Chapter 8

Real-Time IIR Digital Filters

Introduction

Infinite impulse response (IIR) filter design has its roots in traditional analog filter design. One of the main issues in IIR digital filter design is choosing how to convert a well established analog design to its discrete-time counter part. Real-time implementation of IIR design will require recursive algorithms which are prone to stability problems. When using a floating point DSP instability problems should be minimal. In fact, a second-order IIR filter can be used as a sinusoid oscillator, even producing perfect phase quadrature signals ($\sin\omega_o$ and $\cos\omega_o$). Perhaps the most efficient and robust IIR implementation scheme is the cascade of biquad sections. This topology and its real time implementation will be explored in this chapter.

Basic IIR Filter Topologies

An IIR filter has feedback, thus from DSP theory we recall the general form of an *N*-order IIR filter is

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{r=0}^{M} b_r x[n-r] \qquad (8.1)$$

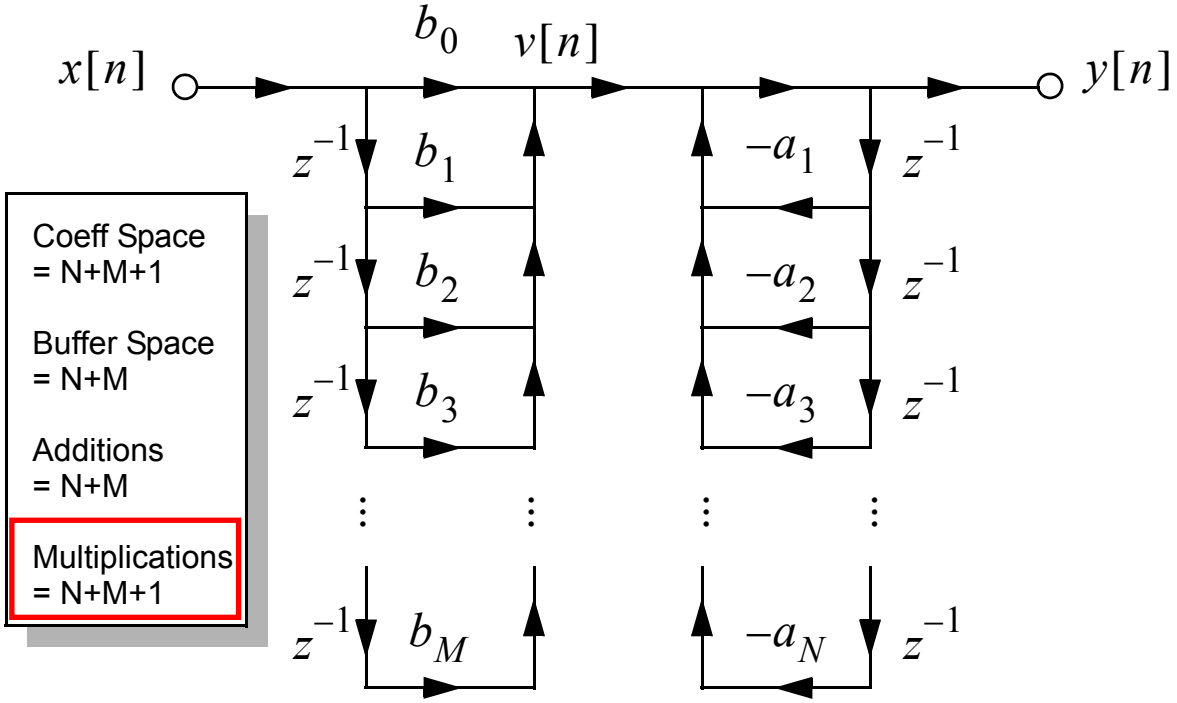
- By z-transforming both sides of (8.1) and using the fact that $H(z) = Y(z)/X(z)$, we can write

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}} \tag{8.2}$$

- IIR filters can be implemented in a variety of topologies, the most common ones, direct form I, II, cascade, and parallel, will be reviewed below

Direct Form I

- Direct implementation of (8.1) leads to the following structure



Direct Form I Structure

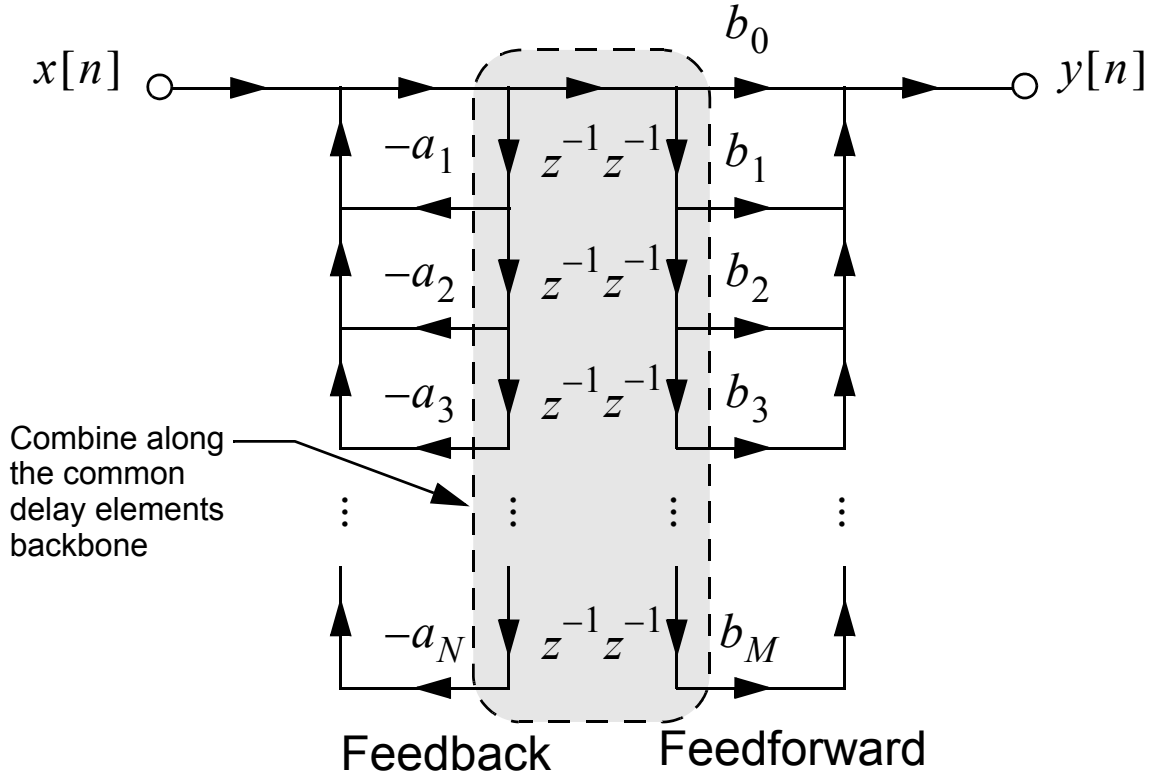
- The calculation of $y[n]$ for each new $x[n]$ requires the ordered solution of two difference equations

$$v[n] = \sum_{k=0}^{M} b_k x[n-k] \tag{8.3}$$

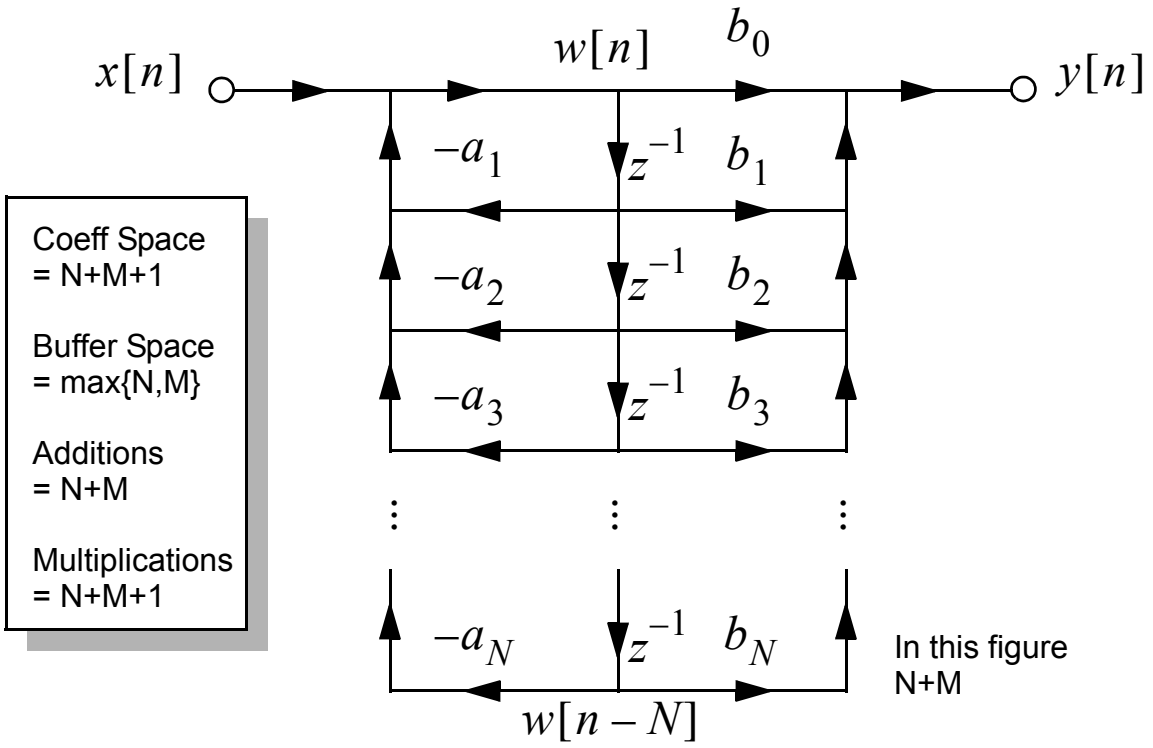
$$y[n] = v[n] - \sum_{k=1}^{N} a_k y[n-k] \tag{8.4}$$

Direct Form II

- A more efficient direct form structure can be realized by placing the feedback section first, followed by the feedforward section
- The first step in this rearrangement is the following



- The final direct form II structure is shown below



Direct Form II Structure

- The ordered pair of difference equations needed to obtain $y[n]$ from $x[n]$ is

$$w[n] = x[n] - \sum_{k=1}^{N} a_k w[n-k] \tag{8.5}$$

$$y[n] = \sum_{k=0}^{M} b_k w[n-k] \tag{8.6}$$

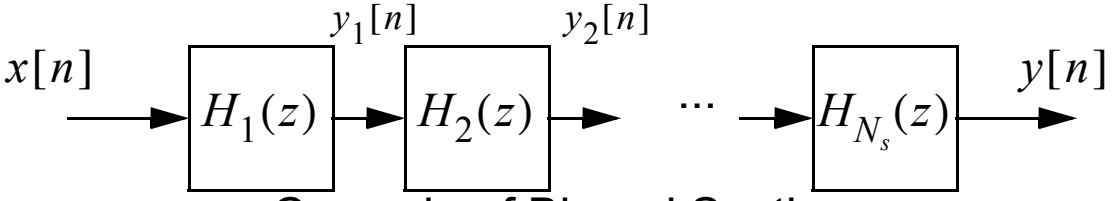
Cascade Form

- Since the system function, $H(z)$, is a ratio of polynomials, it is possible to factor the numerator and denominator polynomials in a variety of ways
- The most popular factoring scheme is as a product of second-order polynomials, which at the very least insures that conjugate pole and zeros pairs can be realized with real coefficients

$$H(z) = \prod_{k=1}^{N_s} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}} = \prod_{k=1}^{N_s} H_k(z) \qquad (8.7)$$

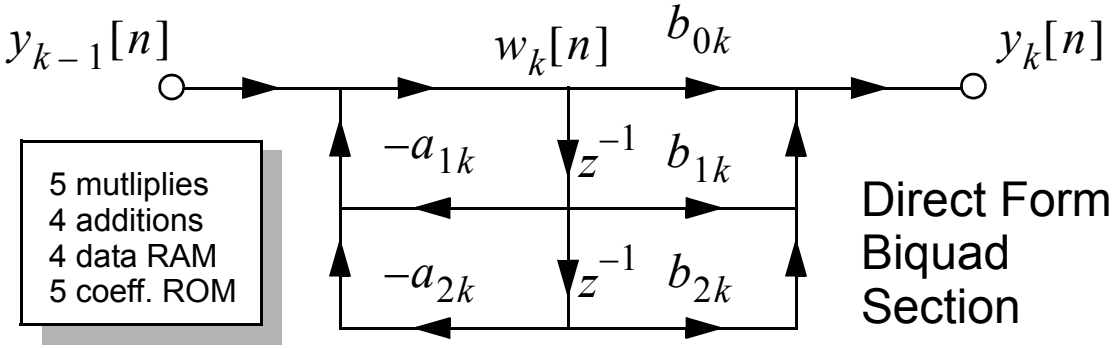
  where $N_s = \lfloor (N+1)/2 \rfloor$ is the largest integer in $(N+1)/2$

- A product of system functions corresponds to a cascade of *biquad* system blocks



Cascade of Biquad Sections

- The *k*th biquad can be implemented using a direct form structure (typically direct form II)



5 mutliplies
4 additions
4 data RAM
5 coeff. ROM

Direct Form II Biquad Section

- The corresponding biquad difference equations are

$$w_k[n] = y_{k-1}[n] - a_{1k}w_k[n-1] - a_{2k}w_k[n-2] \qquad (8.8)$$

$$y_k[n] = b_{0k}w_k[n] + b_{1k}w_k[n-1] + b_{2k}w_k[n-2] \qquad (8.9)$$

- The cascade of biquads is very popular in real-time DSP, is supported by the MATLAB signal processing toolbox, and will be utilized in example code presented later

Parallel Form

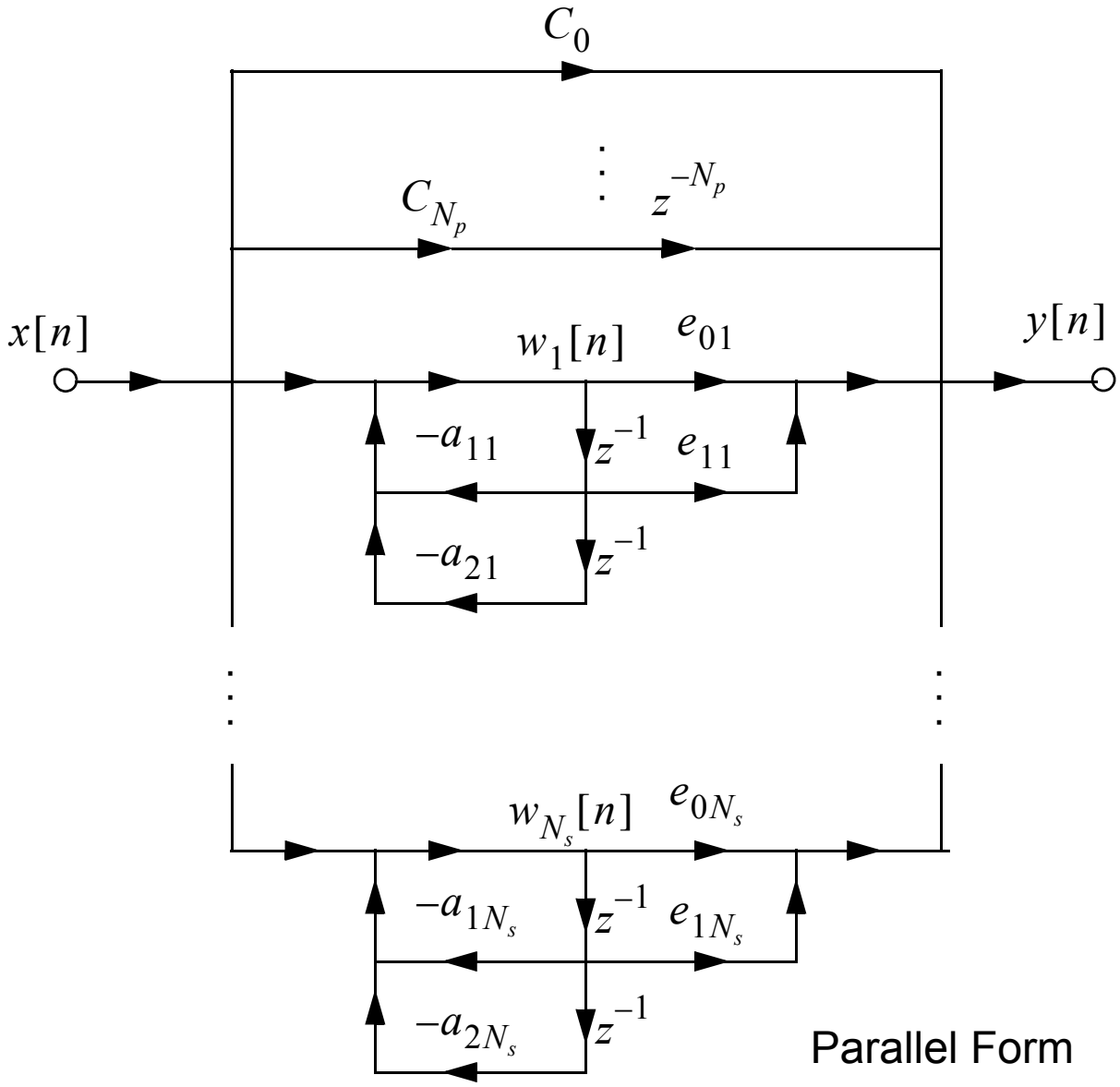
- The parallel form topology is obtained by first using long division to make $H(z)$ a proper rational function, then the remaining system function is expanded using a partial fraction expansion
- Complex conjugate pole pairs are combined into second-order factors, and any real poles are paired up to again make the basic building block a biquad section
- This time the biquad sections are in parallel

$$H(z) = \sum_{k=0}^{N_p} C_k z^{-k} + \sum_{k=1}^{N_s} \frac{e_{0k} + e_{1k}z^{-1}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}} \qquad (8.10)$$

  where for $M \geq N$, $N_p = M - N$ and $N_s = \lfloor (N+1)/2 \rfloor$

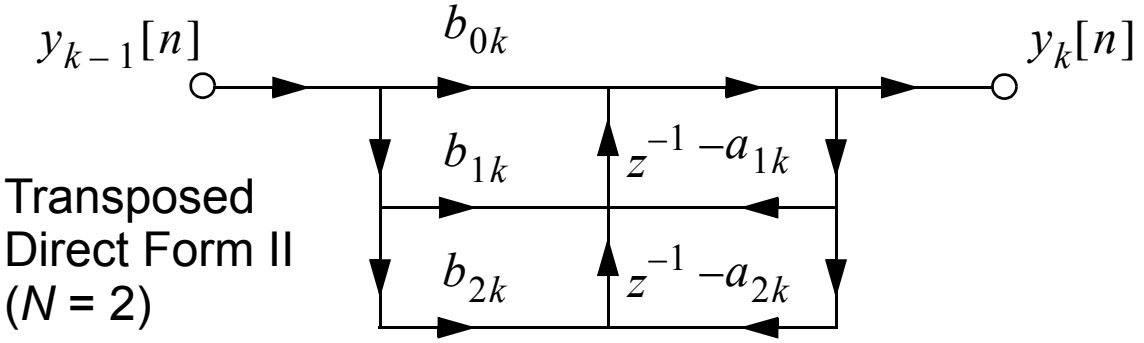
- Again each biquad section is typically implemented using a direct form II structure

