Chapter 7

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

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{r=0}^{M} b_r x[n-r] \qquad (7.1)$$

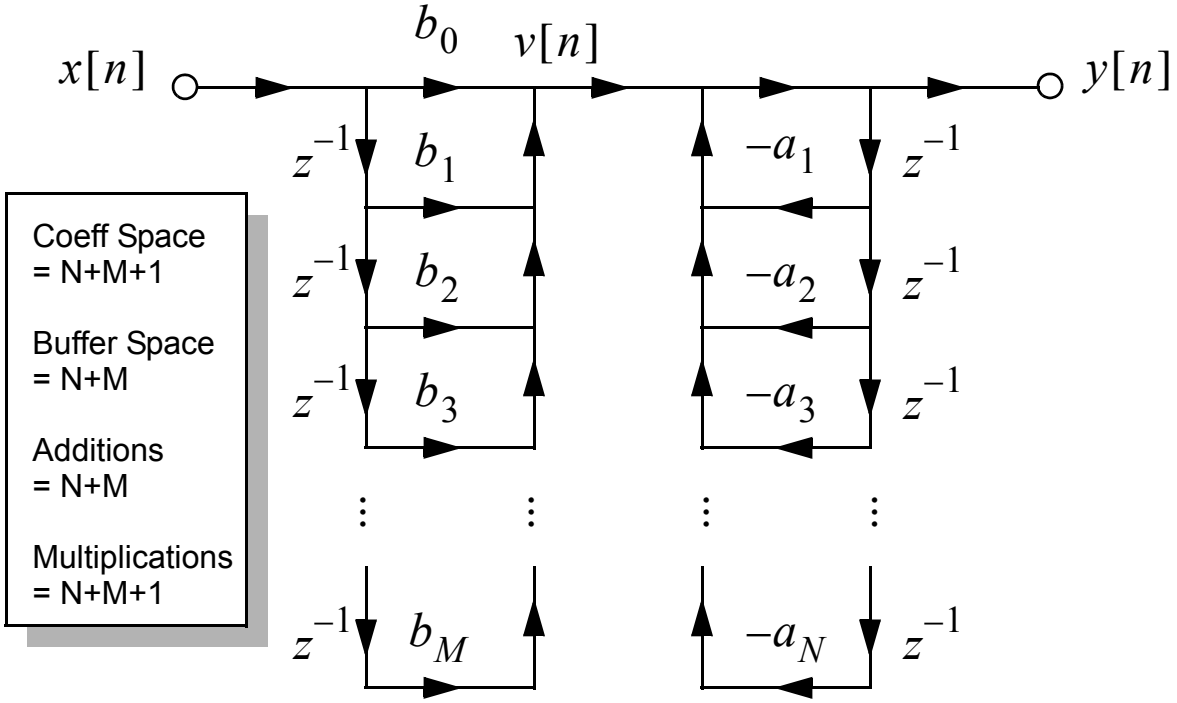
- By z-transforming both sides of (7.1) and using the fact that $H(z) = Y(z)/X(z)$, we can write

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}} \tag{7.2}$$

- 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 (7.1) leads to the following structure



Direct Form I Structure

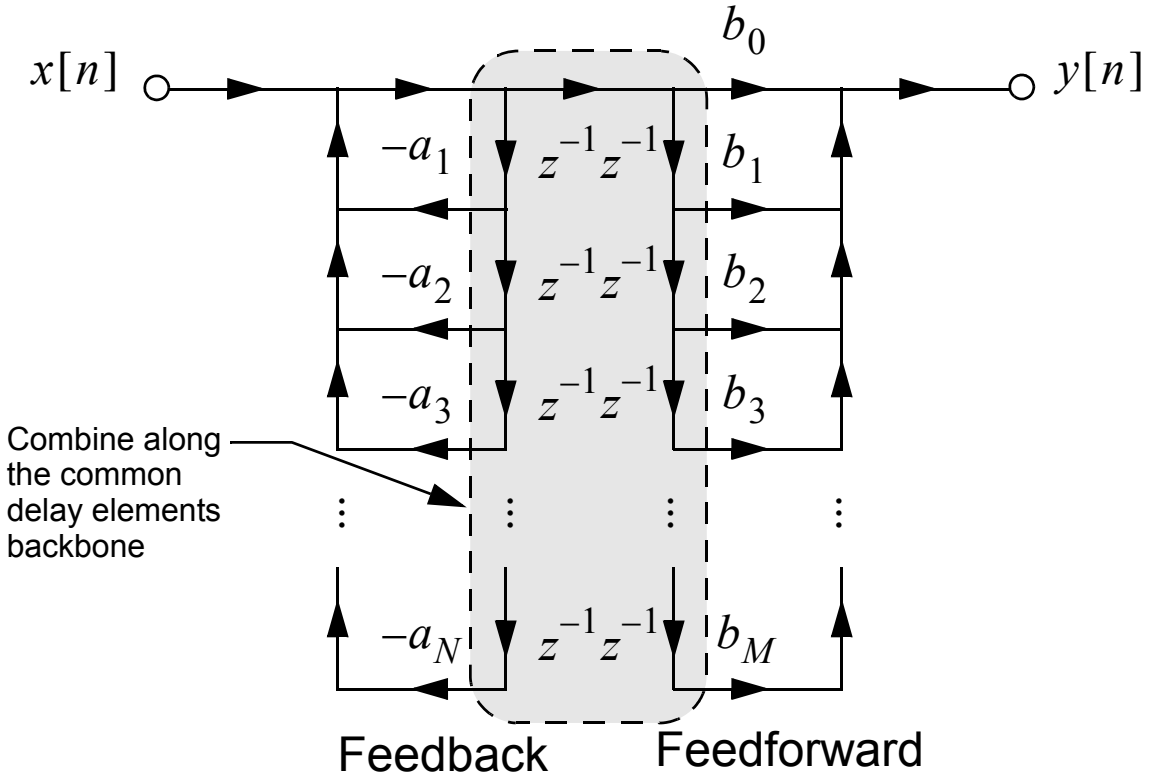
- The calculation of $y[n]$ for each new $x[n]$ requires the ordered solution of two difference equations

$$v[n] = \sum_{k=0}^{M} b_k x[n-k] \tag{7.3}$$

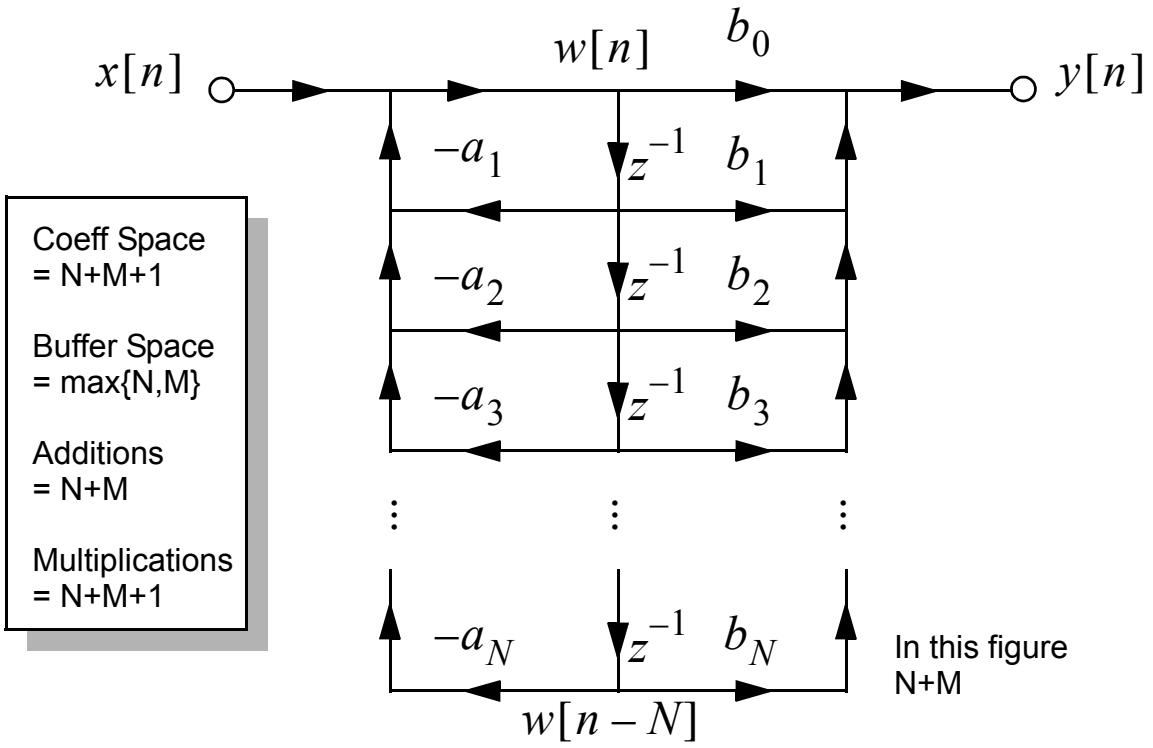
$$y[n] = v[n] - \sum_{k=1}^{N} a_k y[n-k] \tag{7.4}$$

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

$$w[n] = x[n] - \sum_{k=1}^{N} a_k w[n-k] \tag{7.5}$$

$$y[n] = \sum_{k=0}^{M} b_k w[n-k] \tag{7.6}$$

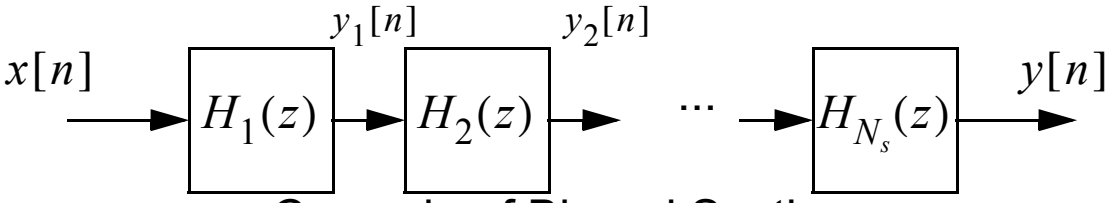
Cascade Form

- 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

$$H(z) = \prod_{k=1}^{N_s} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}} = \prod_{k=1}^{N_s} H_k(z) \qquad (7.7)$$

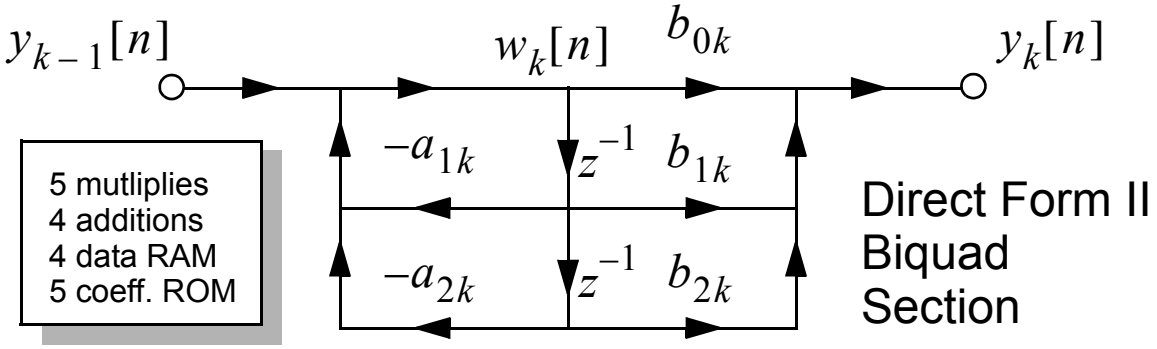
  where $N_s = \lfloor (N+1)/2 \rfloor$ is the largest integer in $(N+1)/2$

- 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)



