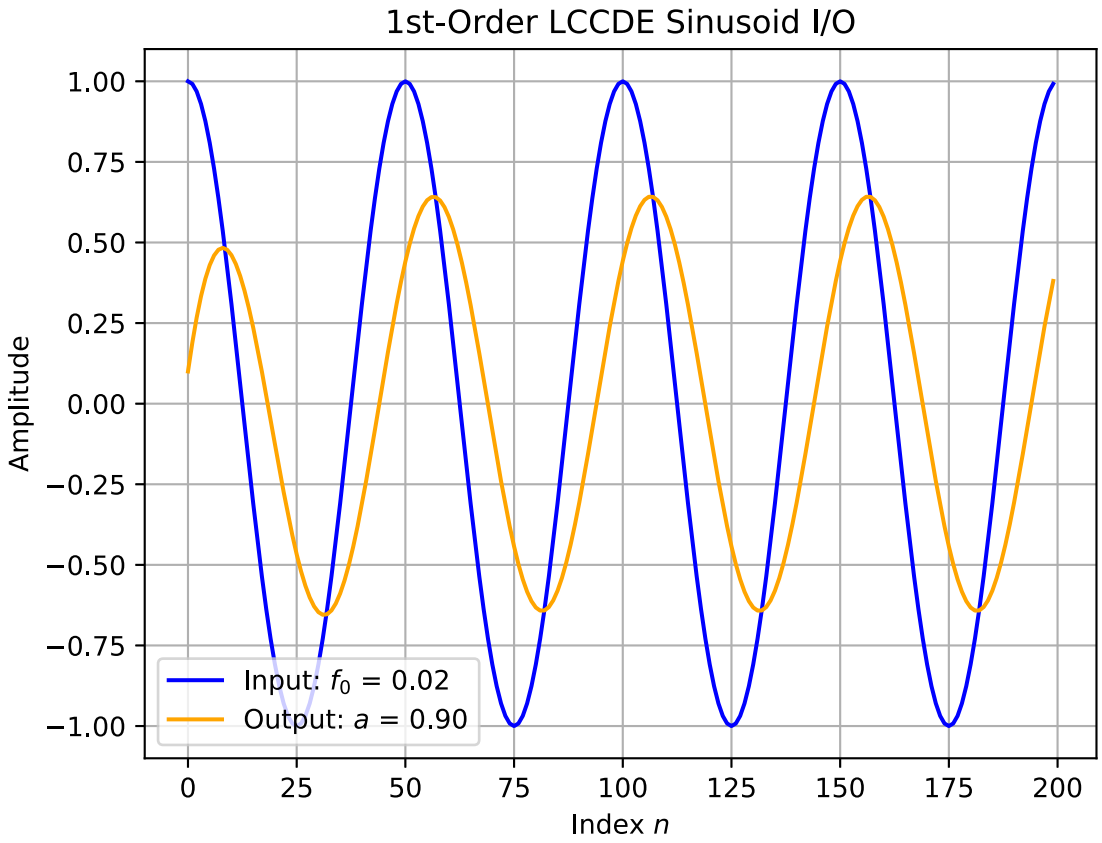
The analytical expression looks nice, but it is easient to just use `freqz()` in either Python or Julia to plot the frequency response. It. is convenient to use the normalized frequency f for plotting, where $\omega = 2\pi f$. For a system with real impulse response we only need to plot over the interval $f \in [0, 0.5]$ or $\omega \in [0, \pi]$.

```
f = arange(0,0.5,1/2048)
a = 0.8
w, H_norm = signal.freqz(1 - a, [1, -a],2*pi*f) # unity normalized gain
subplot(211)
plot(f,20*log10(abs(H_norm)))
title(r'$H_{norm}(e^{j\omega})$ Magnitude in dB and Phase in Degrees')
ylabel(r'Gain (dB)')
grid()
subplot(212)
plot(f,angle(H_norm)*180/pi)
xlabel(r'Normalized Frequency ($\omega/(2\pi)$)')
ylabel(r'Phase (deg.)')
grid()
tight_layout()
```



Pass A Real Sinusoid Through the System

```
n = arange(0,200)
f0 = 0.02 # normalized frequency f0 = w0/(2*pi)
a = 0.9
x = cos(2*pi*0.02*n) # Frequency f = 0.02
y = signal.lfilter([1-a],[1,-a],x) # LCCDE
plot(n,x,'b',label=r'Input: $f_0$ = %4.2f' % f0) # connect the dots
plot(n,y,'orange',label=r'Output: $a$ = %4.2f' % a)
xlabel(r'Index $n$')
ylabel(r'Amplitude')
title(r'1st-Order LCCDE Sinusoid I/O')
legend()
grid();
```



Evaluate the Gain and Phase at DC and `f0`

You should see the amplitude and phase of the output shifted accordingly.

```
f = array([0,f0])
w,H = signal.freqz([1-a],[1,-a],2*pi*f)
```

```
print(abs(H))
print(angle(H)*180/pi)
```

```
[1.         0.64291019]
[  0.         -46.48566452]
```

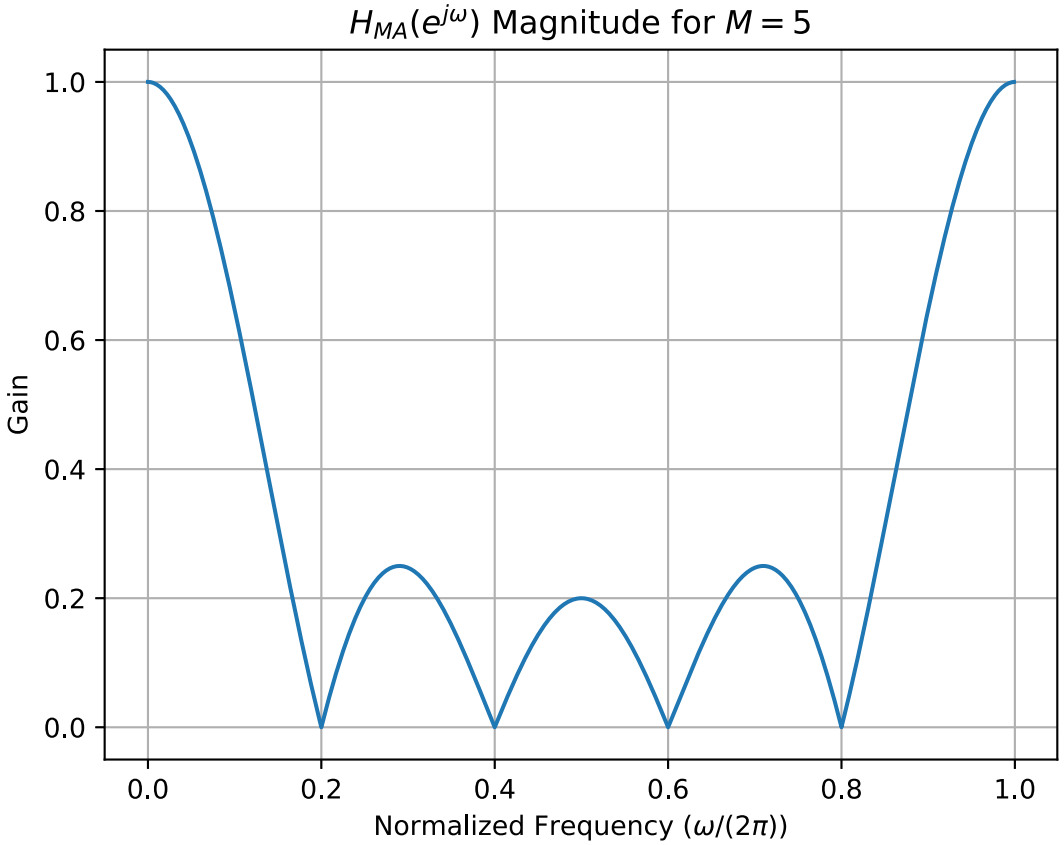
The Moving Average System (Filter)

Consider $M = 5$ filter coefficients (causal).