- The block diagram follows from (8.10)



Transposed Forms

- Any of the topologies discussed thus far can be represented in a transposed form by simply reversing all of the flow graph arrows and swapping the input and output labels
- As an example the biquad section becomes the following:



- The MATLAB `filter()` function implements a transposed direct form II topology

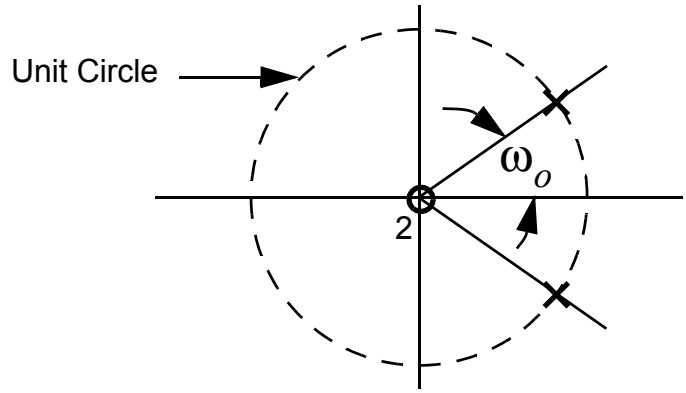
Pole-Zero Lattice

- The all-zero lattice topology mentioned in Chapter 7 is also available in a pole-zero form
- MATLAB has a function available which converts direct form (what MATLAB calls transfer function) coefficients into pole-zero lattice form

Digital Sinusoidal Oscillators

- By placing the poles of a two-pole resonator on the unit circle a *digital sinusoidal oscillator* can be obtained

Pole-Zero Map of the Resonator



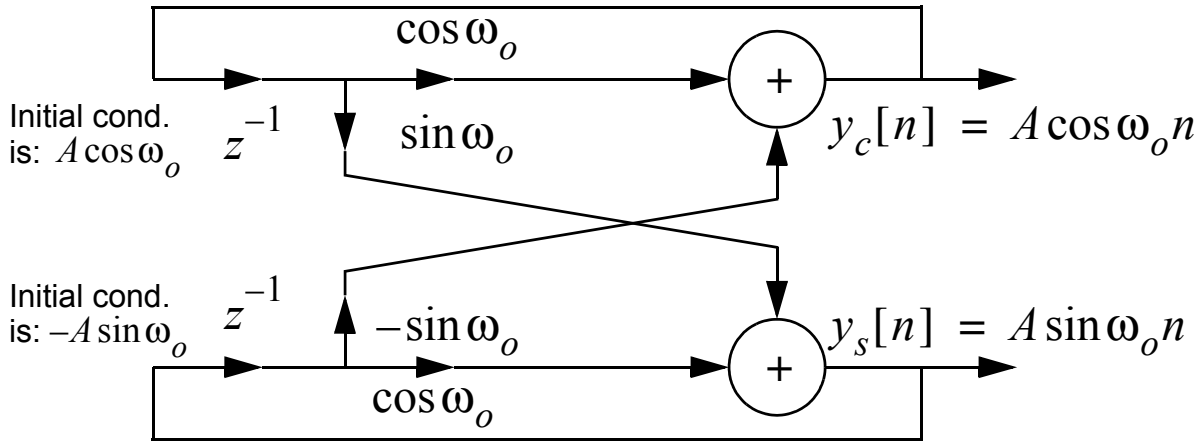
$$H(z) = \frac{b_0}{1 - 2\cos(\omega_o)z^{-1} + z^{-2}} \tag{8.11}$$

– The above can be easily implemented in a direct form II topology

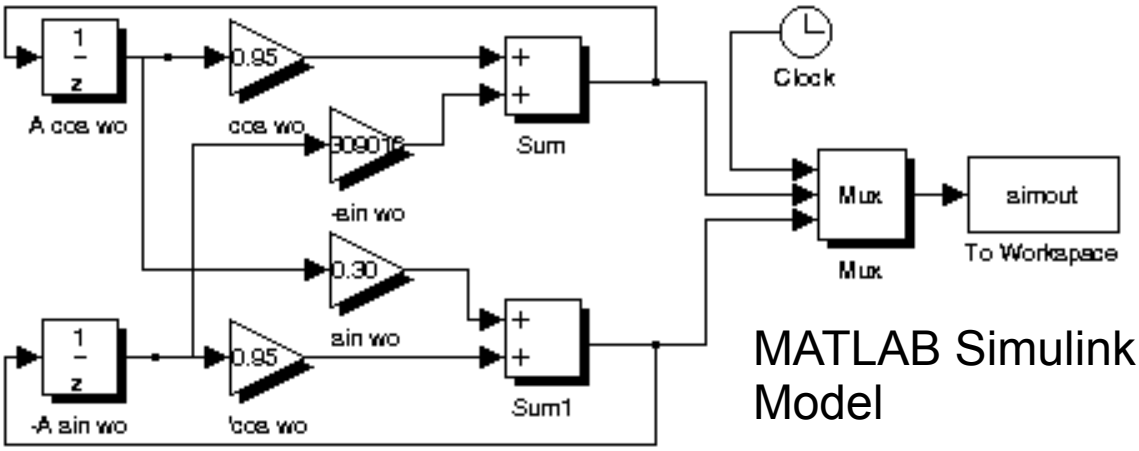
- By letting $b_0 = A\cos\omega_o$ and $x[n] = \delta[n]$ we obtain

$$y[n] = A\sin[(n+1)\omega_o]u[n] \tag{8.12}$$

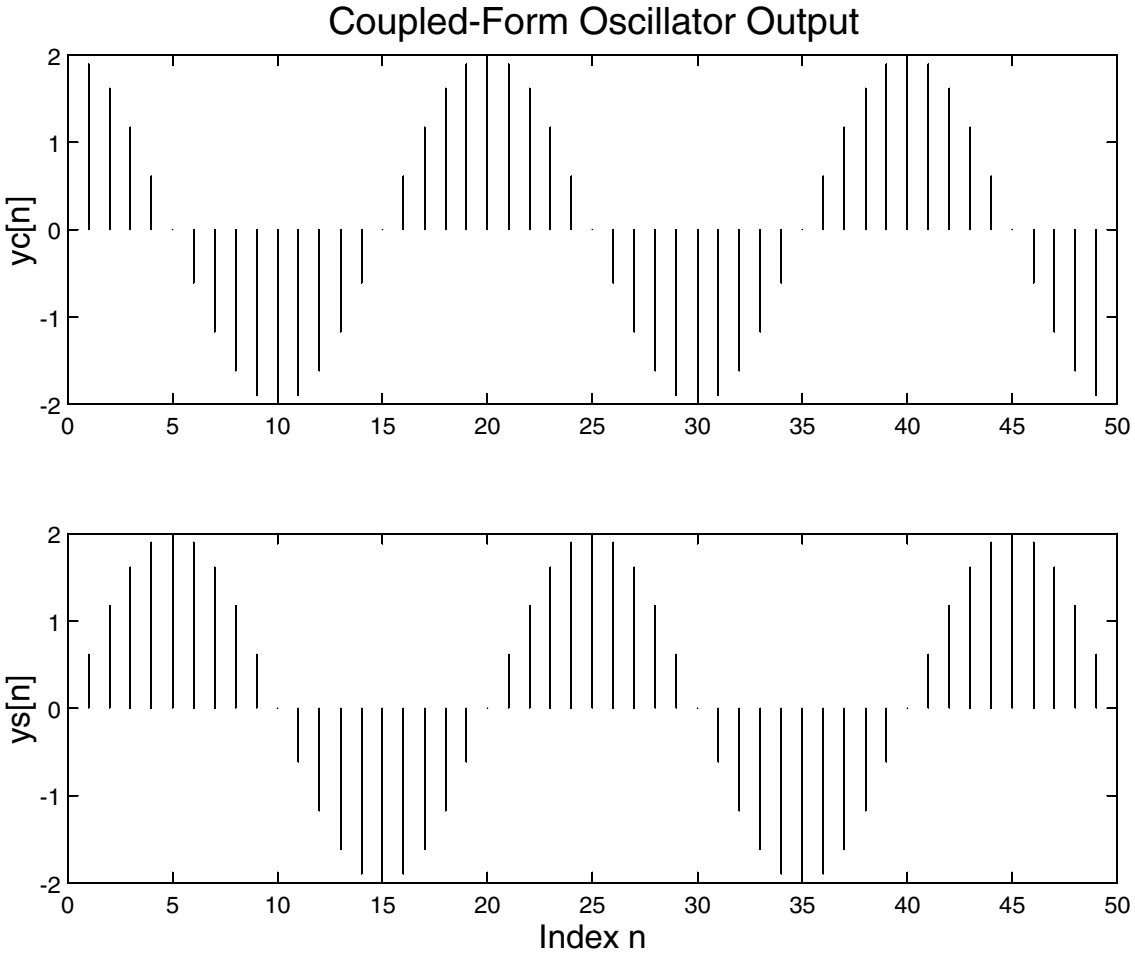
- For applications requiring in-phase (cosine) and quadrature (sine) carriers, the *coupled form oscillator* is useful



- A MATLAB Simulink simulation was constructed to verify the above model



- In the simulation we set $A = 2$ and $\omega_o = \pi/10$
- The results are as expected



Overview of IIR Filter Design

- The design steps follows those presented in Chapter 7 for FIR design

IIR Approximation Approaches

- In IIR filter design it is common to have the digital filter design follow from a standard analog prototype, or at least a known ratio of polynomials in the *s*-domain
- The most common approaches are:
  - Placement of poles and zeros (ad-hoc)
  - Impulse invariant (step invariant, etc.); conversion from *s*-domain
  - Bilinear transformation; conversion from *s*-domain
  - Minimum mean-square error (frequency domain)
- The *s*-domain starting point is often a continuous-time system function of the form

$$H_a(s) = \frac{b_0 + b_1 s + \ldots + b_M s^M}{1 + a_1 s + \ldots + a_N s^N} \tag{8.13}$$

  where the coefficients in (8.13) are different from those in (8.2)
  - The classical analog prototypes most often considered in texts and in actual filter design packages are: Butterworth (maximally flat), Chebyshev type I and II, and elliptical

  - A custom $H_a(s)$ is of course a valid option as well
- Filter design usually begins with a specification of the desired frequency response
- The filter requirements may be stated in terms of
  - Amplitude response vs. frequency
  - Phase or group delay response vs. frequency
  - A combination of amplitude and phase

Characteristics of Analog Prototypes

- The Butterworth design maintains a constant amplitude response in the passband and stopband; For an *n*th-order Butterworth $\left|H_a(\Omega)\right|^2$ has the first $2N-1$ derivatives equal to zero at $\Omega = 0$ and ∞
- The Chebyshev design achieves a more rapid rolloff rate near the cutoff frequency by allowing either ripple in the passband (type I) or ripple in the stopband (type II); the opposing band is still monotonic
  - The group delay fluctuation of the Chebyshev is greater than the Butterworth, except for small ripple values, e.g., $\varepsilon_{dB} \approx 0.1$, in which case the Chebyshev is actually better than the Butterworth
- An elliptic design allows both passband and stopband ripple, and is optimum in the sense that no other filter of the same order can provide a narrower transition band (the ratio stopband critical frequency to passband critical frequency)

Converting $H(s)$ to $H(z)$

In converting the s-domain system function to the z-domain two popular choices are impulse invariance and bilinear transformation

Impulse Invariance

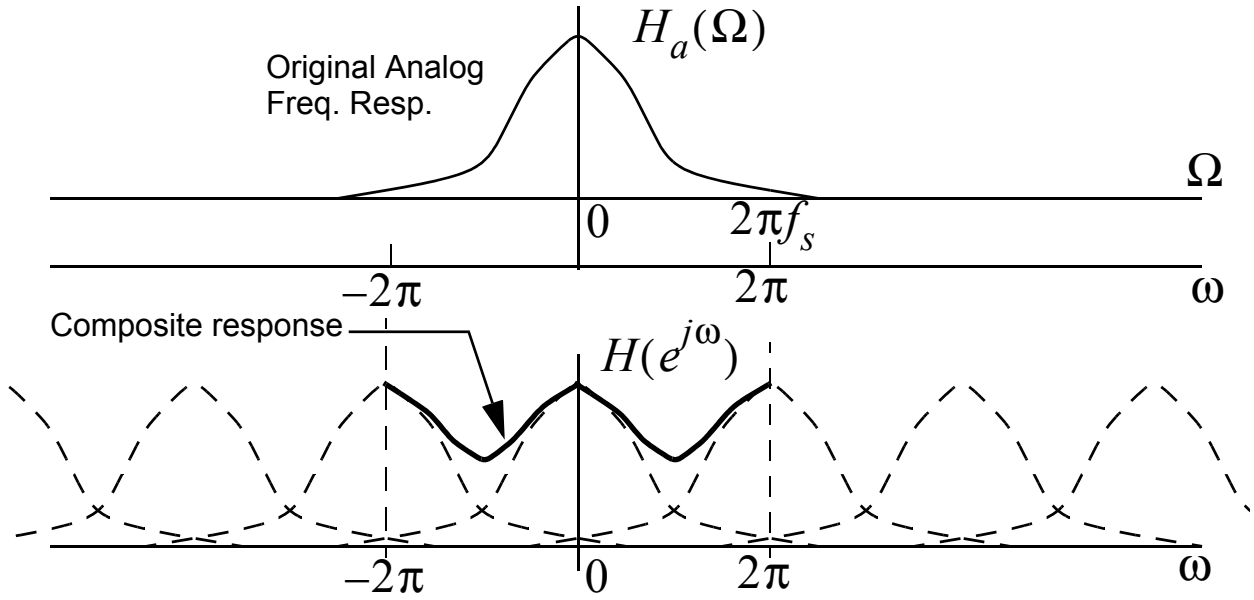
- With impulse invariance we simply chose the discrete-time system impulse response to be a sampled version of the corresponding continuous-time impulse response
- Starting from $h_a(t) \leftrightarrow H_a(s)$, we set

$$h[n] = Th_a(nT) \tag{8.14}$$

  – Note: Here $\leftrightarrow$ denotes a Laplace transform pair

- The frequency response of the impulse invariant filter is an aliased version of the analog frequency response with appropriate remapping of the frequency axis

$$H(e^{j\omega}) = \sum_{k=-\infty}^{\infty} H_a\left(\frac{\omega}{T} + \frac{2\pi k}{T}\right) \tag{8.15}$$



- A serious problem with this technique is the aliasing imposed by (8.15)
- To reduce aliasing:
  - Restrict $H_a(\Omega)$ to be lowpass or bandpass with a monotonic stopband
  - Decrease T (increase f_s) or decrease the filter cutoff frequency
- Since the frequency response in the discrete-time domain corresponds to the z-transform evaluated around the unit circle, it follows that

$$H(z) = Z\left[\{\mathcal{L}^{-1}[H_a(s)]\}\Big|_{t=nT}\right] \tag{8.16}$$

- As a simple example consider

$$H_a(s) = \frac{0.5(s+4)}{(s+1)(s+2)} = \frac{1.5}{s+1} - \frac{1}{s+2} \tag{8.17}$$