- The corresponding biquad difference equations are

$$w_k[n] = y_{k-1}[n] - a_{1k}w_k[n-1] - a_{2k}w_k[n-2] \qquad (7.8)$$

$$y_k[n] = b_{0k}w_k[n] + b_{1k}w_k[n-1] + b_{2k}w_k[n-2] \qquad (7.9)$$

- 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

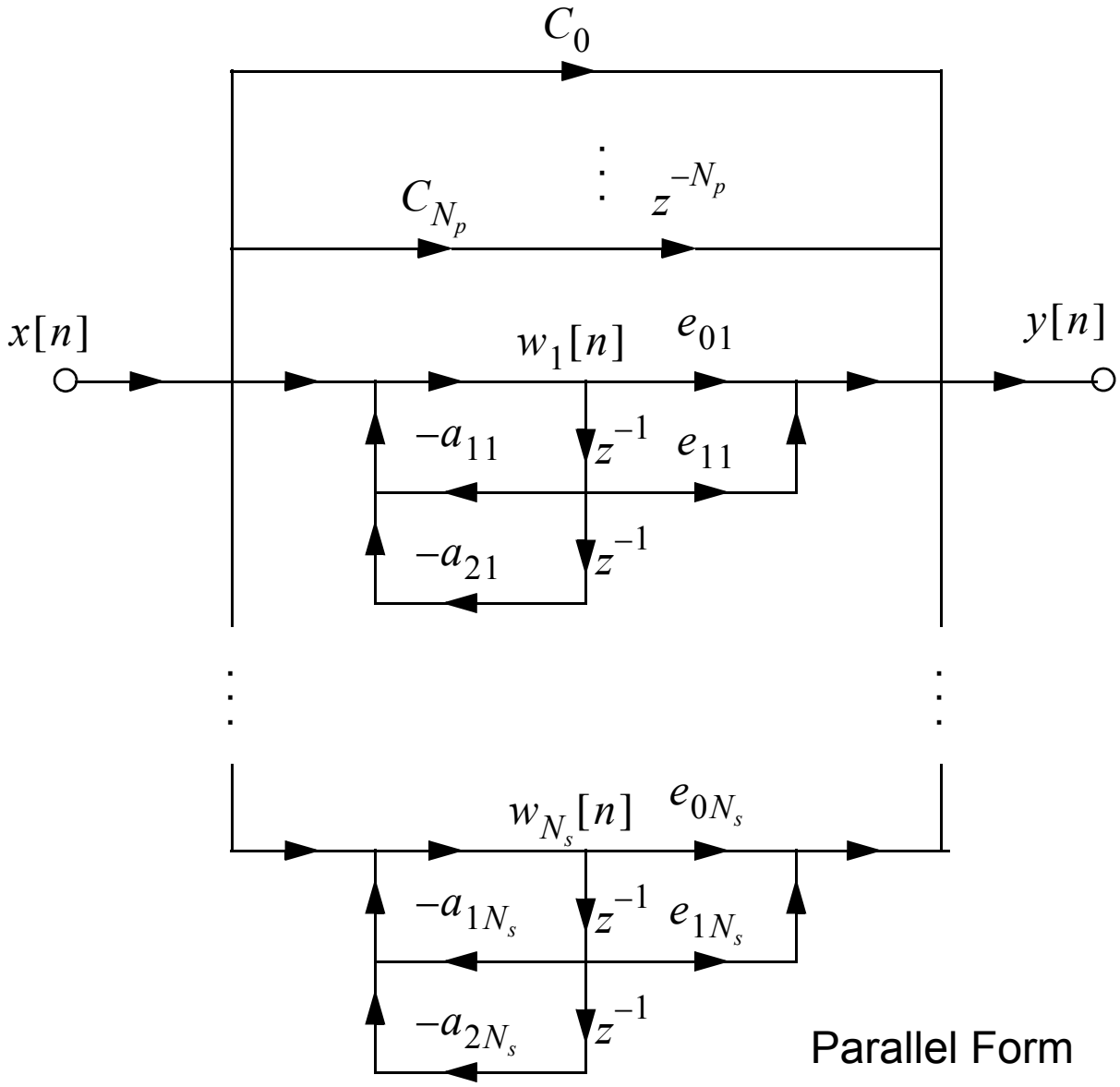
- The parallel form topology is obtained by first using long division to make $H(z)$ a proper rational function, then the remaining system function is expanded using a partial fraction expansion
- Complex conjugate pole pairs are combined into second-order factors, and any real poles are paired up to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

$$H(z) = \sum_{k=0}^{N_p} C_k z^{-k} + \sum_{k=1}^{N_s} \frac{e_{0k} + e_{1k}z^{-1}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}} \qquad (7.10)$$

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

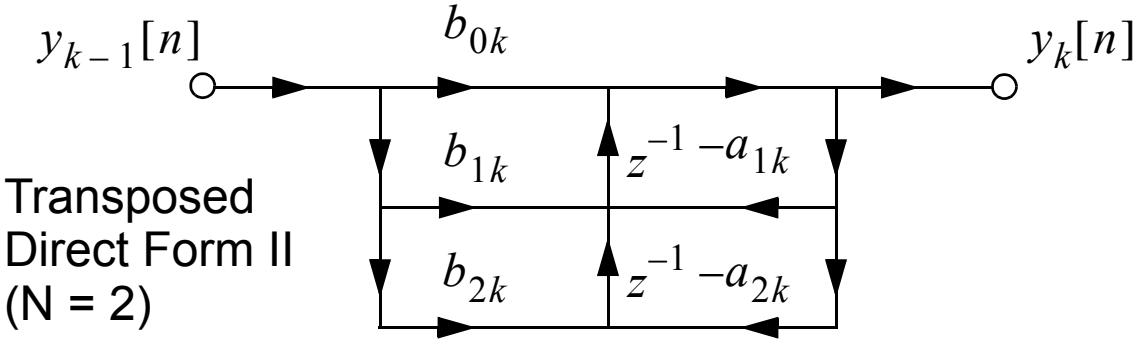
- Again each biquad section is typically implemented using a direct form II structure

- The block diagram follows from (7.10)



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

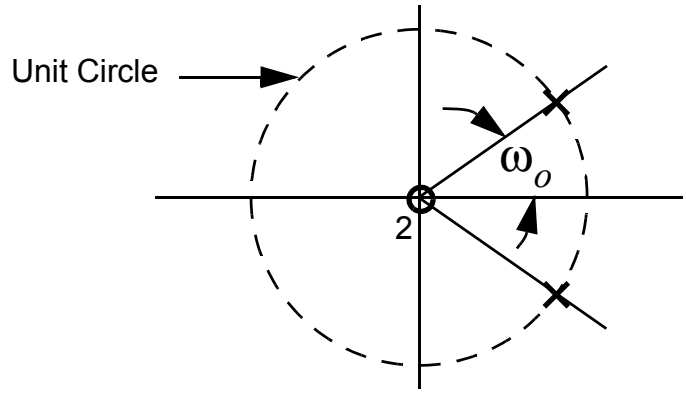
Pole-Zero Lattice

- 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



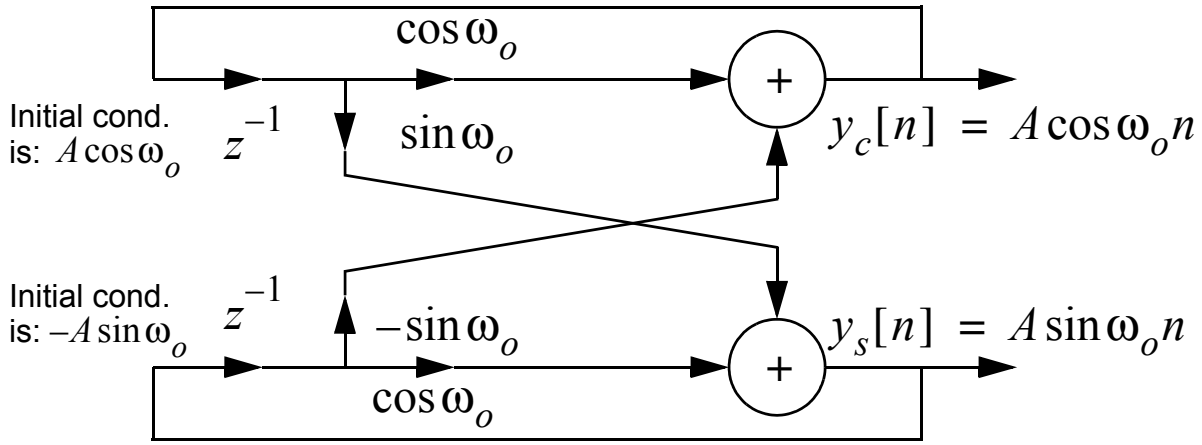
$$H(z) = \frac{b_0}{1 - 2\cos(\omega_o)z^{-1} + z^{-2}} \tag{7.11}$$

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

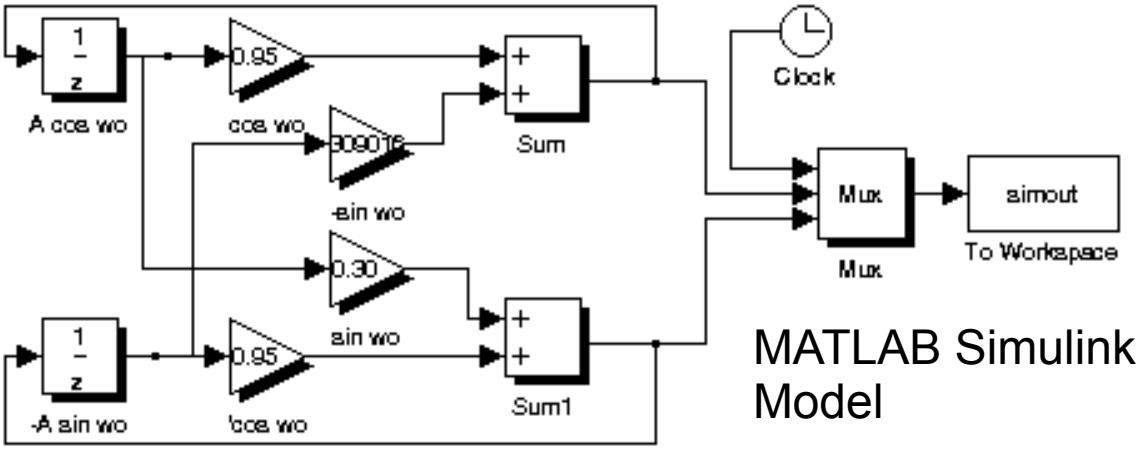
- By letting $b_0 = A\cos\omega_o$ and $x[n] = \delta[n]$ we obtain

$$y[n] = A\sin[(n+1)\omega_o]u[n] \tag{7.12}$$

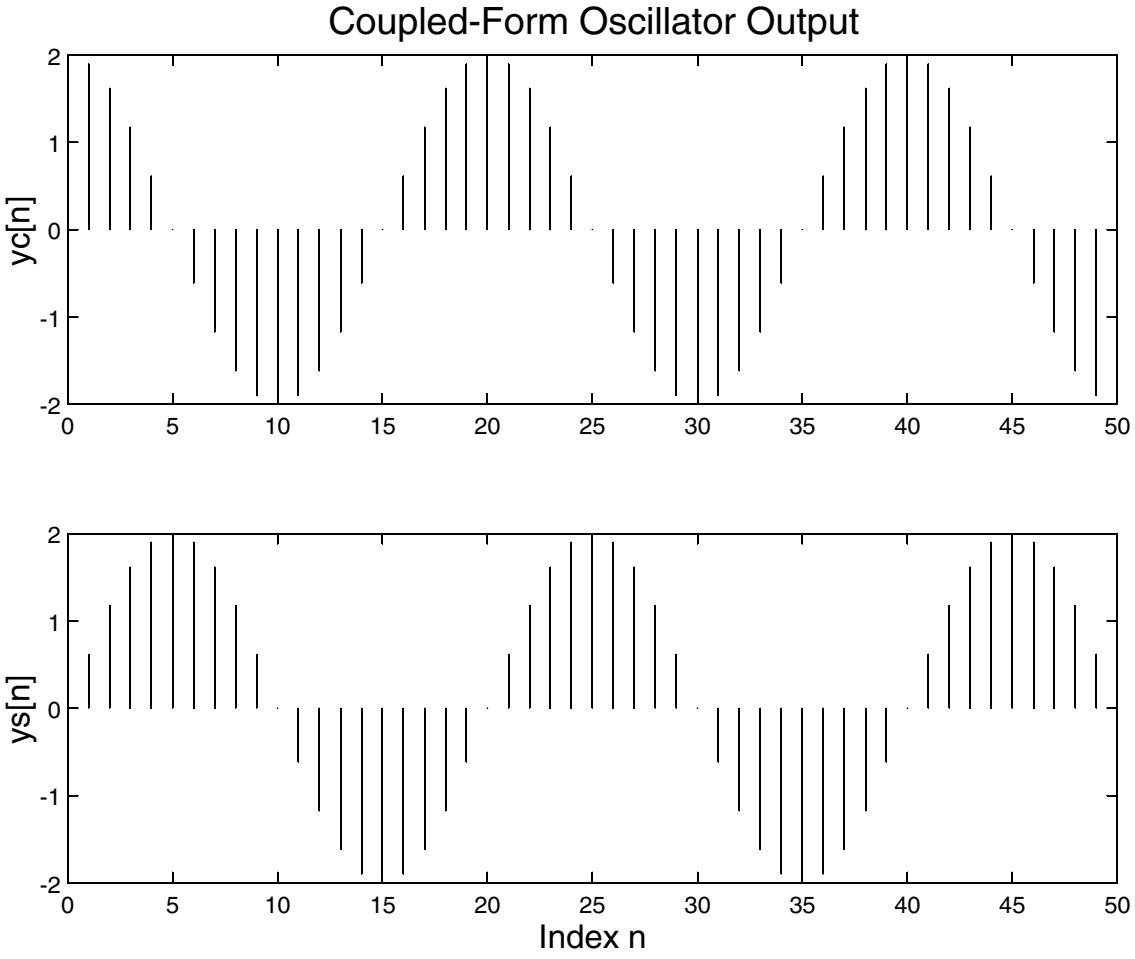
- For applications requiring in-phase (cosine) and quadrature (sine) carriers, the *coupled form oscillator* is useful



