Structures for Discrete-time Systems

The lecture material for Chapter 6 definitely covers stuctures. In this notebook we deal primarily with IIR and FIR filter design and some fixed-point modeling.

- For IIR filters we explore the use of `second-order-sections` and capabilities provided by `scipy.signal` and `sk_dsp_comm.iir_design_helper`

In [1]:
```
# %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
```

IIR Filter Design

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

In [3]:
```
b_b,a_b,sos_b = iir_d.IIR_lpf(1500,2500,1,70,10000,'butter')
b_c1,a_c1,sos_c1 = iir_d.IIR_lpf(1500,2500,1,70,10000,'cheby1')
b_c2,a_c2,sos_c2 = iir_d.IIR_lpf(1500,2500,1,70,10000,'cheby2')
b_e,a_e,sos_e = iir_d.IIR_lpf(1500,2500,1,70,10000,'ellip')
```

Second-Order Sections (SOS)

The SOS matrix (2D `ndarray`) has six columns. The first three are the numerator coefficients `b` and the last three are the denominator coefficients, `a`.

In [4]:
```
roots(sos_c1[0,:3])
```

Out[4]:
```
array([-1., -1.])
```

In [5]:
```
len(a_c1)
```

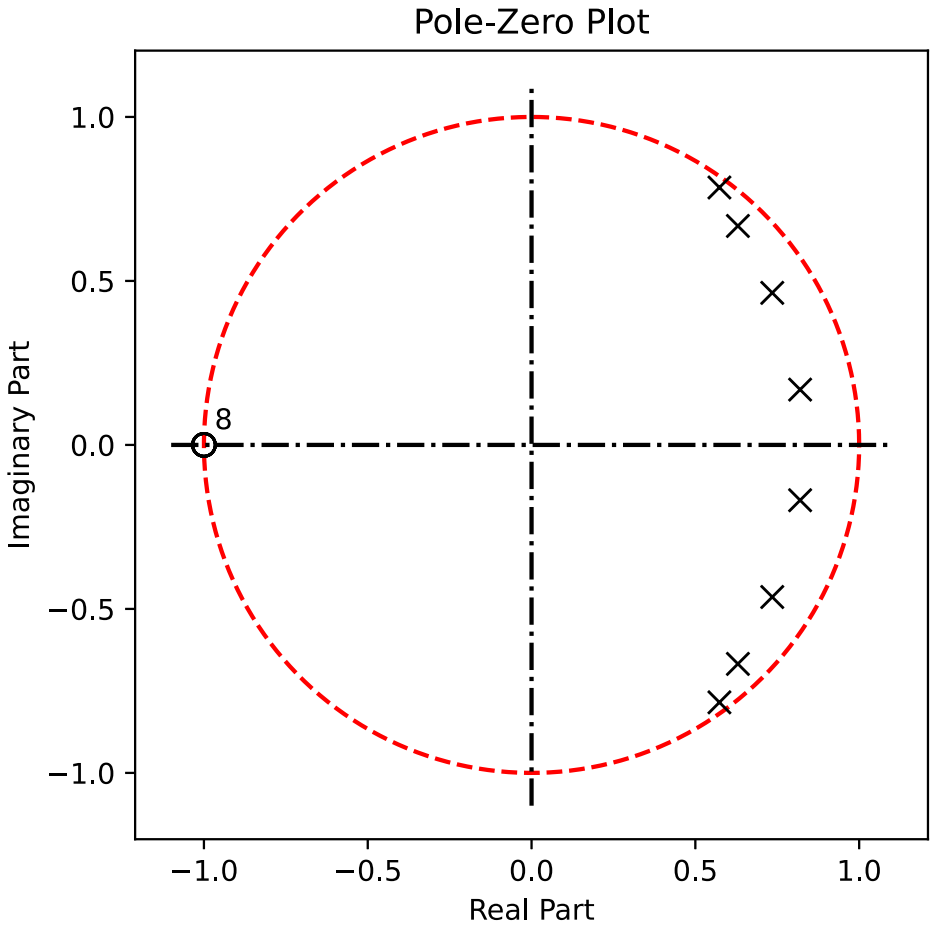
Out[5]:
```
9
```

In [6]:
```
sos_c1
```

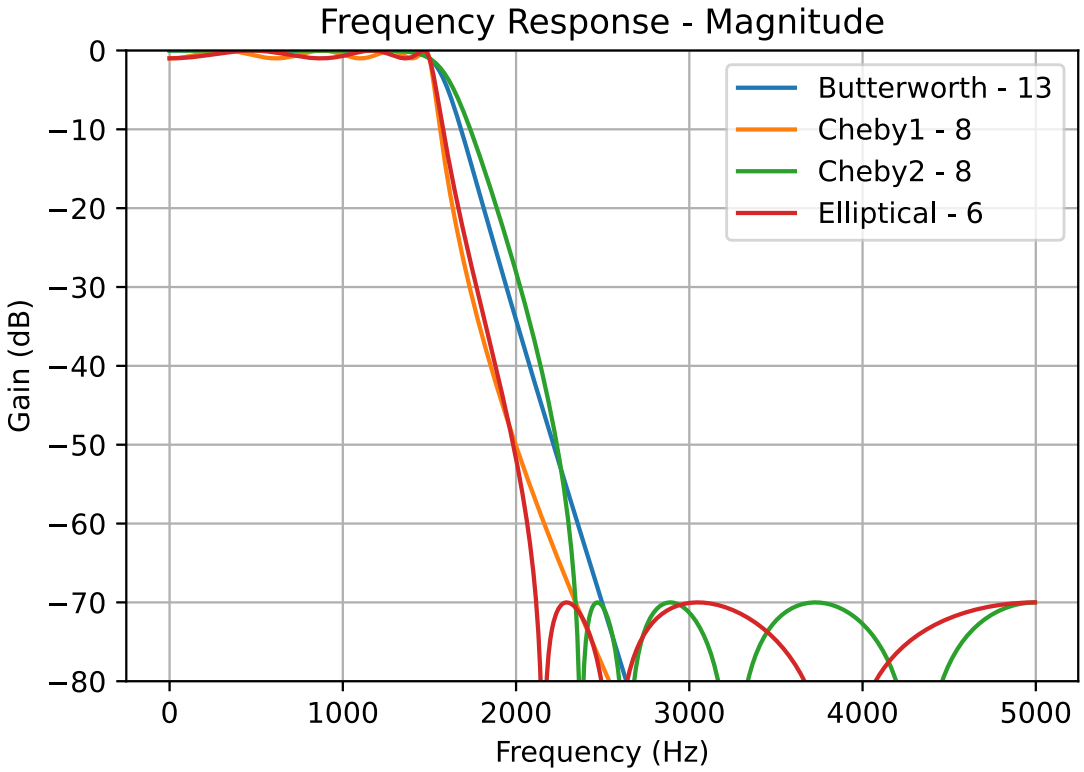
Out[6]:
```