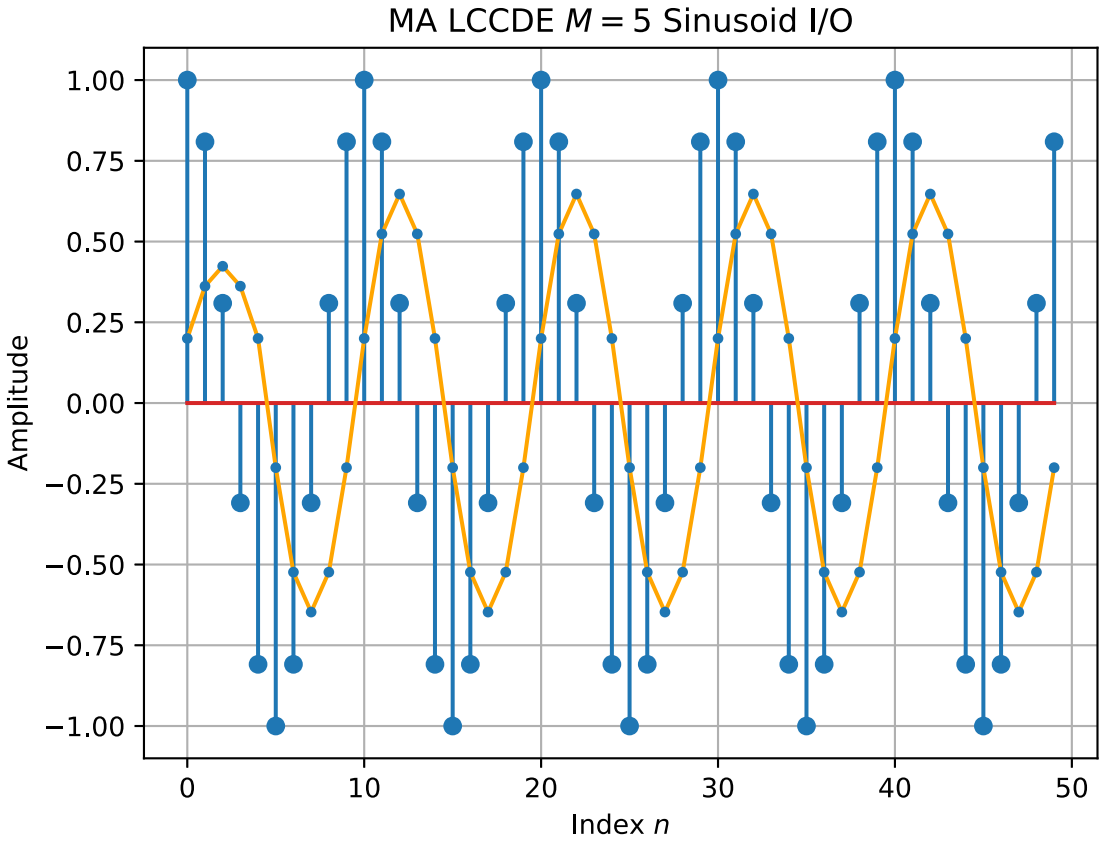
```
h = ones(5)/5
h
```

```
array([0.2, 0.2, 0.2, 0.2, 0.2])
```

```
f = arange(0,1.0,.001)
w,H = signal.freqz(h,[1],2*pi*f)
plot(f,abs(H))
# ylim([-80,0])
title(r'$H_{MA}(e^{j\omega})$ Magnitude for $M=5$')
ylabel(r'Gain')
xlabel(r'Normalized Frequency ($\omega/(2\pi)$)')
grid();
```




```
n = arange(0,50)
f0 = 0.1
x = cos(2*pi*f0*n)
y = signal.lfilter(h,[1],x)
stem(n,x)
# plot(n,x)
plot(n,y,'orange')
plot(n,y,'.')
xlabel(r'Index $n$')
ylabel(r'Amplitude')
title(r'MA LCCDE $M=5$ Sinusoid I/O')
grid();
```



Aliased Sinc Function

The sinc function is typically defined as

$$\text{sinc}(x) \equiv \frac{\sin(\pi x)}{\pi x}$$

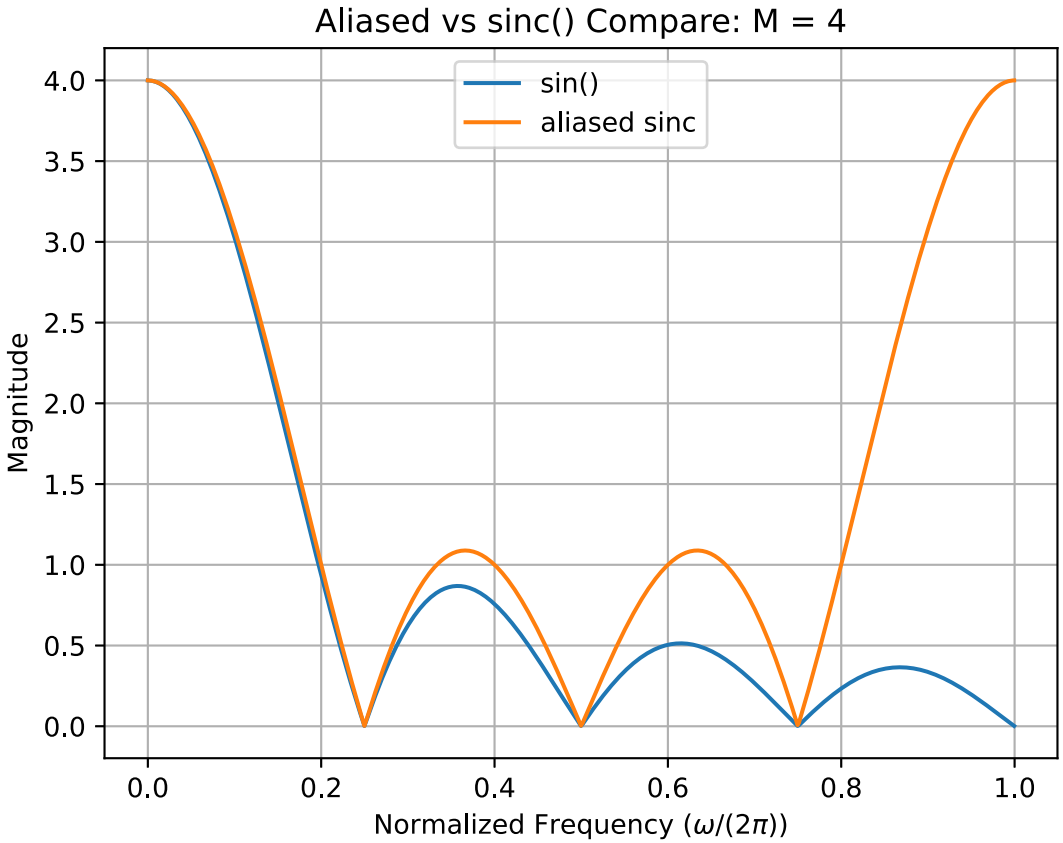
Notice also that the aliased sinc can be defined in terms of the sinc function:

$$\frac{\sin(\omega M/2)}{\sin(\omega/2)} = \frac{\sin(\omega M/2)}{\omega M/2} \cdot \frac{\omega M/2}{\sin(\omega/2)} = M \frac{\text{sinc}(\omega M/(2\pi)}{\text{sinc}(\omega/(2\pi)}$$

This expansion also shows how the scale factor M appears. Writing in terms of sinc functions manages the singularity when the argument is zero.

```
M = 4
w = arange(0,2*pi,.01)
W_s = sinc(M*w/2/pi)
plot(w/2/pi,M*abs(W_s),label=r'sin()')
# W_as = sin(w*M/2)/sin(w/2)
W_as = M*sinc(w*M/2/pi)/sinc(w/2/pi)
plot(w/2/pi,abs(W_as),label=r'aliased sinc')
title(r'Aliased vs sinc() Compare: M = {}'.format(M))
ylabel(r'Magnitude')
xlabel(r'Normalized Frequency ($\omega/(2\pi)$)')
```

```
legend()
grid();
```