- Inverse Laplace transforming using partial fraction expansion yields

$$h_a(t) = [1.5e^{-t} - e^{-2t}]u(t) \tag{8.18}$$

- Sampling and scaling to obtain $h[n]$ yields

$$h[n] = G[1.5e^{-nT} - e^{-2nT}]u[n] \tag{8.19}$$

which implies that

$$H(z) = G\left[\frac{1.5}{1-e^{-T}z^{-1}} - \frac{1}{1-e^{-2T}z^{-1}}\right] \tag{8.20}$$

- We choose the scaling constant G so that $H_a(0) = H(e^{j0})$, thus

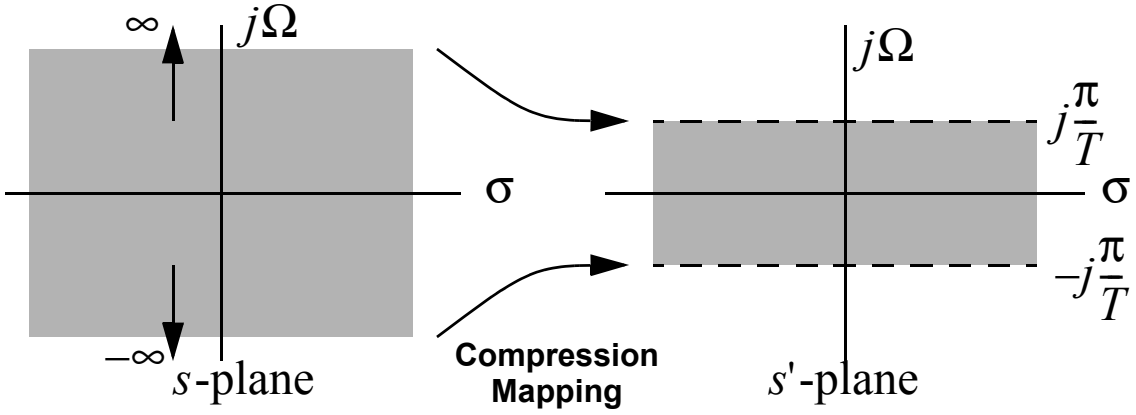
$$G = \left[\frac{1.5}{1-e^{-T}} - \frac{1}{1-e^{-2T}}\right]^{-1} \tag{8.21}$$

- This transformation technique is fully supported by the MATLAB signal processing toolbox

Bilinear Transformation

- A more popular technique than impulse invariance, since it does not suffer from aliasing, is the bilinear transformation method

- The impulse invariant technique used the many-to-one mapping $z = e^{sT}$
- To correct the aliasing problem we first employ a one-to-one mapping which compresses the entire s-plane into a strip



$$s' = \frac{2}{T}\tanh^{-1}\left(\frac{sT}{2}\right) \tag{8.22}$$

- Following the compression mapping we convert to the z-plane as before, except this time there is nothing that can alias
- The complete mapping from s to z is

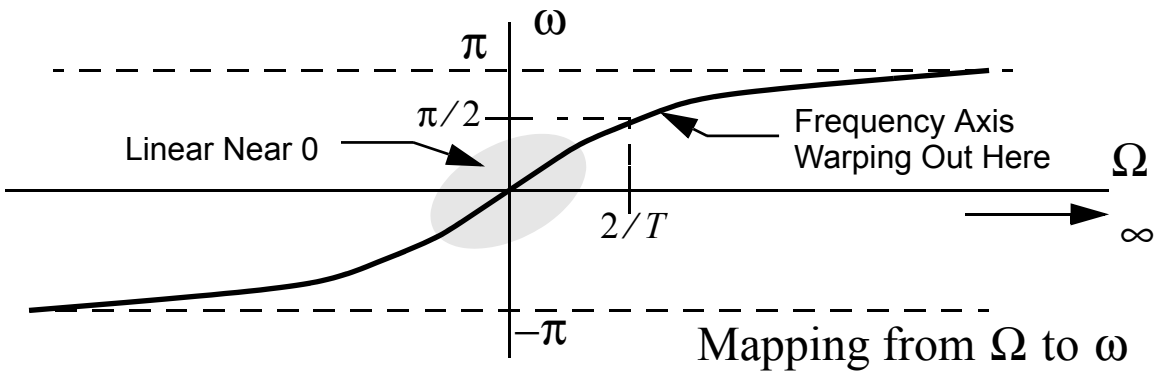
$$s = \frac{2}{T}\left(\frac{1 - z^{-1}}{1 + z^{-1}}\right) \tag{8.23}$$

or in reverse

$$z = \frac{1 + \frac{T}{2}s}{1 - \frac{T}{2}s} \tag{8.24}$$

- The frequency axis mappings become

$$\Omega = \frac{2}{T}\tan\left(\frac{\omega}{2}\right) \text{ or } \omega = 2\tan^{-1}(\Omega T/2) \tag{8.25}$$



- The basic filter design equation is

$$H(z) = H_a(s)\Big|_{s=\frac{2}{T}\left(\frac{1-z^{-1}}{1+z^{-1}}\right)} \tag{8.26}$$

- In a practical design in order to preserve desired discrete-time critical frequencies, such as the passband and stopband cutoff frequencies, we use frequency *prewarping*

$$\Omega_i = \frac{2}{T}\tan\left(\frac{\omega_i}{2}\right) \tag{8.27}$$

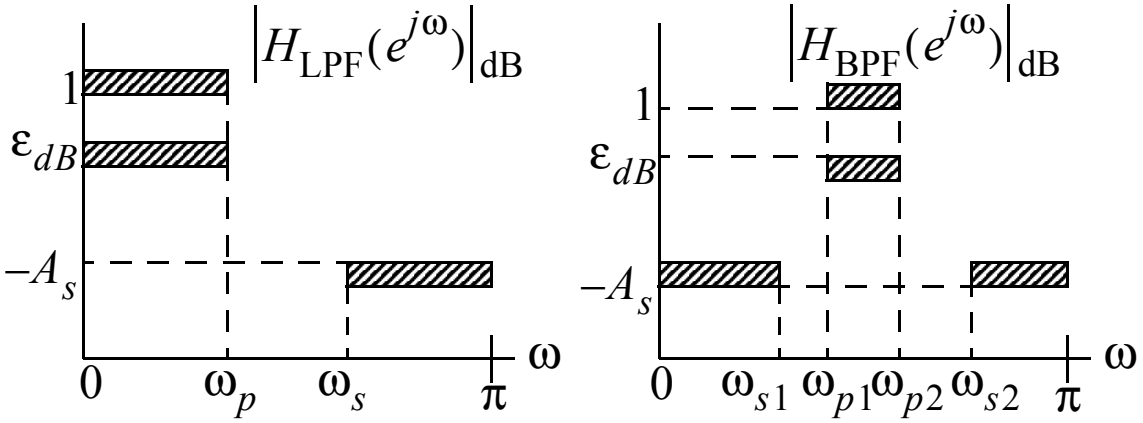
  where ω_i is a discrete-time critical frequency that must be used in the design of an analog prototype with corresponding continuous-time critical frequency Ω_i

  – The frequency axis compression imposed by the bilinear transformation can make the transition ratio in the discrete-time domain smaller than in the continuous-time domain, thus resulting in a lower order analog prototype than if the design was implemented purely as an analog filter

- Given a ratio of polynomials in the s-domain, or an amplitude response specification, we can proceed to find $H(z)$
- A detailed example could be worked at this time, but this transformation technique is fully supported by the MATLAB signal processing toolbox, so we will wait until these functions are introduced first

Classical Design from Analog Prototypes

- Start with amplitude response specifications



- From the amplitude response specifications determine the filter order of a particular analog prototype
  - When mapping the critical frequencies back to the continuous-time domain apply prewarping per (8.27)
- Using the bilinear transformation determine $H(z)$ from $H_a(s)$

Obtaining the Second-Order Section Coefficients

- Once $H(z)$ has been determined it is often best from a numerical precision standpoint, to convert $H(z)$ to a cascade of biquadratic sections (assuming of course that $\dot{M}, N > 2$)
- This requires polynomial rooting of the numerator and denominator polynomials making up $H(z)$, followed by grouping the roots first by conjugate pairs, and then pairing up simple roots
- The MATLAB signal processing toolbox has a function for doing this in one step

MATLAB Design Functions

The following function list is a subset of the filter design functions contained in the MATLAB signal processing toolbox, useful for IIR filter design. The function groupings match those of the toolbox manual.

Filter Analysis/Implementation	
`y = filter(b,a,x)`	Direct form II filter vector `x`
`[H,w] = freqs(b,a)`	s-domain frequency response computation
`[H,w] = freqz(b,a)`	z-domain frequency response computation
`[Gpd,w] = grpdelay(b,a)`	Group delay computation
`h = impz(b,a)`	Impulse response computation
`unwrap`	Phase unwrapping
`zplane(b,a)`	Plotting of the z-plane pole/zero map

Linear System Transformations	
`residuez`	z-domain partial fraction conversion; can use for parallel form design
`tf2zp`	Transfer function to zero-pole conversion
`ss2sos`	State space to second-order biquadratic sections conversion

IIR Filter Design	
`[b,a] = bes-self(n,Wn)`	Bessel analog filter design. Near constant group delay filters, but if transformed to a digital filter this property is lost
`[b,a] = butter (n,Wn,'ftype','s')`	Butterworth analog and digital filter designs. Use 's' for s-domain design. 'ftype' is optional for bandpass, highpass and bandstop designs.
`[b,a] = cheby1(n,Rp,Wn, 'ftype', 's')`	Chebyshev type I analog and digital filter designs. Use 's' for s-domain design. 'ftype' is optional for bandpass, highpass and bandstop designs.
`[b,a] = cheby2(n,Rs,Wn, 'ftype', 's')`	Chebyshev type II analog and digital filter designs. Use 's' for s-domain design. 'ftype' is optional for bandpass, highpass and bandstop designs.
[b,a] = ellip(n,Rp,Rs,Wn,'ftype','s')	Elliptic analog and digital filter designs. Use 's' for s-domain design. 'ftype' is optional for bandpass, highpass and band-stop designs.

IIR Filter Order Selection	
`[n,Wn] = buttord (Wp,Ws,Rp,Rs,'s')`	Butterworth analog and digital filter order selection. Use 's' for s-domain designs.
`[n,Wn] = cheb1ord (Wp,Ws,Rp,Rs,'s')`	Chebyshev type I analog and digital filter order selection. Use 's' for s-domain designs.
`[n,Wn] = cheb2ord (Wp,Ws,Rp,Rs,'s')`	Chebyshev type II analog and digital filter order selection. Use 's' for s-domain designs.
`[n,Wn] = ellipord (Wp,Ws,Rp,Rs,'s')`	Elliptic analog and digital filter order selection. Use 's' for s-domain designs.

- As discussed in Chapter 7, the GUI environment `fdatool` offers a point-and-click approach to filter design, harnessing the power of MATLAB's many filter design functions, including the quantization design features of the filter design toolbox itself

MATLAB Filter Design Examples

Impulse Invariant Design Example #1

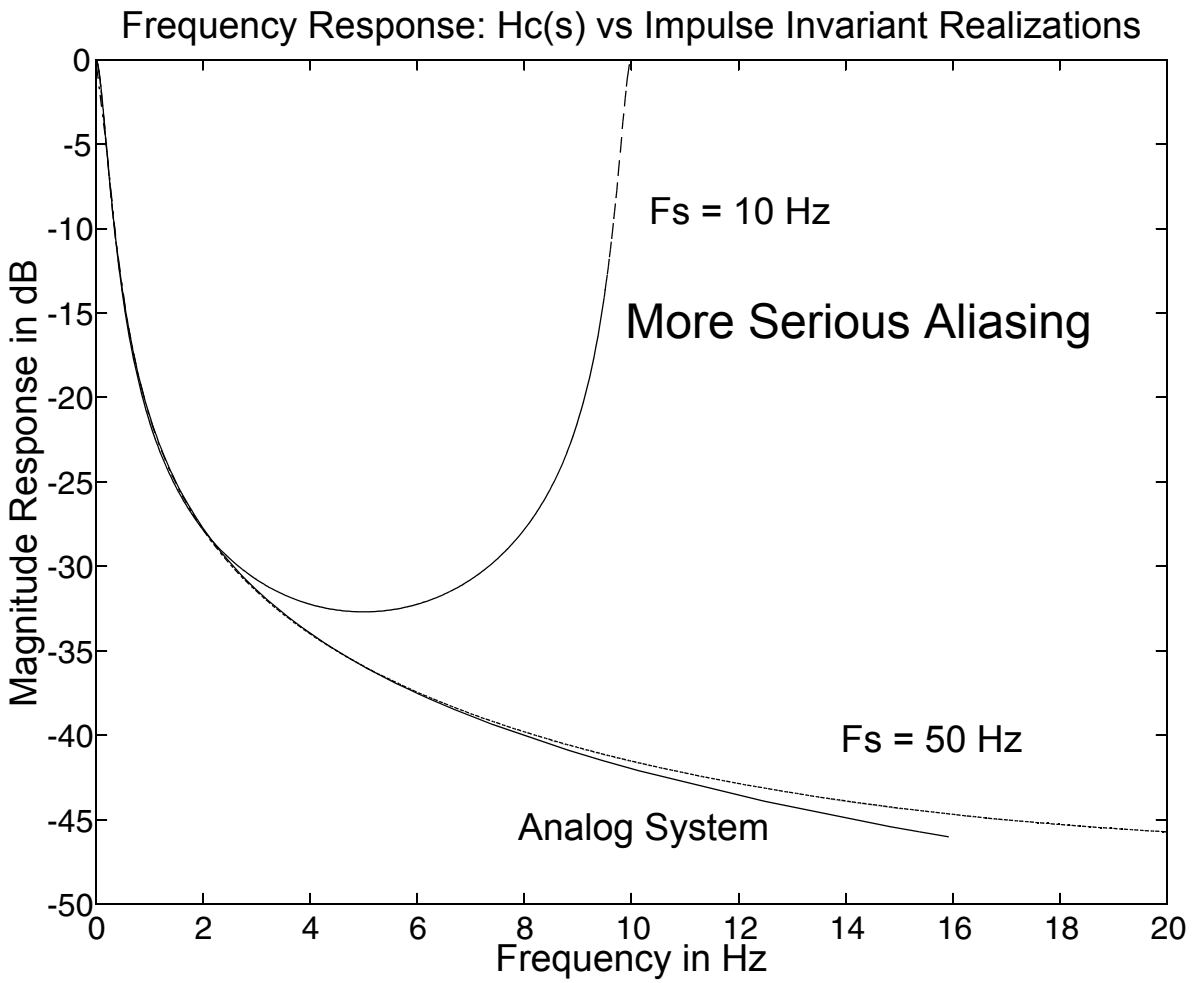
- Suppose we are we are given

$$H_a(s) = \frac{0.5(s+4)}{(s+1)(s+2)}$$

- The MATLAB command line dialog to complete the design is the following:

```
» as = conv([1 1],[1 2])
as = 1     3     2
» bs = 0.5*[1 4]
```

```
bs = 0.5000    2.0000
» %Now use the impinvar design function
» %The third arg is the sampling frequency in Hz
» [bz10,az10] = impinvar(bs,as,10)
bz10 =  0.5000   -0.3233
az10 =  1.0000   -1.7236    0.7408
» [bz50,az50] = impinvar(bs,as,50)
bz50 =  0.5000   -0.4610
az50 =  1.0000   -1.9410    0.9418
» [Hc,wc] = freqs(bs,as);
» [H10,F10] = freqz(bz10,az10,256,'whole',10);
» [H50,F50] = freqz(bz50,az50,256,'whole',50);
» plot(wc/(2*pi),20*log10(abs(Hc)))
» hold
Current plot held
» plot(F10,20*log10(abs(H10/H10(1))),'--')
» plot(F50,20*log10(abs(H50/H50(1))),'-.')
» % In the above gain normalization to unity at dc
» % is accomplished by dividing through by H(1).
```



