5655 Chapter 7

April 25, 2015

Contents


```
In [1]: %pylab inline
        #%matplotlib qt
        from __future__ import division # use so 1/2 = 0.5, etc.
        import ssd
        import scipy.signal as signal
        from IPython.display import Audio, display
        from IPython.display import Image, SVG
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [100]: pylab.rcParams['savefig.dpi'] = 100 # default 72
          #pylab.rcParams['figure.figsize'] = (6.0, 4.0) # default (6,4)
          %config InlineBackend.figure_formats=['svg'] # SVG inline viewing
          #%config InlineBackend.figure_formats=['pdf'] # render pdf figs for LaTeX
```

```
In [101]: from IPython.display import display
          from sympy.interactive import printing
          printing.init_printing(use_latex='mathjax')
          import sympy as sym
          x,y,z,a,b,c = sym.symbols("x y z a b c")
```

Frequency Response Support Function

The function below is useful for overlaying plots of frequency response (magnitude, phase, and group delay) when you may want to compare several filter types. Doing this entirely using `freqz()` on your own is another option, of course.

You get to see this funtion in action in the first example in this notebook.

```
In [102]: 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
              """
              if type(b) == list:
                  # We have a list of filters
                  N_filt = len(b)
              f = np.arange(0,Npts)/(2.0*Npts)
              for n in range(N_filt):
                  w,H = signal.freqz(b[n],a[n],2*np.pi*f)
                  if n == 0:
                      plt.figure(figsize=fsize)
                  if mode.lower() == 'db':
                      plt.plot(f*fs,20*np.log10(np.abs(H)))
                      if n == N_filt-1:
                          plt.xlabel('Frequency (Hz)')
                          plt.ylabel('Gain (dB)')
                          plt.title('Frequency Response - Magnitude')

                  elif mode.lower() == 'phase':
                      plt.plot(f*fs,np.angle(H))
                      if n == N_filt-1:
                          plt.xlabel('Frequency (Hz)')
                          plt.ylabel('Phase (rad)')
                          plt.title('Frequency Response - Phase')

                  elif (mode.lower() == 'groupdelay_s') or (mode.lower() == 'groupdelay_t'):
                      """
                      Notes
                      -----
```

```
        Since this calculation involves finding the derivative of the
        phase response, care must be taken at phase wrapping points
        and when the phase jumps by +/-pi, which occurs when the
        amplitude response changes sign. Since the amplitude response
        is zero when the sign changes, the jumps do not alter the group
        delay results.
        """
        theta = np.unwrap(np.angle(H))
        # Since theta for an FIR filter is likely to have many pi phase
        # jumps too, we unwrap a second time 2*theta and divide by 2
        theta2 = np.unwrap(2*theta)/2.
        theta_dif = np.diff(theta2)
        f_diff = np.diff(f)
        Tg = -np.diff(theta2)/np.diff(w)
        # For gain almost zero set groupdelay = 0
        idx = pylab.find(20*np.log10(H[:-1]) < -400)
        Tg[idx] = np.zeros(len(idx))
        max_Tg = np.max(Tg)
        #print(max_Tg)
        if mode.lower() == 'groupdelay_t':
            max_Tg /= fs
            plt.plot(f[:-1]*fs,Tg/fs)
            plt.ylim([0,1.2*max_Tg])
        else:
            plt.plot(f[:-1]*fs,Tg)
            plt.ylim([0,1.2*max_Tg])
        if n == N_filt-1:
            plt.xlabel('Frequency (Hz)')
            if mode.lower() == 'groupdelay_t':
                plt.ylabel('Group Delay (s)')
            else:
                plt.ylabel('Group Delay (samples)')
            plt.title('Frequency Response - Group Delay')
    else:
        s1 = 'Error, mode must be "dB", "phase, '
        s2 = '"groupdelay_s", or "groupdelay_t"'
        print(s1 + s2)
```

IIR Filter Design

The focus here floating point IIR filters implemented as a cascade of biquadratic sections. Will also need a means to export the filter coefficients to header files. Header export functions for `float32_t` are provided below. The provided function in particular exports headers in a format the meshes wel with the IIR filter routines of CMSIS-DSP. The next step is to actually design some filters using functions found in `scipy.signal`.