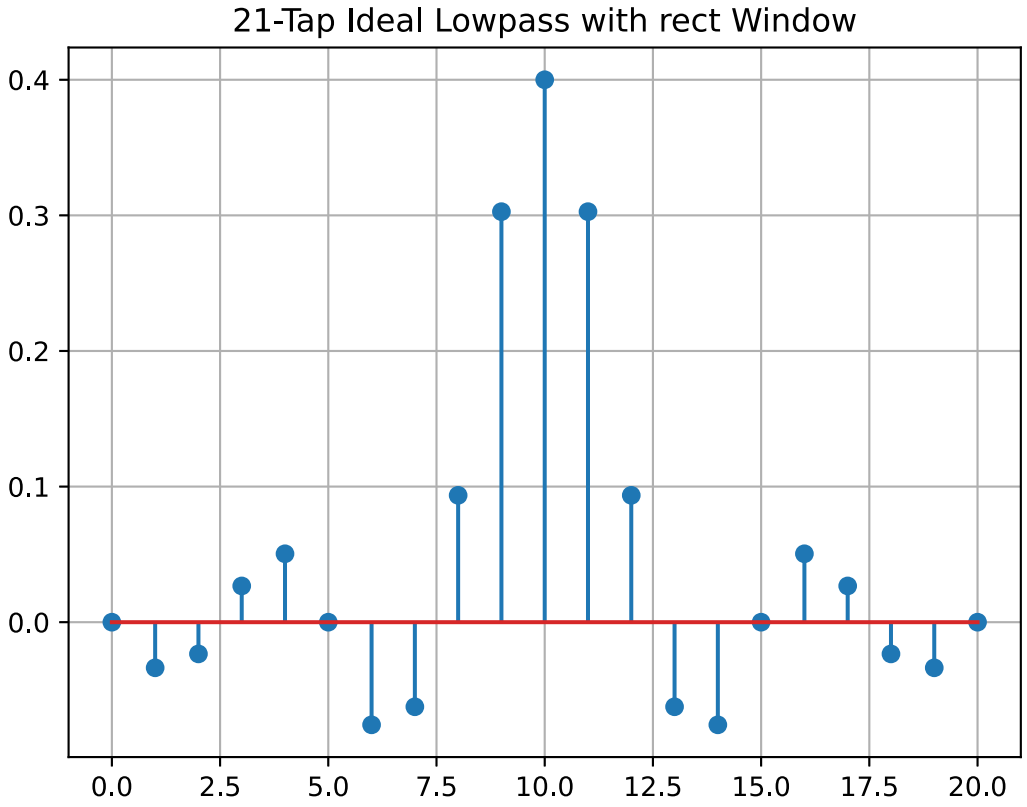


Ideal Lowpass

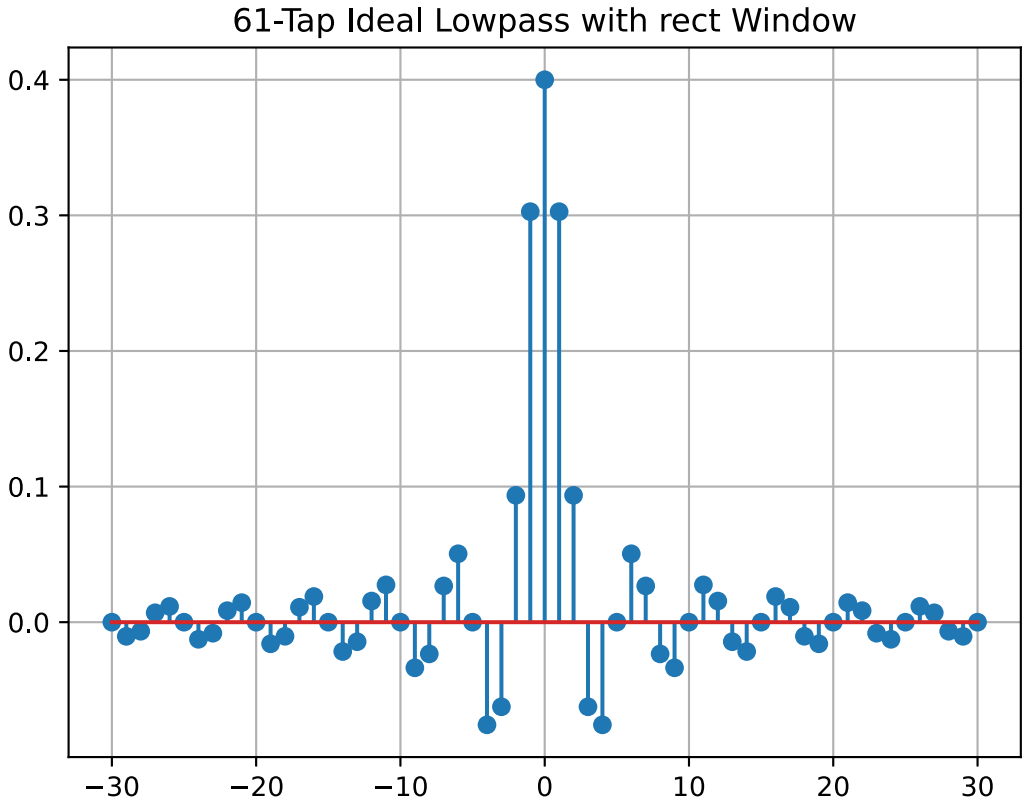
$$h[n] = \frac{\sin(2\pi f_c n)}{\pi n} = 2f_c \cdot \mathrm{sinc}(2f_c n)$$

where $\omega_c = 2\pi f_c$ and $\mathrm{sinc}(x) = \sin(\pi x)/(\pi x)$.

```
n21 = arange(-10,11)
h21 = 2*0.2*sinc(2*0.2*n21)
stem(n21+10,h21)
title(r'%d-Tap Ideal Lowpass with rect Window' % len(n21))
grid();
```



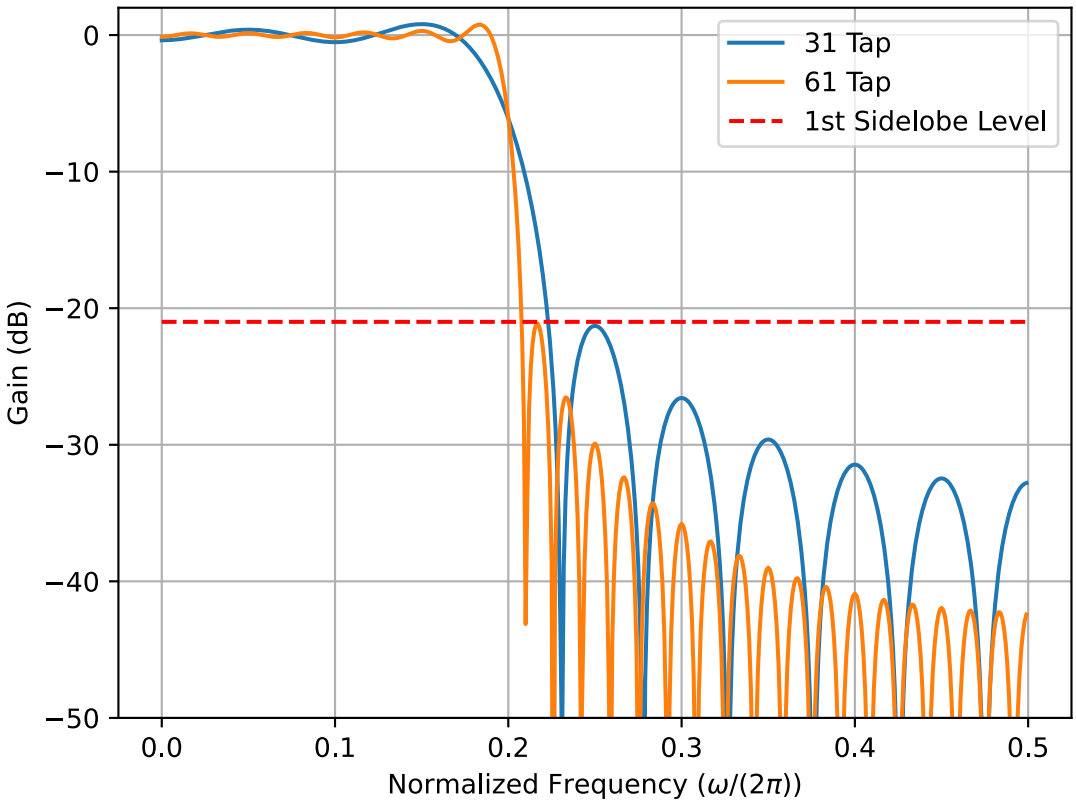

```
n61 = arange(-30,31)
h61 = 2*0.2*sinc(2*0.2*n61)
stem(n61,h61)
title(r'%d-Tap Ideal Lowpass with rect Window' % len(n61))
grid();
```



Compare the Frequency Response of the Truncated Impulse Responses

You will see that in spite of the longer approximation to the impulse response the first sidelobe is still only down 21 dB!

```
f = arange(0,.5,0.001)
w,H21 = signal.freqz(h21,1,2*pi*f)
w,H61 = signal.freqz(h61,1,2*pi*f)
plot(f,20*log10(abs(H21)),label=r'31 Tap')
plot(f,20*log10(abs(H61)),label=r'61 Tap')
plot([0,.5],[-21,-21],'r--',label=r'1st Sidelobe Level') # first sidelobe down 21 dB
ylim([-50,2])
xlabel(r'Normalized Frequency ($\omega/(2\pi)$)')
ylabel(r'Gain (dB)')
legend()
grid();
```



Frequency Response and Spectrum

Next we move on to frequency response and the spectrum by considering a simple rectangular window signal of the form

$$w[n] = u[n] - u[n-M]$$

In applications of spectral estimation we might also view the rectangle as a window function, that is perhaps we form a *windowed* signal $x_w[n]$ via

$$x_w[n] = x[n] \times w[n]$$

where now $w[n]$ is the rectangular pulse signal and $x[n]$ is a signal of interest being acquired from an analog signal $x(t)$ via an ADC.

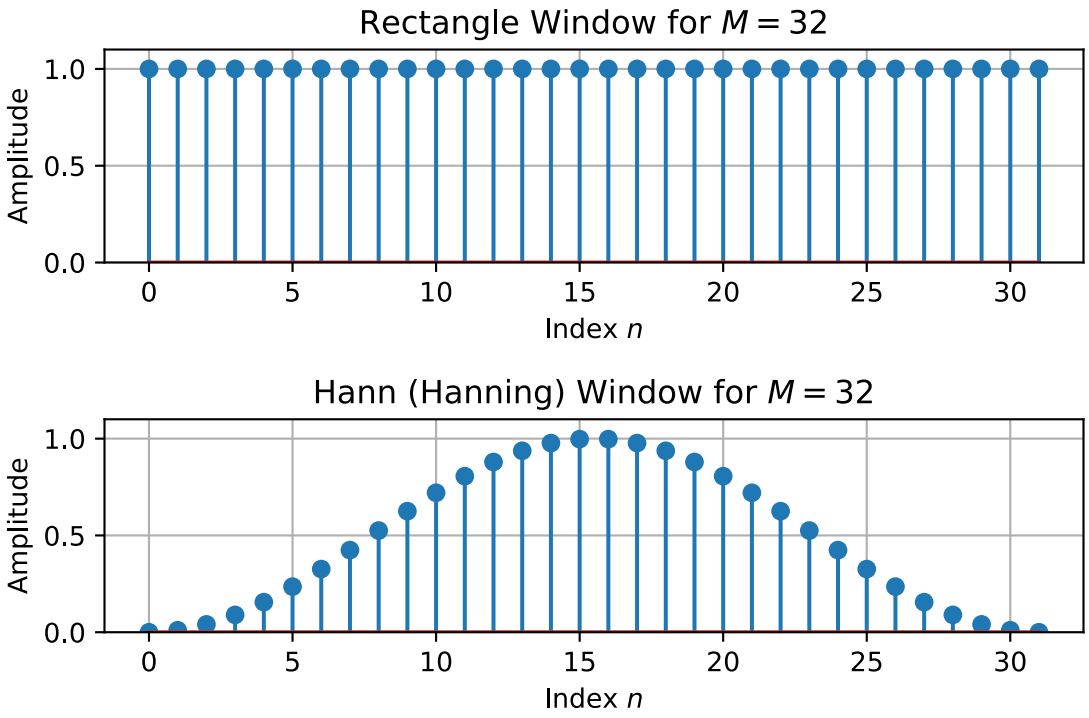
For comparison purposes consider also a shaped window formed using `scipy.signal.windows`. To see the available window functions we can list them using

```
dir(signal.windows)
```

