5650_chapter3_fa2025

October 13, 2025

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

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

1 Chapter 3: The z-Transform

The scipy stack offers some nice capability for z-transform work. With `ss.py` more capabilty is added.

1.1 Basics

Consider

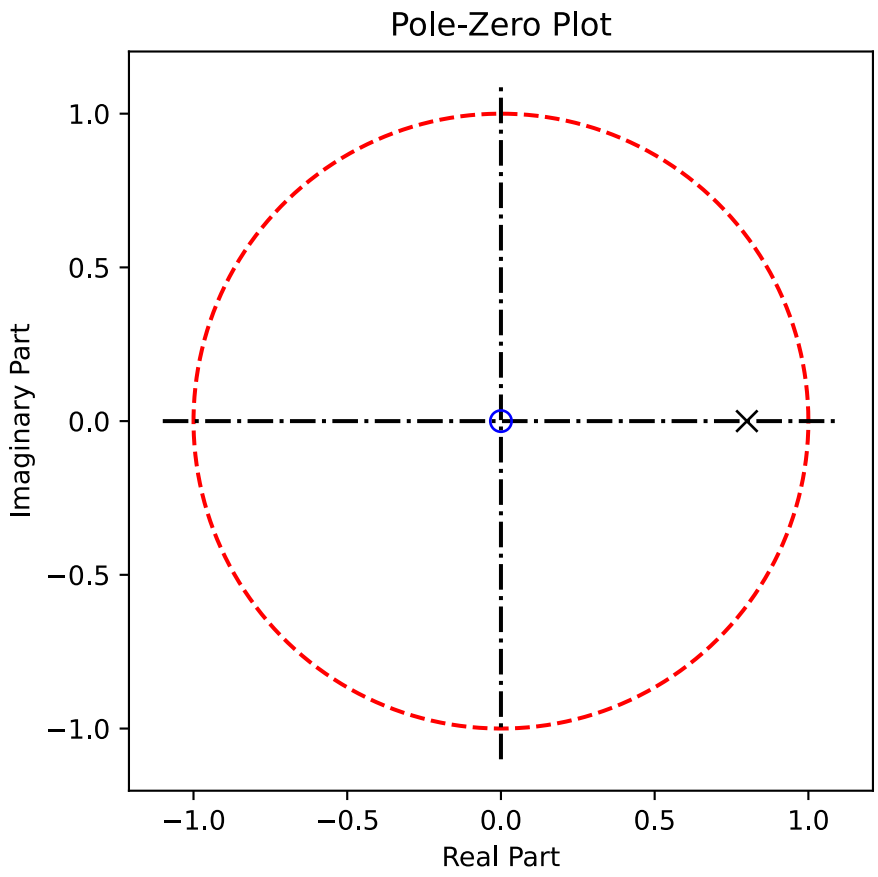
$$x[n] = a^n u[n], \ |a| < 1 \tag{1}$$

$$X(z) = \frac{1}{1 - az^{-1}}, \ \text{ROC:}|z| > |a| \tag{2}$$

Using `ss.zplane(b,a)` you can view the pole-zero plot. The ROC is not plotted however.

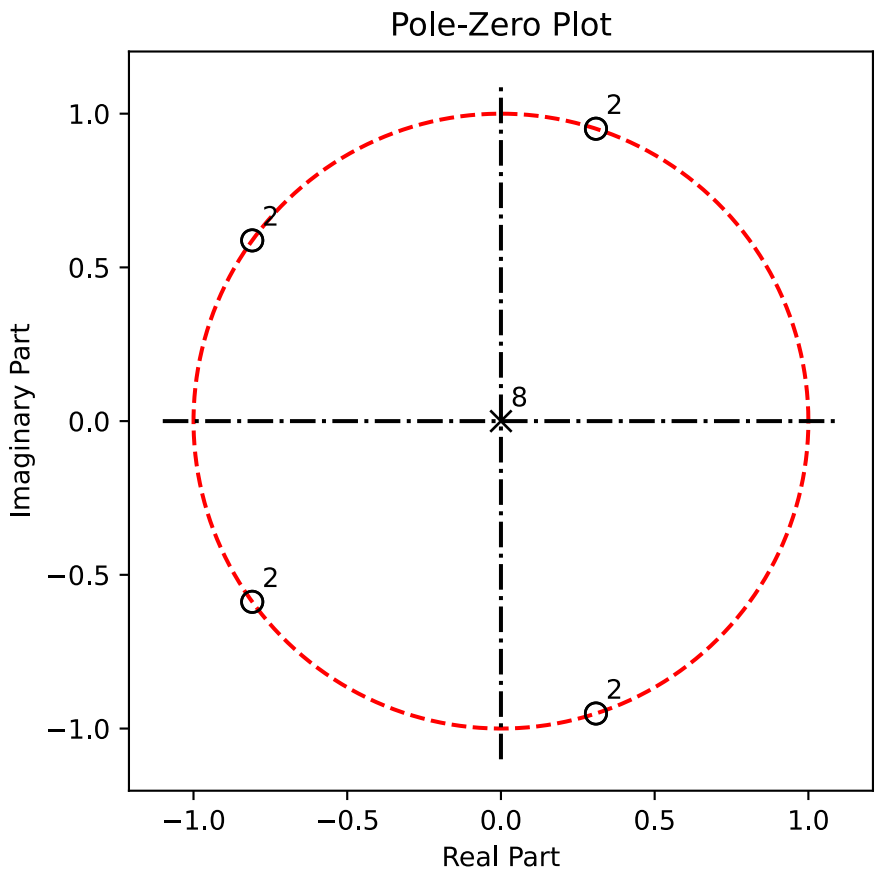
[2]:
```
a = 0.8
ss.zplane([1],[1,-a])
#tight_layout()
```