Note: The MATLAB signal processing toolbox is extremely comprehensive in its support of digital filter design. The use of Python is adequate for this, but do not ignore the power available in MATLAB.

Exporting Coefficients to Header Files

Float Header Export for CMSIS `arm_biquad_cascade_df2T_f32` and Custom

```
In [103]: def IIR_sos_header(fname_out,b,a):
              """
              Write IIR SOS Header Files
              File format is compatible with CMSIS-DSP IIR
              Directform II Filter Functions

              Mark Wickert March 2015
              """
              SOS_mat, G_stage = tf2sos(b,a)
              Ns,Mcol = SOS_mat.shape
              f = open(fname_out,'wt')
              f.write('//define a IIR SOS CMSIS-DSP coefficient array\n\n')
              f.write('#include <stdint.h>\n\n')
              f.write('#ifndef STAGES\n')
              f.write('#define STAGES %d\n' % Ns)
              f.write('#endif\n')
              f.write('/*************************************************************/\n');
              f.write('/*                       IIR SOS Filter Coefficients       */\n');
              f.write('float32_t ba_coeff[%d] = { //b0,b1,b2,a1,a2,... by stage\n' % (5*Ns))
              for k in range(Ns):
                  if (k < Ns-1):
                      f.write('    %15.12f, %15.12f, %15.12f,\n' % \
                              (SOS_mat[k,0],SOS_mat[k,1],SOS_mat[k,2]))
                      f.write('    %15.12f, %15.12f,\n' % \
                              (-SOS_mat[k,4],-SOS_mat[k,5]))
                  else:
                      f.write('    %15.12f, %15.12f, %15.12f,\n' % \
                              (SOS_mat[k,0],SOS_mat[k,1],SOS_mat[k,2]))
                      f.write('    %15.12f, %15.12f\n' % \
                              (-SOS_mat[k,4],-SOS_mat[k,5]))
              f.write('};\n')
              f.write('/*************************************************************/\n')
              f.close()
```

Transfer Function to Second-Order Sections Conversion

```
In [104]: def tf2sos(b,a):
              """
              Cascade of second-order sections (SOS) conversion.
              Convert IIR transfer function coefficients, (b,a), to a
              matrix of second-order section coefficients, sos_mat. The
```

```
        gain coefficients per section are also available.
        SOS_mat,G_array = tf2sos(b,a)

        b = [b0, b1, ..., bM-1], the numerator filter coefficients
        a = [a0, a1, ..., aN-1], the denominator filter ceofficients

        SOS_mat = [[b00, b01, b02, 1, a01, a02],
                   [b10, b11, b12, 1, a11, a12],
                    ...]
        G_stage = gain per full biquad; square root for 1st-order stage

        where K is ceil(max(M,N)/2).

        Mark Wickert March 2015
        """
        Kactual = max(len(b)-1,len(a)-1)
        Kceil = 2*int(np.ceil(Kactual/2))
        z_unsorted,p_unsorted,k = signal.tf2zpk(b,a)
        z = shuffle_real_roots(z_unsorted)
        p = shuffle_real_roots(p_unsorted)
        M = len(z)
        N = len(p)
        SOS_mat = np.zeros((Kceil//2,6))
        # For now distribute gain equally across all sections
        G_stage = k**(2/Kactual)
        for n in range(Kceil//2):
            if 2*n + 1 < M and 2*n + 1 < N:
                SOS_mat[n,0:3] = array([1,-(z[2*n]+z[2*n+1]).real, (z[2*n]*z[2*n+1]).real])
                SOS_mat[n,3:] = array([1,-(p[2*n]+p[2*n+1]).real, (p[2*n]*p[2*n+1]).real])
                SOS_mat[n,0:3] = SOS_mat[n,0:3]*G_stage
            else:
                SOS_mat[n,0:3] = array([1,-(z[2*n]+0).real, 0])
                SOS_mat[n,3:] = array([1,-(p[2*n]+0).real, 0])
                SOS_mat[n,0:3] = SOS_mat[n,0:3]*np.sqrt(G_stage)
        return SOS_mat, G_stage

    def shuffle_real_roots(z):
        """
        Move real roots to the end of a root array so
        complex conjugate root pairs can form proper
        biquad sections.

        Need to add root magnitude re-ordering largest to
        smallest or smallest to largest.

        Mark Wickert April 2015
        """
        z_sort = zeros_like(z)
        front_fill = 0
        end_fill = -1
        for k in range(len(z)):
            if z[k].imag == 0:
                z_sort[end_fill] = z[k]
                end_fill -= 1
```

```
            else:
                z_sort[front_fill] = z[k]
                front_fill += 1
        return z_sort
```