array([[ 2.81418610e-05,  5.62837220e-05,  2.81418610e-05,
         1.00000000e+00, -1.63955493e+00,  7.00480335e-01],
       [ 1.00000000e+00,  2.00000000e+00,  1.00000000e+00,
         1.00000000e+00, -1.46953427e+00,  7.54870129e-01],
       [ 1.00000000e+00,  2.00000000e+00,  1.00000000e+00,
         1.00000000e+00, -1.25970664e+00,  8.42420778e-01],
       [ 1.00000000e+00,  2.00000000e+00,  1.00000000e+00,
         1.00000000e+00, -1.14688834e+00,  9.44850837e-01]])
```

In [7]:
```
iir_d.sos_zplane(sos_c1)
```

Out[7]:
```
(8, 8)
```



In [8]:
```
iir_d.freqz_resp_cas_list([sos_b,sos_c1,sos_c2,sos_e],'dB',10000)
ylim([-80,0])
legend((r'Butterworth - 13',r'Cheby1 - 8',r'Cheby2 - 8',r'Elliptical -
grid();
```



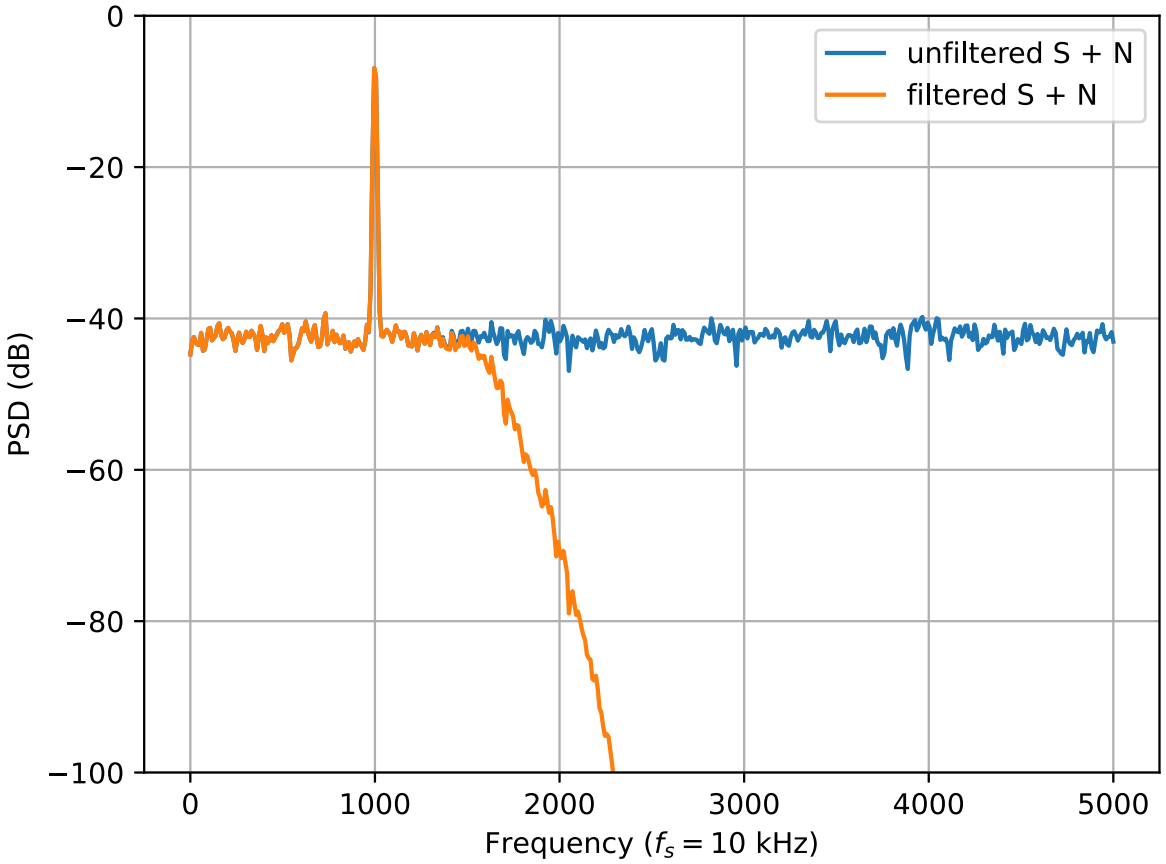
Example: Filtering Using SOS

With `scipy.signal` there is a special version of `lfilter` for working directly with sos filter designs. This method offers greater accuracy as even with double precision coefficients, errors creep in. Below is a simple example of filtering a 1 kHz sinusoid that is corrupted by additive noise

$$x[n] = \cos(2\pi f_0/f_s n) + w[n] \tag{1}$$

where $f_0 = 1$ kHz and $w[n]$ is Gaussian noise having variance $\sigma_w^2 = 0.2$.