- 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

$$H_a(s) = \frac{b_0 + b_1 s + \ldots + b_M s^M}{1 + a_1 s + \ldots + a_N s^N} \qquad (7.13)$$

  where the coefficients in (7.13) are different from those in (7.2)
  - 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

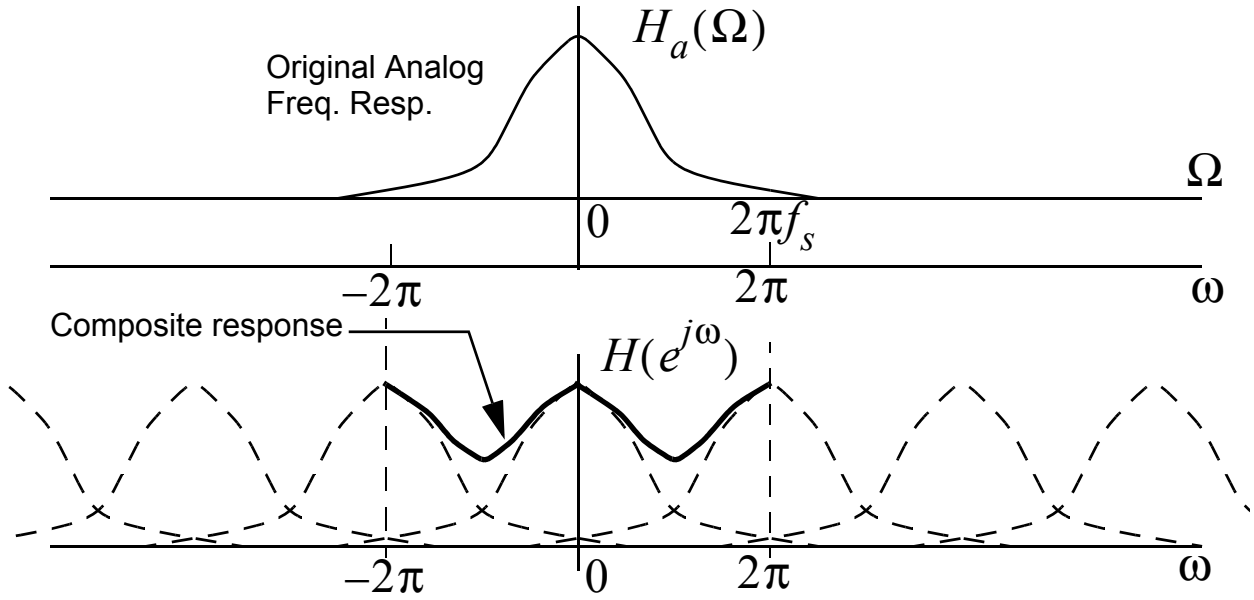
- With impulse invariance we simply chose the discrete-time system impulse response to be a sampled version of the corresponding continuous-time impulse response
- Starting from $h_a(t) \leftrightarrow H_a(s)$, we set

$$h[n] = Th_a(nT) \tag{7.14}$$

  – 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

$$H(e^{j\omega}) = \sum_{k=-\infty}^{\infty} H_a\left(\frac{\omega}{T} + \frac{2\pi k}{T}\right) \tag{7.15}$$



- A serious problem with this technique is the aliasing imposed by (7.15)
- 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

$$H(z) = \mathcal{Z}\left[\{\mathcal{L}^{-1}[H_a(s)]\}\Big|_{t=nT}\right] \tag{7.16}$$

- As a simple example consider

$$H_a(s) = \frac{0.5(s+4)}{(s+1)(s+2)} = \frac{1.5}{s+1} - \frac{1}{s+2} \tag{7.17}$$

- Inverse Laplace transforming using partial fraction expansion yields

$$h_a(t) = [1.5e^{-t} - e^{-2t}]u(t) \tag{7.18}$$

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

$$h[n] = G[1.5e^{-nT} - e^{-2nT}]u[n] \tag{7.19}$$

which implies that

$$H(z) = G\left[\frac{1.5}{1-e^{-T}z^{-1}} - \frac{1}{1-e^{-2T}z^{-1}}\right] \tag{7.20}$$

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

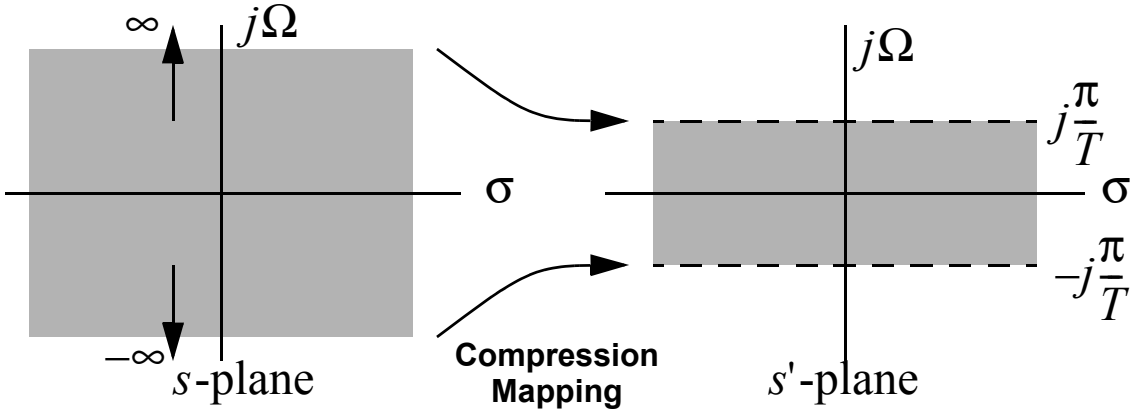
$$G = \left[\frac{1.5}{1-e^{-T}} - \frac{1}{1-e^{-2T}}\right]^{-1} \tag{7.21}$$

- 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



$$s' = \frac{2}{T}\tanh^{-1}\left(\frac{sT}{2}\right) \tag{7.22}$$

- 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

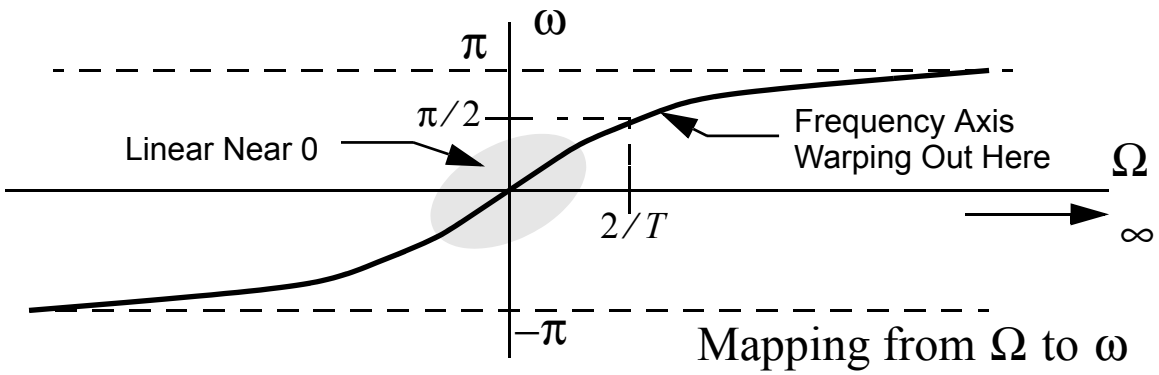
$$s = \frac{2}{T}\left(\frac{1 - z^{-1}}{1 + z^{-1}}\right) \tag{7.23}$$

or in reverse

$$z = \frac{1 + \frac{T}{2}s}{1 - \frac{T}{2}s} \tag{7.24}$$

- The frequency axis mappings become

$$\Omega = \frac{2}{T}\tan\left(\frac{\omega}{2}\right) \text{ or } \omega = 2\tan^{-1}(\Omega T/2) \tag{7.25}$$



- The basic filter design equation is

$$H(z) = H_a(s)\Big|_{s = \frac{2}{T}\left(\frac{1 - z^{-1}}{1 + z^{-1}}\right)} \tag{7.26}$$

- 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*

$$\Omega_i = \frac{2}{T}\tan\left(\frac{\omega_i}{2}\right) \tag{7.27}$$

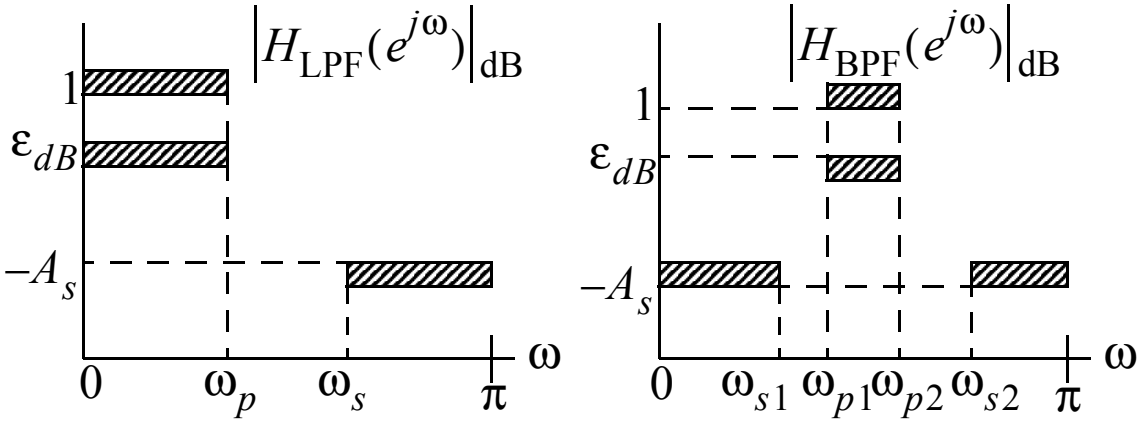
  where ω_i is a discrete-time critical frequency that must be used in the design of an analog prototype with corresponding continuous-time critical frequency Ω_i

  – 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 (7.27)
- 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 $M, N > 2$)
- 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] = che-by1(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] = che-by2(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 bandstop 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