IIR Filter Design Using the Bilinear Transformation

signal.iirdesign():

```
"""
Complete IIR digital and analog filter design.

Given passband and stopband frequencies and gains, construct an analog or
digital IIR filter of minimum order for a given basic type.  Return the
output in numerator, denominator ('ba') or pole-zero ('zpk') form.

Parameters
----------
wp, ws : float
    Passband and stopband edge frequencies.
    For digital filters, these are normalized from 0 to 1, where 1 is the
    Nyquist frequency, pi radians/sample.  (`wp` and `ws` are thus in
    half-cycles / sample.)  For example:

        - Lowpass:   wp = 0.2,          ws = 0.3
        - Highpass:  wp = 0.3,          ws = 0.2
        - Bandpass:  wp = [0.2, 0.5],   ws = [0.1, 0.6]
        - Bandstop:  wp = [0.1, 0.6],   ws = [0.2, 0.5]

    For analog filters, `wp` and `ws` are angular frequencies (e.g. rad/s).

gpass : float
    The maximum loss in the passband (dB).
gstop : float
    The minimum attenuation in the stopband (dB).
analog : bool, optional
    When True, return an analog filter, otherwise a digital filter is
    returned.
ftype : str, optional
    The type of IIR filter to design:

        - Butterworth   : 'butter'
        - Chebyshev I   : 'cheby1'
        - Chebyshev II  : 'cheby2'
        - Cauer/elliptic: 'ellip'
        - Bessel/Thomson: 'bessel'

output : {'ba', 'zpk'}, optional
    Type of output:  numerator/denominator ('ba') or pole-zero ('zpk').
    Default is 'ba'.

Returns
-------
b, a : ndarray, ndarray
    Numerator (`b`) and denominator (`a`) polynomials of the IIR filter.
```

```
    Only returned if ``output='ba'``.
z, p, k : ndarray, ndarray, float
    Zeros, poles, and system gain of the IIR filter transfer
    function.  Only returned if ``output='zpk'``.
"""
```

Example: Design from Frequency Response Requirements

```
In [105]: bfr1,afr1 = signal.iirdesign(1/5,1.44/5,2,50,ftype = 'butter')
          print('Filter order = %d' % (len(afr1)-1,))
          print('b coefficients')
          print(bfr1)
          print('a coefficients')
          print(afr1)
```

```
Filter order = 15
b coefficients
[  2.72320677e-09   4.08481016e-08   2.85936711e-07   1.23905908e-06
   3.71717724e-06   8.17778993e-06   1.36296499e-05   1.75238356e-05
   1.75238356e-05   1.36296499e-05   8.17778993e-06   3.71717724e-06
   1.23905908e-06   2.85936711e-07   4.08481016e-08   2.72320677e-09]
a coefficients
[  1.00000000e+00  -8.89058468e+00   3.77186024e+01  -1.01018190e+02
   1.90599634e+02  -2.67913023e+02   2.89426526e+02  -2.44410932e+02
   1.62506188e+02  -8.49989798e+01   3.46630271e+01  -1.08161062e+01
   2.49831047e+00  -4.03052007e-01   4.05912898e-02  -1.92289977e-03]
```

```
In [106]: bfr2,afr2 = signal.iirdesign(1/5,1.44/5,2,50,ftype = 'cheby1')
          print('Filter order = %d' % (len(afr2)-1,))
          print('b coefficients')
          print(bfr2)
          print('a coefficients')
          print(afr2)
```

```
Filter order = 8
b coefficients
[  8.32305860e-07   6.65844688e-06   2.33045641e-05   4.66091281e-05
   5.82614102e-05   4.66091281e-05   2.33045641e-05   6.65844688e-06
   8.32305860e-07]
a coefficients
[  1.          -6.80853767  20.96016981 -38.03057568  44.42542937
 -34.18774941  16.92096145  -4.92535486   0.64592523]
```