[2]:
```
(0, 1)
```



[3]:
```
ss.zplane([1,2,3,4,5,4,3,2,1],1)
```

[3]: (8, 0)



1.1.1 Rectangular Window

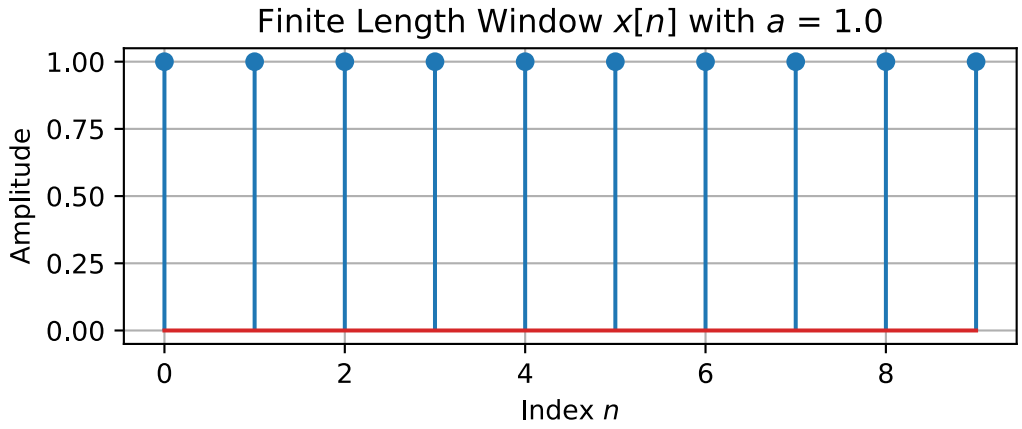
An important finite length sequence is the rectangular window of length N with exponential weighting a^n,

$$x[n] = \begin{cases} a^n, & 0 \leq n \leq N-1 \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

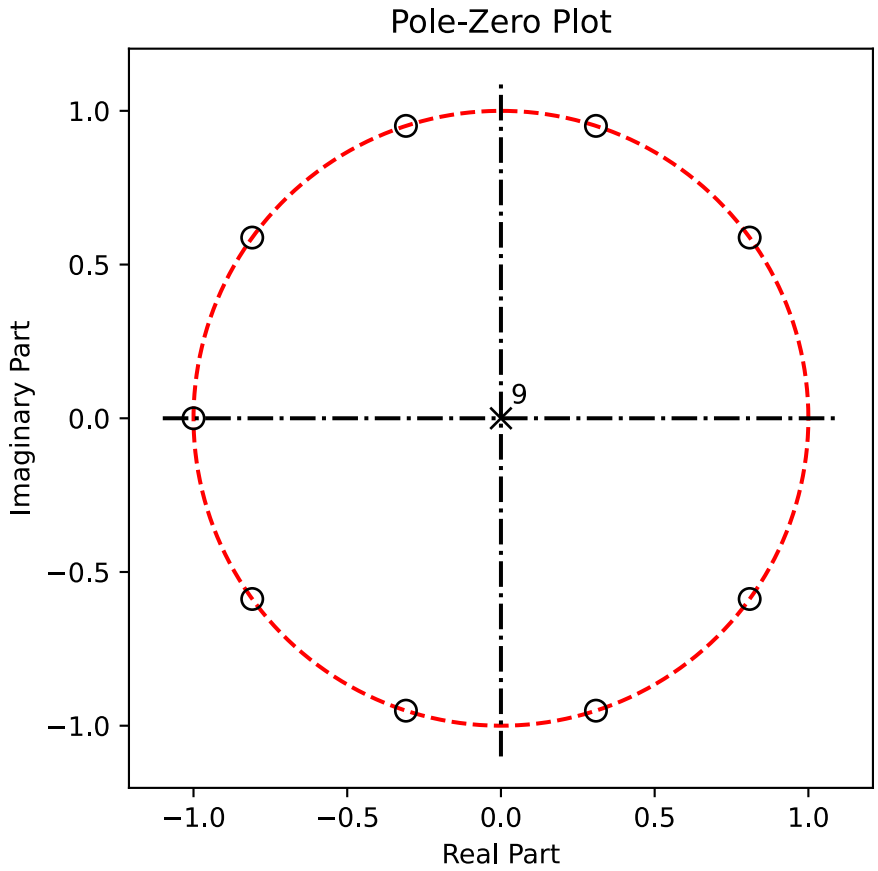
[5]:
```
N = 10
a = 1.0
n = arange(0,N)
x = a**n
figure(figsize=(6,2))
stem(n,x)
title(r'Finite Length Window $x[n]$ with $a$ = {}'.format(a))
ylabel(r'Amplitude')
xlabel(r'Index $n$')
grid();
figure(figsize=(6,3));
```