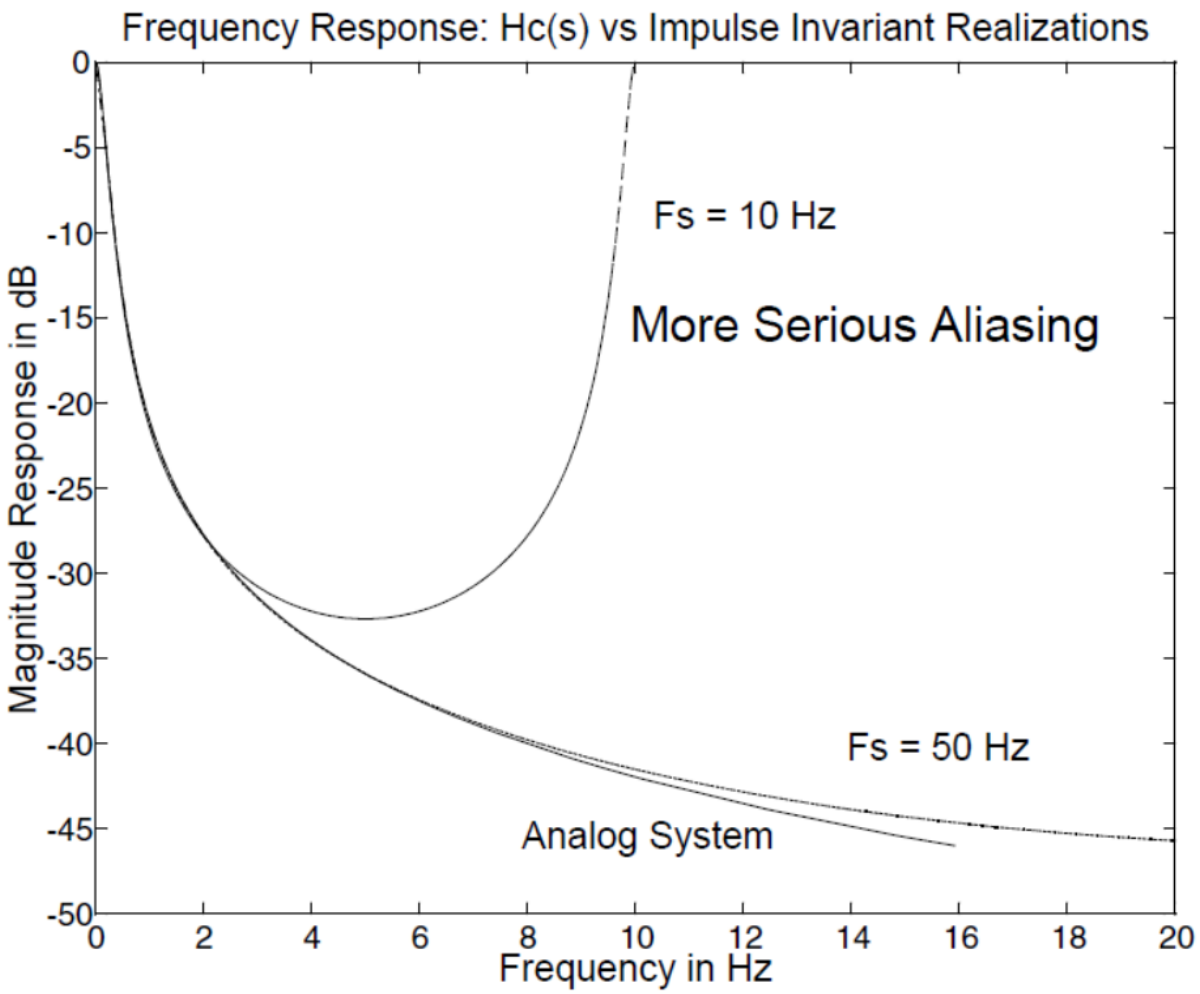
- Suppose we are we are given

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

- 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

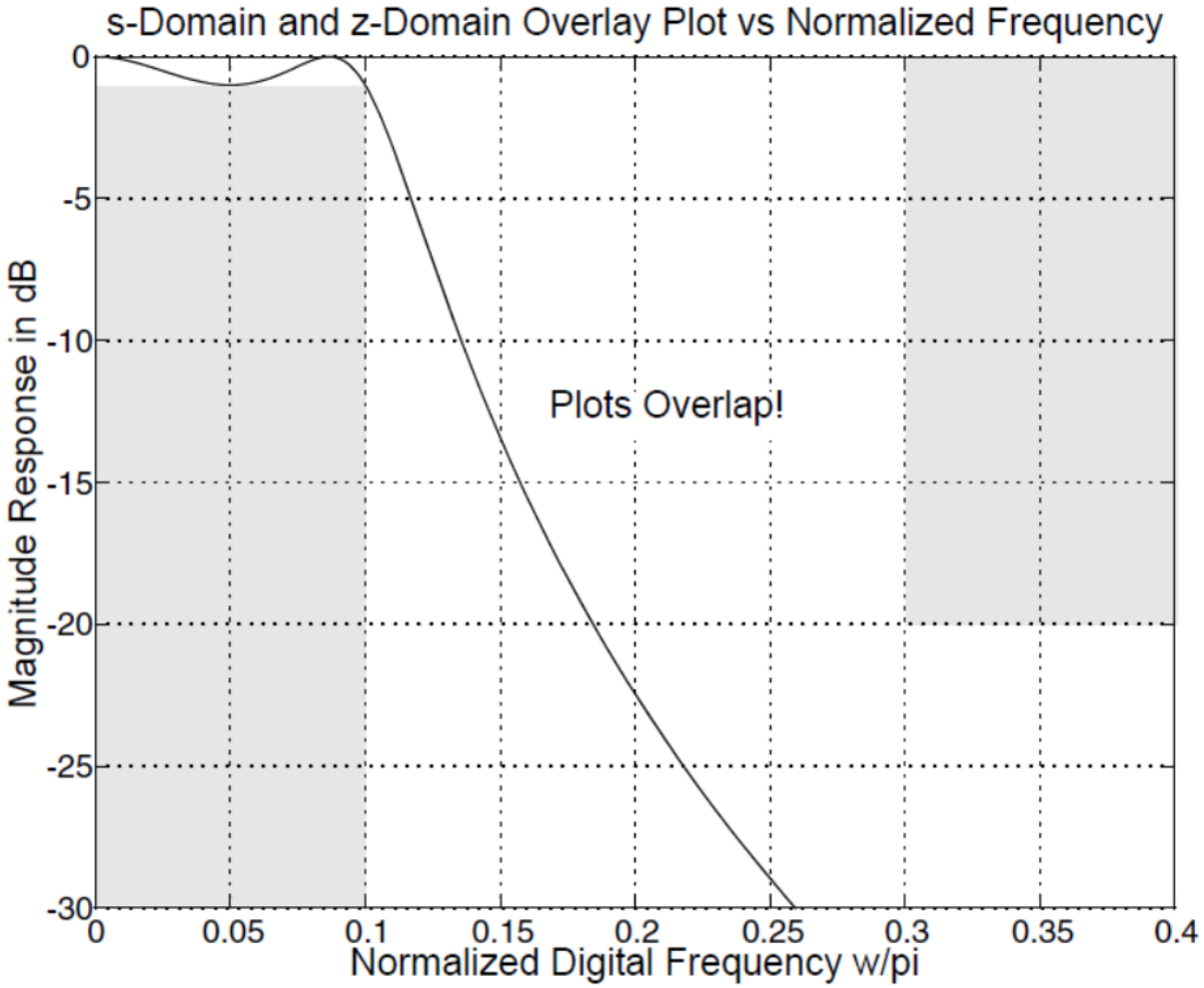
- Consider the following amplitude response requirements

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

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

```
» [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.
```



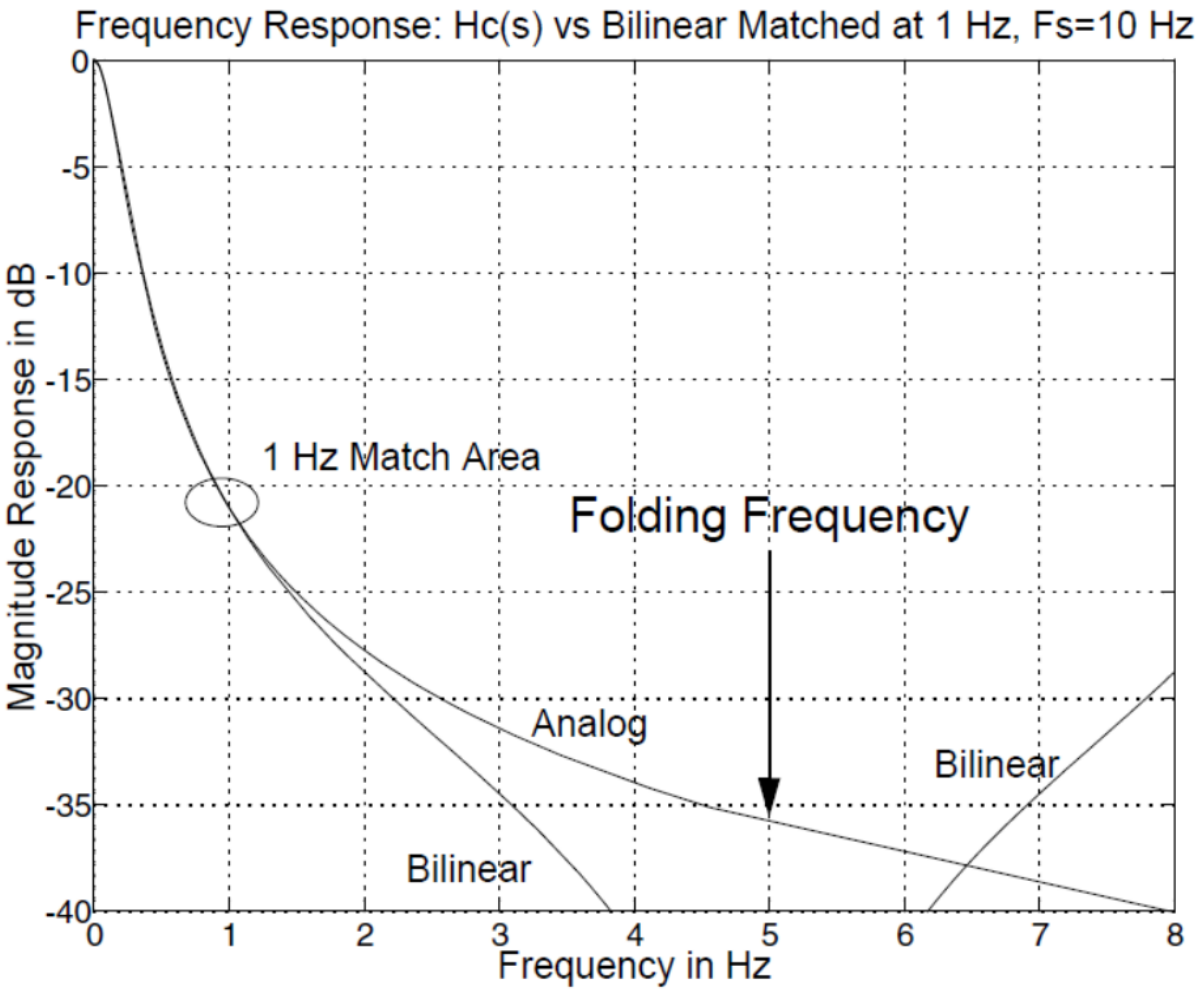
Bilinear Transform Design From Amplitude Specifications (rework of a previous example using the bilinear transform)