Impulse Invariant Design From Amplitude Specifications

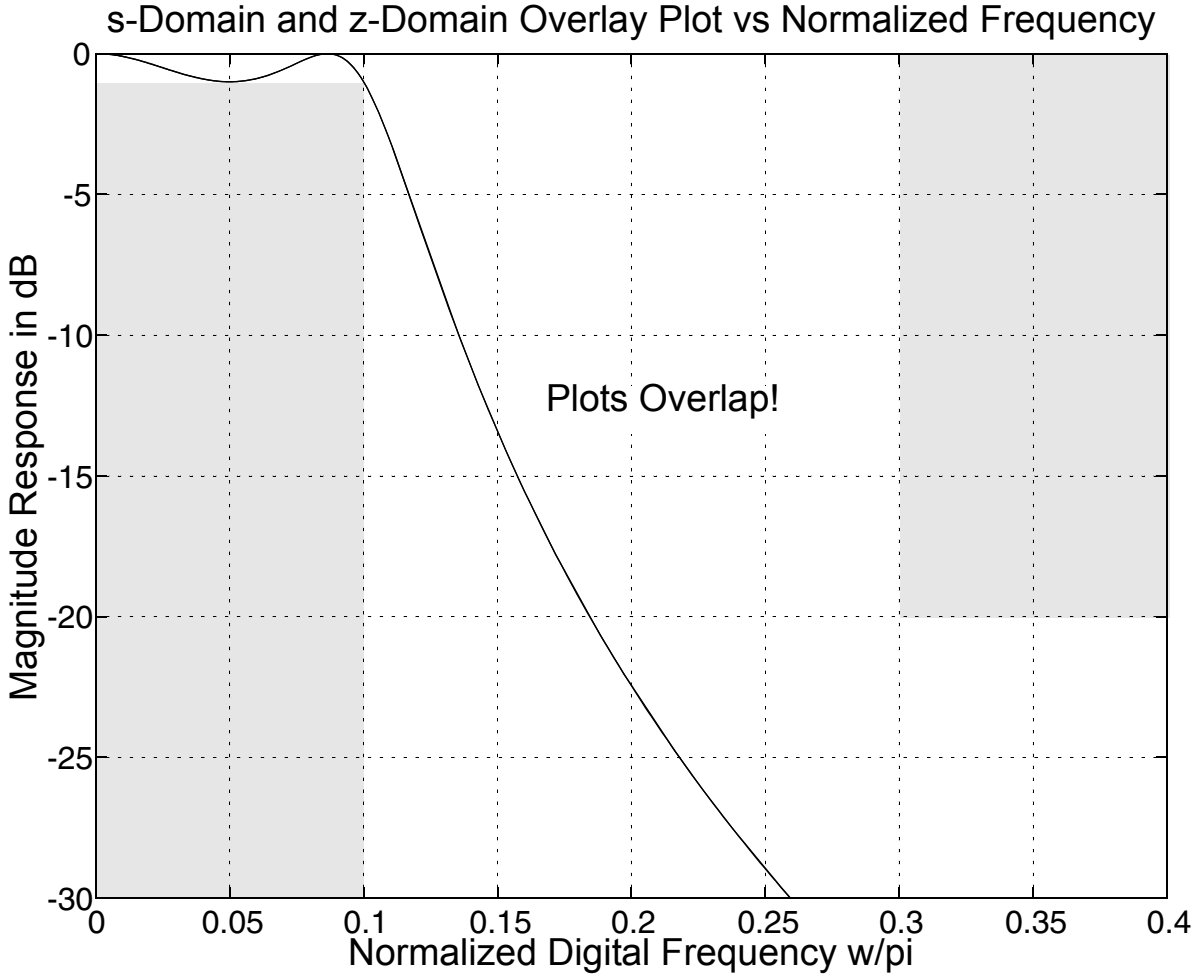
- Consider the following amplitude response requirements

$$\omega_p = 0.1\pi \text{ and } \varepsilon_{\text{dB}} = 1.0 \text{ dB}$$

$$\omega_s = 0.3\pi \text{ and } A_s = 20 \text{ dB}$$

```
» [n,Wn] = cheb1ord(0.1*pi,0.3*pi,1,20,'s')
n =     3
Wn =    0.3142
» [bs,as] = cheby1(n,1,Wn,'s')
» % s-domain design with Fs = 1
bs =  0    0    0    0.0152
as = 1.0000    0.3105    0.1222    0.0152
```

```
» [Hc,W] = freqs(bs,as); %Compute s-domain freq. resp.
» [bz,az] = impinvar(bs,as,1)
» %impulse invariant design using Fs = 1
bz = 0  6.8159e-003  6.1468e-003
az = 1.0000e+000 -2.6223e+000  2.3683e+000 -7.3308e-001
» [H,w] = freqz(bz,az); %Compute z-domain aliased resp.
```



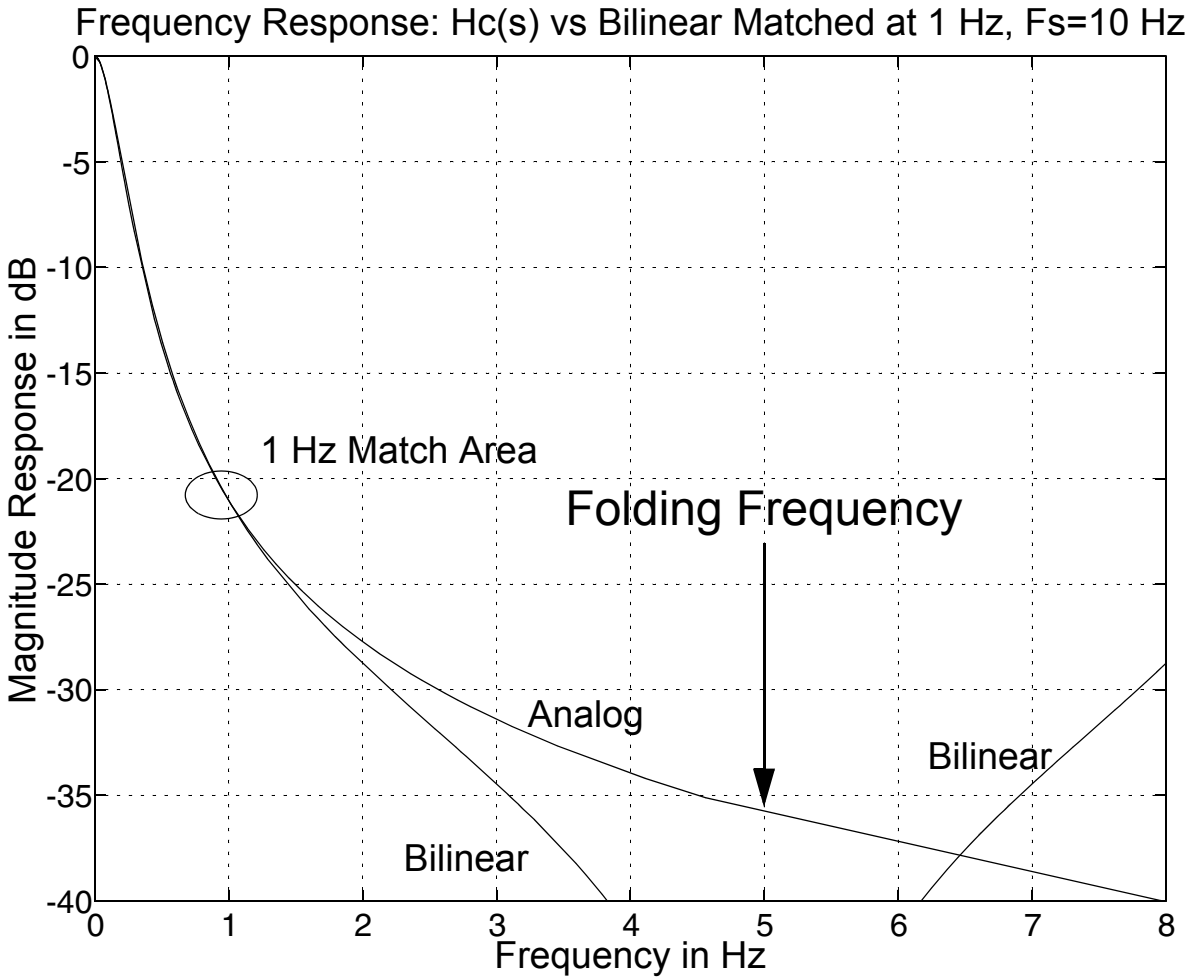
Bilinear Transform Design From Amplitude Specifications (rework of a previous example using the bilinear transform)

- In this problem we are again given

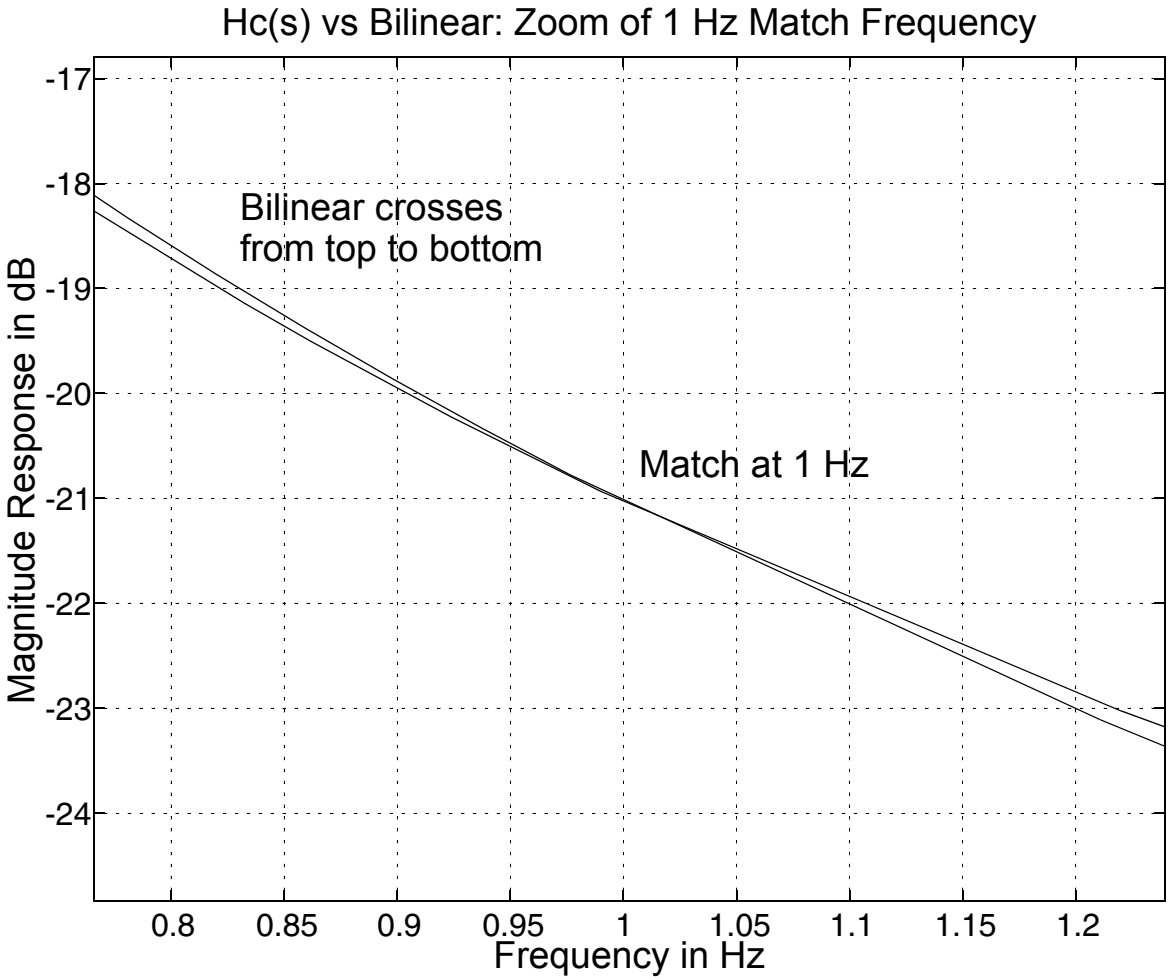
$$H_c(s) = \frac{0.5(s+4)}{(s+1)(s+2)}$$

- Assume a sampling rate of 10 Hz and an analog vs. digital filter matching frequency of 1 Hz (i.e., frequency warping is used to insure the 1 Hz point is invariant under the bilinear transform).

```
» as = conv([1 1],[1 2]);
» bs = 0.5*[1 4];
» [bz,az] = bilinear(bs,as,10,1)
bz =    0.0269    0.0092   -0.0177
az =    1.0000   -1.7142    0.7326
» [Hc,wc] = freqs(bs,as);
» [H10,F10] = freqz(bz,az,256,'whole',10);
```

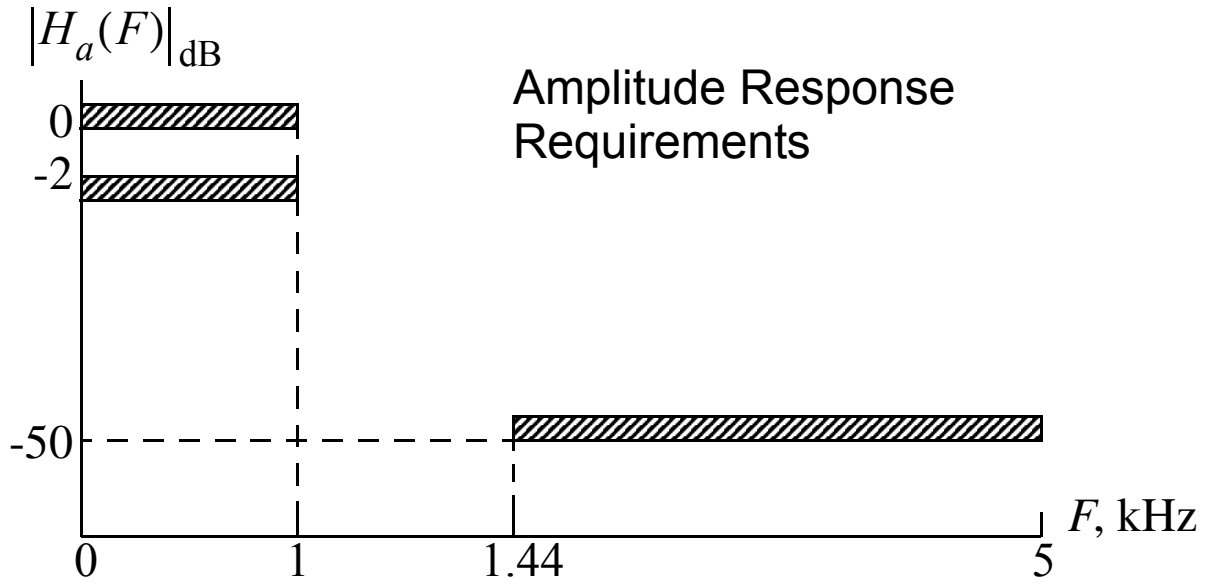


- Zoom in to the 1 Hz point to see if the desired matching condition is satisfied



A Bilinear Transform Design Starting From Analog Frequency Response Requirements

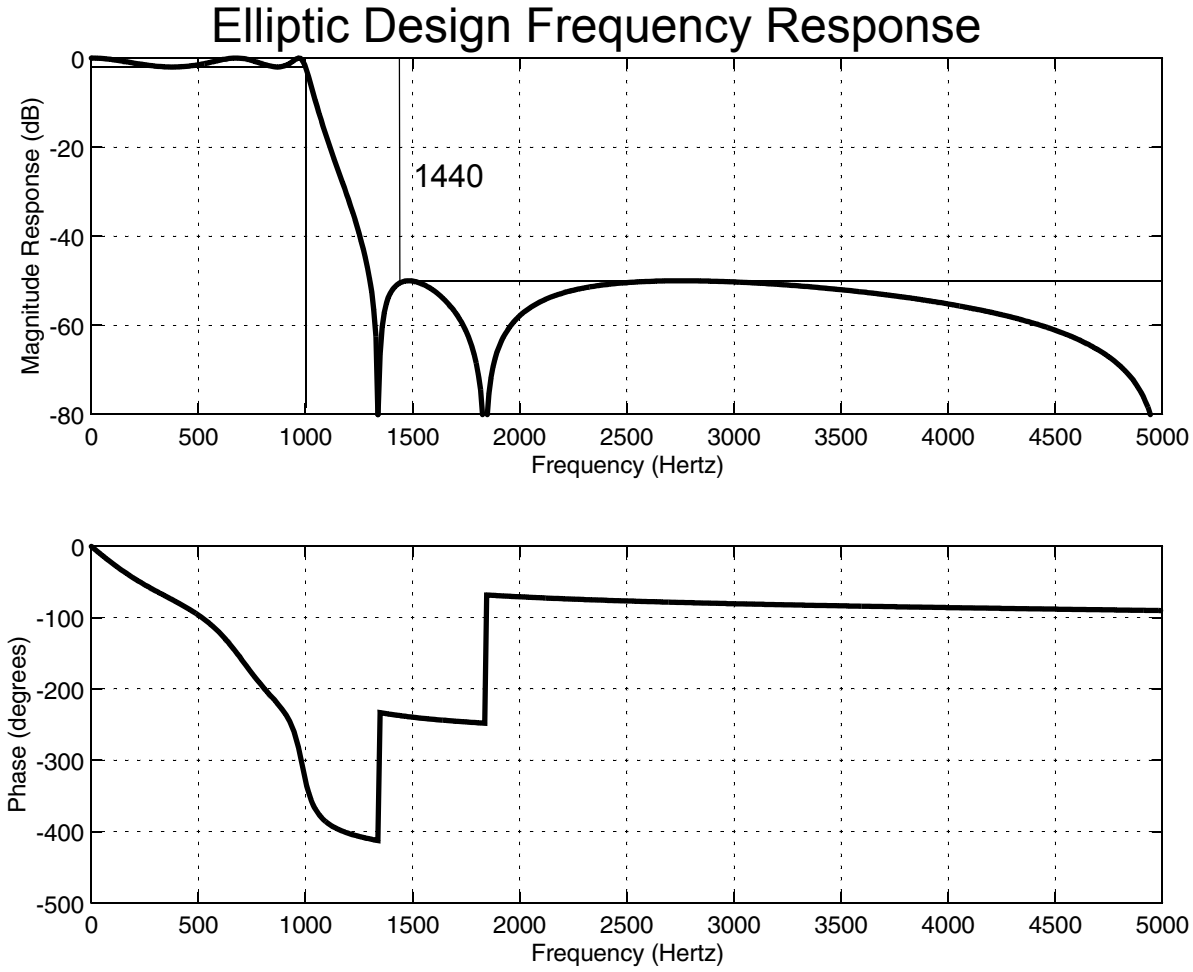
- A analog filter requirement is to be implemented using a system of the form A/D-$H(z)$-D/A
- The sampling rate is chosen to be 10 kHz and the resulting analog frequency response should be as shown below



