A Review of Filter Design

Apple Computer Technical Report #34

By
Malcolm Slaney
Perception Systems–Advanced Technology Group
Apple Computer

This *Mathematica* notebook provides many functions that are useful when designing Infinite Impulse Response (IIR) filters. This notebook is limited to Butterworth and Chebychev polynomial approximations, and transformations of these polynomials into lowpass, highpass, bandpass, and band reject filters. This report also describes how these continuous-domain filters can be implemented as digital filters.

This report is not meant to be a tutorial on filter design but instead to provide polynomial filter design functions in a readily accessible form. For more details on the transformation described here the reader is referred to analog and digital filter design books [Daryanani1976 and Oppenheim1989].

The material in this report started as the Appendix to a previous report on cochlear filters [Slaney1988]. The signal processing and filter design material was extracted and augmented with a number of algorithms for higher-order filter design. These filter design functions are based on a course on filter design taught by Prof. Ray DeCarlo at Purdue University around 1980. Portions of this notebook were used to illustrate two different publications by the author [Slaney1990 and Slaney1992]. A more comprehensive signal processing notebook has been described by Evans [Evans1992]. Finite impulse response (FIR) filter design is also discussed in a series of *Mathematica* notebooks by Julius Smith [Smith1992].

This notebook defines several *Mathematica* functions for signal processing applications. First and second order filters are described since the cochlear model is defined this way. Continuous time filters are described in Section 1 and Section 2 describes the discrete time versions of these filters. In general, the functions provided in the continuous case and in the discrete case are similar. To avoid confusion the word “Continuous” is used in the names of the continuous domain filters.

1. - Continuous Time Filter Design

This section first defines some useful functions when working with continuous (analog) filters. Section 1.2 derives a number of properties of second-order continuous filters, and Section 1.3 describes how to use higher-order polynomials to design lowpass, highpass, bandpass, and band-reject filters.

1.1 - Continuous Filter Functions

Continuous time filters are described by giving the filter's response as a function of complex frequency *s*. In the expressions that follow, filters can be an arbitrary function of the complex frequency s. A typical filter looks like

```
aFilter = a s^3 + b s^2 + c s + d
```

The following functions are used to evaluate the complex response of a filter for a real frequency *f* (in cycles per second), and its corresponding magnitude and phase. A filter's response function is evaluated along the imaginary axis by making the substitution s->I 2 Pi f (or j 2 π f in conventional EE notation.)

```
ContinuousFilterEval[filter_, newS_] :=
        filter /. s->newS;

ContinuousFilterGain[filter_,f_] :=
        ContinuousFilterEval[filter,I 2 Pi f];

ContinuousFilterMag[filter_,f_] :=
        Abs[ContinuousFilterGain[filter,f]]

ContinuousFilterPhase[filter_,f_] :=
        Arg[ContinuousFilterGain[filter,f]]

dB[x_] := 20 Log[10,x]

ContinuousFilterDb[filter_,f_] :=
        dB[ContinuousFilterMag[filter,f]]
```

We define the following function to display the frequency response of a continuous filter. (The plot starts at 0.01Hz to avoid any problems with filters that have a zero at DC.)

```
ContinuousFreqResponse[filter_, maxf_, opts_:{}] :=
    Block[{response},
            response = N[ContinuousFilterDb[filter,f]];
            Plot[response,{f,.01,maxf},
                AxesLabel->{" Hz", "dB"},
                PlotLabel->"Response",
                opts]];
```

We define a similar function for displaying the frequency response of a filter as a function of radian frequency (ω or radians per second, rps).

```
ContinuousFreqResponseRadians[filter_, maxw_, opts_:{}] :=
    Block[{response},
            response = N[ContinuousFilterDb[filter,w/2/Pi]];
            Plot[response,{w,.01,maxw},
                AxesLabel->{" RPS", "dB"},
                PlotLabel->"Response",
                opts]];
```

Note, for each of these functions there is a third optional argument which allows additional options on each plot to be set. We use this feature to plot family of responses by turning off the display until we are ready to show the final accumulated result.

The **ContinuousAdjustGain** function is used to modify a filter so that it has unity gain (and zero phase) at any desired frequency.

```
ContinuousAdjustGain[filter_,f_] :=
        filter/ContinuousFilterGain[filter,f]
```

■ 1.2 - Continuous Second Order Filters

This section describes the behaviour of a second-order polynomial filter. Second-order filters are useful because they represent a simple building block for more complicated filters. This section describes how to use these simple resonators to build bandpass and band-reject filters. The concepts shown here will be useful in the following sections when higher-order filters are described.