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

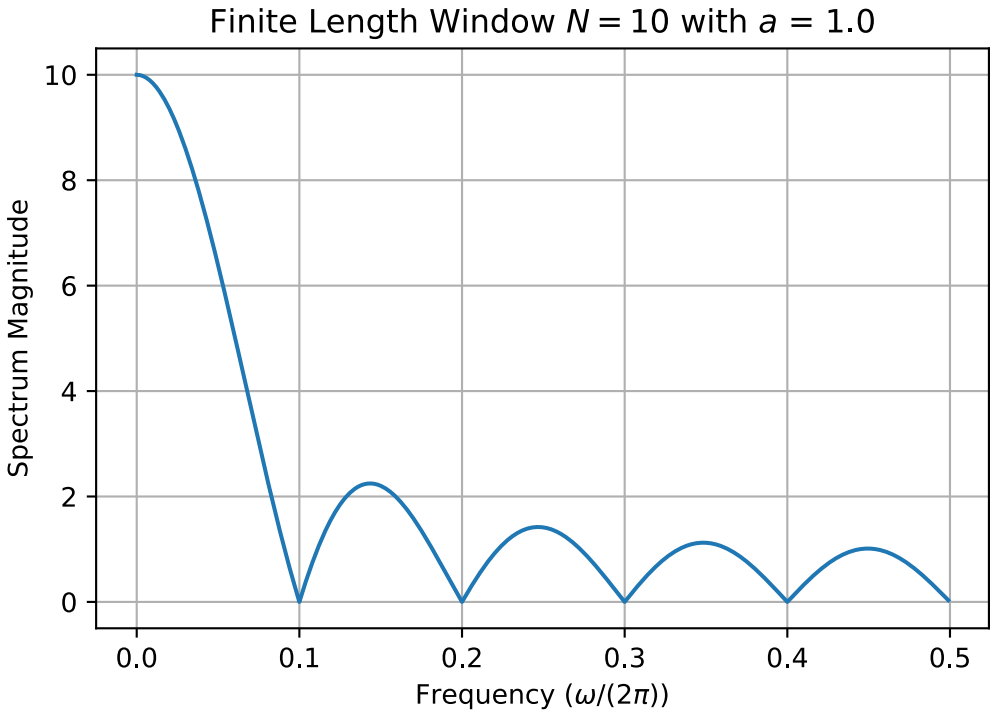
[5]: (9, 0)




```
<Figure size 600x300 with 0 Axes>
```



[5]:
```
f = arange(0,.5,.001)
w,X = signal.freqz(x,1,2*pi*f)
plot(f,abs(X))
title(r'Finite Length Window $N =${} with $a$ = {}'.format(N,a))
ylabel(r'Spectrum Magnitude')
xlabel(r'Frequency ($\omega/(2\pi)$)')
grid();
```



2 Inverse Transforms

```
Signature: signal.residuez(b, a, tol=0.001, rtype='avg')
Docstring:
Compute partial-fraction expansion of b(z) / a(z).

If `M` is the degree of numerator `b` and `N` the degree of denominator
`a`::

          b(z)     b[0] + b[1] z**(-1) + ... + b[M] z**(-M)
  H(z) = ------ = ----------------------------------------
          a(z)     a[0] + a[1] z**(-1) + ... + a[N] z**(-N)
```

```
then the partial-fraction expansion H(z) is defined as::

              r[0]                   r[-1]
      = ---------------- + ... + ----------------- + k[0] + k[1]z**(-1) ...
        (1-p[0]z**(-1))         (1-p[-1]z**(-1))

If there are any repeated roots (closer than `tol`), then the partial
fraction expansion has terms like::

          r[i]              r[i+1]                    r[i+n-1]
    -------------- + ------------------ + ... + ------------------
    (1-p[i]z**(-1))  (1-p[i]z**(-1))**2          (1-p[i]z**(-1))**n

This function is used for polynomials in negative powers of z,
such as digital filters in DSP.  For positive powers, use `residue`.

Parameters
----------
b : array_like
    Numerator polynomial coefficients.
a : array_like
    Denominator polynomial coefficients.

Returns
-------
r : ndarray
    Residues.
p : ndarray
    Poles.
k : ndarray
    Coefficients of the direct polynomial term.
```

2.1 Example 1

- Consider the z-transform

$$X(z) = \frac{1 + 2z^{-1} + z^{-2}}{1 - \frac{3}{2}z^{-1} + \frac{1}{2}z^{-2}}, \text{ ROC:}|z| > 1$$

- Using `residuez()` the solution is

[6]:
```
R,P,K = signal.residuez([1, 2, 1],[1, -3./2, 1./2])
print(R)
print(P)
print(K)
```

```
[-9.  8.]
[0.5 1. ]
[2.]
```

2.2 Example 2

- As a second example consider the z-transform

$$X(z) = \frac{1}{(1+z^{-1})(1-z^{-1})^2}, \text{ ROC:}|z| > 1$$

- Again using `residuez()` the solution is
- To multiply out the denominator polynomials we use the function `signal.convolve(a,b)` to effectively multiply the polynomial coefficients

[7]:
```
signal.convolve([1,1],signal.convolve([1,-1],[1,-1]))
```

[7]:
```
array([ 1, -1, -1,  1])
```

[8]:
```
R,P,K = signal.residuez([1],signal.convolve([1,1],
                  signal.convolve([1,-1],[1,-1])))
print(R)
print(P)
print(K)
```

```
[0.25 0.25 0.5 ]
[-1.  1.  1.]
[]
```