In [9]:
```
# Filter using sos
fs = 10000
n = arange(10000)
x = cos(2*pi*1000/fs*n) + 0.2*random.randn(len(n))
y = signal.sosfilt(sos_c2,x)
# psd()
# Px, f = ss.my_psd(x,2**10,fs)
# Py, f = ss.my_psd(y,2**10,fs)
Px, f = ss.psd(x,2**10,fs,scale_noise=False)
Py, f = ss.psd(y,2**10,fs,scale_noise=False)
plot(f,10*log10(Px))
plot(f,10*log10(Py))
legend((r'unfiltered S + N',r'filtered S + N'),loc='best');
xlabel(r'Frequency ($f_s = 10$ kHz)')
ylabel(r'PSD (dB)')
ylim([-100,0])
grid();
```

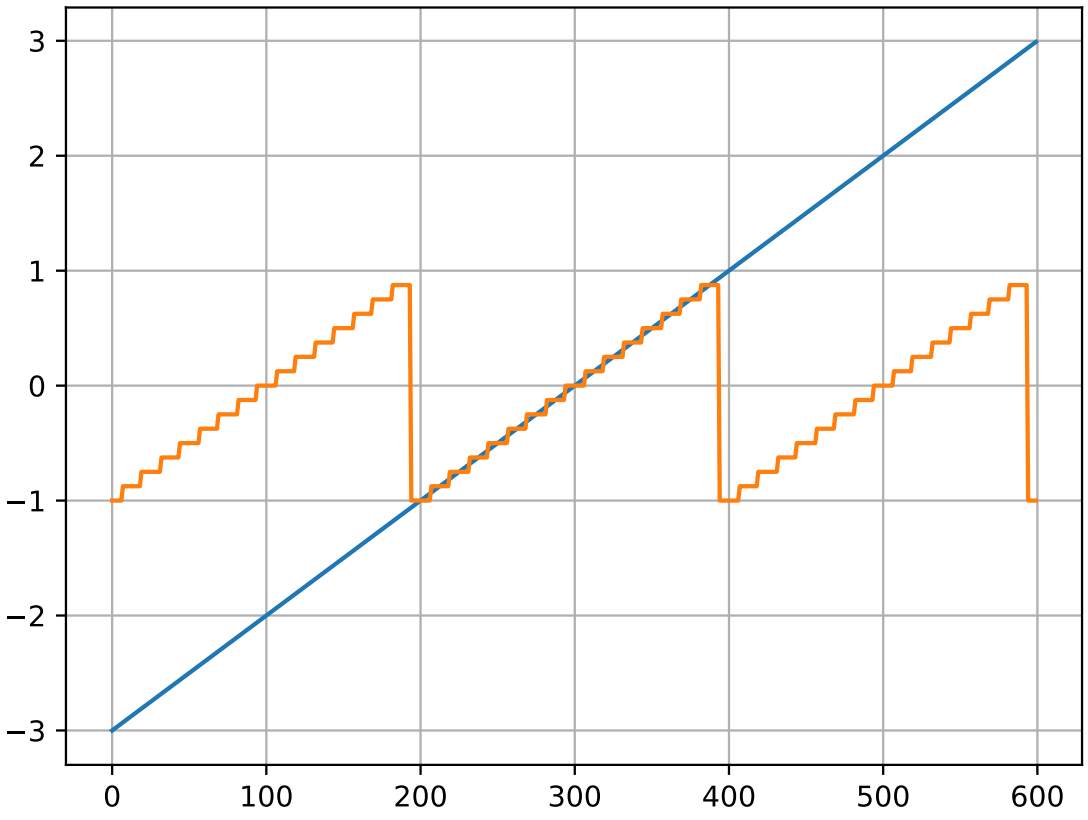


FIR Filters and Fixed-Point

In [10]:
```
import sk_dsp_comm.fir_design_helper as fir_d
```

Simple 2's Complement Overflow

In [11]:
```
s = arange(-3,3,.01)
sq = ss.simple_quant(s,4,1,'over')
plot(s)
plot(sq)
grid();
```