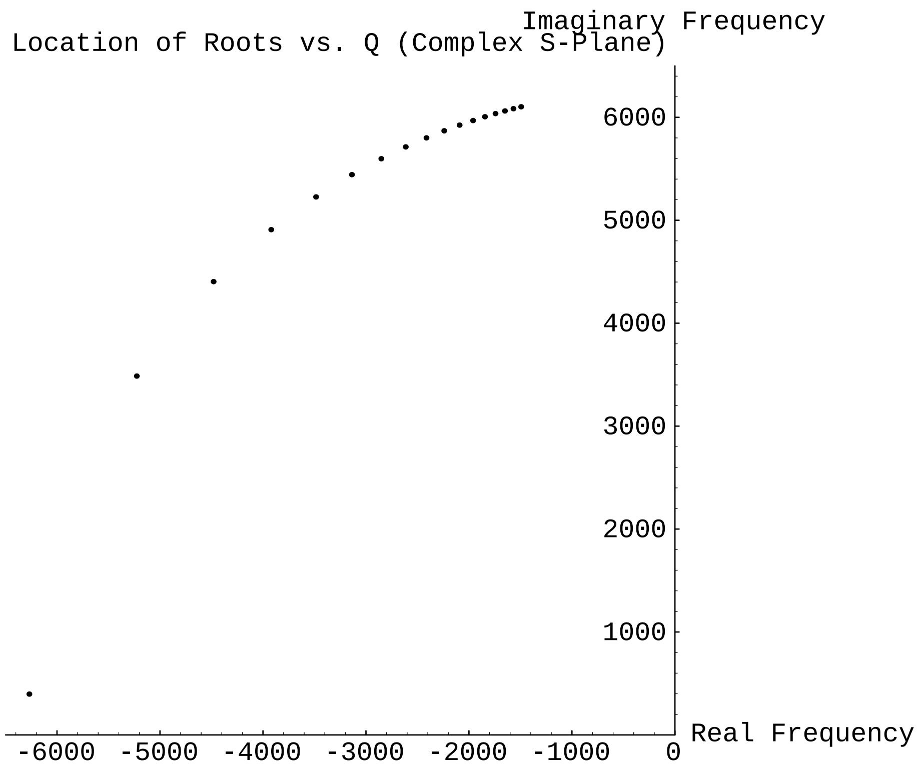
A second order filter is described by its resonant frequency (f) and its quality factor (q). The 3 dB bandwidth of the resulting filter is approximately equal to f/q. The following function computes the roots of a second order polynomial with a given center frequency (f in cycles per second) and bandwidth (q). These roots will be used later in the numerator of a filter function to make a notch in the frequency response or in the denominator to make a peak.

```
ContinuousSecondOrderRoot[f_, q_] :=
  -2 Pi f                          1
  -------- + I 2 Pi f Sqrt[1 - -------]
   q 2                               2
                               (2 q)
```

For any given frequency these roots trace out a circle in the s-plane. For very large q the roots are close to the imaginary axis. Thus as the frequency response is evaluated along the imaginary axis the roots closest to the imaginary axis (high q) will have a sharper response than those that are farther away (low q).

The plot below shows the location of one of the two roots of a second order filter with a constant resonant frequency. Each second order filter has two roots that are complex comjugates of each other. This plot shows only one of the two roots. Those roots that are closest to the imaginary axis have a high q (2.2 in this case) while those on the left near the real axis have a low q (0.501) and thus a broad response in the frequency domain.

The radius of this circle is 2π times the resonant frequency (1000 Hz). For larger resonant frequencies, a cirle with a larger radius will result. In all cases a filter with real coefficients will have a second set of roots that are complex conjugates of the ones shown below. For the filter to be stable all of the roots must be in the left half of the complex plane.



We create a second order filter using a pair of complex conjugate roots as a function of the complex frequency s.

```
ContinuousSecondOrderFilter[f_,q_] :=
    Expand[(s-ContinuousSecondOrderRoot[f,q])
        (s-Conjugate[ContinuousSecondOrderRoot[f,q]])]//N
```

The maximum response of this filter is not at its “center” frequency but at a slightly lower frequency given by.

```
ContinuousNaturalResonance[f_,q_] :=
                                    f Sqrt[1 - 1/(2q^2)]
```

The approximate location of the 3 dB points of the response curve are defined by the following two equations.

```
Lower3DbPoint[f_,q_] :=
        ContinuousNaturalResonance[f,q] - f/(2q)
Upper3DbPoint[f_,q_] :=
        ContinuousNaturalResonance[f,q] + f/(2q)
```