- In this problem we are again given

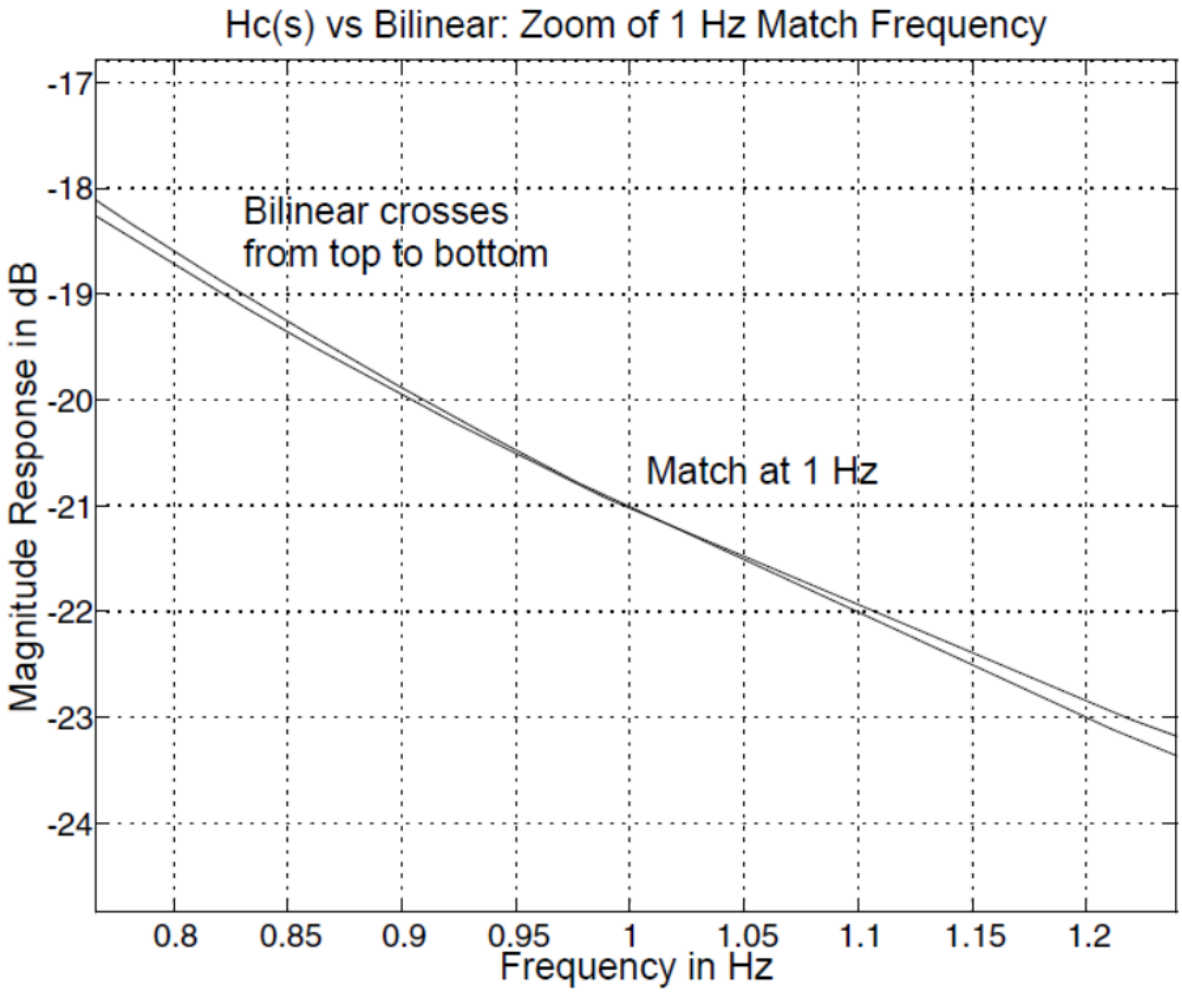
$$H_c(s) = \frac{0.5(s+4)}{(s+1)(s+2)}$$

- 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



Scipy.signal IIR Design Functions

- The Scipy.signal has a good collection of IIR design functions
- At the highest level the function signal.iirdesign is available to create designs from amplitude response requirements

scipy.signal.iirdesign

scipy.signal.iirdesign(*wp, ws, gpass, gstop, analog=False, ftype='ellip', output='ba'*) [source]

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'`.

See also:

butter Filter design using order and critical points
cheby1, cheby2, ellip, bessel

buttord Find order and critical points from passband and stopband spec
cheb1ord, cheb2ord, ellipord

iirfilter General filter design using order and critical frequencies

- There are other functions in `signal` that relate to IIR filter design
  - Order determining functions as MATLAB, e.g., `ord,wn = ellipord`
  - Digital and analog coefficient generating functions that work from the desired order and amplitude response requirements, e.g., `b,a = ellip()`
  - Bilinear transform from s to z at a given sample rate f_s

scipy.signal.bilinear

scipy.signal.bilinear(*b, a, fs=1.0*) [source]

Return a digital filter from an analog one using a bilinear transform.

The bilinear transform substitutes `(z-1) / (z+1)` for `s`.

  - Frequency response calculation

scipy.signal.freqz

scipy.signal.freqz(*b, a=1, worN=None, whole=0, plot=None*) [source]

Compute the frequency response of a digital filter.

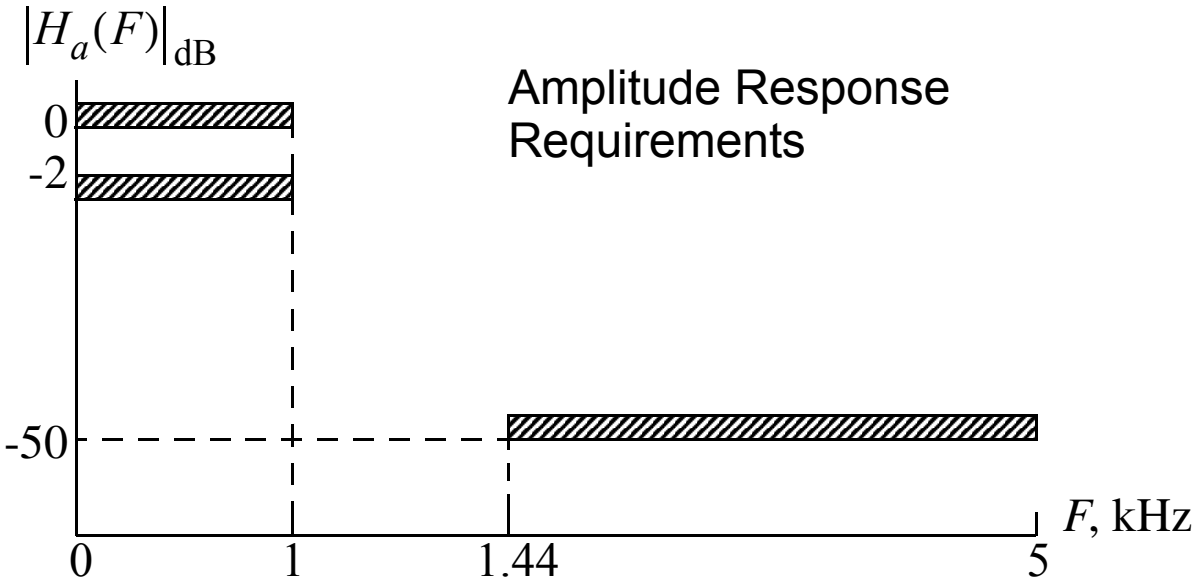
Given the numerator *b* and denominator *a* of a digital filter, compute its frequency response:

$$H(e^{jw}) = \frac{B(e^{jw})}{A(e^{jw})} = \frac{b[0] + b[1]e^{-jw} + \ldots + b[m]e^{-jmw}}{a[0] + a[1]e^{-jw} + \ldots + a[n]e^{-jnw}}$$

- Many others that can be found at `http://docs.scipy.org/doc/scipy/reference/generated/scipy.signal`

Example: 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 `scipy.signal.iirdesign` design procedure is the following:

```
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]
```

```
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]
```

```
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)
```

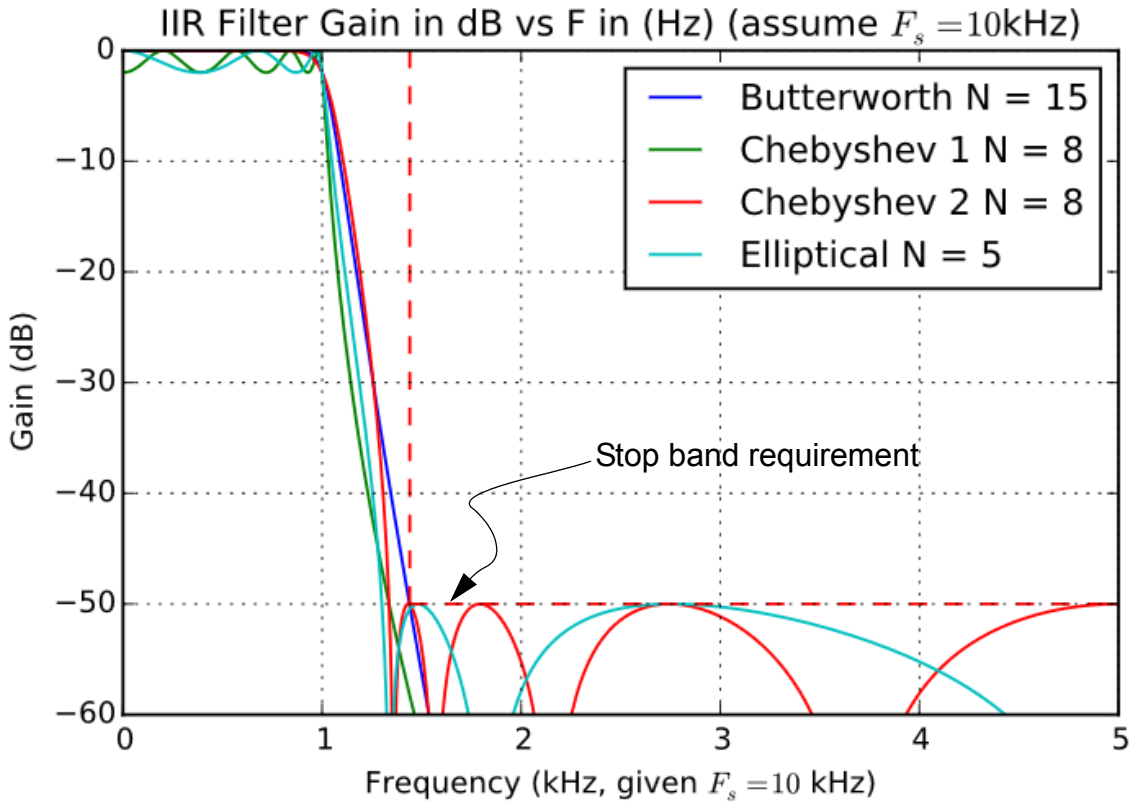
```
Filter order = 8
b coefficients
[ 0.00761324 -0.01135259  0.01996792 -0.01302755  0.01824046 -0.01302755
  0.01996792 -0.01135259  0.00761324]
a coefficients
[  1.          -4.31595778   8.71886201 -10.50954035   8.20030971
  -4.211867     1.38599447  -0.26598822   0.02282966]
```

```
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]
```

- Plot the gain in dB for the four filters

```
freqz_resp_list([bfr1,bfr2,bfr3,bfr4],[afr1,afr2,afr3,afr4],'dB'
                ,fs=10,fsize=(6,4))
ylim(-60,0)
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();
```



- Moving forward with the elliptic $N = 5$ design, we need to convert the direct form coefficients `bfr4` and `afr4` into a cascade of biquad sections
- In MATLAB there is a function available in the signal processing toolbox for doing this
- In Python the function `tf2sos()` does the conversion with one helper function,

```
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
```

- The matrix organization is

```
SOS_mat = [[b00, b01, b02, 1, a01, a02],
           [b10, b11, b12, 1, a11, a12],
                                  ...]
```

- Three sections will be required, with one section really only being first-order

```
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.        ]
```

  – Note the last line is a first-order section since it contains

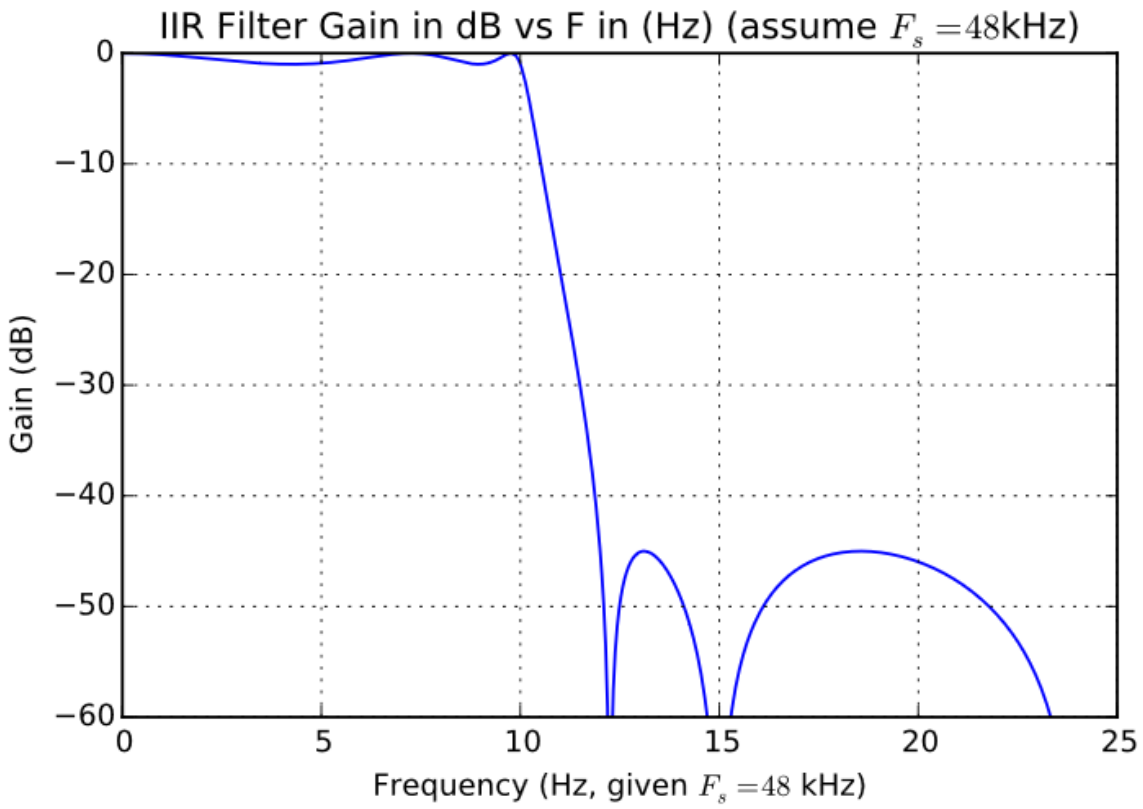
zeros

Elliptical Lowpass and Real-Time Implementation

- 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:
  - Passband ripple of 1 dB over the band $0 \leq f \leq 10$ kHz
  - Stopband attenuation of 45 dB
- Since the order already set at $N = 5$, we can just use the function `signal.ellip()` to design the filter