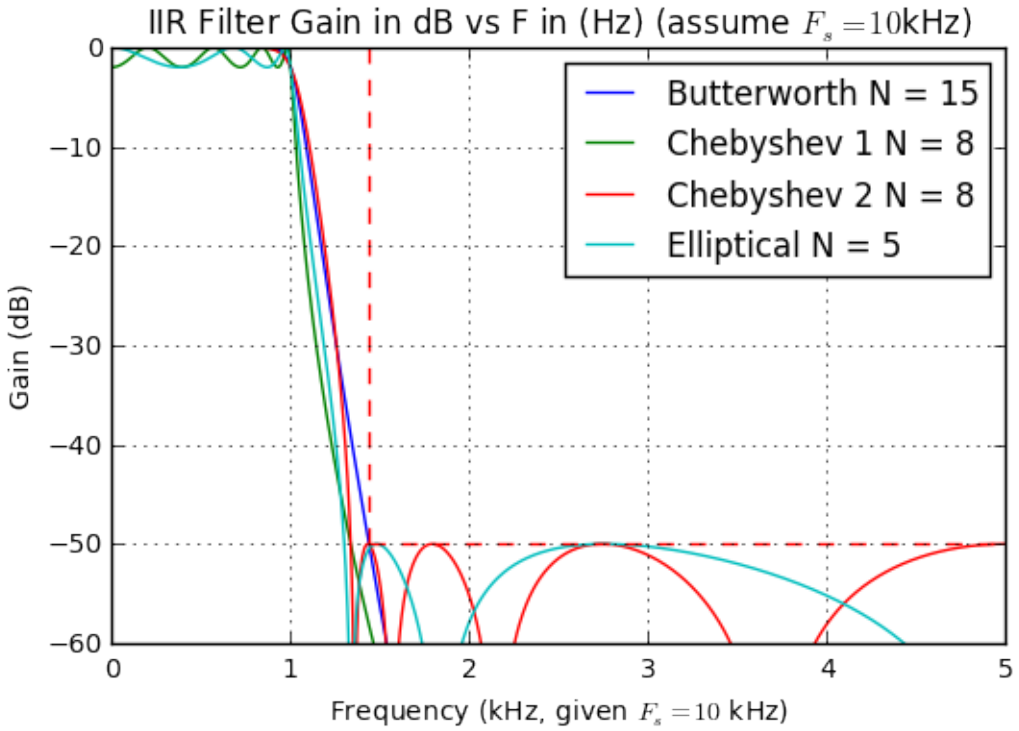
```
In [107]: bfr3,afr3 = signal.iirdesign(1/5,1.44/5,2,50,ftype = 'cheby2')
          print('Filter order = %d' % (len(afr3)-1,))
          print('b coefficients')
          print(bfr3)
          print('a coefficients')
          print(afr3)
```

```
Filter order = 8
b coefficients
[ 0.00761324 -0.01135259  0.01996792 -0.01302755  0.01824046 -0.01302755
  0.01996792 -0.01135259  0.00761324]
a coefficients
[  1.          -4.31595778   8.71886201 -10.50954035   8.20030971
  -4.211867     1.38599447  -0.26598822   0.02282966]
```

In [108]:
```
bfr4,afr4 = signal.iirdesign(1/5,1.44/5,2,50,ftype = 'ellip')
print('Filter order = %d' % (len(afr4)-1,))
print('b coefficients')
print(bfr4)
print('a coefficients')
print(afr4)
```

```
Filter order = 5
b coefficients
[ 0.00670872 -0.00766339  0.00628423  0.00628423 -0.00766339  0.00670872]
a coefficients
[ 1.         -4.05164787  7.03344388 -6.46669653  3.13999072 -0.64443106]
```