□ 1.2.1 - Continuous Second Order Filter Design

A bandpass filter (or resonator) is defined by placing the roots of a second order polynomial in the denominator. The gain of the filter is normalized so at its peak (the natural resonance frequency) it has unity gain. A unity gain bandpass filter is defined by the following function as a function of center frequency and filter quality (q).

```
MakeContinuousResonance[f_,q_] :=
    ContinuousAdjustGain[1/ContinuousSecondOrderFilter[f,q],
                               ContinuousNaturalResonance[f,q]]
```

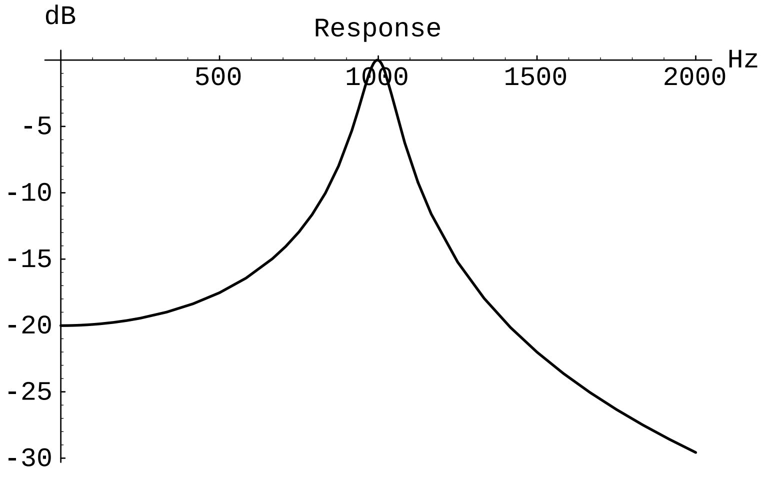
A typical bandpass filter (or resonator) with a center frequency of 1000 Hz and a q of 10 looks like this.

```
MakeContinuousResonance[1000,10]//N
```

```
                      6
  197392. + 3.93796 10  I
---------------------------
          7              2
3.94784 10  + 628.319 s + s
```

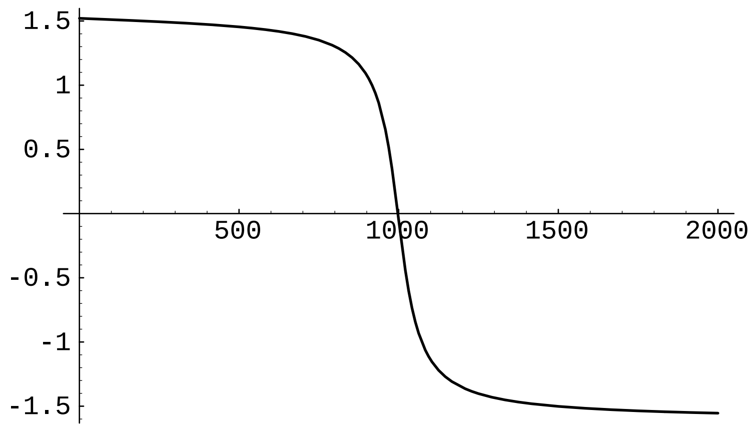
The gain of this bandpass filter is shown below.

```
ContinuousFreqResponse[MakeContinuousResonance[1000,10],
                               2000];
```

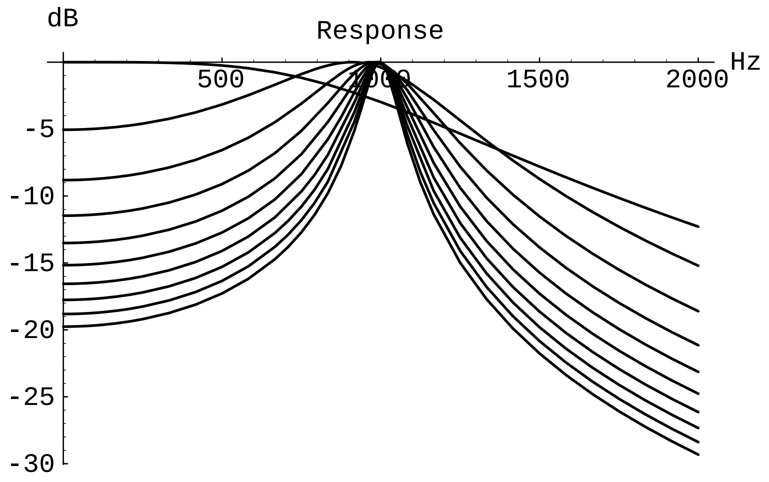


The phase of this bandpass filter shifts as the frequency passes through the resonant frequency. The phase is equal to zero near the center frequency since the gain of the filter has been normalized so that it is equal to one at its resonance frequency.

```
Plot[Release[ContinuousFilterPhase[
            MakeContinuousResonance[1000,10],f]],
        {f,0,2000}];
```



The plot below shows the change in frequency response of second order bandpass filters with a center frequency of 1000 Hz as the quality factor (q) is varied. The flattest curve is the frequency response of the filter with a q of 0.71 while the sharpest (narrowest) filter has a q of 10.