Editing the list and placing it in this markdown cell, we have: `barthann`, `bartlett`, `blackman`, `blackmanharris`, `bohman`, `boxcar`, `chebwin`, `cosine`, `flattop`, `gaussian`, `general_gaussian`, `hamming`, `hann`, `hanning`, `kaiser`, `nuttall`, `parzen`, `slepian`, `triang`

```
M = 32
w_rect = signal.windows.boxcar(M) # rect window
w_hann = signal.windows.hann(M)
```

```
figure(figsize=(6,4))
n = arange(0,len(w_rect))
subplot(211)
stem(n,w_rect)
xlabel(r'Index $n$')
ylabel(r'Amplitude')
title(r'Rectangle Window for $M = $%d' % M)
ylim([0,1.1])
grid();
subplot(212)
stem(n,w_hann)
xlabel(r'Index $n$')
ylabel(r'Amplitude')
title(r'Hann (Hanning) Window for $M = $%d' % M)
ylim([0,1.1])
grid();
tight_layout()
```

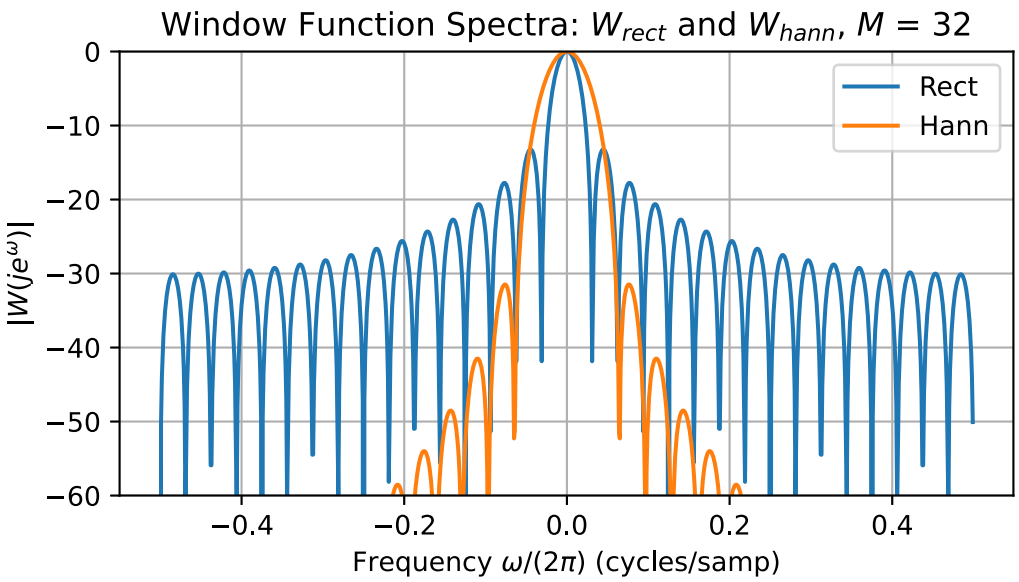


Consider the Spectrum of the Window Alone

Next plot the spectrum of each of the two windows:

```
f = arange(-.5,.5,.001)
w,W_rect = signal.freqz(w_rect,[1],2*pi*f)
w,W_hann = signal.freqz(w_hann,[1],2*pi*f)
```

```
figure(figsize=(6,3))
plot(f,20*log10(abs(W_rect)/sum(w_rect))) #normalize
plot(f,20*log10(abs(W_hann)/sum(w_hann))) #normalize
xlabel(r'Frequency $\omega/(2\pi)$ (cycles/samp)')
ylabel(r'$|W(je^{\omega})|$')
title(r'Window Function Spectra: $W_{rect}$ and $W_{hann}$, $M$ = %d' % M)
ylim([-60,0])
legend((r'Rect',r'Hann'),loc='best')
grid();
```



Note how rapidly the spectrum of the *hann* (or hanning) window drops off compared with the simple rectangle.

Spectral Estimation and Windowing

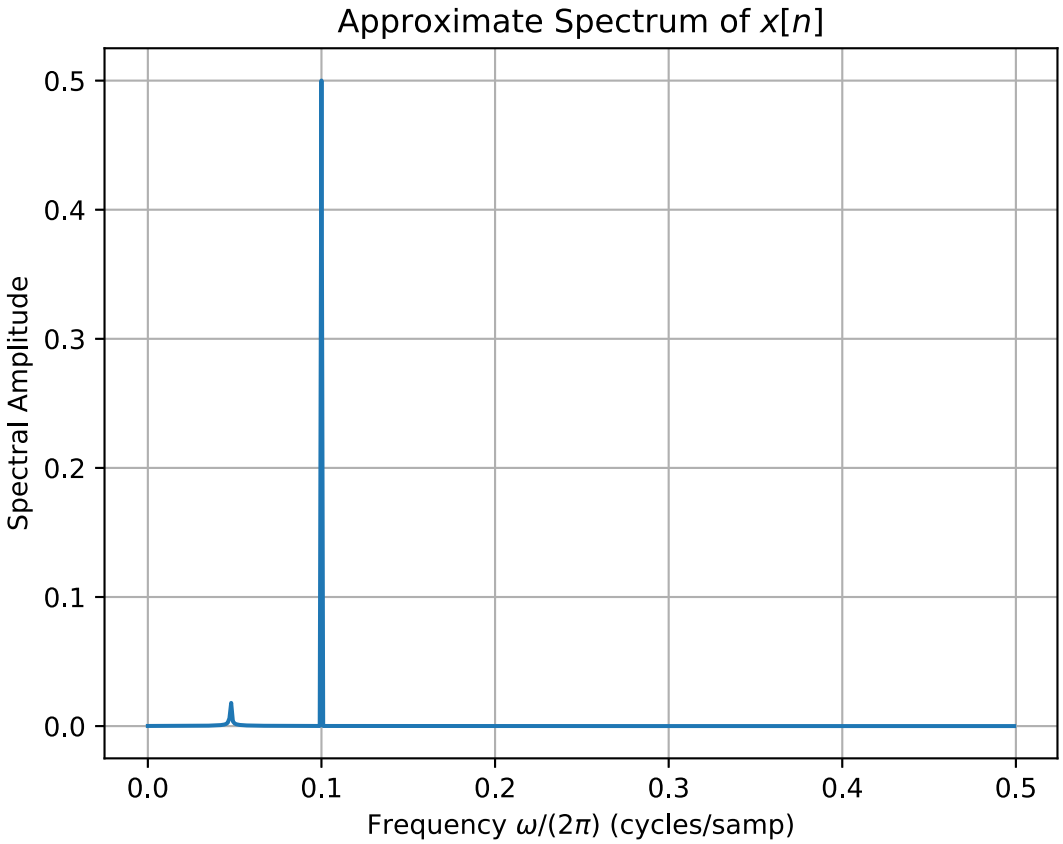
A simple spectrum estimation example is to now define signal $x[n]$ to be

$$x[n] = \cos(\omega_0 n), \ -\infty < n < \infty$$

multiply by the window functions and then consider the DTFT of the results.

```
n = arange(0,1000)
w0 = 1/5
x = cos(2*pi*1/10*n) + 2/50*cos(1.5*w0*n)
f = arange(0,0.5,.001)
```

```
w,X = signal.freqz(x,[1],2*pi*f)
plot(f,abs(X)/1000)
xlabel(r'Frequency $\omega/(2\pi)$ (cycles/samp)')
ylabel(r'Spectral Amplitude')
title(r'Approximate Spectrum of $x[n]$')
grid();
```

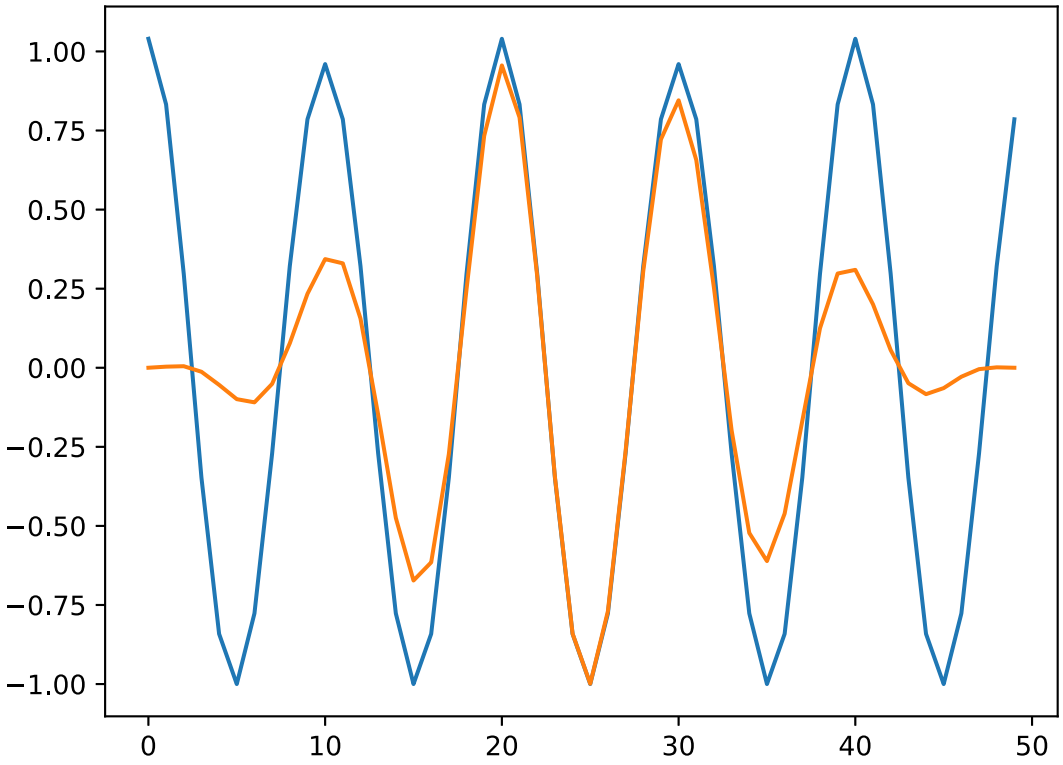



```
#Form x
w0 = 2*pi/10
M = 50
n = arange(M)
x = cos(w0*n)+ 2/50*cos(1.5*w0*n)
w_rect = ones(M) # also signal.windows.boxcar()
w_hann = signal.windows.hann(M)
#Form x_w_rect and x_w_hann
x_w_rect = x*w_rect
x_w_hann = x*w_hann
```

```
signal.windows.
```