In [109]:
```
freqz_resp_list([bfr1,bfr2,bfr3,bfr4],[afr1,afr2,afr3,afr4],'dB'
                ,fs=10,fsize=(6,4))
ylim(-60,0)
plot([1.44,1.44],[-50,0],'r--')
plot([1.44,5],[-50,-50],'r--')
title(r'IIR Filter Gain in dB vs F in (Hz) (assume $F_s = 10 $kHz)')
ylabel(r'Gain (dB)')
xlabel(r'Frequency (kHz, given $F_s = 10$ kHz)')
legend((r'Butterworth N = 15',r'Chebyshev 1 N = 8',
        r'Chebyshev 2 N = 8',r'Elliptical N = 5'),loc='best')
grid();
```



In [110]:
```
SOS_mat, G_stage = tf2sos(bfr4,afr4)
print('SOS Matrix')
print(SOS_mat)
```

```
SOS Matrix
[[ 0.13510017 -0.18002968  0.13510017  1.         -1.5859549   0.94825772]
 [ 0.13510017 -0.10939581  0.13510017  1.         -1.6261702   0.80950139]
 [ 0.36755975  0.36755975  0.          1.         -0.83952277  0.        ]]
```

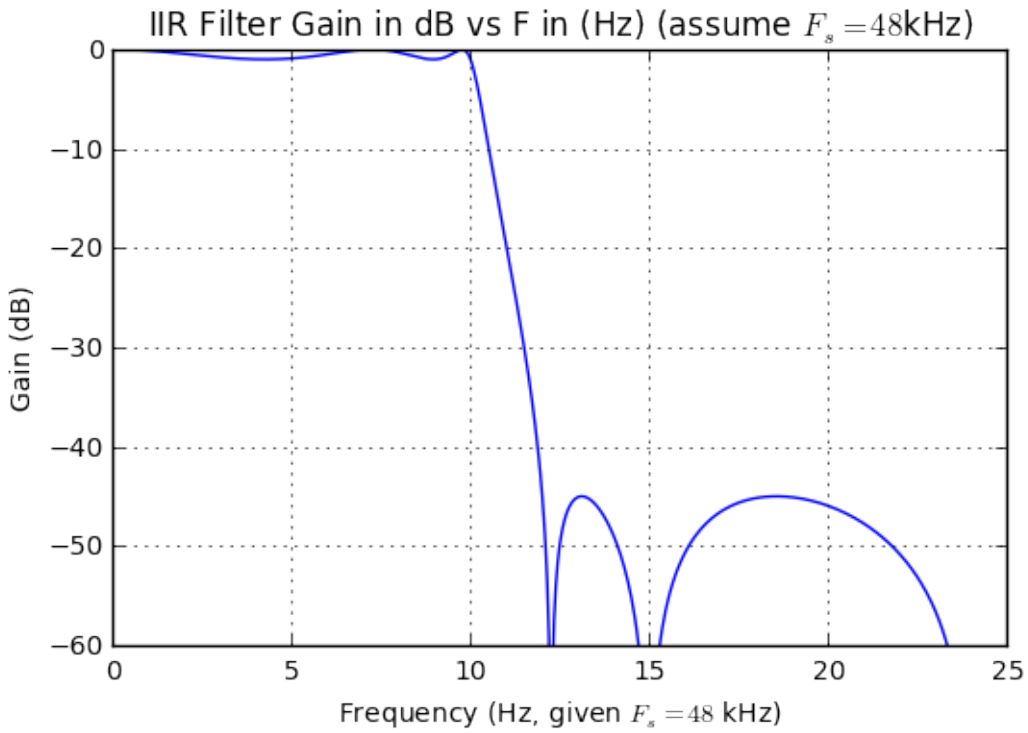
Example: Elliptical Lowpass

As a specific example consider a 5-order elliptic lowpass filter. The desired frequency response characteristics at a sampling rate of $f_s = 48$ kHz are:

1. Passband ripple of 1 dB over the band $0 \leq f \leq 10$ kHz
2. Stopband attenuation of 45 dB

```
In [111]: b1,a1 = signal.ellip(5,1,45,2*10/48)
          #b1a,a1a = signal.iirdesign(2*10/48,2*13/48,1,45,ftype = 'ellip')
```

```
In [112]: f = arange(0,0.5,0.001)
          w,H1 = signal.freqz(b1,a1,2*pi*f)
          plot(f*48,20*log10(abs(H1)))
          title(r'IIR Filter Gain in dB vs F in (Hz) (assume $F_s = 48$kHz)')
          ylabel(r'Gain (dB)')
          xlabel(r'Frequency (Hz, given $F_s = 48$ kHz)')
          ylim([-60,0])
          grid();
```