Likewise we can create a second order filter with a notch in the frequency response. In this case all of the roots are in the numerator so there is a zero (or a notch) in the frequency response. We normalize the filter's gain so that at DC there is unity gain.

```
MakeContinuousAntiResonance[f_,q_] :=
    ContinuousAdjustGain[ContinuousSecondOrderFilter[f,q],
                            0]
```

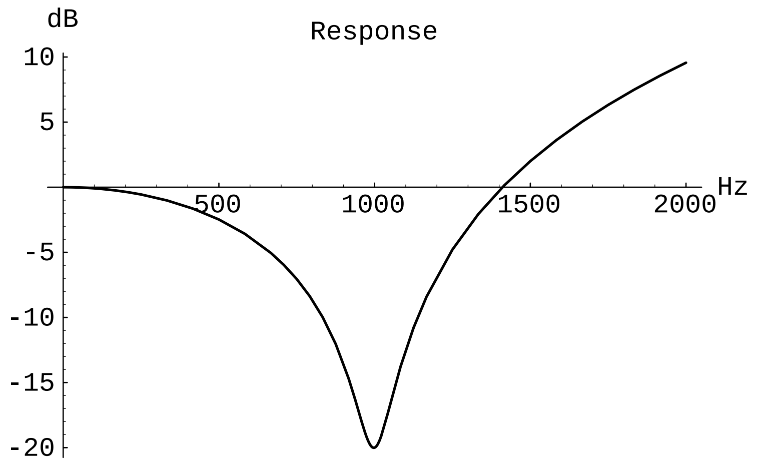
A filter with a notch centered at 1000 Hz and with a q of 10 is described by the following equation.

```
MakeContinuousAntiResonance[1000,10]
```

```
             -8            7                   2
2.53303 10   (3.94784 10  + 628.319 s + s )
```

The response of this filter in the frequency domain is shown below.

```
ContinuousFreqResponse[
                MakeContinuousAntiResonance[1000,10],2000];
```



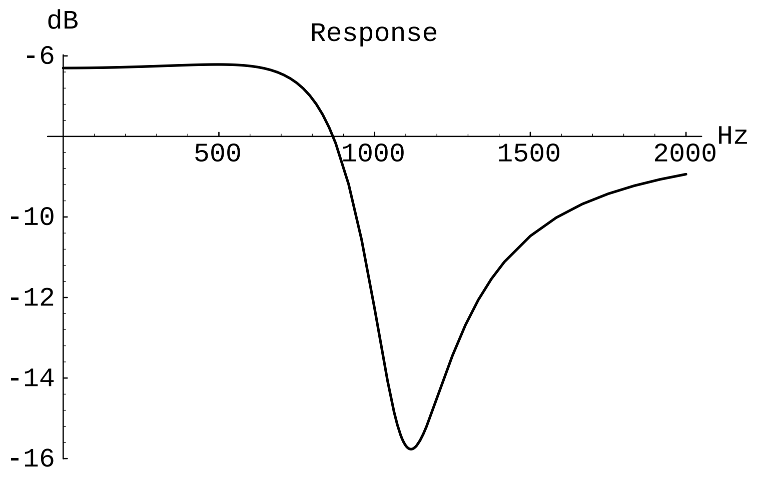
▫ 1.2.2 - Continuous Second Order Examples

We can combine notch and resonance filters to get more interesting responses. The following combination of a resonance with a q of 2 at 1000 Hz and a notch with a q of 5 at 1100 Hz is similar to the cascade-only ear filters described in *Lyon’s Cochlear Model* [Slaney88].

```
MakeContinuousResonance[1000,2] *
          MakeContinuousAntiResonance[1100,5]//N
```

```
                                         7                   2
(0.103306 + 0.386535 I) (4.77689 10  + 1382.3 s + s )
----------------------------------------------------------
                             7                    2
              3.94784 10  + 3141.59 s + s
```

```
ContinuousFreqResponse[MakeContinuousResonance[1000,2]*
        MakeContinuousAntiResonance[1100,5]//N,2000];
```