- The above amplitude response translates directly to the discrete-time domain by simply re-scaling the frequency axis
- The MATLAB design procedure is the following:

```
» % Requirements for a Butterworth Design
» [n_but,Wn_but] = buttord(2*1/10,2*1.44/10,2,50);
» n_but =    15
» Wn_but =  2.0356e-001 %normalized to the
» %                      folding frequency
» % Requirements for a Chebyshev Type I Design
» [n_cheb1,Wn_cheb1] = cheb1ord(2*1/10,2*1.44/10,2,50);
» n_cheb1 =     8
» Wn_cheb1 =  2.0000e-001
» % Requirements for a Chebyshev Type II Design
» [n_cheb2,Wn_cheb2] = cheb2ord(2*1/10,2*1.44/10,2,50);
» n_cheb2 =     8
» Wn_cheb2 =  2.6727e-001
» % Requirements for an Elliptic Design
» [n_ellip,Wn_ellip] = ellipord(2*1/10,2*1.44/10,2,50);
» n_ellip =     5
» Wn_ellip =  2.0000e-001
```

```
» % Design the Elliptic Filter
» [b_ellip,a_ellip] = ellip(n_ellip,2,50,Wn_ellip);
» b_ellip =  6.7088e-003 -7.6635e-003  6.2843e-003
            6.2843e-003 -7.6635e-003  6.7088e-003
» a_ellip =  1.0000e+000 -4.0516e+000  7.0334e+000
           -6.4667e+000  3.1400e+000 -6.4443e-001
» freqz(b_ellip,a_ellip,512,10000)
```



- To implement this $N = 5$ design as a cascade of biquad sections we need to convert `b_ellip` and `a_ellip`, the numerator and denominator filter coefficient vectors into products of second-order polynomials

- Three sections will be required, with one section really only being first-order
- The MATLAB function we need to use is either `zp2sos` or `ss2sos`
  - `zp2sos` converts a zero-pole form (`zp`) to second-order sections (`sos`); we must first use `tf2zp` which converts the present transfer function (`tf`) form to zero-pole form
  - `ss2sos` converts a state space form (`ss`) to second-order sections; we must first use `tf2ss` which converts transfer function form to state space from
- Here we will transform to `sos` via `ss`

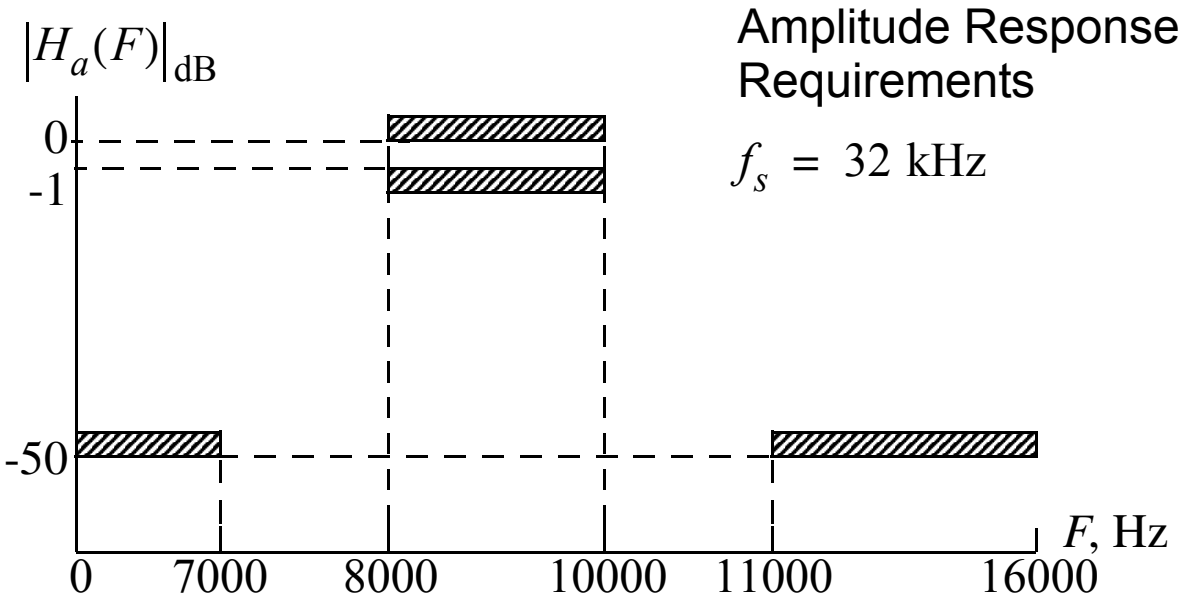
```
» % Get the Second-Order Section Coefficients
» [A,B,C,D] = tf2ss(b_ellip,a_ellip);
» sos_ellip = ss2sos(A,B,C,D);
» sos_ellip =
 0.0802    0.0802         0    1.0000   -0.8395         0
 0.1540   -0.1247    0.1540    1.0000   -1.6262    0.8095
 0.5428   -0.7234    0.5428    1.0000   -1.5860    0.9483
```

- The `sos` form produced by MATLAB is a matrix where each row is the coefficients of a biquad arranged as follows

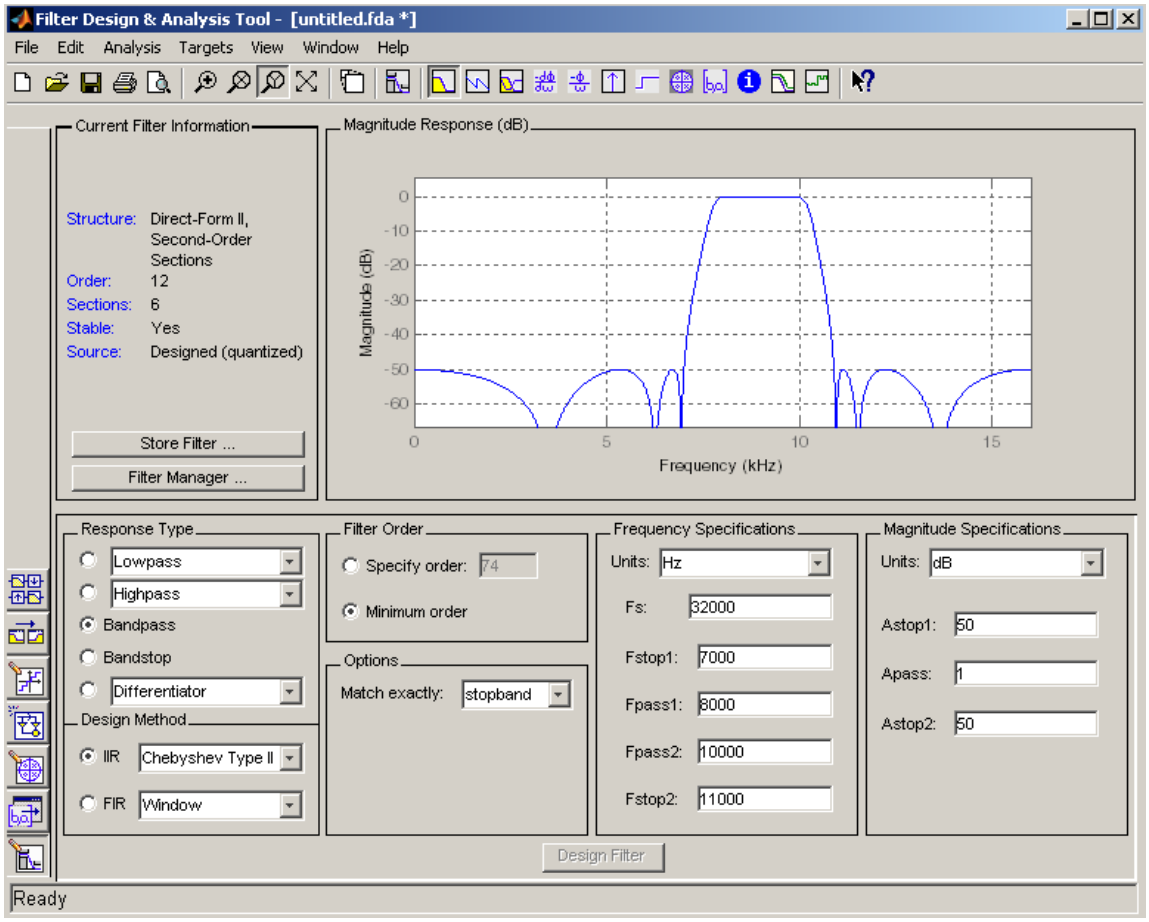
```
SOS = [ b01 b11 b21  a01 a11 a21;
        b02 b12 b22  a02 a12 a22;
            ...
        b0L b1L b2L  a0L a1L a2L ]
```

  - Note: that the first row of `sos_ellip` is the first-order section

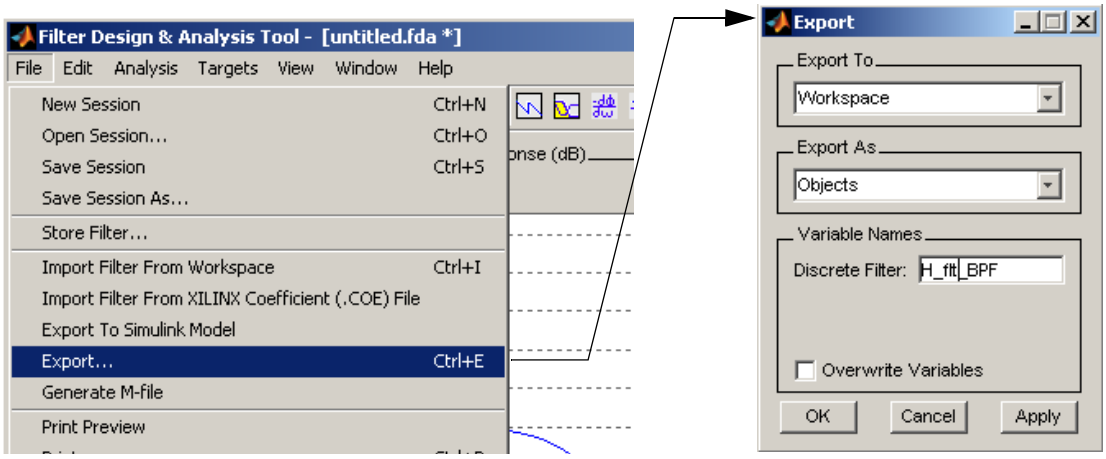
Chebyshev Type II Bandpass Filter Design



- In this example we will use `fdatool` to design a quantized coefficient filter using second-order direct form II sections



- The quantization invoked is single precision float
- The lowpass prototype is 6th-order, but when transformed to a bandpass design the number of poles increases from 6 to 12
- We export the filter to the workspace as an object:



- Once in the MATLAB we can study filter object and its data structures

```
>> H_flt_BPF
>> H_flt_BPF

H_flt_BPF =

     FilterStructure: 'Direct-Form II, Second-Order Sections'
          Arithmetic: 'single' % <<---- This single precision float
           sosMatrix: [6x6 double]
         ScaleValues: [7x1 double]
    PersistentMemory: false

>> H_flt_BPF.ScaleValues

ans =

   0.60013782978058
   0.60013782978058
   0.46401774883270
   0.46401774883270
   0.28024104237556
   0.28024104237556
   1.00000000000000
```

```
>> H_flt_BPF.sosMatrix

ans =

  Columns 1 through 4 % Note 1-->3 = num coeffs, 4-->5 den coeffs

   1.00000000000000   1.09410989284515   1.00000000000000   1.00000000000000
   1.00000000000000  -0.41578763723373   1.00000000000000   1.00000000000000
   1.00000000000000   1.27380359172821   1.00000000000000   1.00000000000000
   1.00000000000000  -0.67229676246643   1.00000000000000   1.00000000000000
   1.00000000000000   1.78818154335022   1.00000000000000   1.00000000000000
   1.00000000000000  -1.55479288101196   1.00000000000000   1.00000000000000

  Columns 5 through 6

  -0.08468919992447   0.90287345647812
   0.80284541845322   0.91137528419495
   0.68900525569916   0.73233383893967
  -0.03038517385721   0.71177399158478
   0.46334517002106   0.56319200992584
   0.15073190629482   0.54895496368408
```

- To get the quantized coefficients into a usable form for a C program, we could use the generate C header option
- Instead we will just manipulate the filter object to create a custom header file that combines the section-by-section scale values into the numerator of each biquad
- For each biquad we need to export 3 numerator coefficients and 2 denominator coefficients

Writing C Coefficient Files

- To make it easier to port filter design results from MATLAB to CCS an m-file that generates C style header files in cascade form was written

```
function sos_C_header(Hq,mode,filename);
%      sos_C_header(Hq,mode,filename): Used to create a C-style header file
%      containing filter coefficients. This reads Hq filter objects assuming a
%      direct-form II cascade of second-order sections is present
%
```

```
%        Hq = quantized filter object containing desired coefficients
%      mode = specify 'float' (plan for fixed and hex to be added later
% file_name = string name of file to be created

%Check to see what type of Hq object we have
if strcmp(Hq.FilterStructure,'Direct-Form II, Second-Order Sections') == 0,
      disp('Wrong structure type, no file written.')
      disp(['Type found is: ' Hq.FilterStructure ' not Direct-Form II, Second-
Order Sections!'])
      return
end

dimSOS = size(Hq.sosMatrix);
Ns = dimSOS(1); % Number of biquad sections

num = zeros(Ns,3);
den = zeros(Ns,3);
for i=1:Ns,
    num(i,:) = Hq.sosMatrix(i,1:3);
    num(i,:) = num(i,:)*Hq.ScaleValues(i);
    den(i,:) = Hq.sosMatrix(i,4:6);
end

if length(Hq.ScaleValues) == Ns+1
    scalevalue = Hq.ScaleValues(Ns+1);
else
    scalevalue = 1.0;
end

fid = fopen(filename,'w'); % use 'a' for append

fprintf(fid,'//define number of 2nd-order stages\n');
fprintf(fid,'#define STAGES %d\n',Ns);

kk = 1;
switch lower(mode)
case 'float '
    fprintf(fid,'float b[STAGES][3] =    {                              ');
    fprintf(fid,'/*numerator coefficients */\n');
    for i=1:Ns,
            if i==Ns
                  fprintf(fid,'{%15.12f, %15.12f, %15.12f} ',...
                               num(i,1),num(i,2),num(i,3));
            else
                  fprintf(fid,'{%15.12f, %15.12f,...
                               %15.12f},',num(i,1),num(i,2),num(i,3));
            end
                  fprintf(fid,' /*b0%1d, b1%1d, b2%1d  */\n',i,i,i);
    end
    fprintf(fid,'};\n');
```

```
    fprintf(fid,'float a[STAGES][2] =    {                              ');
    fprintf(fid,'/*denominator coefficients*/\n');
    for i=1:Ns,
            if i==Ns
                  fprintf(fid,'{%15.12f, %15.12f} ',den(i,2),den(i,3));
            else
                  fprintf(fid,'{%15.12f, %15.12f},',den(i,2),den(i,3));
            end
            fprintf(fid,'                  /*a1%1d, a2%1d  */\n',i,i);
    end
    fprintf(fid,'};\n');
    fprintf(fid,'float scalevalue =  %15.12f;', scalevalue);
    fprintf(fid,'                  /* final output scale value */\n');
otherwise
      disp('Unknown mode!')
end
fprintf(fid,'/
*****************************************************************/\n');
fclose(fid);
```

- Currently this program only supports `float` format, but support for fixed i.e., decimal or hex format would be easy to add if we knew how best to scale the coefficients
- Using the `H_flt_BPF` object for the Chebyshev bandpass filter we enter into MATLAB:

```
>>
sos_C_header(H_flt_BPF,'float','iir_sos_fltcoeff.h');
```