2.3 Example 3

Consider

$$X(z) = \frac{1+z^{-1}}{1-z^{-1}+\frac{1}{2}z^{-2}}$$

[9]:
```
A,p,K = signal.residuez([1,1],[1,-1,1/2])
(A,p,K)
```

[9]:
```
(array([0.5-1.5j, 0.5+1.5j]),
 array([0.5+0.5j, 0.5-0.5j]),
 array([], dtype=float64))
```

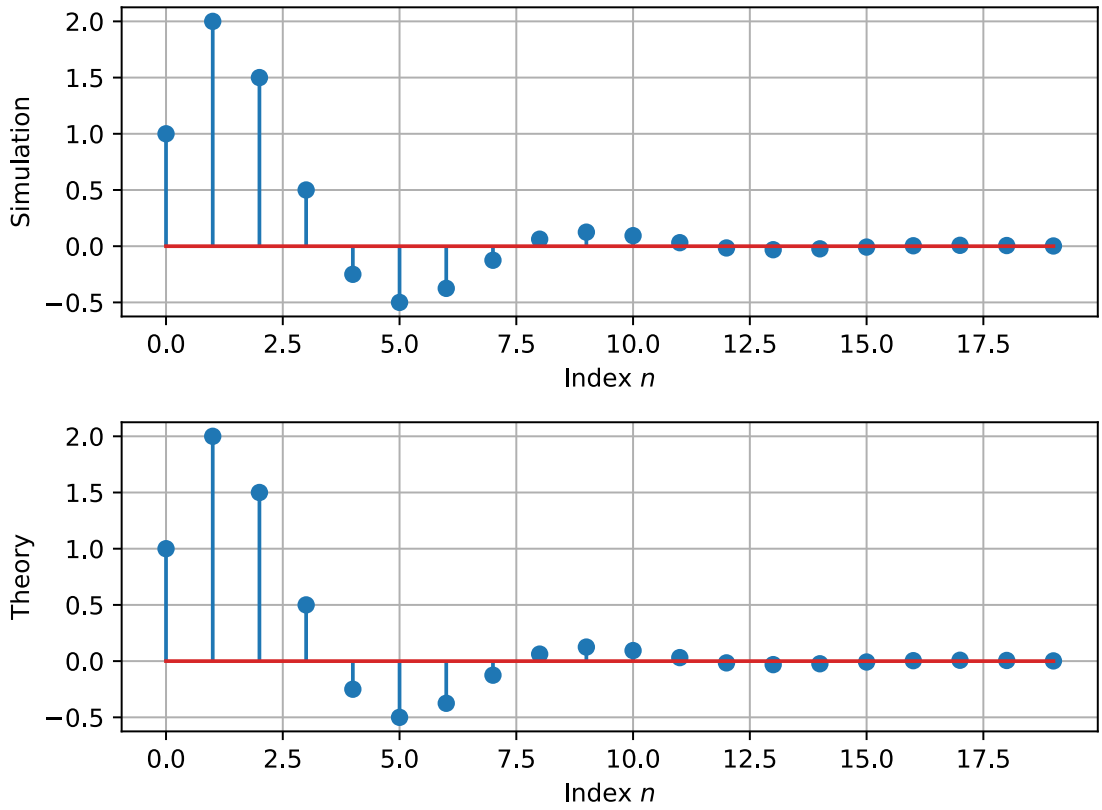
We can use the partial fraction expansion results to obtain the theoretical $x[n]$ and then compare that with a simulated result for $x[n]$ obtained by driving a filter having the same `b` and `a` polynomial coefficients as tion $X(z)$, with input $\delta[n]$. This simulation result follows from the convolution theorem for z-transforms:

$$x[n] = \mathcal{Z}^{-1}\{1 \cdot X(z)\} = \delta[n] * \mathcal{Z}^{-1}\{X(z)\} = \delta[n] \Rightarrow \text{LCCDE (lfilter)}$$

with the convolution now represented as $\delta[n]$ passing through a difference equation representation of $X(z)$ when viewing it as a system rather than a signal.

[7]:
```
n = arange(0,20)
subplot(211)
u = ss.dimpulse(n)
```

```
x = signal.lfilter([1,1],[1,-1,1/2],u) # pass delta through LCCDE
stem(n,x)
ylabel(r'Simulation')
xlabel(r'Index $n$')
grid();
subplot(212)
x_thy = sqrt(10)*(1/sqrt(2))**n*cos(pi*n/4 - 71.565*pi/180)
stem(n,x_thy)
ylabel(r'Theory')
xlabel(r'Index $n$')
grid();
tight_layout()
```