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

```
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();
```



- The SOS coefficients are:

```
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.        ]
```

- To implement this filter in real-time we need store the coefficients in a header, preferrably one that is compatible with CMSIS-DSP
- The Python function `IIR_sos_header()` takes care of this:

```
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()
```

- For the present design, the header file is as follows:

```
IIR_sos_header('IIR_sos_lpf10.h',b1,a1)
```

```
//define a IIR SOS CMSIS-DSP coefficient array
```

```
#include <stdint.h>

#ifndef STAGES
#define STAGES 3
#endif
/*************************************************************/
/*                      IIR SOS Filter Coefficients          */
float32_t ba_coeff[15] = { //b0,b1,b2,a1,a2,... by stage
     0.282640639291,  0.215128075306,  0.282640639291,
     0.493040104197, -0.895991535298,
     0.282640639291,  0.018608611888,  0.282640639291,
     0.798892481894, -0.589599370586,
     0.531639576491,  0.531639576491,  0.000000000000,
     0.563240296737, -0.000000000000
};
/*************************************************************/
```

- To implement in C we write a custom IIR SOS filter module to be clear on how this is done
  - A data structure is used, similar to the interface of CMSIS-DSP
  - A frames capability is also implemented
  - No real attempt at optimization is made however
- The module `IIR_filter.c/h` (definition and implementation is listed below

```
//IIR_filters.h
// IIR Filters header
// Mark Wickert April 2015

#define ARM_MATH_CM4
#include <stdint.h>
#include "arm_math.h"

/*
Structures to hold IIR filter state information. Two direct form FIR
types are implemented at present: (1) float32_t and in the future (2)
int16_t. Each type requires both an initialization function and a
filtering function. The functions feature both sample-based and
frame-based capability.

More IIR implementation types can/should be added, say for a folded design,
```

```
use of intrinsics inthe fixed point version. The use of a circular buffer
in all versions.
*/

struct IIR_struct_float32
{
  int16_t N_stages;    //number of biquad stages = ceil(N_order/2)
  float32_t *state;    //two per stage w1,w2,... so 2*N_stages total
  float32_t *ba_coeff; //five per stage b0,b1,b2,a1,a2,...
                       //so 5*N_stages total
};


void IIR_sos_init_float32(struct IIR_struct_float32 *IIR,
                          int16_t N_stages,
                          float32_t *ba_coeff,
                          float32_t *state);
void IIR_sos_filt_float32(struct IIR_struct_float32 *IIR,
                          float32_t *x_in,
                          float32_t *x_out,
                          int16_t Nframe);


//IIR_filters.c
// IIR SOS Filter Implementation
// Mark Wickert April 2015

#include "IIR_filters.h"

void IIR_sos_init_float32(struct IIR_struct_float32 *IIR,
                          int16_t N_stages,
                          float32_t *ba_coeff,
                          float32_t *state) {
  // Load the filter coefficients and initialize the filter state.
  int16_t iIIR;
  IIR->N_stages = N_stages;
  IIR->ba_coeff = ba_coeff;
  IIR->state = state;
  for (iIIR = 0; iIIR < 2*IIR->N_stages; iIIR++)
  {
    IIR->state[iIIR] = 0;
  }
}

void IIR_sos_filt_float32(struct IIR_struct_float32 *IIR,
                          float32_t *x_in, float32_t *x_out,
                          int16_t Nframe) {
  int16_t iframe;
  int16_t iIIR;
  int16_t icstride;
```

```
  int16_t isstride;
  float32_t input;
  float32_t result;
  float32_t wn;

  //Process each sample of the frame with this loop
  for (iframe = 0; iframe < Nframe; iframe++)
  {
    input = x_in[iframe];
    //Biquad section filtering stage-by-stage using coefficients
    //and biquad states stored in a 1D array.
    //A float accumulator is used.
    wn = 0; // Clear the accumulator
    for (iIIR = 0; iIIR < IIR->N_stages; iIIR++)
    {
      icstride = 5*iIIR; //filter coefficient stride into linear array
      isstride = 2*iIIR; //filter state stride into linear array
      //biquad 2nd-order LCCDE code from notes eqn 7.8 & 7.9
      //Note: The sign of the a[1] and a[2] coefficients is flipped to match
      //the format used by CMSIS-DSP. Thus there is a sign change in the
      // difference equation below for ba_coeff[icstride+3]
      //& ba_coeff[icstride+4]
      wn = input
         + IIR->ba_coeff[icstride+3] * IIR->state[isstride]
         + IIR->ba_coeff[icstride+4] * IIR->state[isstride+1];
      result = IIR->ba_coeff[icstride]*wn
             + IIR->ba_coeff[icstride+1]*IIR->state[isstride]
             + IIR->ba_coeff[icstride+2]*IIR->state[isstride+1];
      //Update filter state for stage i
      IIR->state[isstride+1] = IIR->state[isstride];
      IIR->state[isstride] = wn;
      input = result; //the output is the next stage input
    }
    x_out[iframe] = result;
  }
}
```

- Test and measurement is performed using a standard sample-based program
- In the program listing below key areas are colored blue
- Also, CMSIS-DSP code is along side the custom functions `IIR_sos_init_float32()` and `IIR_sos_filt_-float32()`

```
// stm32_IIR_intr.c
```

```
#include "defines.h"
#include "tm_stm32f4_delay.h"
#include "tm_stm32f4_gpio.h"
#include "tm_stm32f4_usart.h"
#include "stm32_wm5102_init.h"
#include "IIR_filters.h"

#include "IIR_sos_lpf10.h" //the filter design in this subsection
//#include "IIR_sos_bpf8_10.h"

int32_t rand_int32(void);

#define NUMBER_OF_FIELDS 6 //how many parameters to receive from GUI

//Globals for parameter sliders via serial port on USART6
float32_t P_vals[NUMBER_OF_FIELDS] = {1.0,1.0,0.0,0.0,0.0,0.0};
int16_t P_idx;
int16_t H_found = 0;

// IIR filter related variables
float32_t x, y, IIRstate[2*STAGES];
struct IIR_struct_float32 IIR1; //structure for custom IIR SOS
arm_biquad_cascade_df2T_instance_f32 IIR2; //stucture for CMSIS-DSP IIR DF2

void SPI2_IRQHandler()
{
  int16_t left_out_sample = 0;
  int16_t right_out_sample = 0;
  int16_t left_in_sample = 0;
  int16_t right_in_sample = 0;

      TM_GPIO_SetPinHigh(GPIOC, GPIO_Pin_8);

  if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) == SET)
  {
    //TM_GPIO_SetPinHigh(GPIOC, GPIO_Pin_9); //moved down to IIR call
    left_in_sample = SPI_I2S_ReceiveData(I2Sx);
    if (P_vals[3] < 1) // for GUI control of noise or signal input
    {
      x = (float32_t) left_in_sample;
    }
    else
    {
      x = (float32_t)(((int16_t)rand_int32())>>2);
    }
    TM_GPIO_SetPinHigh(GPIOC, GPIO_Pin_9); //timing around filter call
    IIR_sos_filt_float32(&IIR1, &x, &y, 1);
    //arm_biquad_cascade_df2T_f32(&IIR2,&x,&y,1); CMSIS-DSP equivalent
    TM_GPIO_SetPinLow(GPIOC, GPIO_Pin_9);
```

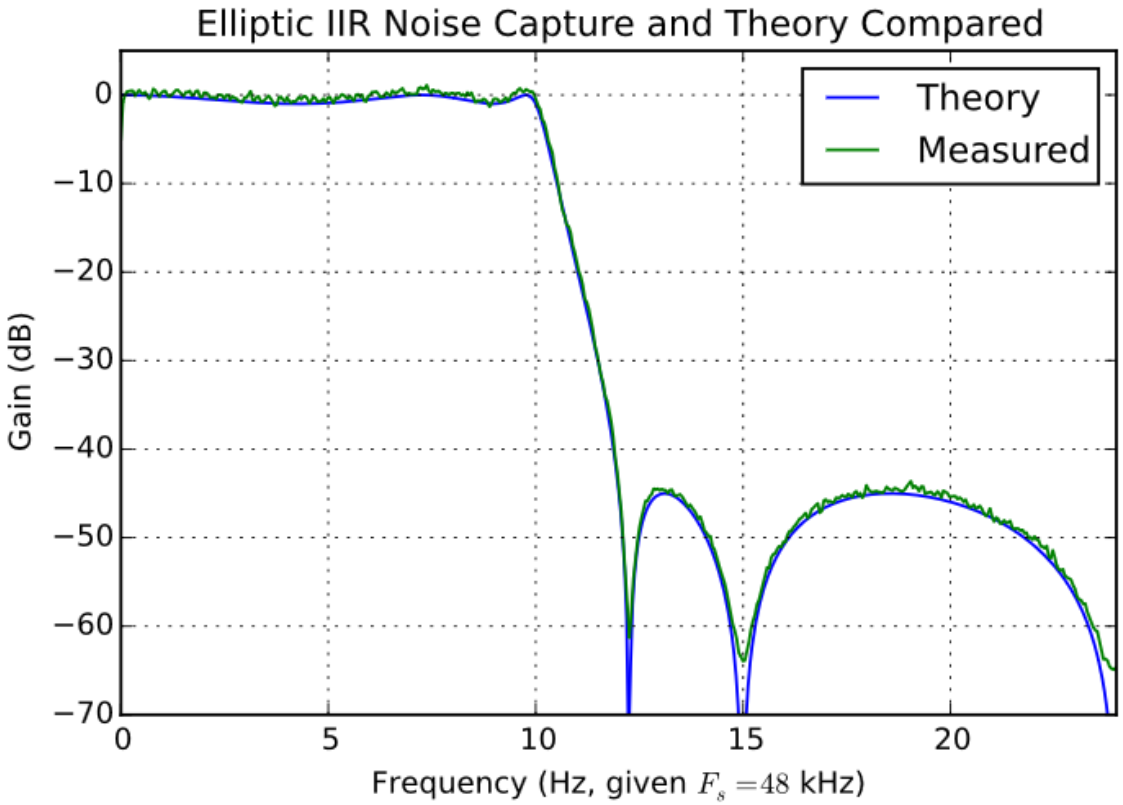
```
    left_out_sample = (int16_t) (P_vals[0]*y);

    while (SPI_I2S_GetFlagStatus(I2Sxext, SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, left_out_sample);
    //TM_GPIO_SetPinLow(GPIOC, GPIO_Pin_9);
  }
  else
  {
    TM_GPIO_SetPinHigh(GPIOC, GPIO_Pin_14);
    right_in_sample = SPI_I2S_ReceiveData(I2Sx);
    right_out_sample = (int16_t)(P_vals[1]*right_in_sample);
    while (SPI_I2S_GetFlagStatus(I2Sxext, SPI_I2S_FLAG_TXE ) != SET){}
    SPI_I2S_SendData(I2Sxext, right_out_sample);
    TM_GPIO_SetPinLow(GPIOC, GPIO_Pin_14);
  }
  TM_GPIO_SetPinLow(GPIOC, GPIO_Pin_8);
}

int main(void)
{
  //Variables for parameter slider communication
  uint8_t c;
  char P_tx[20]; // tx echo chr array
  char P_rcvd[10]; // received chr array
  uint8_t i = 0;
  //Initialize system
  SystemInit();
  /* Initialize delay */
  TM_DELAY_Init();
  //Initialize USART6 at some baud, TX: PC6, RX: PC7
  // Timer error with TM_USART requires a recalc of the baud rate by 231/127
  TM_USART_Init(USART6, TM_USART_PinsPack_1, (231*230400)/127);
  /* Init pins PC8, 9, 14, & 15 for output, no pull-up, and high speed*/
  TM_GPIO_Init(GPIOC, GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_14 | GPIO_Pin_15,
               TM_GPIO_Mode_OUT, TM_GPIO_OType_PP, TM_GPIO_PuPd_NOPULL,
               TM_GPIO_Speed_High);
  IIR_sos_init_float32(&IIR1, STAGES, ba_coeff, IIRstate);
  //arm_biquad_cascade_df2T_init_f32(&IIR2,STAGES,ba_coeff,IIRstate);
  stm32_wm5102_init(FS_48000_HZ, WM5102_LINE_IN, IO_METHOD_INTR);
  while(1)
  {
    //Get character from internal buffer and
    //decode char string into slider float32_t
    //held in P_vals[] array
    c = TM_USART_Getc(USART6);
    if (c) {
      //Wait for header char 'H'
      if (H_found == 0) {
        if (c == 'H') {
          H_found = 1;
```

```
        }
      }
      else {
        //TM_USART_Putc(USART6, c);
        if ((c >= '0' && c <= '9') || (c == '.') || (c == '-')) {
          P_rcvd[i] = c;
          i++;
        }
        else if (c == ':') {
          P_idx = (int16_t) atoi(P_rcvd);
          i = 0;
          memset(P_rcvd, 0, sizeof(P_rcvd)); //clear received char array
        }
        else if (c == 'T') {
          P_vals[P_idx] = (float32_t) atof(P_rcvd);
          //For debug echo parameter value back to GUI
          sprintf(P_tx, "H%d:%1.3fT", P_idx,P_vals[P_idx]);
          TM_USART_Puts(USART6, P_tx);
          memset(P_rcvd, 0, sizeof(P_rcvd));
          i = 0;
          H_found = 0;
        }
        else {
          memset(P_rcvd, 0, sizeof(P_rcvd));
        }
      }
    }
  }
}

int32_t rand_int32(void)
{
      static int32_t a_start = 100001;

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

- We now run the program with the slider in the internal noise source input to the left channel
- Results are captured on the Analog Discovery scope channel using FFT averaging