- The header file obtained is:

```
//define number of 2nd-order stages
#define STAGES 6
float b[STAGES][3] =    {                          /*numerator coefficients */
{ 0.600137829781,  0.656616736634,  0.600137829781}, /*b01, b11, b21  */
{ 0.600137829781, -0.249529890259,  0.600137829781}, /*b02, b12, b22  */
{ 0.464017748833,  0.591067475089,  0.464017748833}, /*b03, b13, b23  */
{ 0.464017748833, -0.311957630267,  0.464017748833}, /*b04, b14, b24  */
{ 0.280241042376,  0.501121859665,  0.280241042376}, /*b05, b15, b25  */
{ 0.280241042376, -0.435716777653,  0.280241042376}  /*b06, b16, b26  */
};
float a[STAGES][2] =    {                          /*denominator coefficients*/
{-0.084689199924,  0.902873456478},                /*a11, a21  */
{ 0.802845418453,  0.911375284195},                /*a12, a22  */
{ 0.689005255699,  0.732333838940},                /*a13, a23  */
```

```
{-0.030385173857,  0.711773991585},                    /*a14, a24  */
{ 0.463345170021,  0.563192009926},                    /*a15, a25  */
{ 0.150731906295,  0.548954963684}                     /*a16, a26  */
};
float scalevalue =   1.000000000000;                   /* final output scale value */
/*************************************************************/
```

- To utilize these arrays of filter coefficients we need a corresponding filter algorithm

Filter Implementation Code Examples

A Simple Floating-Point Cascade of Biquads Implementation

- The CCS 5.1 project `sos_iir_float_AIC3106` contains the ISR module `ISRs_sos_iir_float.c`

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011
// Modified by Mark Wickert February 2012 to include GPIO ISR start/stop postings

////////////////////////////////////////////////////////////////////////
// Filename: ISRs_sos_iir_float
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
////////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"
#include "iir_sos_fltcoeff.h"   //coefficients in decimal format

// Function Prototypes
long int rand_int(void);

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word.  The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
```

```
	Uint32 UINT;
	Int16 Channel[2];
} CodecDataIn, CodecDataOut;


/* add any global variables here */
float dly[STAGES][2];		//buffer for delay samples

interrupt void Codec_ISR()
////////////////////////////////////////////////////////////////////////
// Purpose:   Codec interface interrupt service routine
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
	/* add any local variables here */
	WriteDigitalOutputs(1); // Write to GPIO J15, pin 6; begin ISR timing pulse
	float xRight;
	//Define filter variables
	int i;
	float wn, input;
	float result = 0; //initialize the accumulator

	if(CheckForOverrun())// overrun error occurred (i.e. halted DSP)
		return;					// so serial port is reset to recover

	CodecDataIn.UINT = ReadCodecData();// get input data samples

	//input  = CodecDataIn.Channel[ LEFT];
	//xRight = CodecDataIn.Channel[ RIGHT];
	//*********** Input Noise Testing ***************
	//Generate left and right noise samples
	input = ((short)rand_int())>>2;
	xRight = ((short)rand_int())>>2;
	//*******************************************

	/* add your code starting here */
	//Biquad section filtering stage-by-stage
	//using a float accumulator.
	for (i = 0; i < STAGES; i++)
	{
```

```
            //2nd-order LCCDE code
            wn = input - a[i][0] * dly[i][0] - a[i][1] * dly[i][1];     //8.8
            result = b[i][0]*wn + b[i][1]*dly[i][0] + b[i][2]*dly[i][1];//8.9
            //Update filter buffers for stage i
            dly[i][1] = dly[i][0];
            dly[i][0] = wn;
            input = result; /*in case we have to loop again*/
        }
        //result *= scalevalue; //Apply cascade final stage scale factor

        CodecDataOut.Channel[ LEFT] = (short) result; /* scaled L output */
        CodecDataOut.Channel[RIGHT] = (short) xRight; /* scaled R output */
        /* end your code here */

        WriteCodecData(CodecDataOut.UINT);// send output data to  port
        WriteDigitalOutputs(0); // Write to GPIO J15, pin 6; end ISR timing pulse
}

//White noise generator for filter noise testing
long int rand_int(void)
{
        static long int a = 100001;

        a = (a*125) % 2796203;
        return a;
}
```

- The C implementation of the biquad difference equations follows directly from (8.8) and (8.9)
- In particular the array `dly[STAGES][2]` holds the filter state information for each stage of the cascade
  - Ideally this array should be initialized to zeros
- The feedforward filter coefficients for each cascade section are contained in the array `b[STAGES][3]` where

$$\texttt{b[i][0]} = b_{i0}, i = 0, 2, \ldots N_s - 1 \quad (8.28)$$

$$\texttt{b[i][1]} = b_{i1}, i = 0, 2, \ldots N_s - 1 \quad (8.29)$$

$$\texttt{b[i][2]} = b_{i2}, i = 0, 2, \ldots N_s - 1 \quad (8.30)$$

- The feedback filter coefficients for each cascade section are contained in the array `a[STAGES][2]`

$$\texttt{a[i][0]} = a_{i1}, i = 0, 2, \ldots N_s - 1 \quad (8.31)$$

$$\texttt{a[i][1]} = a_{i2}, i = 0, 2, \ldots N_s - 1 \quad (8.32)$$

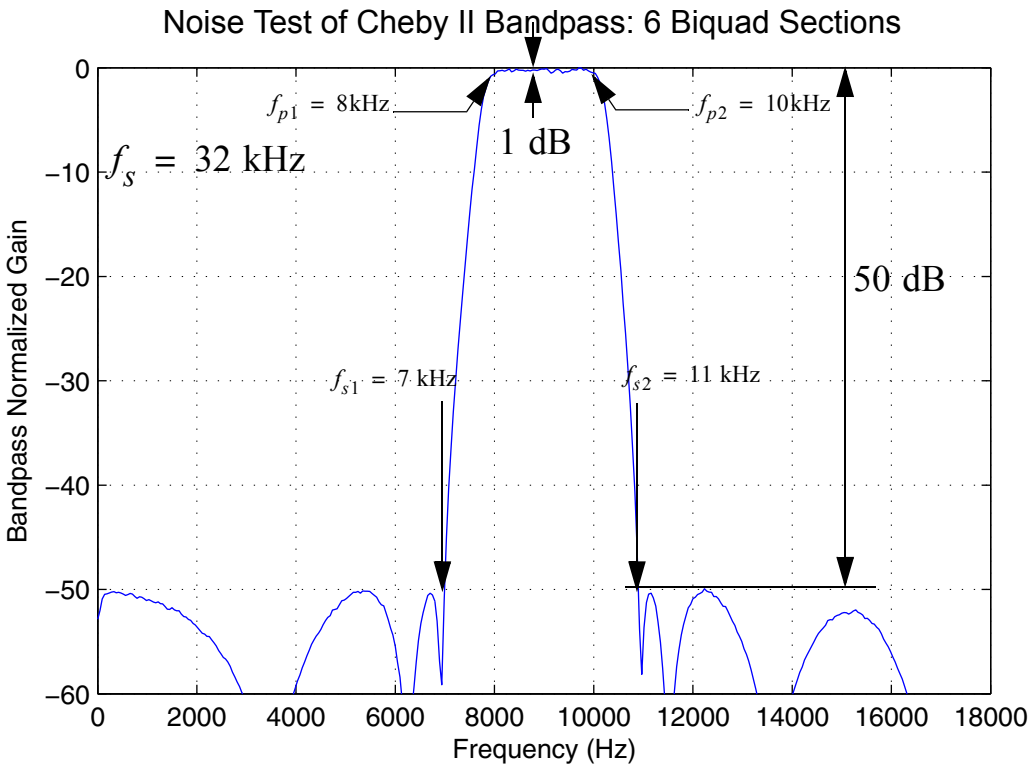
  – Note that we do not need to store a_{i0} since this coefficient is by definition unity

- Each section takes in `input` and produces output in `result`
- We set `input` = `result` at the end of each biquad loop to allow the sample to propagate to the next section until looping is complete
- The filter was tested using a white noise input generated in software

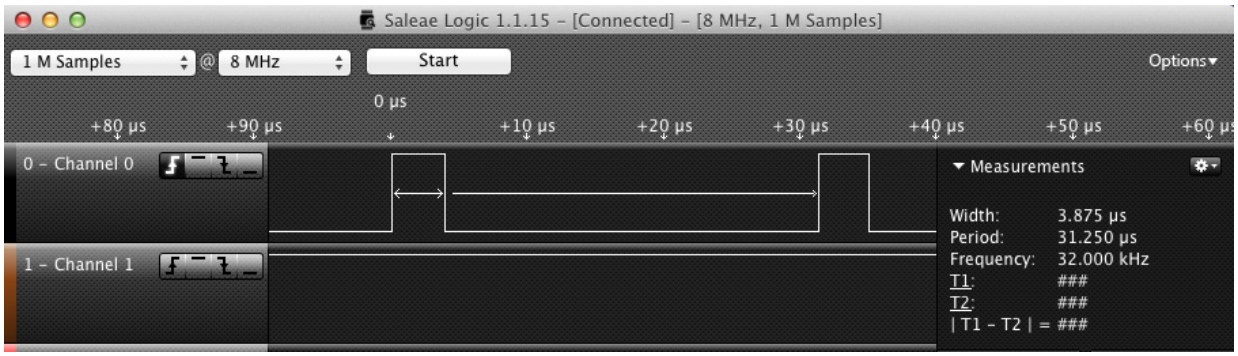
```
//*********** Input Noise Testing ***************
//Generate left and right noise samples
input = ((short)rand_int())>>2;
xRight = ((short)rand_int())>>2;
//***********************************************
```

- A 30 second sound file created at 48 ksps, 16-bits, was imported into MATLAB and then the power spectrum was estimated using `psd()`

```
[Px,F] = simpleSA(x(:,1),2^10,48000,-90,10,'b');
>> plot(F,10*log10(Px/max(Px)))
```



- The results are as expected (again watch for over-loading on the PC audio input)
  - Note: The capture here was done on the Mac
- Code timing results:



  - Here we see that the 6-stage IIR (12th-order filter) is using 3.875 µs

Coupled Form Oscillator in C

- To implement the coupled form oscillator we modify `ISRs_sos_iir_float.c` code is modified:

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011
// Modified by Mark Wickert March 2012 to include GPIO ISR start/stop postings

////////////////////////////////////////////////////////////////////////
// Filename: ISRs_osc_iir_float
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
////////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"

// Function Prototypes
long int rand_int(void);

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word.  The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
      Uint32 UINT;
      Int16 Channel[2];
} CodecDataIn, CodecDataOut;


/* add any global variables here */
float dly1;               //filter LCCDE variables
float dly2;               //initialized in main
float cos_w0;
float sin_w0;

interrupt void Codec_ISR()
////////////////////////////////////////////////////////////////////////
// Purpose:   Codec interface interrupt service routine
//
// Input:     None
```

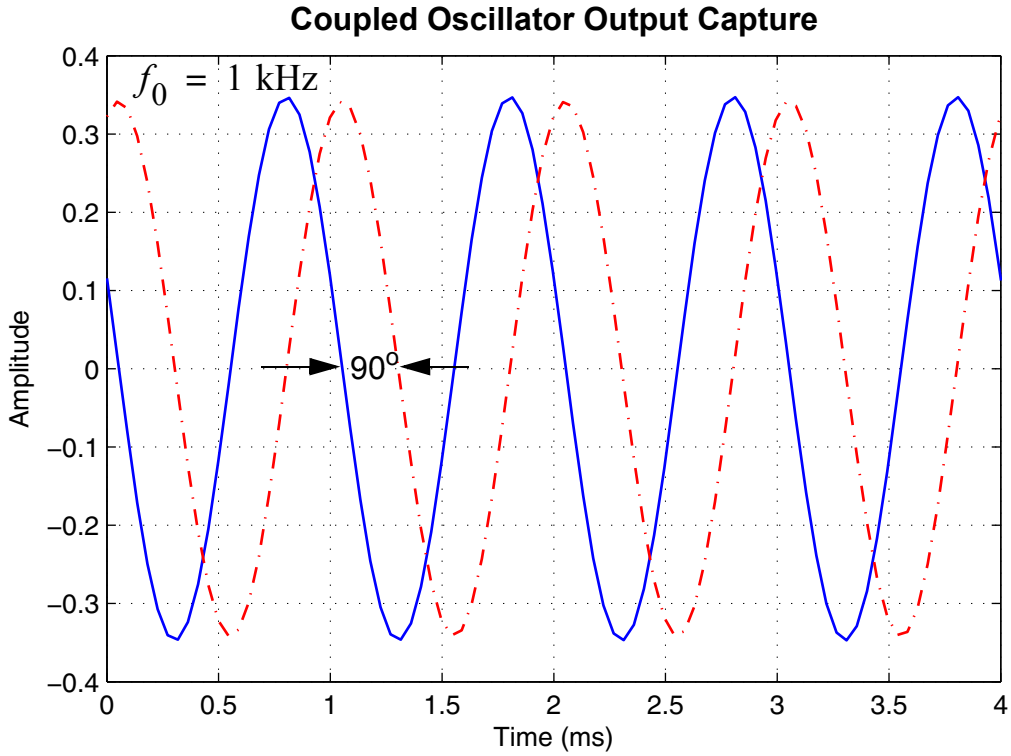
```
//
// Returns:   Nothing
//
// Calls:     CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes:     None
/////////////////////////////////////////////////////////////////////////
{
       /* add any local variables here */
       WriteDigitalOutputs(1); // Write to GPIO J15, pin 6; begin ISR timing pulse
       //Define output variables
       float yc, ys;

       if(CheckForOverrun())// overrun error occurred (i.e. halted DSP)
              return;                    // so serial port is reset to recover

       CodecDataIn.UINT = ReadCodecData();// get input data samples

       //input  = CodecDataIn.Channel[ LEFT];
       //xRight = CodecDataIn.Channel[ RIGHT];

       /* add your code starting here */
       //Coupled form oscillator difference equations
    yc = cos_w0*dly1 - sin_w0*dly2;
    ys = sin_w0*dly1 + cos_w0*dly2;
       //Update filter states
    dly1 = yc;
    dly2 = ys;

       CodecDataOut.Channel[ LEFT] = (short) yc; /* scaled L output */
       CodecDataOut.Channel[RIGHT] = (short) ys; /* scaled R output */
       /* end your code here */

       WriteCodecData(CodecDataOut.UINT);// send output data to  port
       WriteDigitalOutputs(0); // Write to GPIO J15, pin 6; end ISR timing pulse
}

//White noise generator for filter noise testing
long int rand_int(void)
{
       static long int a = 100001;

       a = (a*125) % 2796203;
       return a;
}
```

- Variables are initialized in `main.c`

```
//////////////////////////////////////////////////////////////////////
// Filename: main.c
//
// Synopsis: Main program file for demonstration code
//
//////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"

#include <math.h>        // ANSI C math functions
extern float dly1;  //filter LCCDE variables
extern float dly2;  //initialized in main
extern float cos_w0;
extern float sin_w0;

int main()
{

      //Define filter variables
      float A = 10000;
      float pi = 3.14159265359;
      //initialize oscillator variables
      cos_w0 = cos(2*pi/32.0); //Fo ~1 kHz as Fs ~32 kHz
      sin_w0 = sin(2*pi/32.0);
      dly1 = A*cos_w0;  //delay samples                */
      dly2 = A*sin_w0;

      // initialize DSP board
      DSP_Init();

      // call StartUp for application specific code
      // defined in each application directory
      StartUp();

      // main stalls here, interrupts drive operation
      while(1) {
            ;
      }
```

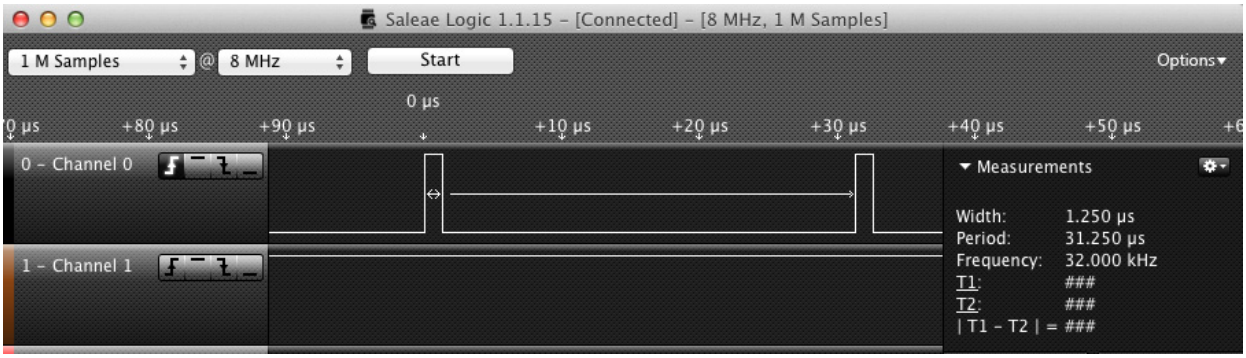
- The program outputs a 1 KHz cosine wave with a sampling rate of 32 ksps
- Output is captured using a PC sound card and imported into MATLAB to create a plot showing the quadrature relationship between the two 1 kHz outputs

```
>> t = ((0:200)*1/44100)';
>> plot(t,osc_1khz.data(100:300,1));
>> hold
Current plot held
>> plot(t,osc_1khz.data(100:300,2),'r');
```



- For a VCO or numerically controlled oscillator (NCO) design, the phase accumulator based oscillator of Problem Set #3 is better
- This can be implemented more efficiently than in the home-work, using a lookup table e.g., with fixed-point values