Filter Design with Quantization

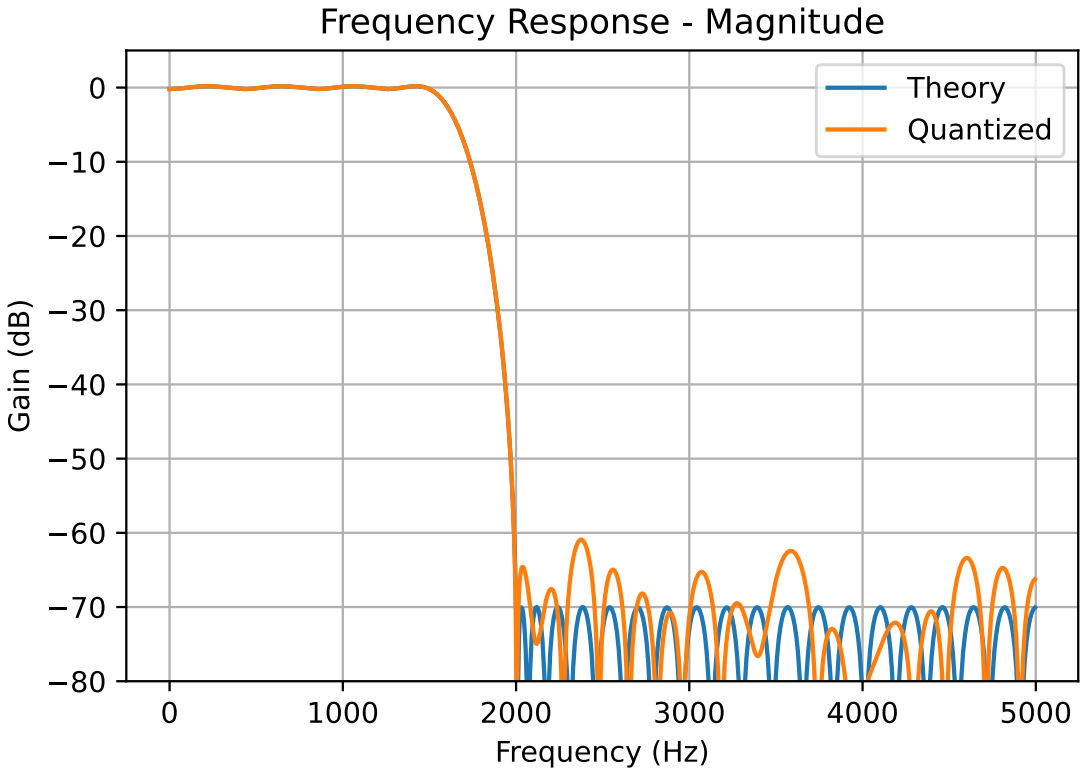
In [12]:
```
# Obtain FIR coefficients
b = fir_d.fir_remez_lpf(1500,2000,0.2,70,10000,n_bump=4)
```

In [13]:
```
max(abs(b))
```

Out[13]:
```
np.float64(0.3369557804095827)
```

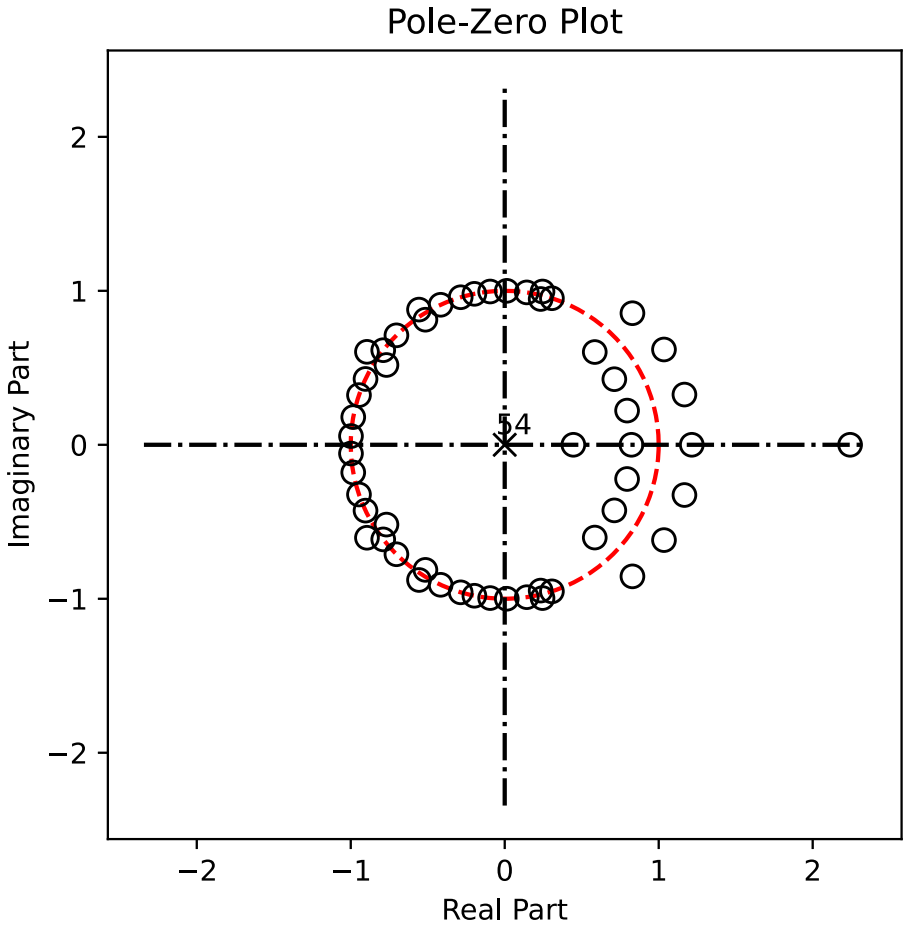
In [14]:
```
# Obtain fixed-point coefficients
b_q = ss.simple_quant(b,14,1.0,'over')
```

In [15]:
```
fir_d.freqz_resp_list([b,b_q],[[1],[1]],'dB',10000)
ylim([-80,5])
legend(('Theory','Quantized'))
grid();
```



In [16]:
```
# ss.zplane(b,[1])
ss.zplane(b_q,[1])
```

Out[16]: (54, 0)