```
fs_48,P5lpf10_float = loadtxt('spec_noise_iir5lpf10_fs48.csv',delimiter=',
                              skiprows=1,usecols=(0,1),unpack=True)
```

```
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();
```



- Results compare very favorably
- Timing results were taken just around the call to the IIR function:
  - Sample-based 3-stage `float32_t` is 1.48µs

CMSIS-DSP Cascade of Biquads Transposed Direct Form II

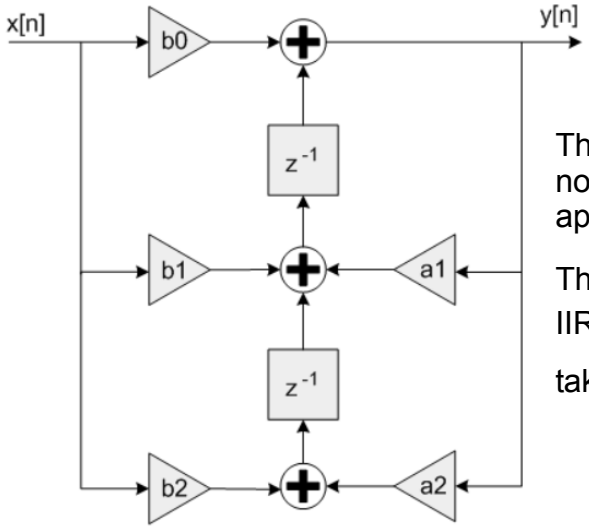
- Continue the elliptic lowpass example, except now using CMSIS-DSP
- As in the special IIR SOS function documented above, CMSIS-DSP also requires one special data type (structure) and two functions

Each Biquad stage implements a second order filter using the difference equation:

```
y[n] = b0 * x[n] + d1
d1 = b1 * x[n] + a1 * y[n] + d2
d2 = b2 * x[n] + a2 * y[n]
```

where d1 and d2 represent the two state values.

A Biquad filter using a transposed Direct Form II structure is shown below.



Single transposed Direct Form II Biquad

The sign error noted here does apply!

The Python function:
IIR_sos_CMSIS_header(fname_out,b,a)

takes this into account

Coefficients `b0, b1, and b2` multiply the input signal `x[n]` and are referred to as the feedforward coefficients. Coefficients `a1` and `a2` multiply the output signal `y[n]` and are referred to as the feedback coefficients. Pay careful attention to the sign of the feedback coefficients. Some design tools flip the sign of the feedback coefficients:

```
y[n] = b0 * x[n] + d1;
d1 = b1 * x[n] - a1 * y[n] + d2;
d2 = b2 * x[n] - a2 * y[n];
```

In this case the feedback coefficients `a1` and `a2` must be negated when used with the CMSIS DSP Library.

- The data structure is described below:

arm_biquad_cascade_df2T_instance_f32 Struct Reference

Instance structure for the floating-point transposed direct form II Biquad cascade filter.

Data Fields

uint8_t	**numStages**
float32_t *	**pState**
float32_t *	**pCoeffs**

Field Documentation

uint8_t arm_biquad_cascade_df2T_instance_f32::numStages

number of 2nd order stages in the filter. Overall order is 2*numStages.

Referenced by **arm_biquad_cascade_df2T_f32()**, and **arm_biquad_cascade_df2T_init_f32()**.

float32_t* arm_biquad_cascade_df2T_instance_f32::pCoeffs

points to the array of coefficients. The array is of length 5*numStages.

Referenced by **arm_biquad_cascade_df2T_f32()**, and **arm_biquad_cascade_df2T_init_f32()**.

float32_t* arm_biquad_cascade_df2T_instance_f32::pState

points to the array of state coefficients. The array is of length 2*numStages.

Referenced by **arm_biquad_cascade_df2T_f32()**, and **arm_biquad_cascade_df2T_init_f32()**.

- The two functions (methods) that use this data structure are the `init` function placed in main and the actual filtering function, placed somewhere in a processing loop/function:

void arm_biquad_cascade_df2T_init_f32 (arm_biquad_cascade_df2T_instance_f32 * S,
uint8_t numStages,
float32_t * pCoeffs,
float32_t * pState
)

Parameters

[in,out]	***S**	points to an instance of the filter data structure.
[in]	**numStages**	number of 2nd order stages in the filter.
[in]	***pCoeffs**	points to the filter coefficients.
[in]	***pState**	points to the state buffer.

Returns

none

Coefficient and State Ordering:

The coefficients are stored in the array `pCoeffs` in the following order:

```
{b10, b11, b12, a11, a12, b20, b21, b22, a21, a22, ...}
```

where `b1x` and `a1x` are the coefficients for the first stage, `b2x` and `a2x` are the coefficients for the second stage, and so on. The `pCoeffs` array contains a total of `5*numStages` values.

The `pState` is a pointer to state array. Each Biquad stage has 2 state variables `d1`, and `d2`. The 2 state variables for stage 1 are first, then the 2 state variables for stage 2, and so on. The state array has a total length of `2*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

LOW_OPTIMIZATION_ENTER void arm_biquad_cascade_df2T_f32 (const arm_biquad_cascade_df2T_instance_f32 * S,
float32_t * pSrc,
float32_t * pDst,
uint32_t blockSize
)

Parameters

[in]	***S**	points to an instance of the filter data structure.
[in]	***pSrc**	points to the block of input data.
[out]	***pDst**	points to the block of output data
[in]	**blockSize**	number of samples to process.

Returns

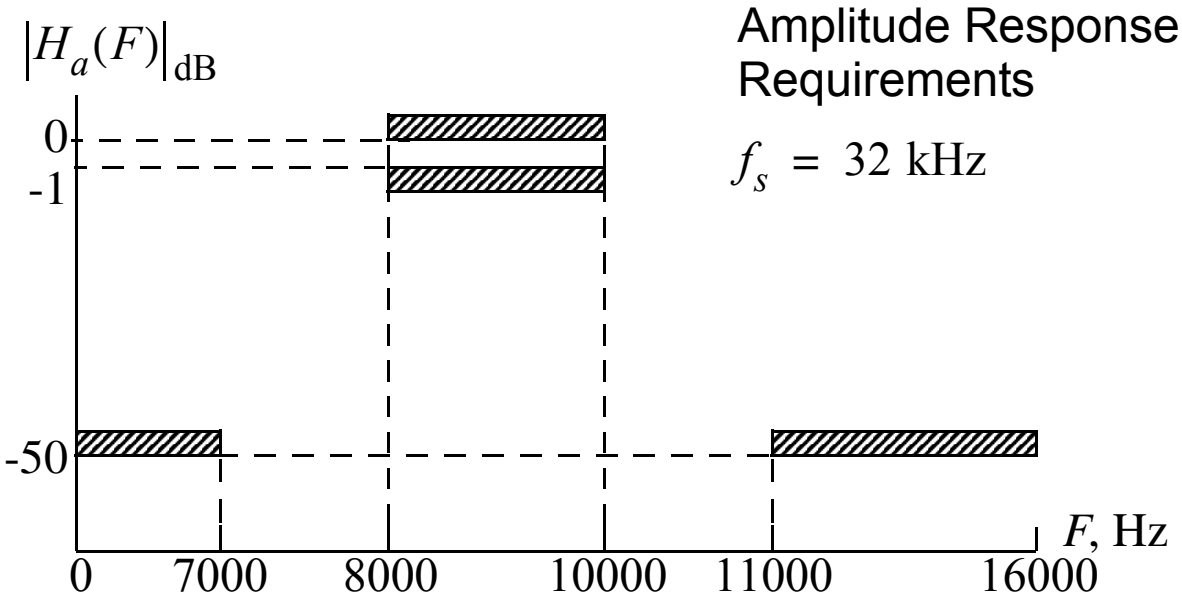
none.

- The code for using `biquad_cascade_df2T_f32()` was shown in the previous C-code main module, except the code was commented out
- Running this code and checking the execution time with the

Analog Discovery, we find

- Sample-based 3-stage `float32_t` is 1.11µs
- The CMSIS-DSP code is faster by about 25%!

Chebyshev Type II Bandpass Filter Design



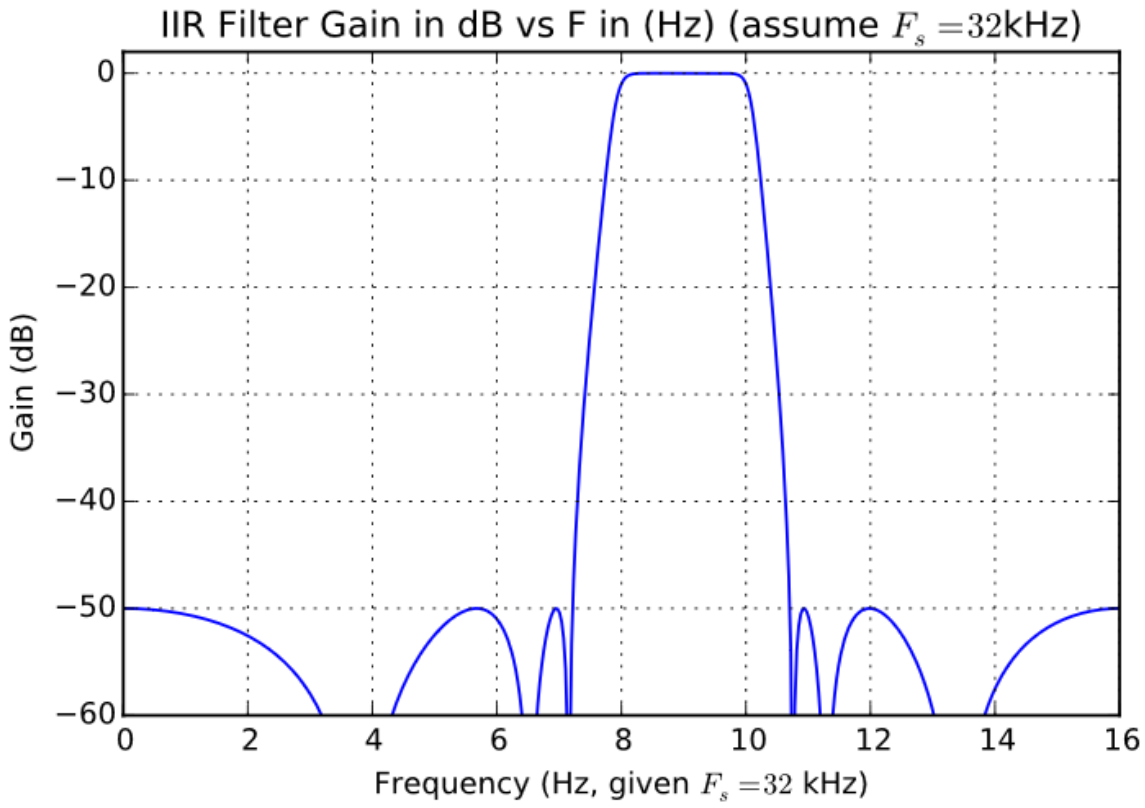
- In this example we will use `signal.iirdesign` to design II sections