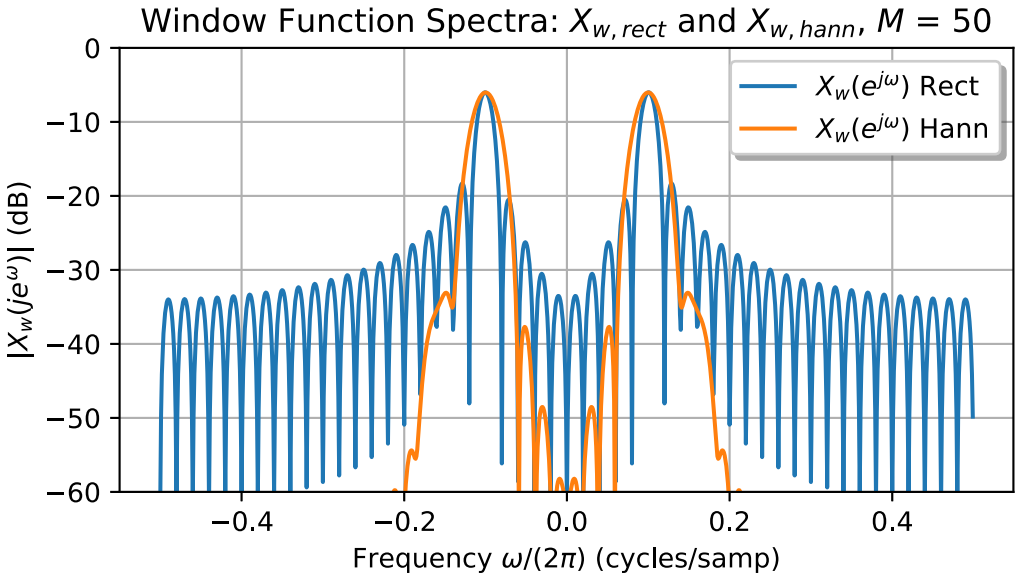
```
plot(x_w_rect)
plot(x_w_hann)
```

```
[<matplotlib.lines.Line2D at 0x21c0f10ea30>]
```



- Compute the DTFT using `freqz` with the signal values loaded into the `b` array and the `a` array just 1.

```
f = arange(-.5,.5,.001)
w,X_w_rect = signal.freqz(x_w_rect,[1],2*pi*f)
w,X_w_hann = signal.freqz(x_w_hann,[1],2*pi*f)
figure(figsize=(6,3))
plot(f,20*log10(abs(X_w_rect)/sum(w_rect))) #normalize
plot(f,20*log10(abs(X_w_hann)/sum(w_hann))) #normalize
xlabel(r'Frequency $\omega/(2\pi)$ (cycles/samp)')
ylabel(r'$|X_w(je^{\omega})|$ (dB)')
title(r'Window Function Spectra: $X_{w,rect}$ and $X_{w,hann}$, $M$ = %d'\
      % M)
ylim([-60,0])
legend((r'$X_w(e^{j\omega})$ Rect',r'$X_w(e^{j\omega})$ Hann'),
       loc='best',fontsize='medium',shadow=True)
grid();
```



Notice how much more dynamic range there is when using the windowing. The frequency resolution is still not very good, but the window size is still very small. Try increasing it to see what happens. **Note** The true spectrum should be delta functions at $f = 1.0f_0$ and $f = 1.5f_0$, with the higher frequency sinusoid having amplitude 2/50.

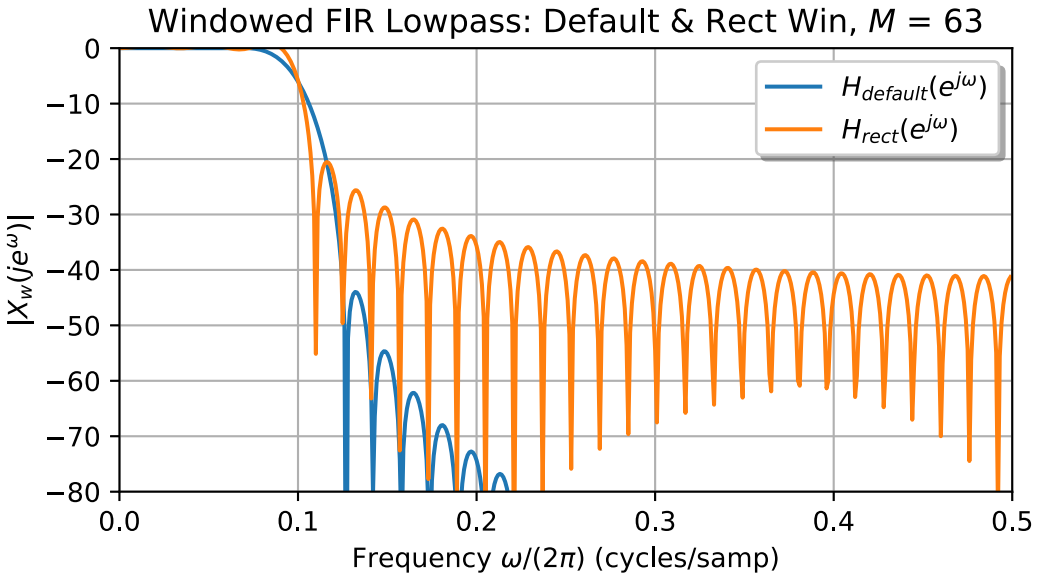
Windowed FIR Filters

In this short study we again look at windows, but this time FIR lowpass filters that use windowing.

```
b_default = signal.firwin(63,2*0.1,window='hann') # uses a Hamming window
b_rect  = signal.firwin(63,2*0.1,window='boxcar')
```

```
w,H_default = signal.freqz(b_default,[1],2*pi*f)
w,H_rect = signal.freqz(b_rect,[1],2*pi*f)

figure(figsize=(6,3))
plot(f,20*log10(abs(H_default)))
plot(f,20*log10(abs(H_rect)))
xlabel(r'Frequency $\omega/(2\pi)$ (cycles/samp)')
ylabel(r'$|X_w(je^{\omega})|$')
title(r'Windowed FIR Lowpass: Default & Rect Win, $M$ = 63')
ylim([-80,0])
xlim([0,0.5])
legend((r'$H_{default}(e^{j\omega})$',r'$H_{rect}(e^{j\omega})$'),
       loc='best',fontsize='medium',shadow=True)
grid();
```



Look at the Coefficients

Compare the default design that uses the non-rectangular window with the rectangular window coefficients. At a distance the coefficients look similar.

```
b_default
```

```
array([  4.83453019e-04,  -6.42932042e-19,  -5.77622799e-04,
        -1.09443334e-03,  -1.31665387e-03,  -9.92895649e-04,
         1.26813269e-18,   1.49067430e-03,   2.93993083e-03,
         3.56070478e-03,   2.64660980e-03,  -2.63400916e-18,
        -3.74912317e-03,  -7.15192928e-03,  -8.38647763e-03,
        -6.04986417e-03,   4.39730745e-18,   8.15799349e-03,
         1.52795495e-02,   1.76759087e-02,   1.26467028e-02,
        -6.11489876e-18,  -1.71012315e-02,  -3.24688076e-02,
        -3.84973985e-02,  -2.86487075e-02,   7.35514077e-18,
         4.50885485e-02,   9.89082455e-02,   1.50132542e-01,
         1.86895906e-01,   2.00256751e-01,   1.86895906e-01,
         1.50132542e-01,   9.89082455e-02,   4.50885485e-02,
         7.35514077e-18,  -2.86487075e-02,  -3.84973985e-02,
        -3.24688076e-02,  -1.71012315e-02,  -6.11489876e-18,
         1.26467028e-02,   1.76759087e-02,   1.52795495e-02,
         8.15799349e-03,   4.39730745e-18,  -6.04986417e-03,
        -8.38647763e-03,  -7.15192928e-03,  -3.74912317e-03,
        -2.63400916e-18,   2.64660980e-03,   3.56070478e-03,
         2.93993083e-03,   1.49067430e-03,   1.26813269e-18,
        -9.92895649e-04,  -1.31665387e-03,  -1.09443334e-03,
        -5.77622799e-04,  -6.42932042e-19,   4.83453019e-04])
```