3 Initial Conditions and the One-Sided z-Transform

To set initial conditions in `signal.lfilter()` we use `lfiltic`

```
Signature: signal.lfiltic(b, a, y, x=None)
Docstring:
Construct initial conditions for lfilter given input and output vectors.
```

```
Given a linear filter (b, a) and initial conditions on the output `y`
and the input `x`, return the initial conditions on the state vector zi
which is used by `lfilter` to generate the output given the input.

Parameters
----------
b : array_like
    Linear filter term.
a : array_like
    Linear filter term.
y : array_like
    Initial conditions.

    If ``N = len(a) - 1``, then ``y = {y[-1], y[-2], ..., y[-N]}``.

    If `y` is too short, it is padded with zeros.
x : array_like, optional
    Initial conditions.

    If ``M = len(b) - 1``, then ``x = {x[-1], x[-2], ..., x[-M]}``.

    If `x` is not given, its initial conditions are assumed zero.

    If `x` is too short, it is padded with zeros.

Returns
-------
zi : ndarray
    The state vector ``zi = {z_0[-1], z_1[-1], ..., z_K-1[-1]}``,
    where ``K = max(M, N)``.
```

As a simple example we take

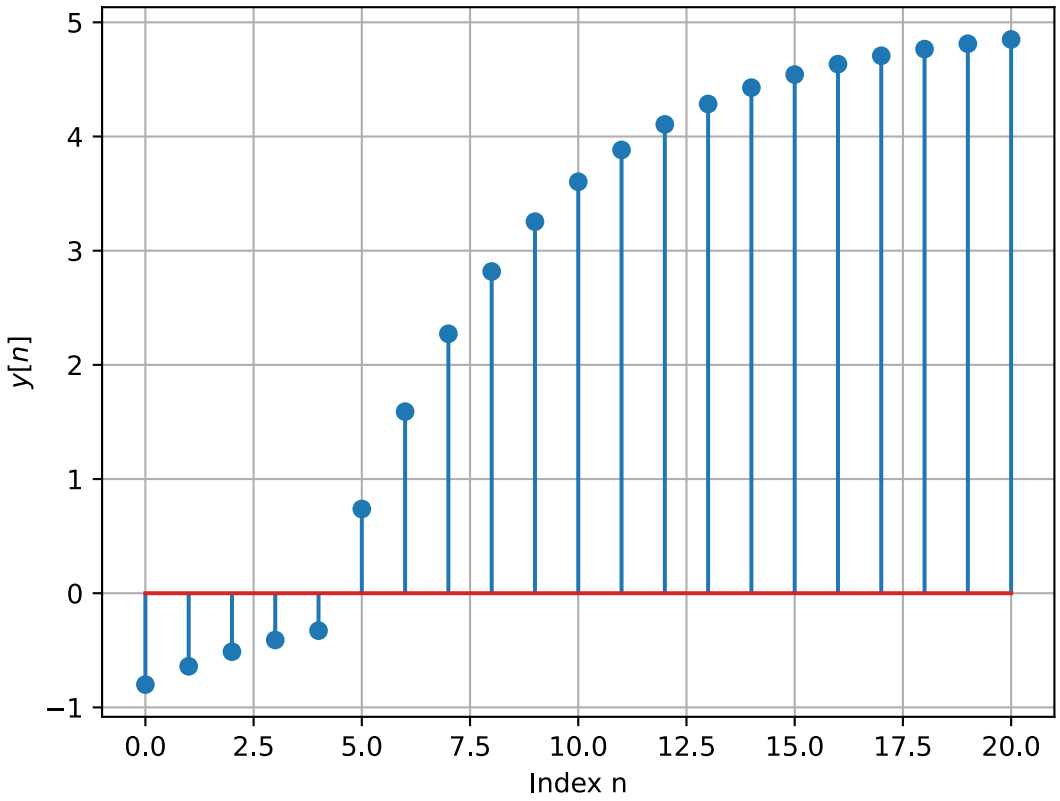
$$H(Z) = \frac{1}{1 - 0.8z^{-1}}$$

and give the system an initial condition of $y[-1] = -0.8$. We drive the system with

$$x[n] = u[n - 5]$$

[8]:
```
n = arange(0,21)
x = ss.dstep(n-5)
y,zf = signal.lfilter([1],[1,-0.8],x,zi=[-1*0.8])
stem(n,y)
grid()
xlabel(r'Index n')
ylabel(r'$y[n]$')
```

[8]:
```
Text(0, 0.5, '$y[n]$')
```



The initial output state of -0.8 begins to decay to zero before the input is applied at $n = 5$. The final value of the output is heading towards 5 as $n \to \infty$.