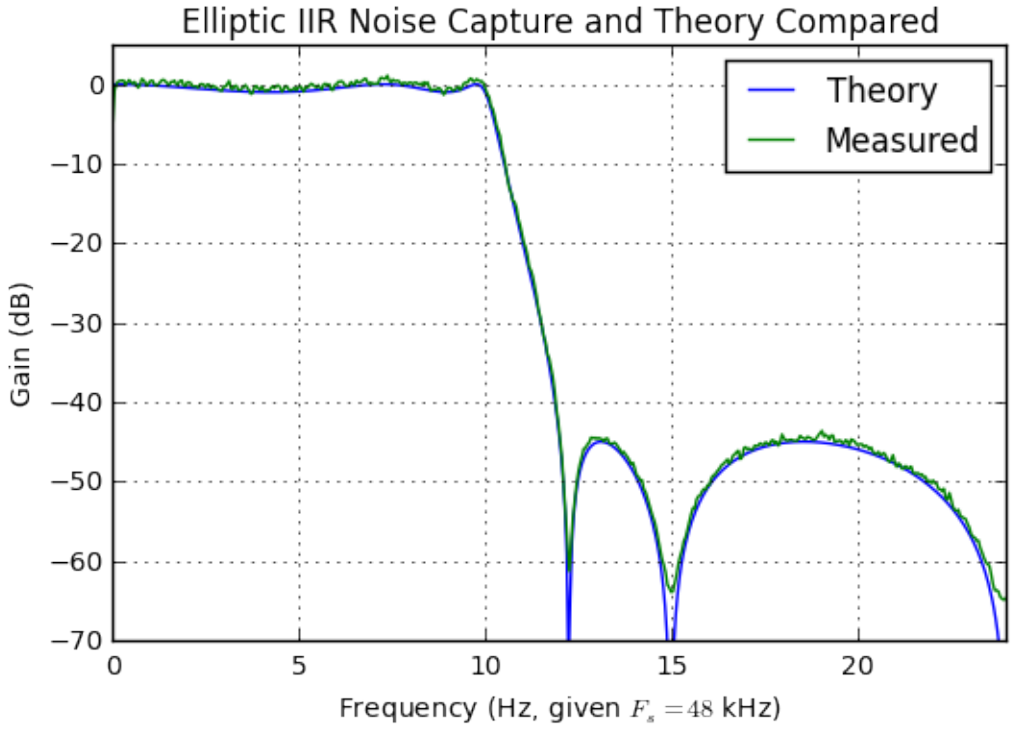

```
In [113]: SOS_mat, G_stage = tf2sos(b1,a1)
          print('SOS Matrix')
          print(SOS_mat)
```

```
SOS Matrix
[[ 0.28264064  0.21512808  0.28264064  1.         -0.4930401   0.89599154]
 [ 0.28264064  0.01860861  0.28264064  1.         -0.79889248  0.58959937]
 [ 0.53163958  0.53163958  0.          1.         -0.5632403   0.        ]]
```

```
In [114]: IIR_sos_header('IIR_sos_lpf10.h',b1,a1)
```

```
In [115]: fs_48,P5lpf10_float = loadtxt('spec_noise_iir5lpf10_fs48.csv',delimiter=',',
                                        skiprows=1,usecols=(0,1),unpack=True)
```

```
In [116]: f = arange(0,1.0,.001)
          w,H1 = signal.freqz(b1,a1,2*pi*f)
          plot(f*48,20*log10(abs(H1)))
          plot(fs_48[:500]/1000,P5lpf10_float[:500]-P5lpf10_float[2])
          ylim([-70,5])
          xlim([0,48/2])
          title(r'Elliptic IIR Noise Capture and Theory Compared')
          ylabel(r'Gain (dB)')
          xlabel(r'Frequency (Hz, given $F_s = 48$ kHz)')
          legend((r'Theory',r'Measured'),loc='best')
          grid();
```