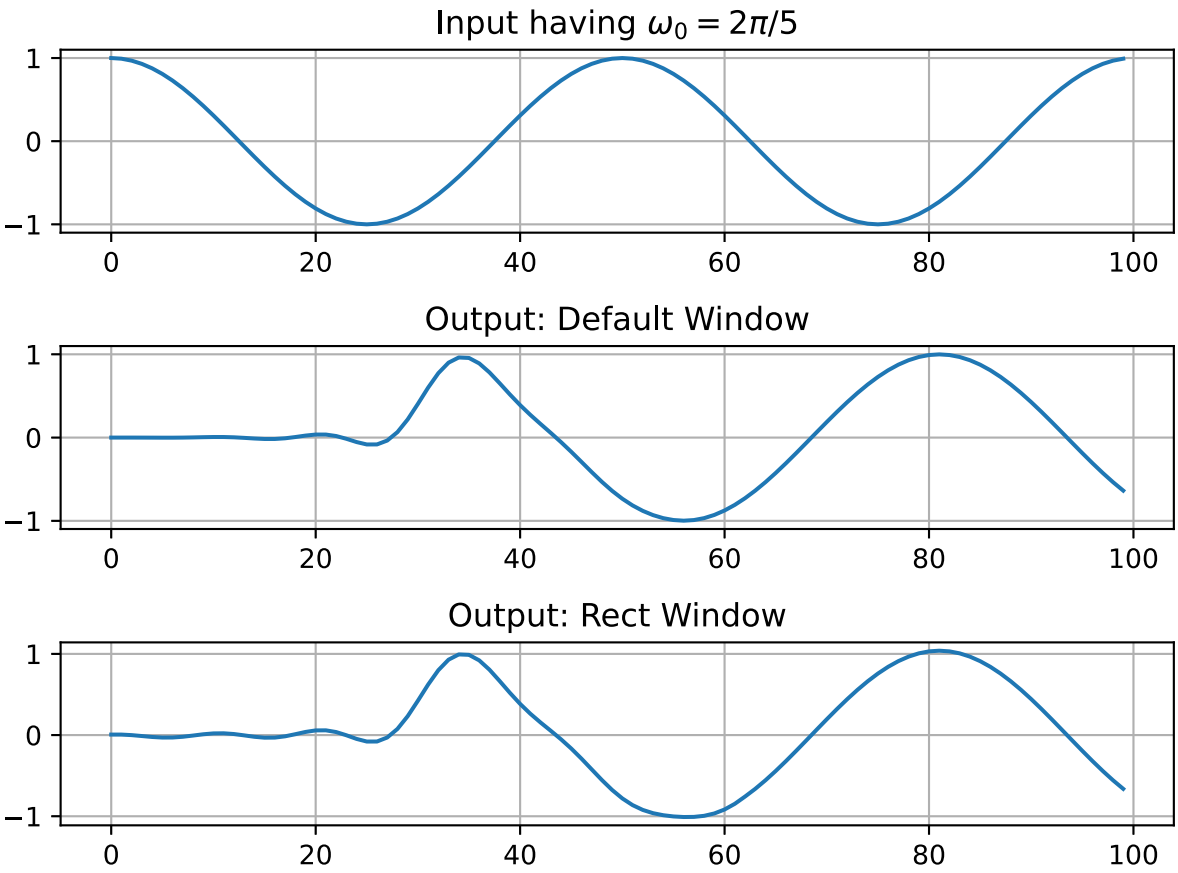
```
b_rect
```

```
array([  6.16110354e-03,  -7.95870419e-18,  -6.58600723e-03,
        -1.10369687e-02,  -1.14457453e-02,  -7.34593114e-03,
         7.95870419e-18,   7.95809207e-03,   1.34363097e-02,
         1.40470510e-02,   9.09496236e-03,  -7.95870419e-18,
        -1.00523268e-02,  -1.71686179e-02,  -1.81785366e-02,
        -1.19371381e-02,   7.95870419e-18,   1.36424435e-02,
         2.37719325e-02,   2.57529269e-02,   1.73631100e-02,
        -7.95870419e-18,  -2.12215788e-02,  -3.86293903e-02,
        -4.41478747e-02,  -3.18323683e-02,   7.95870419e-18,
         4.77485524e-02,   1.03011708e-01,   1.54517561e-01,
         1.90994210e-01,   2.04165043e-01,   1.90994210e-01,
         1.54517561e-01,   1.03011708e-01,   4.77485524e-02,
         7.95870419e-18,  -3.18323683e-02,  -4.41478747e-02,
        -3.86293903e-02,  -2.12215788e-02,  -7.95870419e-18,
         1.73631100e-02,   2.57529269e-02,   2.37719325e-02,
         1.36424435e-02,   7.95870419e-18,  -1.19371381e-02,
        -1.81785366e-02,  -1.71686179e-02,  -1.00523268e-02,
        -7.95870419e-18,   9.09496236e-03,   1.40470510e-02,
         1.34363097e-02,   7.95809207e-03,   7.95870419e-18,
        -7.34593114e-03,  -1.14457453e-02,  -1.10369687e-02,
        -6.58600723e-03,  -7.95870419e-18,   6.16110354e-03])
```

```
n = arange(100)
x = cos(2*pi*1/50*n)
y_default = signal.lfilter(b_default,1,x)
y_rect = signal.lfilter(b_rect,1,x)
```

```
subplot(311)
plot(n,x)
title(r'Input having $\omega_0 = 2\pi/5$')
grid();
subplot(312)
#plot(n,x)
plot(n,y_default)
title(r'Output: Default Window')
grid();
subplot(313)
#plot(n,x)
```

```
plot(n,y_rect)
title(r'Output: Rect Window')
grid();
tight_layout()
```



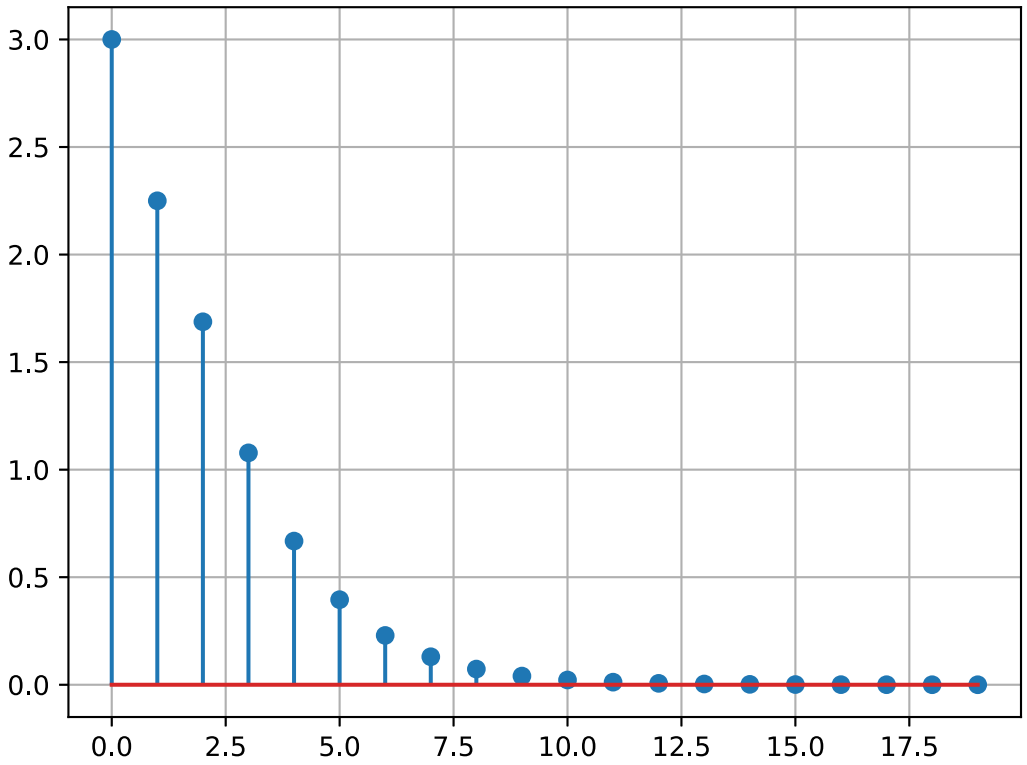
The nominal cutoff frequency for the lowpass filter is $\omega_c = 0.2\pi$. Clearly a design based on the rectangular window `boxcar` is inferior.

More worked Problems with DTFT and Chapter 2 Wrapup

Problem 2.16b

```
n = arange(0,20)
x = (1/2)**n
y_sim = signal.lfilter([3],[1,-1/4,-1/8],x)
#y =
```

```
stem(n,y_sim)
grid();
```




```
y_sim
```

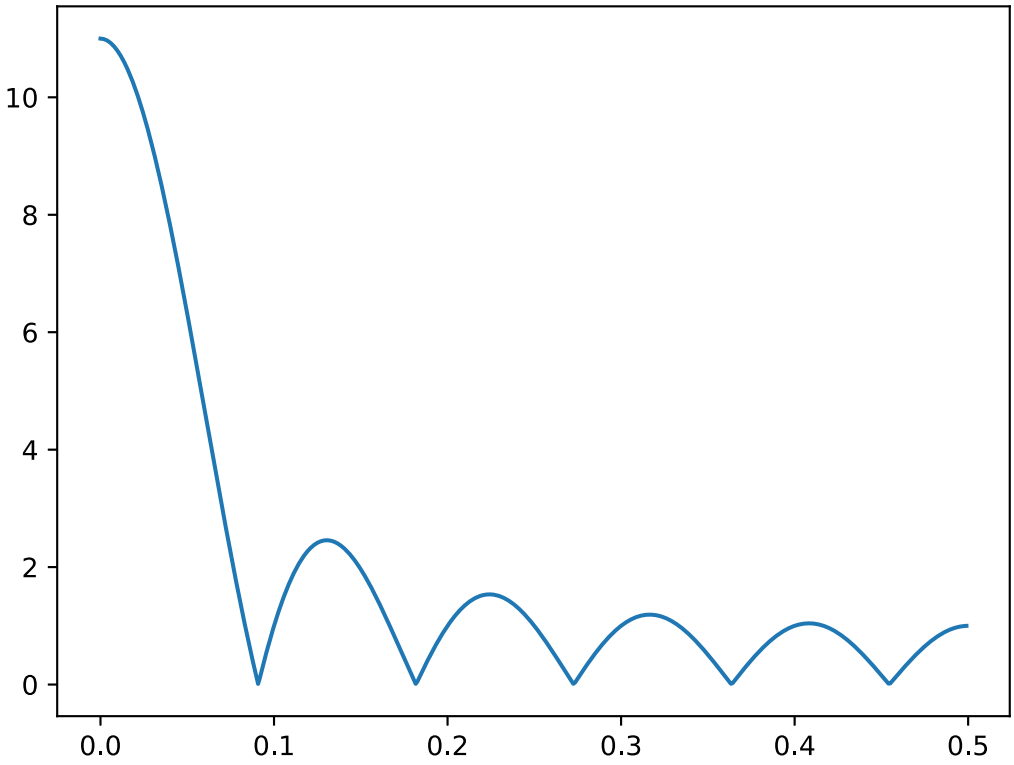
```
array([3.00000000e+00, 2.25000000e+00, 1.68750000e+00, 1.07812500e+00,
       6.67968750e-01, 3.95507812e-01, 2.29248047e-01, 1.30187988e-01,
       7.29217529e-02, 4.03633118e-02, 2.21357346e-02, 1.20441914e-02,
       6.51043653e-03, 3.49934399e-03, 1.87174603e-03, 9.96907242e-04,
       5.28971432e-04, 2.79744447e-04, 1.47501632e-04, 7.75655099e-05])
```