3.1 Using the Initial Conditions to Provide Continuity Between Data Frames

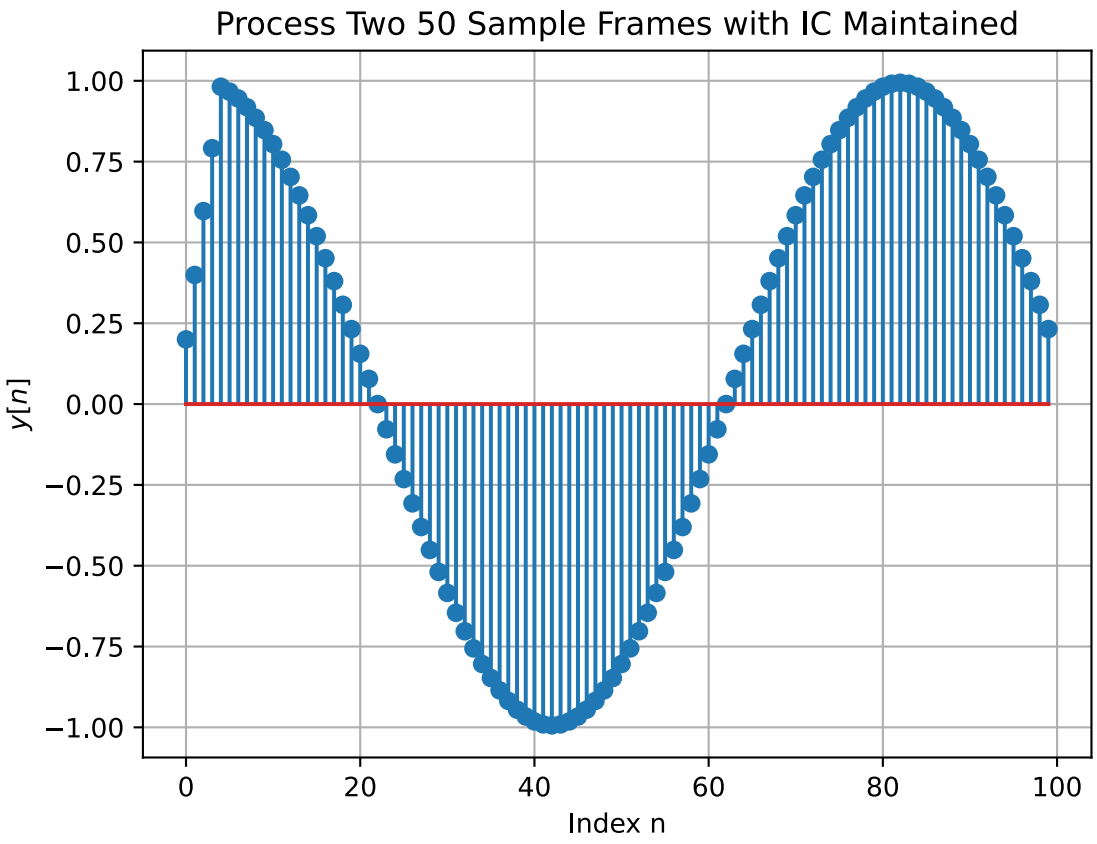
Consider the same 1st-order IIR filter as above. As an input we create a sinusoid at $\omega_0 = \pi/40$. We will send the input through the filter in two blocks of 50 samples each. The filter initial conditions in `zi` will used to start the filter off, then saved back to `zi` at the end of the first block of processing.

[9]:
```
b = [1.0]
a = [1,-0.8]
zi = signal.lfiltic(b,a,[0]) # start with a zero initial condition
```

[10]:
```
b_ma5 = ones(5)/5
a = [1]
zi = signal.lfiltic(b_ma5,a,[0]) # start with a zero initial condition
```

3.1.1 Proper Use of Initial Conditions and Carrying zi Forward on Frame 2

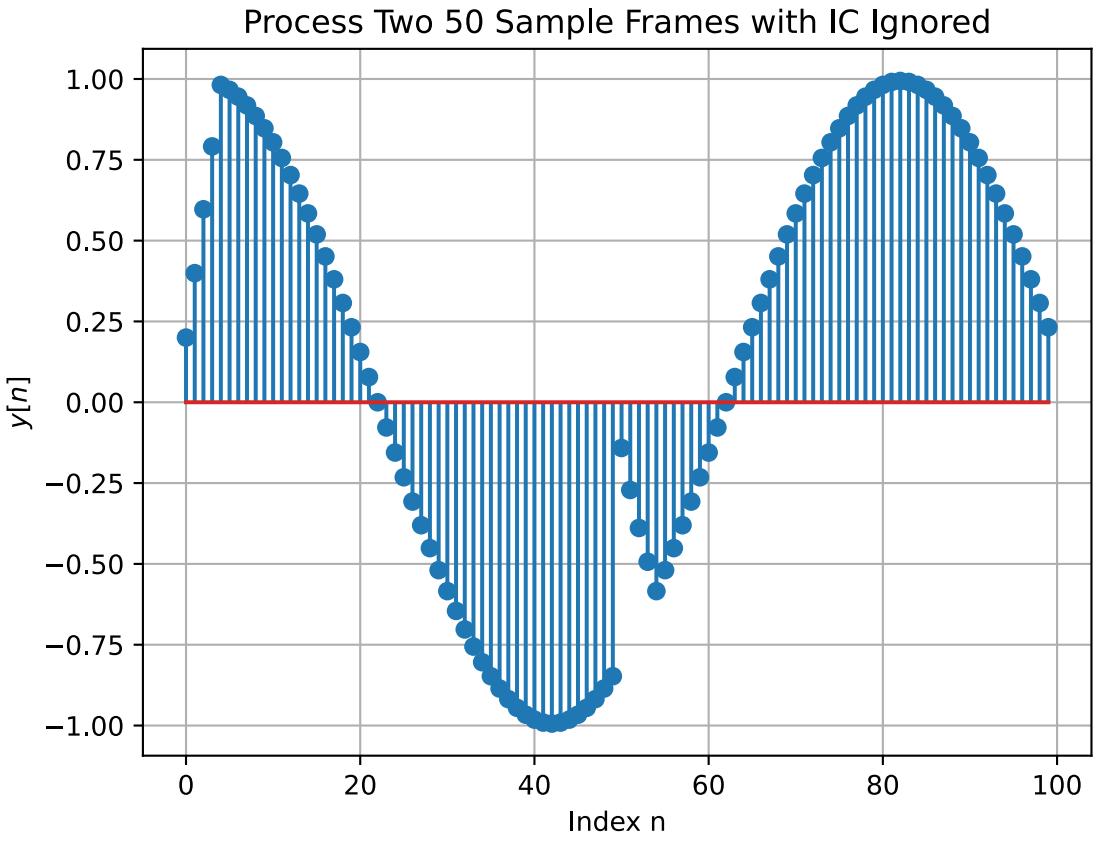
[11]:
```
n = arange(0,100)
x = cos(pi/40*n)
y = zeros_like(x)
y[0:50], zi = signal.lfilter(b_ma5,a,x[0:50],zi=zi)
y[50:], zi = signal.lfilter(b_ma5,a,x[50:],zi=zi)
stem(y)
title(r'Process Two 50 Sample Frames with IC Maintained')
xlabel(r'Index n')
ylabel(r'$y[n]$')
grid()
```