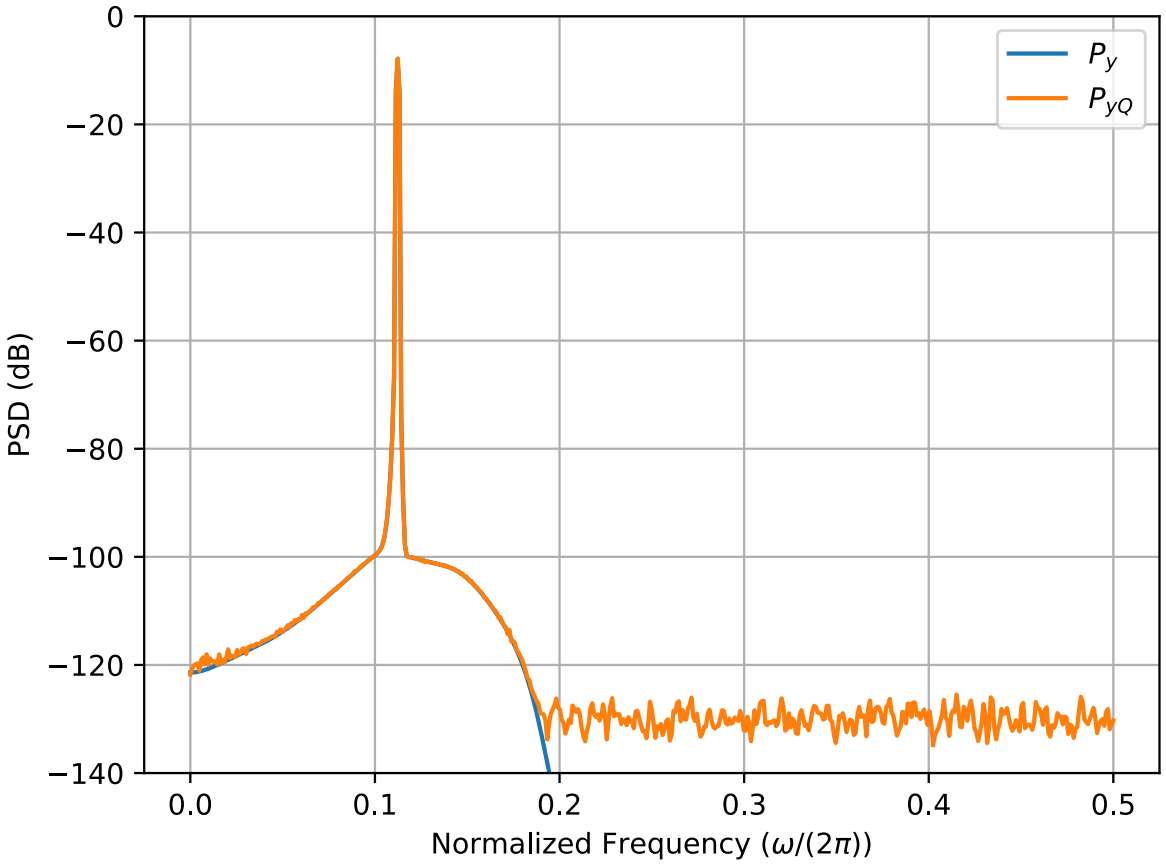
Quantization Noise In Digital Filters

In [17]:
```
def lfilter_fp(bQ,aQ,x,Bx,B,By):
    """
    y Q = lfilter_fp(bQ,aQ,x,Bx,B,By)

    Bx = Input bit width
     B = Sum of products bit width
    By = Output bit width


    Mark Wickert November 2016
    """
    # Quantize input (assume normalized to |x| < 1)
    xQ = ss.simple_quant(x,Bx,1,'over')
    # Initialize the output array
    yQ = zeros(len(xQ))
    # Filter using Direct Form I
    x_state = zeros(len(bQ))
    y_state = zeros(len(aQ)-1)
    # Assume a  very large register for holding the
    # sum of products, both feedforward and feedback.
    # In this case we assume double precision float
```

```
    # Process sample-by-sample in a loop
    for n,xn in enumerate(xQ):
        x_state[0] = xn
        if len(aQ) > 1:
            yQ[n] = ss.simple_quant(sum(x_state * bQ) - \
                    sum(y_state * aQ[1:]),B,1,'over')
        else:
            yQ[n] = ss.simple_quant(sum(x_state * bQ),B,1,'over')
            x_state = roll(x_state,1)
        if len(aQ) > 1:
            y_state = roll(y_state,1)
            y_state[0] = yQ[n]
    if By < B:
        # If the output bit width is less than sum of products width,
        # further quantize the filter output
        yQ = ss.simple_quant(yQ,By,1,'over')
    return yQ
```

In [18]:
```
n = arange(10000)
# x = ss.dstep(n-10)
x = cos(2*pi*0.1123*n)
y = signal.lfilter(b,[1],x*.8)
yQ = lfilter_fp(b_q,[1],x*.8,16,32,16)
```

In [19]:
```
Py, f = ss.psd(y,2**10,1,scale_noise=False)
PyQ, f = ss.psd(yQ,2**10,1,scale_noise=False)
plot(f,10*log10(Py))
plot(f,10*log10(PyQ))
legend((r'$P_y$',r'$P_{yQ}$'),loc='best');
xlabel(r'Normalized Frequency ($\omega/(2\pi)$)')
ylabel(r'PSD (dB)')
ylim([-140,0])
grid();
```



Using `Fxp Math`

To install use:

```
pip install fxpmath
# or (the below did not work without conflict for me)
conda install -c francof2a fxpmath
```

In [20]:
```
from fxpmath import Fxp
```

For details see the *Quick Start*:

- See: https://francof2a.github.io/fxpmath/docs/quick_start.html
- Using `templates` can be useful, although I will not show them in the examples below

```
# Fxp like templates
DATA        = Fxp(None, True, 24, 15)
ADDERS      = Fxp(None, True, 40, 16)
MULTIPLIERS = Fxp(None, True, 24, 8)
CONSTANTS   = Fxp(None, True, 8, 4)

# init
```

```
x1 = Fxp(-3.2).like(DATA)
x2 = Fxp(25.5).like(DATA)
c  = Fxp(2.65).like(CONSTANTS)
m  = Fxp().like(MULTIPLIERS)
y  = Fxp().like(ADDERS)

# do the calc!
m.equal(c*x2)
y.equal(x1 + m)
```

Work with an Fxp `numpy` Array

In the below series of examples I show how the `integer` and `fractional` *bit width* can be used to constrain the fixed-point waveform fidelity of a single sinusoid. Further examples, not yet developed, show the use `Fxp` to quantize say FIR filter coefficients similarly to what was done using `ss.simpleQuant()`.

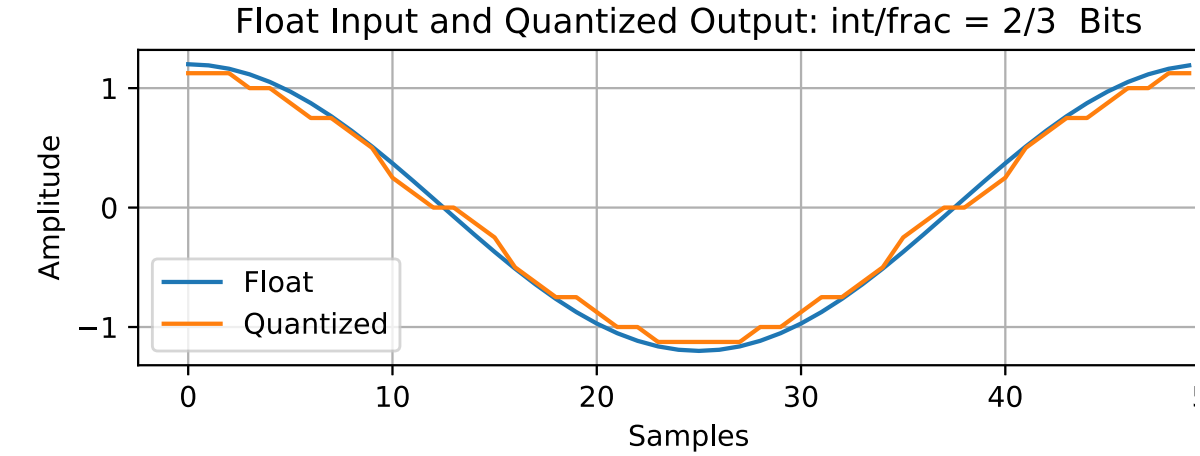
- In the below have switched `overflow` from the default mode of `saturate` to `wrap`
- To look at the errors introduced by fixed-point math you can view the impact of quantization by bringing a value back to the float domain using say `xq.astype(float)`
- Without using `astype()` operations between fixed-point and float become seemed beconde cast to fixed-point, which hides what you are trying to analyze
- To view values in binary form we can use say

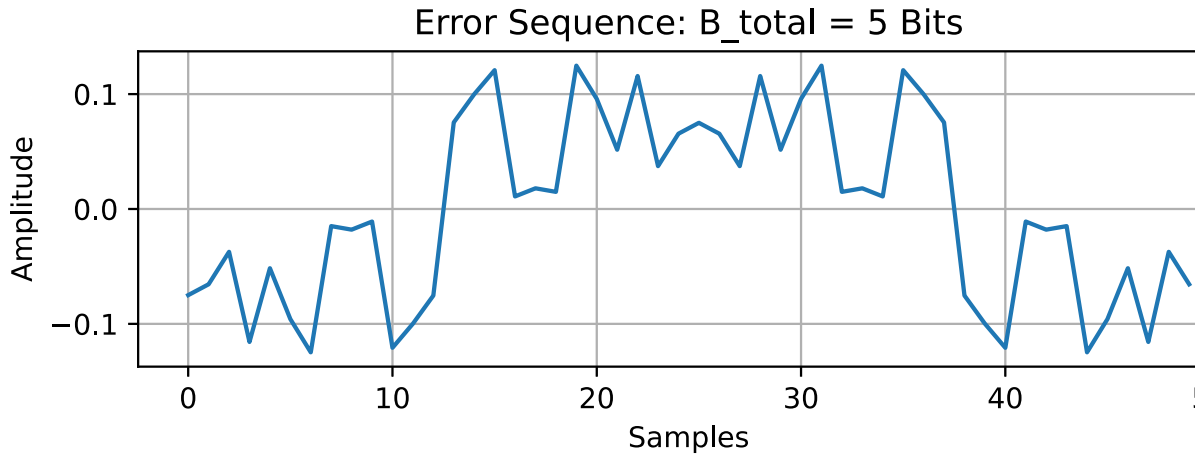
```
x.bin(frac_dot=True)    # binary with fractional dot
```