Problem 2.17c

```
M = 10
n = arange(0,M+1)
r = ones(M+1)
#w = x*
f = arange(0,.5,1/1024)
w,R = signal.freqz(r,[1],2*pi*f)
```

```
plot(f,abs(R))
```

```
[<matplotlib.lines.Line2D at 0x21c0e3ad2e0>]
```

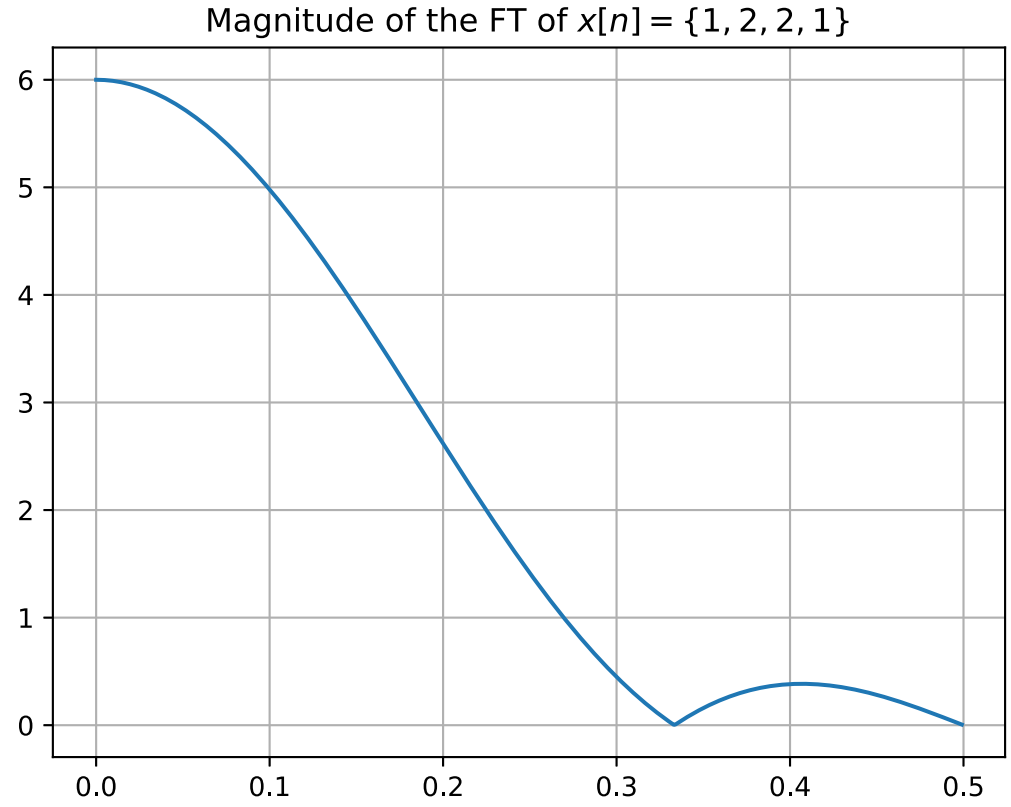


Another Approach

Consider direct evaluation using a loop:

```
x = [1,2,2,1]
f = arange(0,0.5,1/1024)
X = zeros(len(f),dtype=complex128) # Make complex128 to hold X(e^jw) values
for k in range(0,4):
    X += x[k]*exp(-1j*2*pi*f*k)
```

```
plot(f,abs(X))
title(r'Magnitude of the FT of $x[n] = \{1,2,2,1\}$')
grid();
```



Problem 2.44

Creating a function for manipulating $x[n]$. This allows you to answer parts (d) and (e) a little bit easier. As an example function consider

$$x[n] = (3+1j)\delta[n+1] + (2-3j)\delta[n] + (1+1j)\delta[n-1]$$

```
def x_pulse(n):
    """
    Create a function consisting of three impulse functions that fire at
    n = -1, 0, 1, and have complex coefficients.
    """
    x = (3+1j)*ss.dimpulse(n+1) + (2-3j)*ss.dimpulse(n) \
        + (1-1j)*ss.dimpulse(n-1)
    return x
```

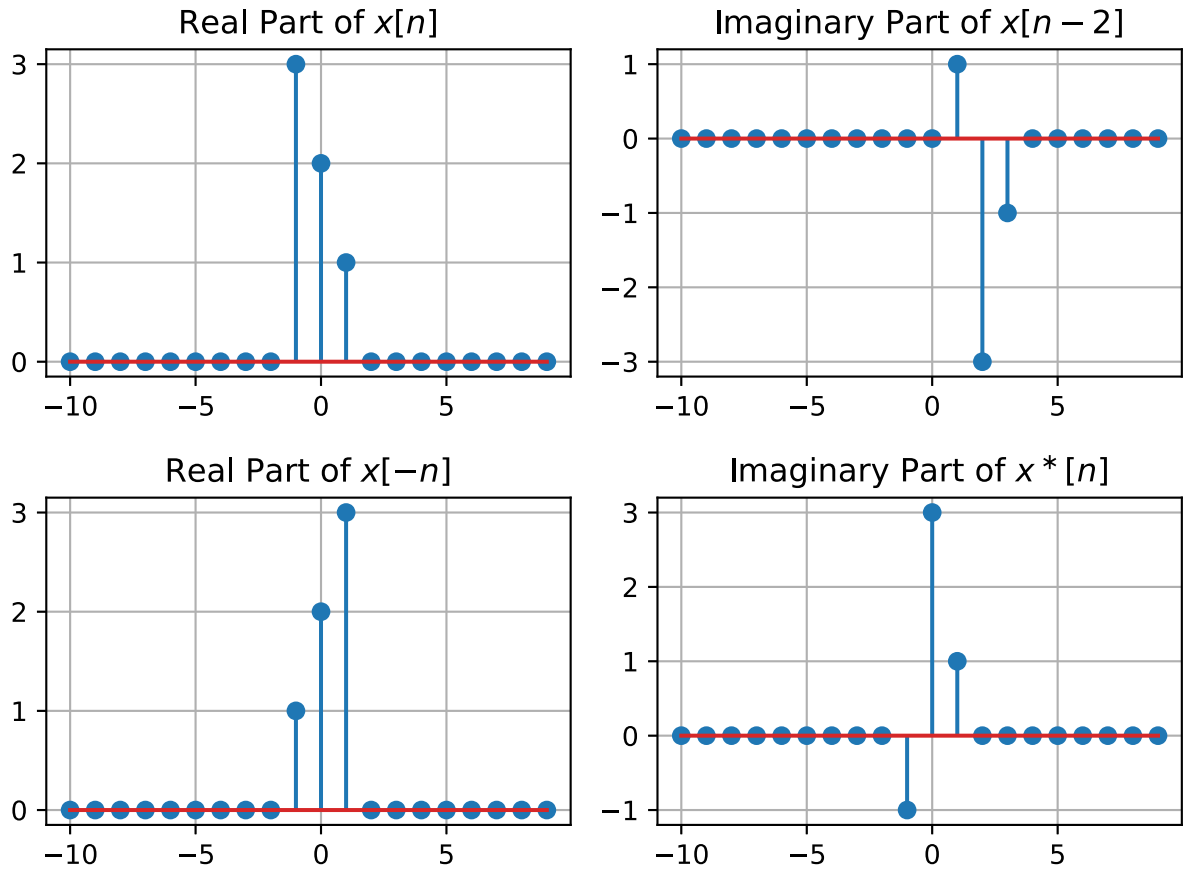
Plot Some Special Cases

Here we plot some special cases of $x[n]$:

1. $\mathrm{Re}x[n]$
2. $\mathrm{Im}x[n-2]$
3. $\mathrm{Re}x[-n]$

4. $\mathrm{Im}x^*[n]$

```
n = arange(-10,10)
subplot(221)
title(r'Real Part of $x[n]$')
stem(n,real(x_pulse(n)))
grid();
subplot(222)
title(r'Imaginary Part of $x[n-2]$')
stem(n,imag(x_pulse(n-2)))
grid();
subplot(223)
title(r'Real Part of $x[-n]$')
stem(n,real(x_pulse(-n)))
grid();
subplot(224)
title(r'Imaginary Part of $x^\ast[n]$')
stem(n,imag(conj(x_pulse(n))))
grid();
tight_layout()
```