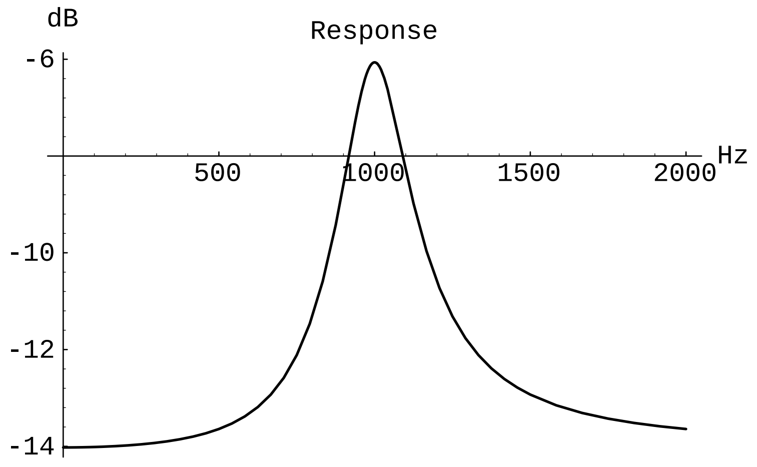


A second filter is shown below. We have combined a broad notch with a narrow resonance to give a bandpass filter.

```
MakeContinuousResonance[1000,5] *
    MakeContinuousAntiResonance[1000,2]//N
```

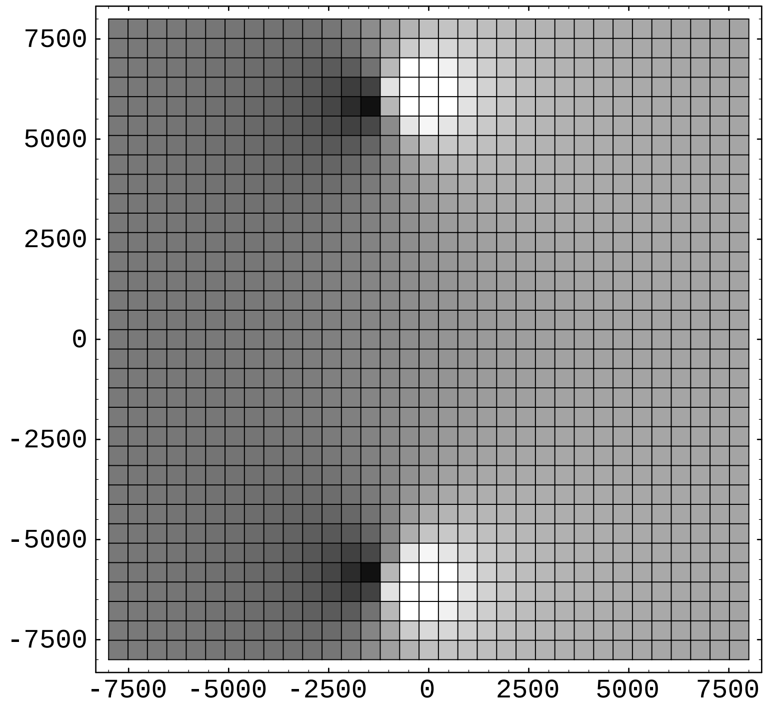
$$\frac{(0.02 + 0.19799\,I)\,(3.94784\,10^{7} + 3141.59\,s + s^{2})}{3.94784\,10^{7} + 1256.64\,s + s^{2}}$$

```
ContinuousFreqResponse[MakeContinuousResonance[1000,5]*
    MakeContinuousAntiResonance[1000,2]//N,2000];
```