In [21]:
```
n = arange(50)
x = 1.2*cos(2*pi*n/50)
B_int = 2 # adequate integer headroom to avoid overflow
B_frac = 3 # poor fractional resolution
B_total = B_int + B_frac
xq = Fxp(x, True, n_int=B_int, n_frac=B_frac, overflow='wrap') #Fxp(x)
subplot(211)
plot(n,x,label='Float')
plot(n,xq,label='Quantized')
title(f'Float Input and Quantized Output: int/frac = {B_int}/{B_frac}
ylabel(r'Amplitude')
xlabel(r'Samples')
legend()
grid();
subplot(212)
plot(n,xq.astype(float) -x)
title(f'Error Sequence: B_total = {B_total} Bits')
ylabel(r'Amplitude')
```

```
xlabel(r'Samples')
grid();
tight_layout()
xq[0:20:5].bin(frac_dot=True) # array values in binary with the fracti
```

Out[21]: ['001.001', '000.111', '000.010', '111.110']

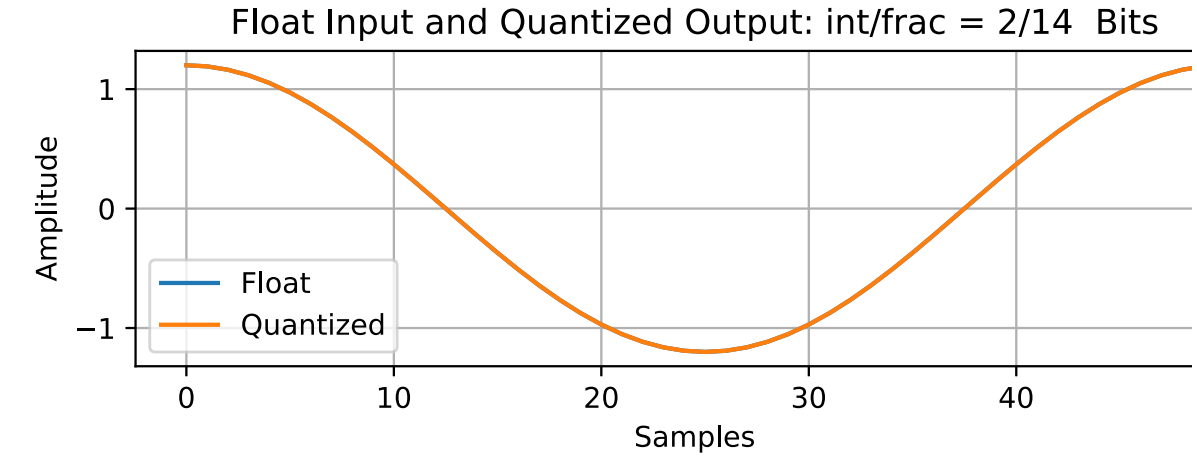


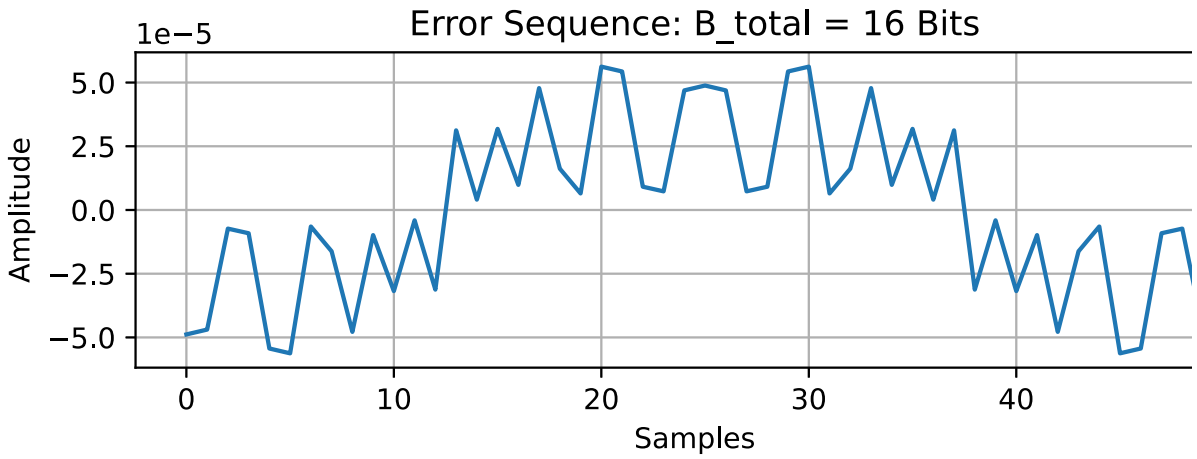


In [22]:
```
n = arange(50)
x = 1.2*cos(2*pi*n/50)
B_int = 2  # adequate integer headroom to avoid overflow
B_frac = 14 # very good fractional resolution
B_total = B_int + B_frac
xq = Fxp(x, True, n_int=B_int, n_frac=B_frac, overflow='wrap') #Fxp(x)
subplot(211)
plot(n,x,label='Float')
plot(n,xq,label='Quantized')
title(f'Float Input and Quantized Output: int/frac = {B_int}/{B_frac}
ylabel(r'Amplitude')
xlabel(r'Samples')
legend()
grid();
subplot(212)
plot(n,xq.astype(float) -x)
title(f'Error Sequence: B_total = {B_total} Bits')
ylabel(r'Amplitude')
```

```
xlabel(r'Samples')
grid();
tight_layout()
xq[0:20:5].bin(frac_dot=True) # array values in binary with the fracti
```

Out[22]:
```
['001.00110011001100',
 '000.11111000100001',
 '000.01011110111011',
 '111.10100001000101']
```