- The interrupt timing results are shown below:



  – The single biquad is relatively fast

A *Peaking Filter* with a MATLAB GUI to Control the App Using WinDSK8

- A peaking filter is used to provide gain or loss (attenuation) at a specific center frequency f_c.
- The peaking filter has unity frequency response magnitude or 0 dB gain, at frequencies far removed from the center frequency
- At the center frequency f_c, the frequency response magnitude in dB is G_{dB}
- At the heart of the peaking filter is a second-order IIR filter

$$H_{\text{pk}}(z) = C_{\text{pk}}\left(\frac{1 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}\right) \tag{8.33}$$

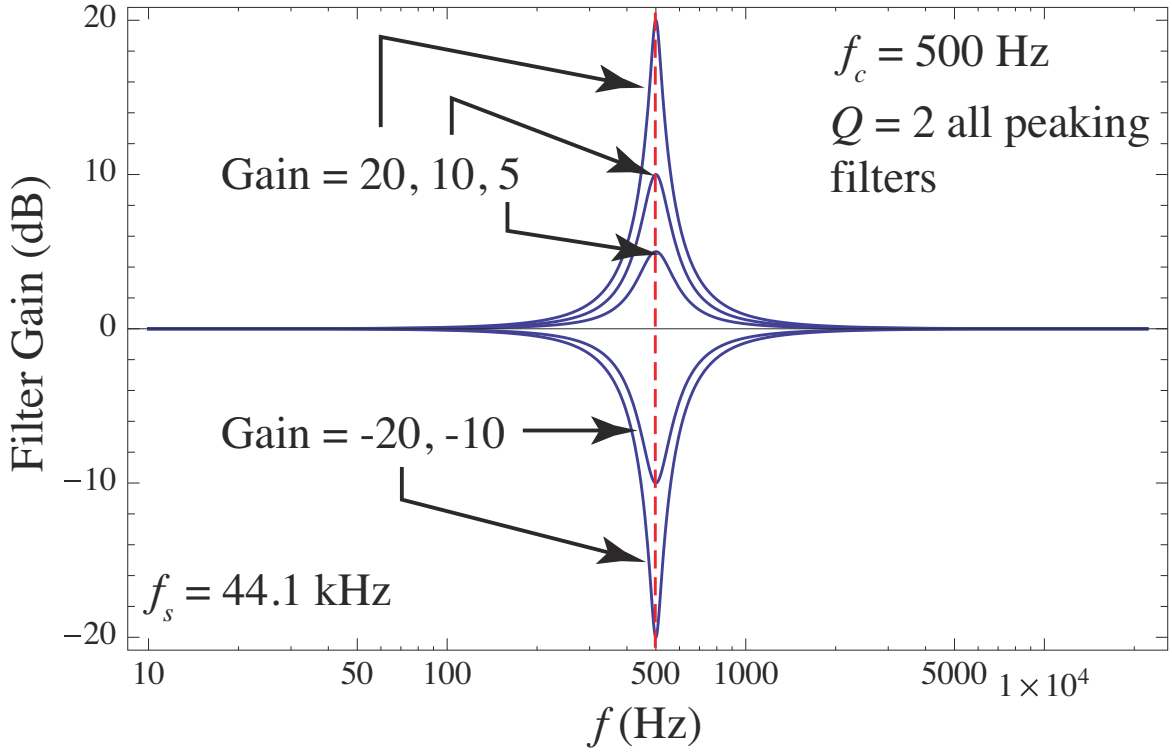
where

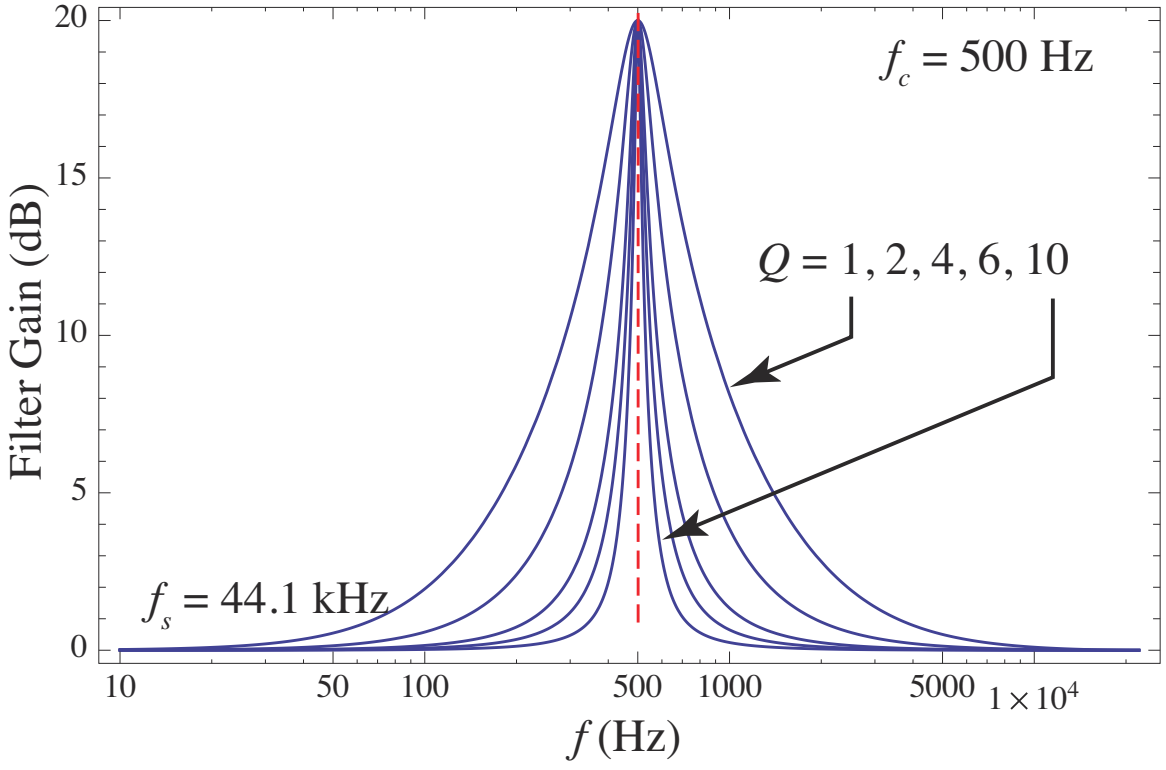
$$
\begin{aligned}
C_{\text{pk}} &= \frac{1 + k_q \mu}{1 + k_q} \\
b_1 &= \frac{-2\cos(\hat{\omega}_c)}{1 + k_q \mu}, \; b_2 = \frac{1 - k_q \mu}{1 + k_q \mu} \\
a_1 &= \frac{-2\cos(\hat{\omega}_c)}{1 + k_q}, \; a_2 = \frac{1 - k_q}{1 + k_q} \\
k_q &= \frac{4}{1 + \mu} \cdot \tan\left(\frac{\hat{\omega}_c}{2Q}\right)
\end{aligned}
\tag{8.34}
$$

and

$$
\mu = 10^{G_{\text{dB}}/20} \tag{8.35}
$$

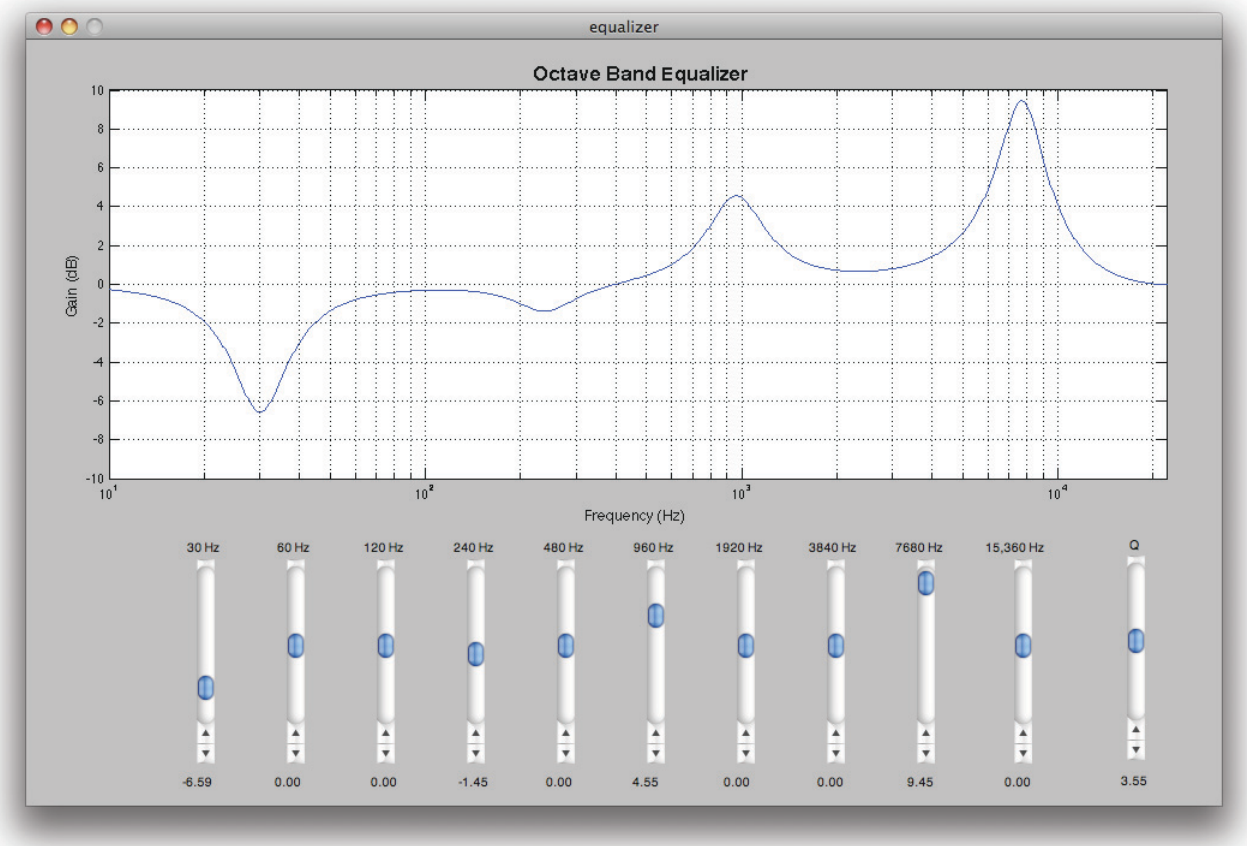
- The peaking filter is parameterized in terms of the peak gain, G_{dB}, the center frequency f_c, and a parameter Q, which is inversely proportional to the peak bandwidth
- Examples of the peaking filter frequency response can be found in the following figures
- The impact of changing the gain at the center frequency, f_c, can be seen in the first figure
- The impact of changing Q can be seen in the second figure





- Peaking filters are generally placed in cascade to form a graphic equalizer, e.g., perhaps using 10 octave spaced center

frequencies



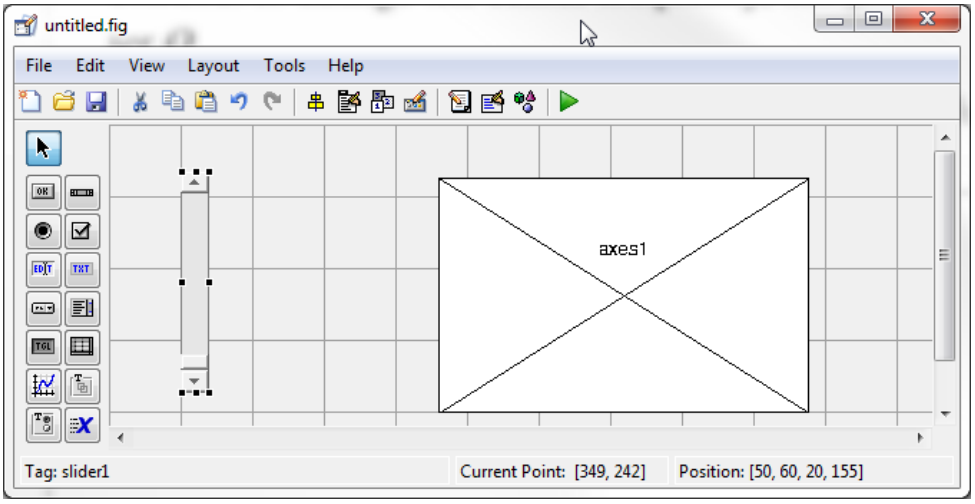
- With the peaking filters in cascade, and the gain setting of each filter at 0 dB, the cascade frequency response is unity gain (0 dB) over all frequencies from 0 to $f_s/2$
- In the figure above the octave band center frequencies are set starting at 30 Hz and then according to the formula

$$f_{ci} = 30 \times 2^i \text{ Hz}, i = 0, 1, \ldots 9 \qquad (8.36)$$

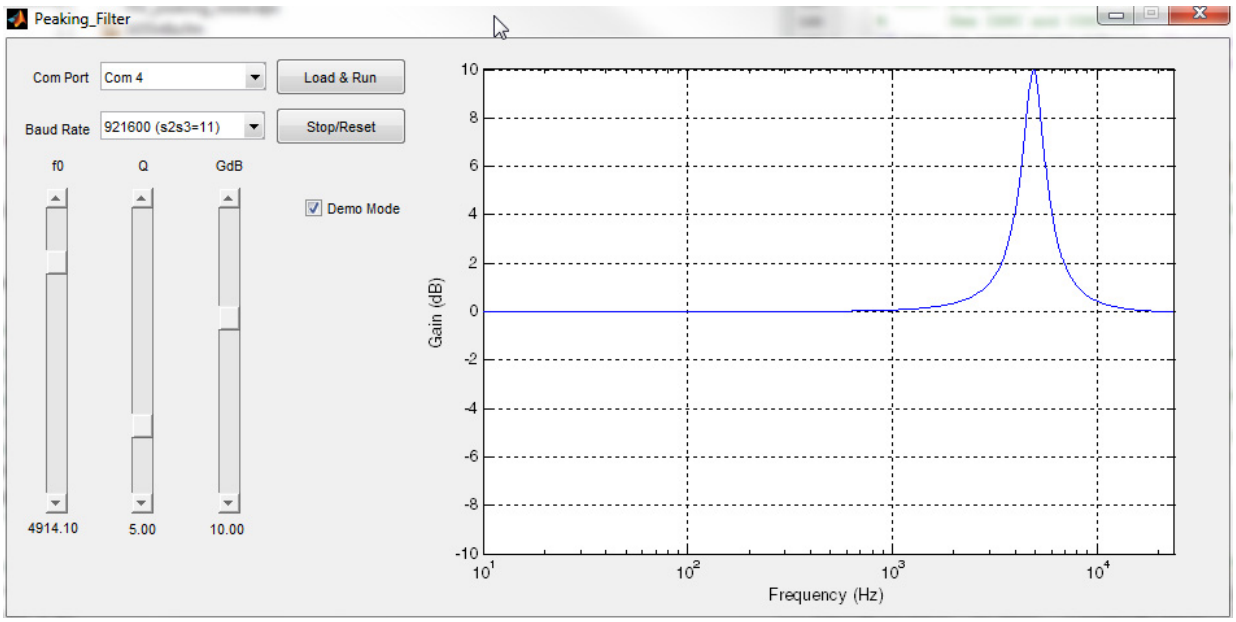
- The idea here being a means to reasonably cover the 20 to 20 kHz audio spectrum
- With $f_s = 44.1$ kHz, we have a usable bandwidth up to $f_s/2 = 22.05$ kHz
- Building the 10 band equalizer will be left as an exercise

A Single Band Implementation

- Here we concentrate on a single band, but will provide slider controls to change the center frequency, the gain, and the filter Q
- We first develop the GUI using MATLAB's *graphical user interface development environment* (GUIDE)



- Screen shot of the completed GUI, with the code created by GUIDE stored in the file `Peaking_Filter.fig` (**do not** attempt to edit this file, it is maintained by GUIDE)