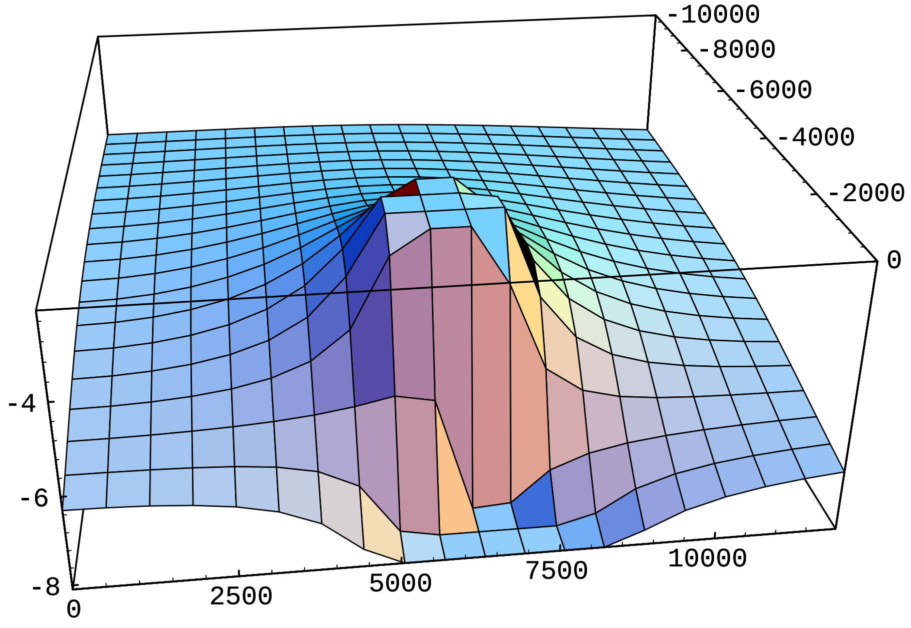
We can gain more insight about this filter by examining the response in the complex s-plane. This is shown below with the white spots representing the location of the resonance (or greatest response) and the black representing the zeros or lowest response.

```
gg[x_] := Release[N[Abs[(MakeContinuousResonance[1000,5] *
                        MakeContinuousAntiResonance[1000,2])
                            /.s->x]]];

DensityPlot[gg[x + I y],{x,-8000,8000},{y,-8000,8000},
            PlotPoints->33,
            AxesLabel->{"Real Frequency","Imaginary Frequency"}];
```



We can also plot this function as a perspective plot. This is shown below. Here we have only shown the upper left hand quadrant of the gray scale picture above. The front axis is the positive jω axis and corresponds to the functions frequency response. The poles and zeros in the upper left half plane are clearly visible. [Thanks to Richard Lyon for suggesting this viewpoint.]

```
gg[x_] := Release[N[dB[Abs[(MakeContinuousResonance[1000,2] *
                  MakeContinuousAntiResonance[1000,5])
                      /.s->x]]]]

Plot3D[gg[x + I y],{x,-10000,0},{y,0,12000},
             PlotPoints->20,
             ViewPoint->{2,-0.25,1},
             Lighting->True];
```



◻ 1.2.3 - Continous Second Order Test Code

The following function prints some interesting information about a second order filter function. (The filter's center frequency and quality factor aren't strictly needed for this analysis since they can be found from the filter polynomial. They are included here to make the code easier to write.)

```
FilterCheck[filter_,f_,q_] :=
         Block[{fd,f1,f2,g1,g2},
                 f1 = N[Lower3DbPoint[f,q]];
                 f2 = N[Upper3DbPoint[f,q]];
                 fd = N[ContinuousNaturalResonance[f,q]];
                 g = N[ContinuousFilterMag[filter,fd]];
                 g1 = N[ContinuousFilterMag[filter,f1]];
                 g2 = N[ContinuousFilterMag[filter,f2]];
                 Print["Gain at f1=",f1," is ",g1];
                 Print[" or ",dB[g1/g],"dB"];
                 Print["Gain at fd=",fd," is ",g];
                 Print["Gain at f2=",f2," is ",g2];
                 Print[" or ",dB[g2/g],"dB"];
                 ]
```

```
FilterCheck[MakeContinuousResonance[1000,10],1000,10]
```

```
Gain at f1=947.497 is 0.71646
 or -2.89616dB
Gain at fd=997.497 is 1.
Gain at f2=1047.5 is 0.698751
 or -3.11355dB
```

Note that the calculation of the 3dB points is only approximate. As the q gets larger the approximation is better.

```
FilterCheck[MakeContinuousResonance[1000,100],1000,100]
```

```
Gain at f1=994.975 is 0.707996
 or -2.99939dB
Gain at fd=999.975 is 1.
Gain at f2=1004.97 is 0.706228
 or -3.0211dB
```

I've written the following function to look for the 3dB points. I couldn't figure out any way to make *Mathematica* solve the equation for me.

```
FilterSearch[f_,v_,l_,h_] := Block[{vl,vm,vh,m},
              m = (l + h) / 2;
              If [Abs[m-l] < m/1000000,
                   Return[N[m]]];
              vl = N[ContinuousFilterMag[f,l]];
              vm = N[ContinuousFilterMag[f,(l+h)/2]];
              vh = N[ContinuousFilterMag[f,h]];
              If [ Between[vl,v,vm],
                   FilterSearch[f,v,l,m],
                   If [ Between [vm, v, vh],
                        FilterSearch[f,v,m,h],
                        Print["Error vl=",vl," vm=",vm," vh=",vh]
                        ]
                   ]
              ]
Between[l_,m_,h_] := (m >= l && m < h) ||
                        (m < l && m >= h)
```