3.1.2 Ignoring Initial Conditions and Not Carrying zi Forward on Frame 2

[12]:
```
n = arange(0,100)
x = cos(pi/40*n)
y = zeros_like(x)
y[0:50] = signal.lfilter(b_ma5,a,x[0:50])
y[50:] = signal.lfilter(b_ma5,a,x[50:])
```

```
stem(y)
title(r'Process Two 50 Sample Frames with IC Ignored')
xlabel(r'Index n')
ylabel(r'$y[n]$')
grid()
```



[15]:
```
roots([1,1/4,-1/8])
```

[15]:
```
array([-0.5 ,  0.25])
```

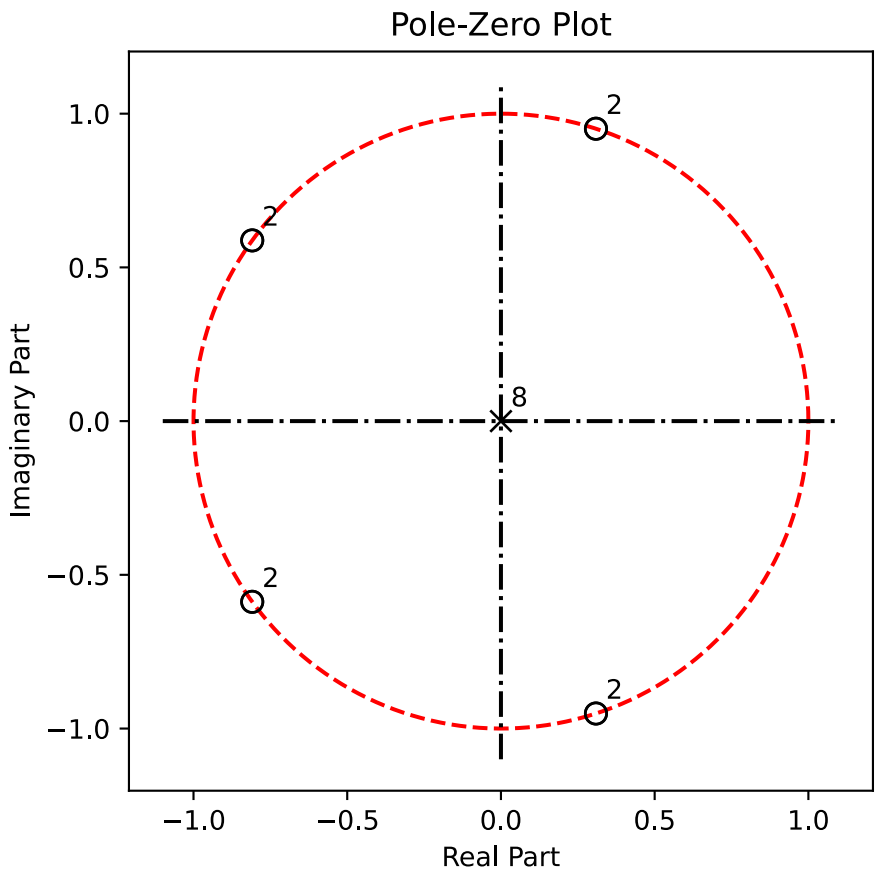
3.2 Homework Checks for 3.3c

[13]:
```
x_c = signal.convolve(ones(5),ones(5))
x_c
```

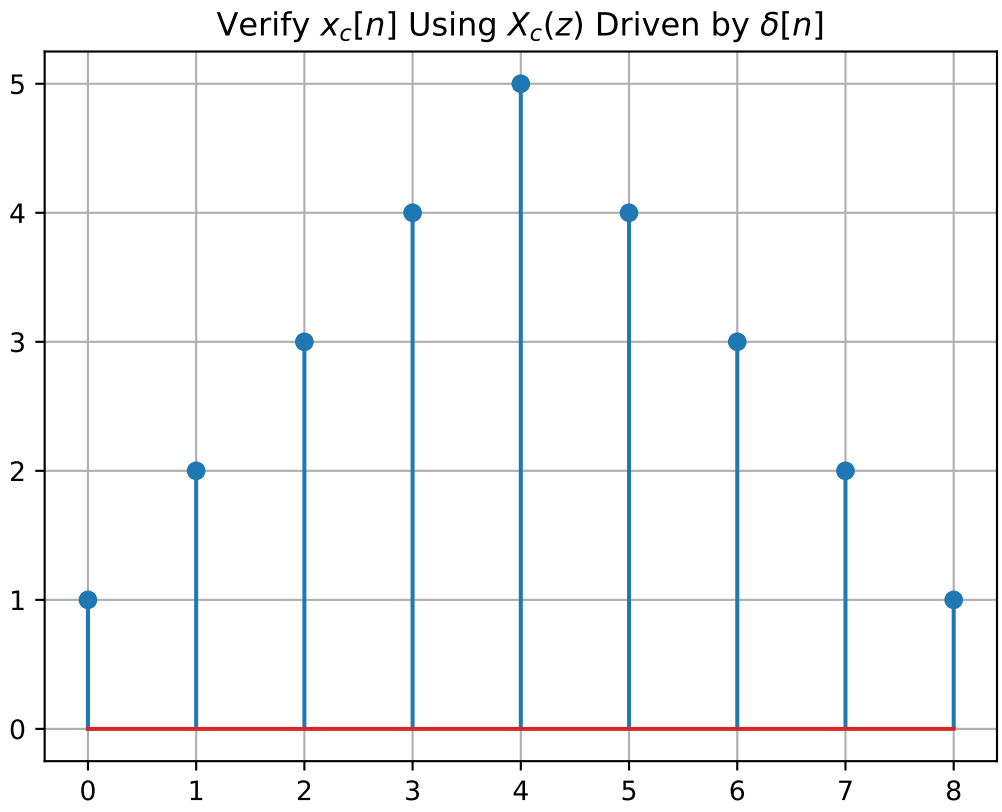
[13]:
```
array([1., 2., 3., 4., 5., 4., 3., 2., 1.])
```

[14]:
```
ss.zplane(x_c,1)
```

[14]:
```
(8, 0)
```

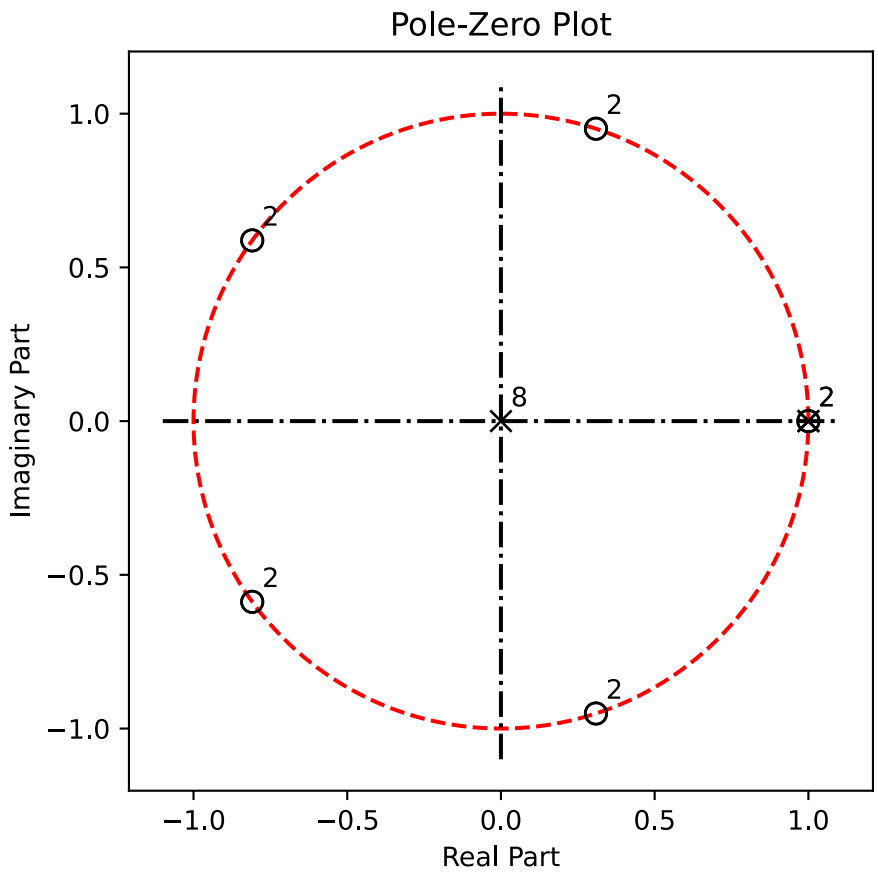


[16]:
```
# N = 5 case
xc_num = signal.convolve([1,0,0,0,0,-1],[1,0,0,0,0,-1])
xc_den = signal.convolve([1,-1],[1,-1])
n = arange(0,9)
x = ss.dimpulse(n)
y = signal.lfilter(xc_num,xc_den,x)
stem(n,y)
title(r'Verify $x_c[n]$ Using $X_c(z)$ Driven by $\delta[n]$')
ylim=([-.1,5.2])
xlim=([-.2,9.2])
grid();
```



[17]:
```
ss.zplane(xc_num,xc_den)
```

[17]: (10, 2)



[18]:
```
ns = 4
m = 6
n = np.arange(-m * ns, m * ns + 1)
a = 0.2
for i in range(len(n)):
    K = abs(1 - 16 * a ** 2 * (n[i] / ns) ** 2)
    if K <= np.finfo(np.float64).eps:
        print(K)
```

```
2.220446049250313e-16
2.220446049250313e-16
```