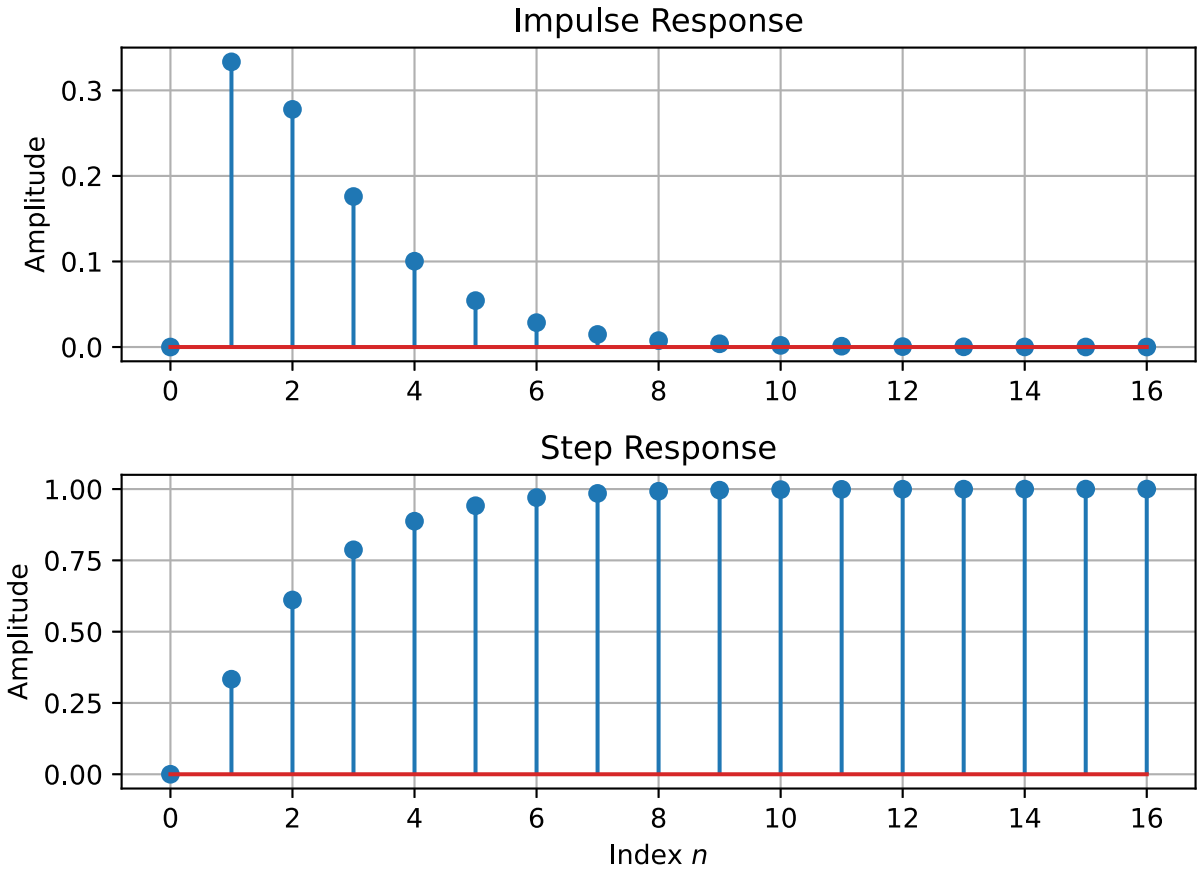
Problem 2.9 LCCDE Impulse and Step Response Modeling

```
n = arange(17)
h = signal.lfilter([0,1/3],[1,-5/6,1/6],ss.dimpulse(n))
y_step = signal.lfilter([0,1/3],[1,-5/6,1/6],ss.dstep(n))
(h,y_step)
```

```
(array([0.00000000e+00, 3.33333333e-01, 2.77777778e-01, 1.75925926e-01,
        1.00308642e-01, 5.42695473e-02, 2.85065158e-02, 1.47105053e-02,
        7.50766842e-03, 3.80463947e-03, 1.91925482e-03, 9.65272441e-04,
        4.84517897e-04, 2.42886174e-04, 1.21652162e-04, 6.08957728e-05,
        3.04711170e-05]),
 array([0.        , 0.33333333, 0.61111111, 0.78703704, 0.88734568,
        0.94161523, 0.97012174, 0.98483225, 0.99233992, 0.99614456,
        0.99806381, 0.99902908, 0.9995136 , 0.99975649, 0.99987814,
        0.99993903, 0.99996951]))
```

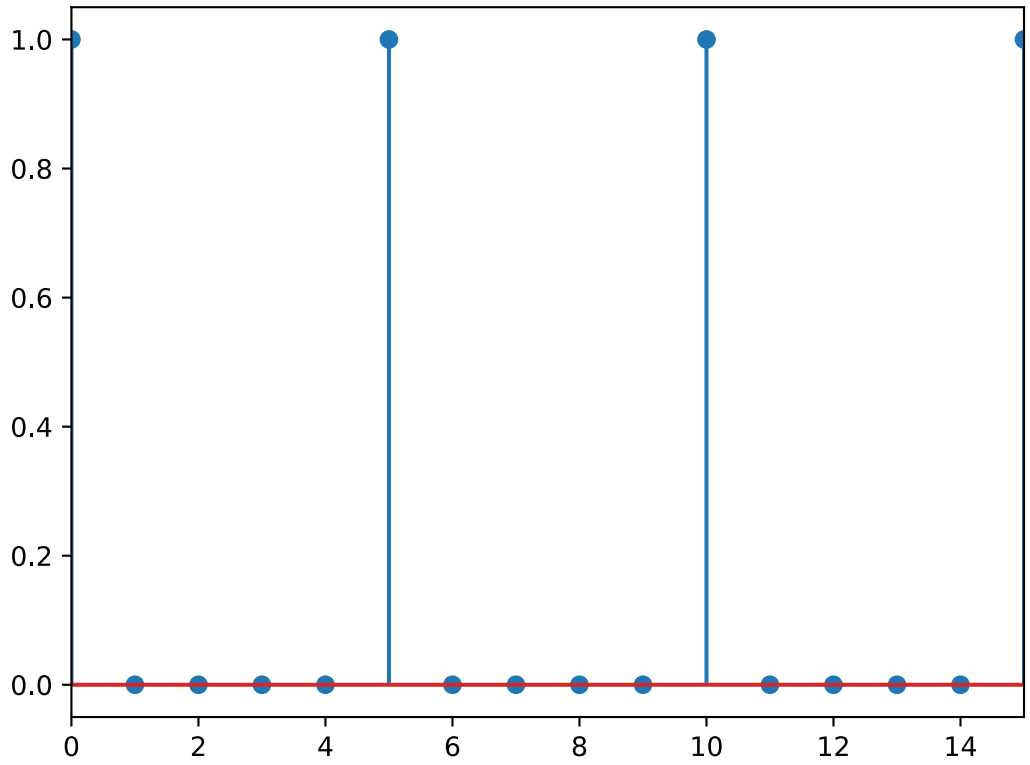
```
subplot(211)
stem(h)
title(r'Impulse Response')
ylabel(r'Amplitude')

grid();
subplot(212)
stem(y_step)
title(r'Step Response')
ylabel(r'Amplitude')
xlabel(r'Index $n$')
grid();
tight_layout()
```




```
x = ones(100)
y = ss.upsample(x,5)
stem(y[0:100])
xlim([0,15])
```

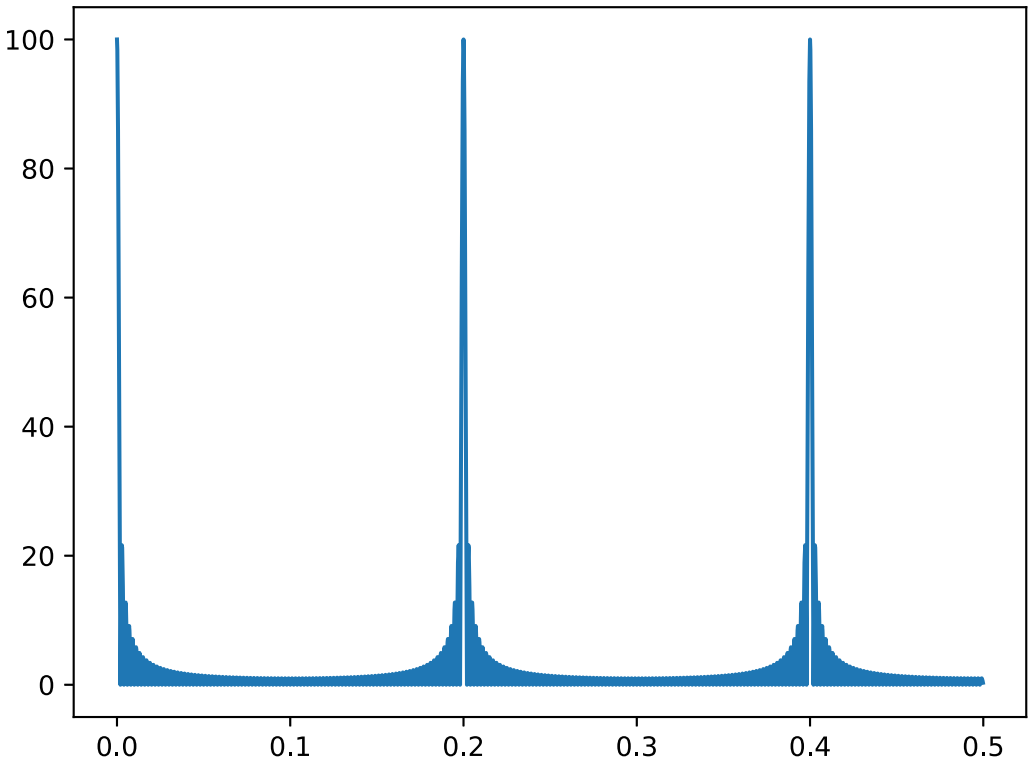
```
(0.0, 15.0)
```




```
f = arange(0,.5,1/5000)
w,Y = signal.freqz(y,1,2*pi*f)
```

```
plot(f,abs(Y))
```

```
[<matplotlib.lines.Line2D at 0x21c0f58eaf0>]
```


100
80
60
40
20
0
0.0
0.1
0.2
0.3
0.4
0.5