Get a simple resonator with center frequency 1000 and a q of 2.

```
aFilter = MakeContinuousResonance[1000,2]//N
```

$$\frac{4.9348\ 10^{6} + 1.84643\ 10^{7}\ I}{3.94784\ 10^{7} + 3141.59\ s + s^{2}}$$

First look for the upper 3dB point. Note that it is few percent lower in frequency than the equations predict.

```
FilterSearch[aFilter,N[Sqrt[2]/2],1000,1200]
```

```
1165.81
```

```
Upper3DbPoint[1000,2]//N
```

```
1185.41
```

Likewise the lower 3dB point is also off by a bit.

```
FilterSearch[aFilter,N[Sqrt[2]/2],500,1000]
```

```
625.202
```

```
Lower3DbPoint[1000,2]//N
```

```
685.414
```

The following two equations verify that the two 3dB frequencies really do solve the defining equations (See Hayt and Kemmerly, *Engineering Circuit Analysis*.)

```
Simplify[q(Upper3DbPoint[f,q]/f-f/Upper3DbPoint[f,q])]
```

```
                           2
(-(f/(f Sqrt[1 - 1/(2 q )] + f/(2 q))) + 
                          2
    (f Sqrt[1 - 1/(2 q )] + f/(2 q))/f) q
```

```
Simplify[q(Lower3DbPoint[f,q]/f-f/Lower3DbPoint[f,q])]
```

```
                           2
(-(f/(f Sqrt[1 - 1/(2 q )] - f/(2 q))) + 
                          2
    (f Sqrt[1 - 1/(2 q )] - f/(2 q))/f) q
```

Note that if we subtract the two 3dB frequencies we get the bandwidth of the filter (in cycles per second) as a function of the center frequency and q.

```
Simplify[Upper3DbPoint[f,q] - Lower3DbPoint[f,q]]
```

```
f/q
```

◻ 1.2.4 - Usage

The following definitions are needed so that *Mathematica* can respond to user's requests for more information about the functions defined here. This means that the Apple-I (for a function template) and ?Function features will now work with this notebook.

```
FilterEval::usage = "FilterEval[filter, w] returns the
response of a filter at real frequency w.";
```

```
FilterMag::usage = "FilterMag[filter, w] returns the
magnitude of the response of a filter at real frequency w.";
```

```
FilterPhase::usage = "FilterPhase[filter, w] returns the
phase of the response of a filter at real frequency w.";
```

```
dB::usage = "dB[x] converts a voltage (or current) into
decibels.";
```

```
FilterDb::usage = "FilterDb[filter, w] returns the
magnitude of the response of a filter in dB at a
frequency w.";
```

```
AdjustGain::usage = "AdjustGain[filter,w] adjust the
gain of a filter so that it has unity gain at the frequency
w.";

SecondOrderRoot::usage = "SecondOrderRoot[w, q] gives the
location on the s-plane of a pole or zero that has a center
frequency of w and a quality factor of q.  The other root
of the second order section is given by the complex
conjugate of this root.";

Omega1::usage = "Omega1[w, q] gives the frequency of the
lower 3dB point of a second order section with center
frequency of w and a quality factor of q.";

Omega2::usage = "Omega2[w, q] gives the frequency of the
upper 3dB point of a second order section with center
frequency of w and a quality factor of q.";

NaturalResonance::usage = "NaturalResonance[w, q] gives
the frequency of the peak (or valley) of the response of a
second order filter with frequency w and quality factor q.";

ContinuousFreqResponse::usage = "ContinuousFreqResponse[
filter,maxf] shows the frequency response of a continuous
filter from DC to the maximum frequency specified.";

MakeResonance::usage = "MakeResonance[w,q] designs a
second order filter with a peak at frequency w and a
quality factor of q.";

MakeAntiResonance::usage = "MakeAntiResonance[w,q] designs
a second order filter with a notch at frequency w and a
quality factor of q.";
```

■ 1.3 - Continuous Filter Design

Section 1.3.1 will introduce higher-order filters and define a data structure that is used throughout this notebook to describing an arbitrary polynomial filter. The simplest filter, the Butterworth, is described in Section 1.3.2 and a higher performance alternative, the Chebychev, is described in Section 1.3.3. These polynomials are transformed into lowpass filters in Section 1.3.4. Section 1.3.5 shows how to transform these polynomials into highpass, bandpass, and band-reject filters. Finally, Section 1.3.6 shows a number of examples and test cases.