Execution time for sample-based 3-stage `float32_t` is 1.48μs using `IIR_sos_filt_float32()`
Execution time for sample-based 3-stage `float32_t` is 1.11μs using `arm_biquad_cascade_df2T_f32()`

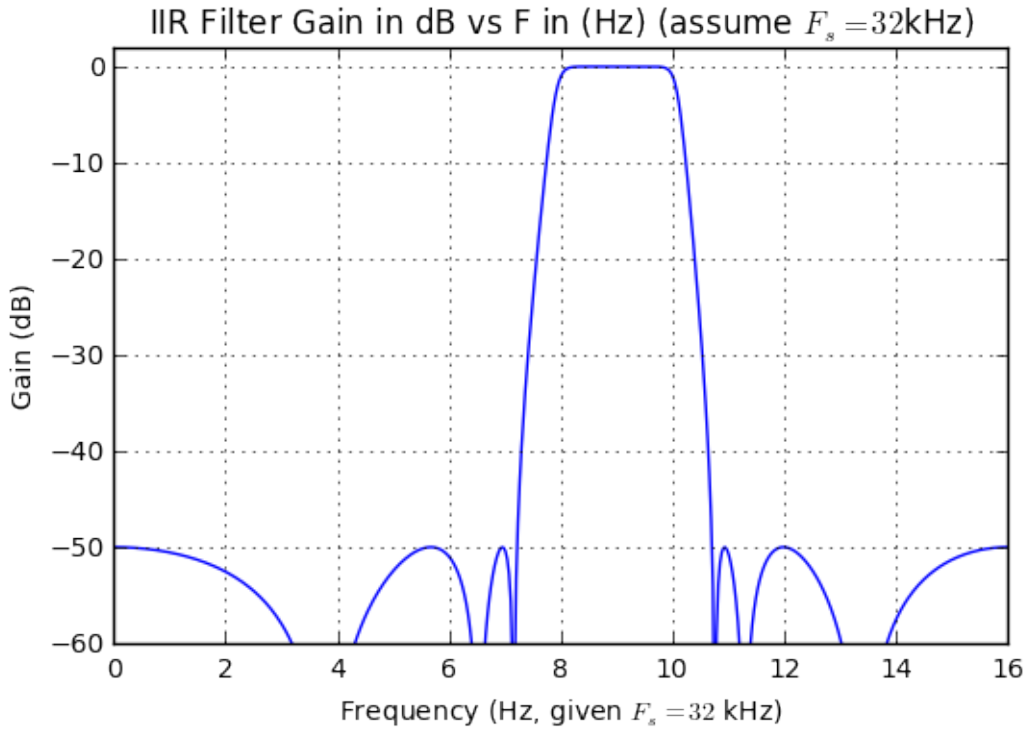
Example: Chebychev Type 2 Bandpass

```
In [117]: b2,a2 = signal.iirdesign([2*8000/32000,2*10000/32000],
                                   [2*7000/32000,2*11000/32000],1,50,ftype = 'cheby2')
```

```
        print('Filter order = %d' % (len(a2)-1,))
        print('b coefficients')
        print(b2)
        print('a coefficients')
        print(a2)
```

```
Filter order = 12
b coefficients
[ 0.0053181   0.00855892  0.01713494  0.02004807  0.02881132  0.02716787
  0.03081476  0.02716787  0.02881132  0.02004807  0.01713494  0.00855892
  0.0053181 ]
a coefficients
[  1.           2.0338808    5.95661525   8.03094851  12.82563179
  12.314038    13.31621541   9.15252827   7.09020731   3.28881527
   1.81220396   0.45490953   0.16604046]
```

In [118]:
```
f = arange(0,0.5,0.001)
w,H1 = signal.freqz(b2,a2,2*pi*f)
plot(f*32,20*log10(abs(H1)))
title(r'IIR Filter Gain in dB vs F in (Hz) (assume $F_s = 32$kHz)')
ylabel(r'Gain (dB)')
xlabel(r'Frequency (Hz, given $F_s = 32$ kHz)')
ylim([-60,2])
grid();
```