- To support the operation of the interface two supporting m-code functions are also created (1) `peaking.m` which outputs a and b filters vector given design parameters, and (2) a plotting utility, `sl_freqz_mag.m`, that facilitates log frequency plots in the discrete-time domain

```
function [b,a] = peaking(GdB, fc, Q, fs)
% [b,a] = peaking(GdB, fc, Q, fs)
% Seond-order peaking filter having GdB gain at fc and approximately
% and 0 dB otherwise.
%
%////// Inputs ////////////////////////////////////////////////////////
%
% GdB = Lowpass gain in dB
%  fc = Center frequency in Hz
%   Q = Filter Q which is inversely proportional to bandwidth
%  fs = Sampling frquency in Hz
%
%////// Outputs ///////////////////////////////////////////////////////
%
%   b = Numerator filter coefficients in MATLAB form
%   a = Denominator filter coefficients in MATLAB form
%
% Mark Wickert, April 2009

mu = 10^(GdB/20);
kq = 4/(1 + mu)*tan(2*pi*fc/fs/(2*Q));
Cpk = (1 + kq *mu)/(1 + kq);
b1 = -2*cos(2*pi*fc/fs)/(1 + kq*mu);
b2 = (1 - kq*mu)/(1 + kq*mu);
a1 = -2*cos(2*pi*fc/fs)/(1 + kq);
a2 = (1 - kq)/(1 + kq);

b = Cpk*[1 b1 b2];
a = [1 a1 a2];

function [H,f] = sl_freqz_mag(b,a,fs,style)
% [H,f] = sl_freqz_mag(b,a,fs,style)
% Digital filter magnitude respponse plot in dB using a logarithmic
% frequnecy axis.
```

```
%
%/////////////// Inputs //////////////////////////////////////////////
%
%      b = Denominator coefficient vector
%      a = Numerator coefficient vector
%     fs = Sampling  frequency in Hz
% style = String variable list of plot options used by plot() etc.
%
%////////////// Output ///////////////////////////////////////////////
%
% Output plot to plot window only
%
% Mark Wickert, April 2009

%f = logspace(0,log10(fs/2)); % start at 1 Hz
f = logspace(1,log10(fs/2),1024); % start at 10 Hz
w = 2*pi*f/fs;
H = freqz(b,a,w);

if nargin == 3
    style = 'b';
end

semilogx(f,20*log10(abs(H)),style);
ymm = ylim;
axis([10,fs/2,ymm(1),ymm(2)]);
grid on
%set(gca,'XminorGrid','off') % turn of minor xaxis grid
xlabel('Frequency (Hz)')
ylabel('Filter Gain (dB)')
```

- The code behind the GUI, `Peaking_Filter.m`, also written by GUIDE, consists of all of the function calls that respond to events associated with user interaction

```
function varargout = Peaking_Filter(varargin)
% PEAKING_FILTER MATLAB code for Peaking_Filter.fig
%      PEAKING_FILTER, by itself, creates a new PEAKING_FILTER or raises the
existing
%      singleton*.
%
%      H = PEAKING_FILTER returns the handle to a new PEAKING_FILTER or the
```

```
handle to
%      the existing singleton*.
%
%      PEAKING_FILTER('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in PEAKING_FILTER.M with the given input
arguments.
%
%      PEAKING_FILTER('Property','Value',...) creates a new PEAKING_FILTER or
raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before Peaking_Filter_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to Peaking_Filter_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help Peaking_Filter

% Last Modified by GUIDE v2.5 01-Apr-2012 22:09:17

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Peaking_Filter_OpeningFcn, ...
                   'gui_OutputFcn',  @Peaking_Filter_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT
```

```
% --- Executes just before Peaking_Filter is made visible.
function Peaking_Filter_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to Peaking_Filter (see VARARGIN)

% Choose default command line output for Peaking_Filter
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes Peaking_Filter wait for user response (see UIRESUME)
% uiwait(handles.figure1);
global fs a_peak1 b_peak1 demo_true
global f0 Q GdB com_port baudrate;
fs = 48000; f0 = 1000; Q = 5.0; GdB = 5.0;
demo_true = 1;
com_port = 4; % set initial values to match control initial settings
baudrate = 3;

[b_peak1, a_peak1] =  peaking(GdB, f0, Q, fs);

set(handles.slider_f0,'Value',log10(f0));
set(handles.text_f0,'String',sprintf('%6.2f',f0));
set(handles.slider_Q,'Value',Q);
set(handles.text_Q,'String',sprintf('%6.2f',Q));
set(handles.slider_GdB,'Value',GdB);
set(handles.text_GdB,'String',sprintf('%6.2f',GdB));
Plot_Response()

% --- Outputs from this function are returned to the command line.
function varargout = Peaking_Filter_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
```

```
varargout{1} = handles.output;


% --- Executes on selection change in popupmenu_port.
function popupmenu_port_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu_port (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns popupmenu_port
contents as cell array
%        contents{get(hObject,'Value')} returns selected item from
popupmenu_port
global com_port;
com_port = get(hObject,'Value');

% --- Executes during object creation, after setting all properties.
function popupmenu_port_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu_port (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on selection change in popupmenu_baudrate.
function popupmenu_baudrate_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu_baudrate (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns popupmenu_baudrate
contents as cell array
%        contents{get(hObject,'Value')} returns selected item from
popupmenu_baudrate
global baudrate;
baudrate = get(hObject,'Value');
baudrate = baudrate - 1; % board number is one less than control 'Value'
```

```
% --- Executes during object creation, after setting all properties.
function popupmenu_baudrate_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu_baudrate (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on button press in pushbutton_run.
function pushbutton_run_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton_run (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global a_peak1 b_peak1
global com_port baudrate;
C8X_DEBUG('silent');%
%C8X_DEBUG('debug');
% Portnumber 1 <> com1, 2, <> com2, etc.
% Baudrate index: 0 <> 115200, 1 <> 230400, 2 <> 460800, 3 <>921600
C8X_DEBUG('init', com_port, baudrate);
% Used to display the C8X_DEBUG mex version
C8X_DEBUG('version');
% Used to load and then the given .out file
C8X_DEBUG('run', 'Peaking_Filter.out');
% Transfer filter coefficients to OMAP
C8X_DEBUG('wfs', a_peak1, '_Avec', 3);
C8X_DEBUG('wfs', b_peak1, '_Bvec', 3);

% --- Executes on button press in pushbutton_stop.
function pushbutton_stop_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton_stop (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
C8X_DEBUG('silent');
C8X_DEBUG('reset');
```

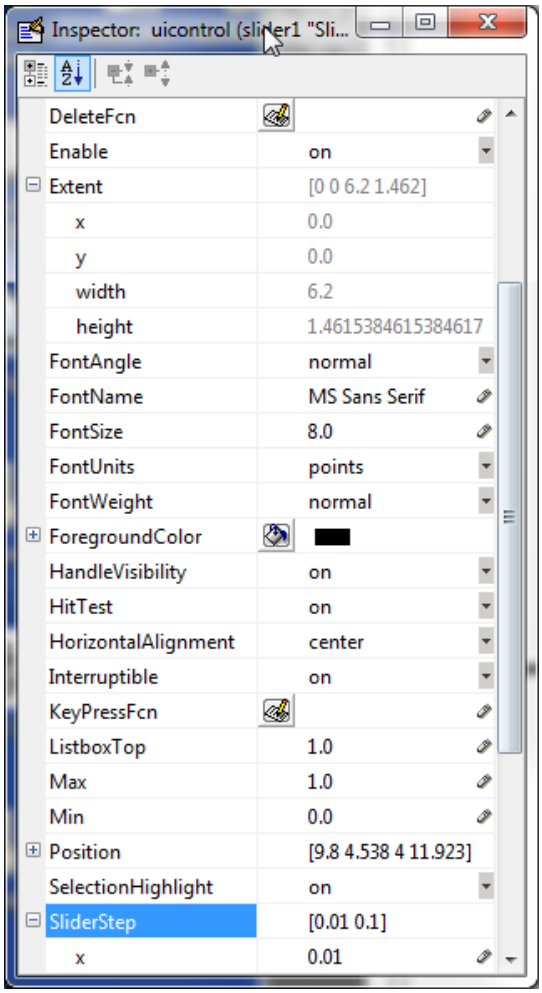
```
% --- Executes on slider movement.
function slider_f0_Callback(hObject, eventdata, handles)
% hObject    handle to slider_f0 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
global fs a_peak1 b_peak1 demo_true
global f0 Q GdB
f0 = 10^get(handles.slider_f0,'Value');
set(handles.text_f0,'String',sprintf('%6.2f',f0));
[b_peak1, a_peak1] =  peaking(GdB, f0, Q, fs);
if demo_true == 0
    C8X_DEBUG('wfs', a_peak1, '_Avec', 3);
    C8X_DEBUG('wfs', b_peak1, '_Bvec', 3);
end
Plot_Response()

% --- Executes during object creation, after setting all properties.
function slider_f0_CreateFcn(hObject, eventdata, handles)
% hObject    handle to slider_f0 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes on slider movement.
function slider_Q_Callback(hObject, eventdata, handles)
% hObject    handle to slider_Q (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
global fs a_peak1 b_peak1 demo_true
global f0 Q GdB
```

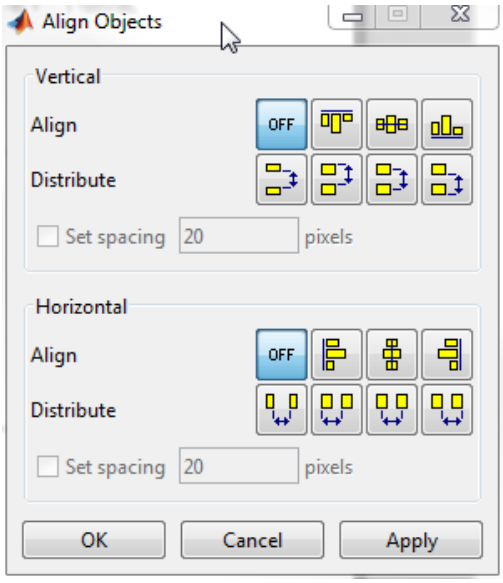
```
Q = get(handles.slider_Q,'Value');
set(handles.text_Q,'String',sprintf('%6.2f',Q));
[b_peak1, a_peak1] =  peaking(GdB, f0, Q, fs);
if demo_true == 0
    C8X_DEBUG('wfs', a_peak1, '_Avec', 3);
    C8X_DEBUG('wfs', b_peak1, '_Bvec', 3);
end
Plot_Response()

% --- Executes during object creation, after setting all properties.
function slider_Q_CreateFcn(hObject, eventdata, handles)
% hObject    handle to slider_Q (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes on slider movement.
function slider_GdB_Callback(hObject, eventdata, handles)
% hObject    handle to slider_GdB (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
global fs a_peak1 b_peak1 demo_true
global f0 Q GdB
GdB = get(handles.slider_GdB,'Value');
set(handles.text_GdB,'String',sprintf('%6.2f',GdB));
[b_peak1, a_peak1] =  peaking(GdB, f0, Q, fs);
if demo_true == 0
    C8X_DEBUG('wfs', a_peak1, '_Avec', 3);
    C8X_DEBUG('wfs', b_peak1, '_Bvec', 3);
end
Plot_Response()

% --- Executes during object creation, after setting all properties.
```

```
function slider_GdB_CreateFcn(hObject, eventdata, handles)
% hObject    handle to slider_GdB (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end

% --- Executes on button press in checkbox_demo.
function checkbox_demo_Callback(hObject, eventdata, handles)
% hObject    handle to checkbox_demo (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of checkbox_demo
global demo_true
demo_true = get(hObject,'Value');

function Plot_Response()
    global fs a_peak1 b_peak1
    [H,f] = sl_freqz_mag(b_peak1, a_peak1,fs,'b');
    semilogx(f,20*log10(abs(H)))
    %axis([0 fs/2 -50 50])
    grid
    set(gca,'XminorGrid','off')
    axis([10 fs/2 -10 10])
    xlabel('Frequency (Hz)')
    ylabel('Gain (dB)')
```

- The last function, `Plot_Response()`, is a support function for updating the magnitude response plot
- With the exception of the last function, the skeleton code for each function is written and maintained by GUIDE
- The GUI code can be tested as it is being written

- GUI control attributes and layout alignment can be set with the GUIDE pallets shown below





- On the OMAP-L138 side we need to implement a single biquad section operating at $f_s = 48\,\text{ksps}$
  - We choose to filter just the left channel and pass the righ channel through, or in the case of noise testing pass the left channel noise through the filter and send the right channel noise directly to the output
  - A default set of biquad filter coefficients will initialize the OMAP program independent of the MATLAB GUI com-

munications; later the GUI can input new filter coefficients

- The complete ISR code is shown below

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011
// Modified by Mark Wickert April 2012 to include GPIO ISR start/stop postings

////////////////////////////////////////////////////////////////////////
// Filename: ISRs_Peaking_iir.c
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
////////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"

// Function Prototypes
long int rand_int(void);

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word.  The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
      Uint32 UINT;
      Int16 Channel[2];
} CodecDataIn, CodecDataOut;


/* add any global variables here */
/* add any global variables here */
float dly1;               //filter LCCDE variables
float dly2;               //initialized in main
// The vectors below are updated from the MATLAB GUI
// Initial values are set with f0  = 1000 Hz, Q = 5, GdB = 5;
volatile float Bvec[3] = { 1.0144F,   -1.9462F,    0.9486F};// b0, b1, and b2 filter
coefficients
volatile float Avec[3] = { 1.0000F,   -1.9462F,    0.9630F};// a0, a1, and a2 filter
coefficients

interrupt void Codec_ISR()
////////////////////////////////////////////////////////////////////////
```

```
// Purpose:    Codec interface interrupt service routine
//
// Input:      None
//
// Returns:    Nothing
//
// Calls:      CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes:      None
////////////////////////////////////////////////////////////////////////
{
       /* add any local variables here */
       WriteDigitalOutputs(1); // Write to GPIO J15, pin 6; begin ISR timing pulse
       float xLeft, xRight;
       float wn, result;


       if(CheckForOverrun())// overrun error occurred (i.e. halted DSP)
              return;                    // so serial port is reset to recover

       CodecDataIn.UINT = ReadCodecData();// get input data samples

       /* add your code starting here */
       // this example simply copies sample data from in to out
       //xLeft  = CodecDataIn.Channel[ LEFT];
       //xRight = CodecDataIn.Channel[ RIGHT];
       //*********** Input Noise Testing ***************
       //Generate left and right noise samples
       xLeft = ((short)rand_int())>>2;
       xRight = ((short)rand_int())>>2;
       //*****************************************

       //2nd-order LCCDE code
       wn = xLeft - Avec[1] * dly1 - Avec[2] * dly2;     //8.8
       result = Bvec[0]*wn + Bvec[1]*dly1 + Bvec[2]*dly2;//8.9
       //Update filter buffers for stage i
       dly2 = dly1;
       dly1 = wn;

       CodecDataOut.Channel[ LEFT] = result;
       CodecDataOut.Channel[RIGHT] = xRight;

       /* end your code here */

       WriteCodecData(CodecDataOut.UINT);// send output data to  port
       WriteDigitalOutputs(0); // Write to GPIO J15, pin 6; end ISR timing pulse
}
```

```
//White noise generator for filter noise testing
long int rand_int(void)
{
      static long int a = 100001;

      a = (a*125) % 2796203;
      return a;
}
```

- The actual communication between the MATLAB GUI and the OMAP-L138 uses Morrow's `MATLAB C8X_DEBUG` interface (see `MATLAB_C8X_DEBUG_API.pdf` for more details)
- The function that runs when the Load & Run button is clicked begins the process:

```
% --- Executes on button press in pushbutton_run.
function pushbutton_run_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton_run (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global a_peak1 b_peak1
global com_port baudrate;
C8X_DEBUG('silent');%
%C8X_DEBUG('debug');
% Portnumber 1 <> com1, 2, <> com2, etc.
% Baudrate index: 0 <> 115200, 1 <> 230400, 2 <> 460800, 3 <>921600
C8X_DEBUG('init', com_port, baudrate);
% Used to display the C8X_DEBUG mex version
C8X_DEBUG('version');
% Used to load and then the given .out file
C8X_DEBUG('run', 'Peaking_Filter.out');
% Transfer filter coefficients to OMAP
C8X_DEBUG('wfs', a_peak1, '_Avec', 3);
C8X_DEBUG('wfs', b_peak1, '_Bvec', 3);
```

  - To minimize the command window text and maximize interface speed we first choose the `'silent'` mode i.e.,

    `C8X_DEBUG('silent');`

  - Most importantly we initialize communications with

    ```
    C8X_DEBUG('init', com_port, baudrate);
    ```

    which is talking via the serial port on the OMAP-L138
  - Finally we can send and receive arrays of data

    ```
    C8X_DEBUG('wfs', a_peak1, '_Avec', 3);
    ```

    which says write floating point data in `a_peak1` to the symbol name `_Avec`, a three element vector created iin CCS 5.1

- Once communications has been established reading writing can take place when needed, e.g., when a slider is moved:

```
% --- Executes on slider movement.
function slider_f0_Callback(hObject, eventdata, handles)
% hObject    handle to slider_f0 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
global fs a_peak1 b_peak1 demo_true
global f0 Q GdB
f0 = 10^get(handles.slider_f0,'Value');
set(handles.text_f0,'String',sprintf('%6.2f',f0));
[b_peak1, a_peak1] =  peaking(GdB, f0, Q, fs);
if demo_true == 0
    C8X_DEBUG('wfs', a_peak1, '_Avec', 3);
    C8X_DEBUG('wfs', b_peak1, '_Bvec', 3);
end
Plot_Response()
```

- We reset the OMAP-L138 when the Stop/Reset button is clicked:

```
% --- Executes on button press in pushbutton_stop.
function pushbutton_stop_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton_stop (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)
C8X_DEBUG('silent');
C8X_DEBUG('reset');
```

Testing

- The complete application is tested by processing left channel filtered noise outputs, then post processing in MATLAB
  - Case 1: $f_0 = 5\,\text{kHz}$, +10 dB gain, and $Q = 5$
  - Case 1: $f_0 = 5\,\text{kHz}$, -10 dB gain, and $Q = 3$
- An overlay plot of the spectra is shown below