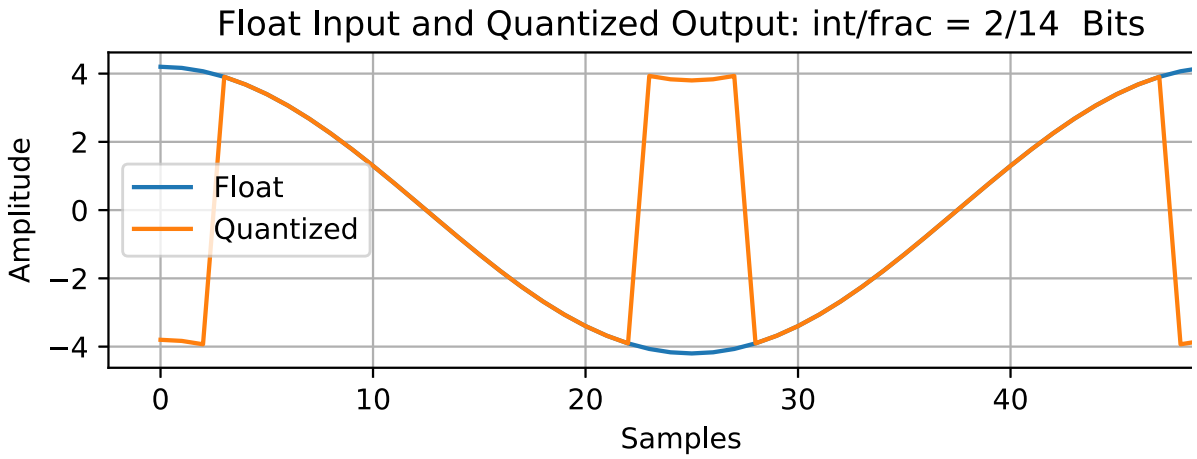


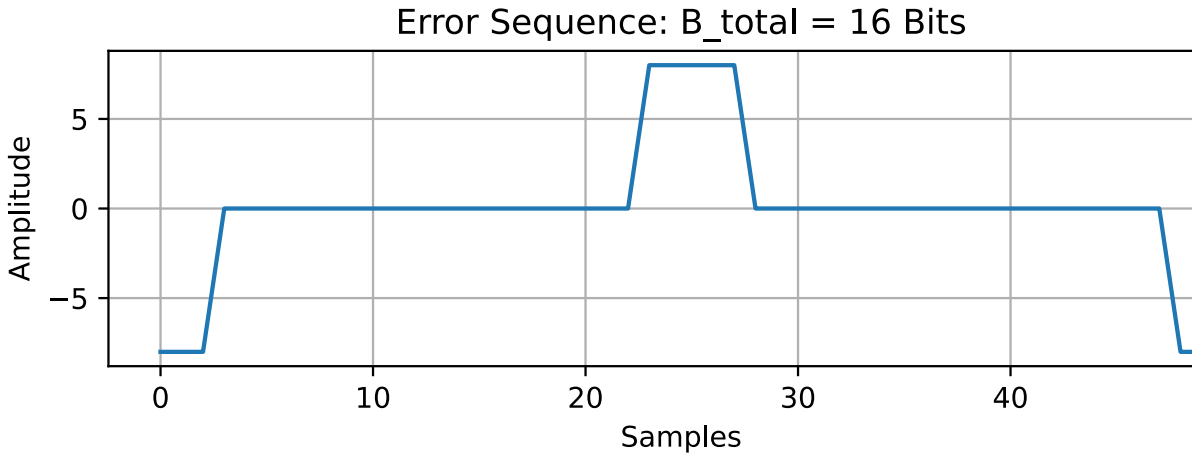


In [23]:
```
n = arange(50)
x = 4.2*cos(2*pi*n/50) # The 4.2 will make overflow occur
B_int = 2 # no longer adequate integer headroom to avoid overflow
B_frac = 14 # very good fractional resolution
B_total = B_int + B_frac
xq = Fxp(x, True, n_int=B_int, n_frac=B_frac, overflow='wrap') #Fxp(x)
subplot(211)
plot(n,x,label='Float')
plot(n,xq,label='Quantized')
title(f'Float Input and Quantized Output: int/frac = {B_int}/{B_frac}
ylabel(r'Amplitude')
xlabel(r'Samples')
legend()
grid();
subplot(212)
plot(n,xq.astype(float) -x)
```

```
title(f'Error Sequence: B_total = {B_total} Bits')
ylabel(r'Amplitude')
xlabel(r'Samples')
grid();
tight_layout()
xq[0:20:5].bin(frac_dot=True) # array values in binary with the fracti
```

Out[23]:
```
['100.00110011001100',
 '011.01100101110110',
 '001.01001100010000',
 '110.10110011110000']
```





In [24]:
```
n = arange(50)
x = 4.2*cos(2*pi*n/50)
B_int = 3 # add one more integer bit and overflow will stop
B_frac = 13 # Good fractional resolution, in spite of moving one bit t
B_total = B_int + B_frac
xq = Fxp(x, True, n_int=B_int, n_frac=B_frac, overflow='wrap') #Fxp(x)
subplot(211)
plot(n,x,label='Float')
plot(n,xq,label='Quantized')
title(f'Float Input and Quantized Output: int/frac = {B_int}/{B_frac}
ylabel(r'Amplitude')
xlabel(r'Samples')
legend()
grid();
```

```
subplot(212)
plot(n,xq.astype(float) -x)
title(f'Error Sequence: B_total = {B_total} Bits')
ylabel(r'Amplitude')
xlabel(r'Samples')
grid();
tight_layout()
xq[0:20:5].bin(frac_dot=True) # array values in binary with the fracti
```

Out[24]:
```
['0100.0011001100110',
 '0011.0110010111011',
 '0001.0100110001000',
 '1110.1011001111000']
```