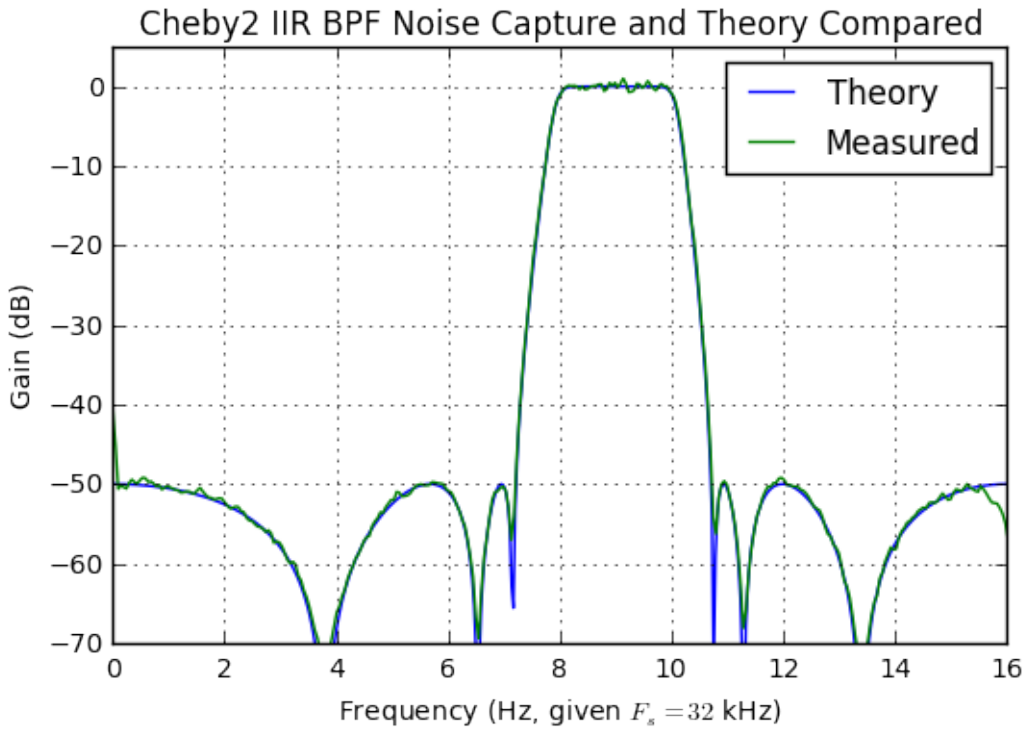


In [119]:
```
IIR_sos_header('IIR_sos_bpf8_10.h',b2,a2)
print('SOS Matrix')
print(SOS_mat)
```

```
SOS Matrix
[[ 0.28264064  0.21512808  0.28264064  1.         -0.4930401   0.89599154]
 [ 0.28264064  0.01860861  0.28264064  1.         -0.79889248  0.58959937]
 [ 0.53163958  0.53163958  0.          1.         -0.5632403   0.        ]]
```

```
In [120]: fs_32,P12bpf8_10_float = loadtxt('spec_noise_iir12bpf8_10_fs32.csv',delimiter=',',
                                           skiprows=1,usecols=(0,1),unpack=True)
```

```
In [121]: f = arange(0,1.0,.001)
          w,H2 = signal.freqz(b2,a2,2*pi*f)
          plot(f*32,20*log10(abs(H2)))
          plot(fs_32[:500]/1000,P12bpf8_10_float[:500]-max(P12bpf8_10_float)+1)
          ylim([-70,5])
          xlim([0,32/2])
          title(r'Cheby2 IIR BPF Noise Capture and Theory Compared')
          ylabel(r'Gain (dB)')
          xlabel(r'Frequency (Hz, given $F_s = 32$ kHz)')
          legend((r'Theory',r'Measured'),loc='best')
          grid();
```



Execution time for sample-based 6-stage `float32_t` is 2.60μs using `IIR_sos_filt_float32()`
Execution time for sample-based 6-stage `float32_t` is 2.04μs using `arm_biquad_cascade_df2T_f32()`

```
In [121]:
```