```
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]
```

- Take a look at the frequency response before the actual implementation

```
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();
```



- We now export a filter coefficient file:

```
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.        ]
```

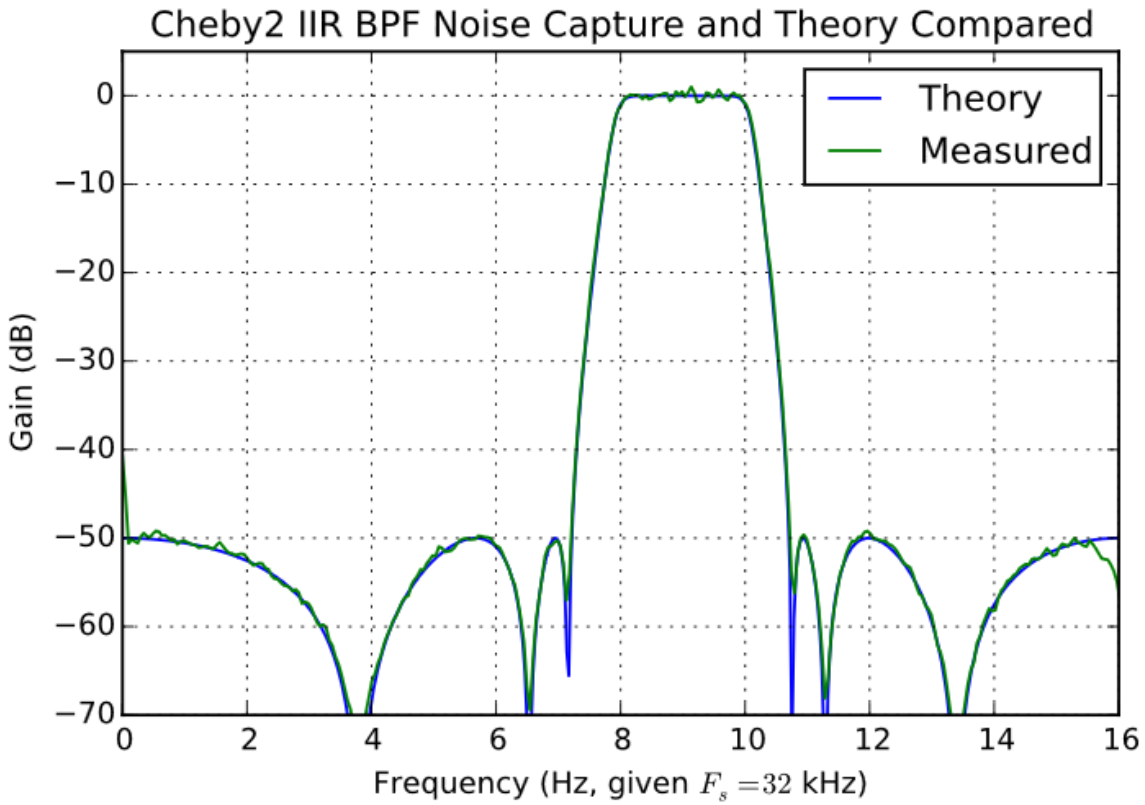
- The actual header file is:

```
//define a IIR SOS CMSIS-DSP coefficient array

#include <stdint.h>

#ifndef STAGES
#define STAGES 6
#endif
/*********************************************************/
/*                     IIR SOS Filter Coefficients       */
float32_t ba_coeff[30] = { //b0,b1,b2,a1,a2,... by stage
     0.417791292456, -0.614128298643,  0.417791292456,
    -0.762606723938, -0.919636130301,
     0.417791292456, -0.237613428130,  0.417791292456,
     0.032894276109, -0.912756121524,
     0.417791292456, -0.137911184868,  0.417791292456,
    -0.661042739522, -0.755715813726,
     0.417791292456,  0.728885035706,  0.417791292456,
    -0.012543850539, -0.738851415396,
     0.417791292456,  0.502674429004,  0.417791292456,
    -0.456139564049, -0.601083617060,
     0.417791292456,  0.430484891836,  0.417791292456,
    -0.174442197979, -0.589376242694
};
/*********************************************************/
```

- In the C-code main module we simple change the header file that is imported to `IIR_sos_bpf8_10.h`
- Results are captured on the Analog Discovery scope channel using FFT averaging

```
fs_32,P12bpf8_10_float = loadtxt('spec_noise_iir12bpf8_10_fs32.csv',delim
                                 skiprows=1,usecols=(0,1),unpack=True)
```

```
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();
```



- The measured results once again are very good
- The execution times for the custom filter and CMSIS are:
  - 6-stage `float32_t` is 2.60µs using `IIR_sos_filt_float32()`
  - 6-stage `float32_t` is 2.04µs using `arm_biquad_cascade_df2T_f32()`

Coupled Form Oscillator in C

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

```
//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;
```

- 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

A Peaking Filter with GUI Control

- 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

$$H_{\text{pk}}(z) = C_{\text{pk}}\left(\frac{1 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}\right) \tag{7.28}$$

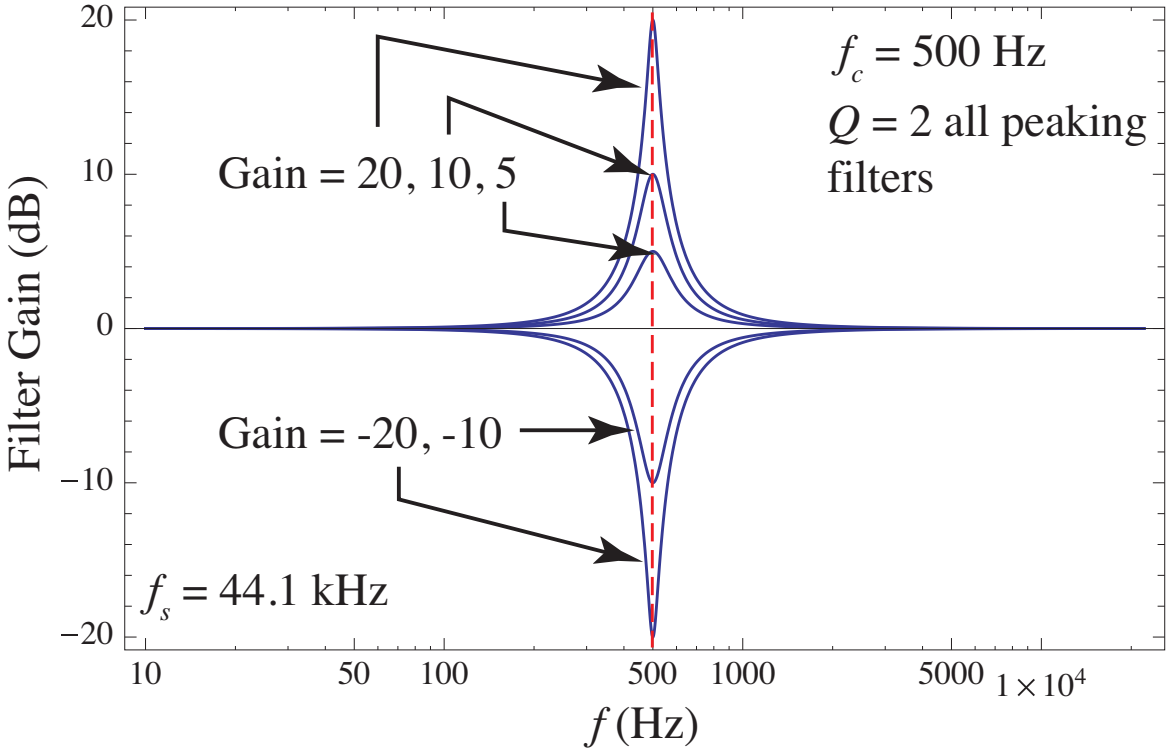
where

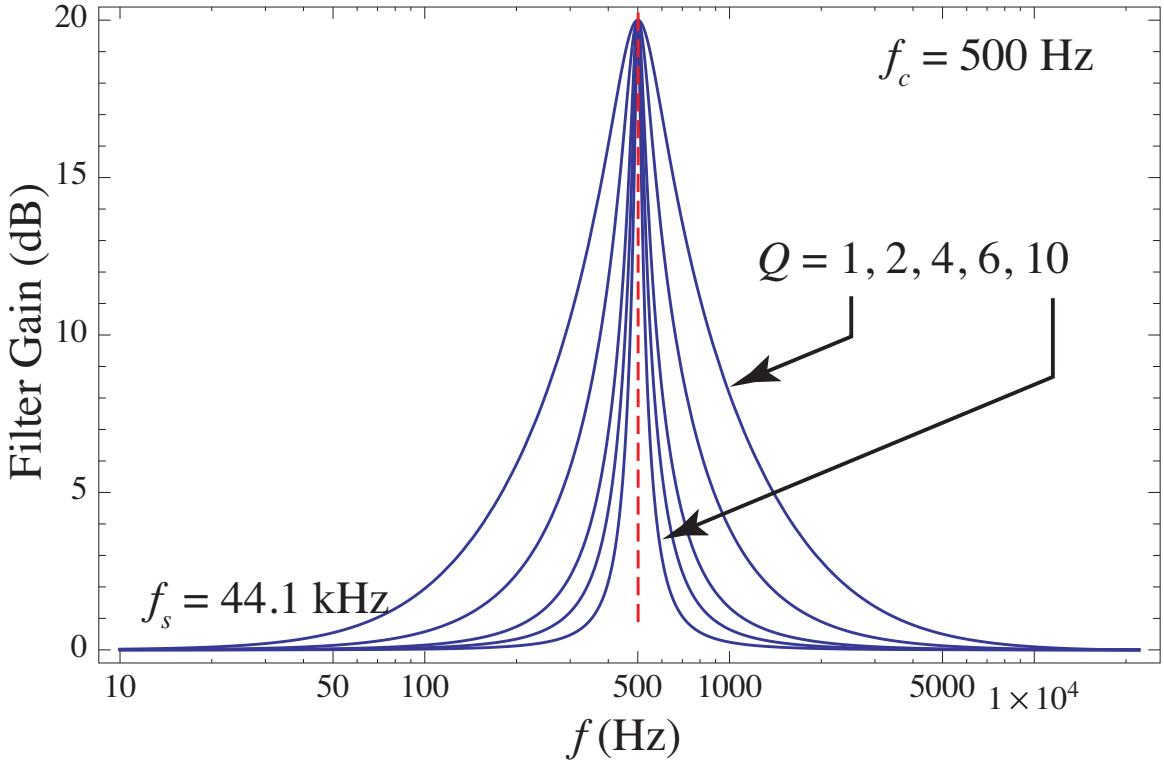
$$\begin{aligned} C_{\text{pk}} &= \frac{1 + k_q \mu}{1 + k_q} \\ b_1 &= \frac{-2\cos(\omega_c)}{1 + k_q \mu}, \; b_2 = \frac{1 - k_q \mu}{1 + k_q \mu} \\ a_1 &= \frac{-2\cos(\omega_c)}{1 + k_q}, \; a_2 = \frac{1 - k_q}{1 + k_q} \\ k_q &= \frac{4}{1 + \mu} \cdot \tan\left(\frac{\omega_c}{2Q}\right) \end{aligned} \tag{7.29}$$

and

$$\mu = 10^{G_{\text{dB}}/20} \tag{7.30}$$

- 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

$$f_{ci} = 30 \times 2^i \text{ Hz}, i = 0, 1, \ldots 9 \qquad (7.31)$$

- 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