Higher order filters have a sharper response and are designed using different techniques. The second order filters designed above have a gain that varies with the square of the frequency. In other words the gain changes by at most 6 dB per octive. Faster gain changes are possible by combining multiple second order sections and this section of the report describes how to place the poles and zeros to get arbitrarily sharp cutoffs.

1.3.1 - Introduction to Continuous Filter Design

Four types of filters we will be studied in this section. The simplest filter is a low pass filter. We study several types of low pass filters, all normalized so that frequencies below 1 radians/second (rps) are passed relatively unchanged and frequencies above 1 rps are attenuated. We then study four transformations that are used to transform the normalized low pass filter into four other types of filters. They are the unnormalized low pass, high pass, band reject and band pass filters. For these filters each of the poles and zeros of the normalized low pass are transformed into new poles and zeros that lead to the desired response.

These filters could be designed with *Mathematica* using either rational polynomials or lists of poles and zeros. Rational polynomials would certainly be nicer since all intermediate results look like filters. Unfortunately we sometimes need to talk about individual poles and zeros (for example when doing partial fractions expansions) and this is difficult if the filter is described as a polynomial. If a filter is described by its poles, zeros and gain and we can always regenerate the polynomial.

A list of polynomial roots are turned into a polynomial in s using this *Mathematica* expression.

```
PolynomialFromRoots[roots_] :=
    If[Length[roots] == 0,
        1,
        First[Apply[Times,Map[{s-#}&,roots]]]]
```

In this notebook we use a list to keep track of the zeros, poles and gain of a filter. Functions that transform filters will take as input a list of three items (gain, zeros and poles) and return a similar structure. We abbreviate the name of this structure to just GZP (Gain, Zeros and Poles.) The following function is then used to take one of these lists and transform it into a filter in the s-domain. Note that we have used the pattern matching facilities of *Mathematica* to pick out the three elements of the input list.

```
FilterFromGZP[{gain_, zeros_, poles_}] :=
    gain * PolynomialFromRoots[zeros]/
                 PolynomialFromRoots[poles]//N

FilterFromGZP[{zeros_, poles_}] :=
    FilterFromGZP[{1,zeros,poles}]
```

1.3.2 - Butterworth Filters

The simplest type of higher order filter is known as the Butterworth. In a Butterworth filter design the poles are evenly spaced along a circle in the left half of the s-plane. (Remember that poles in the right half of the s-plane lead to unstable filters.) The left hand poles of the following equation define the Butterworth filter.

$$\frac{1}{1 + s^{2 n}}$$

The following expression calculates the poles of an n'th order Butterworth filter. Note that the poles are evenly distributed around the left half of a unit circle in the s-plane.

```
ButterworthPoles[n_] :=
    Table[N[Exp[I Pi / 2 ( 2 k + n - 1)/n]],
            {k,1,n}]
```

```
ButterworthPoles[4]
```

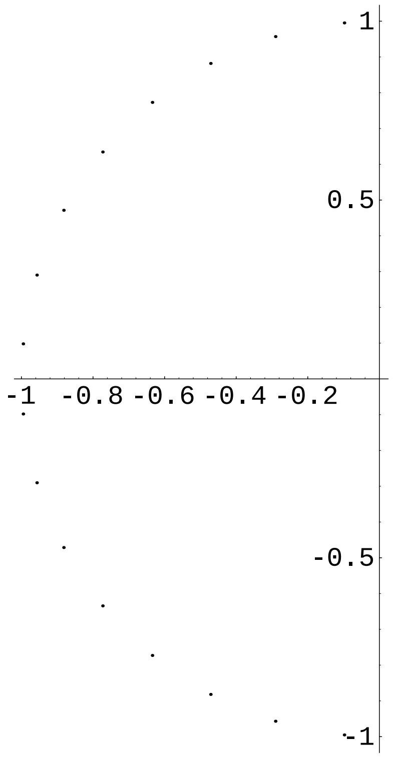
```
{-0.382683 + 0.92388 I, -0.92388 + 0.382683 I,
  -0.92388 - 0.382683 I, -0.382683 - 0.92388 I}
```

We use the following expression to plot the locations of a poles (or zero) list.

```
PlotPoles[complexpolelist_]:=
    ListPlot[Map[{Re[#],Im[#]}&,complexpolelist],
            AspectRatio->Automatic];
```

These last two functions are demonstrated below by showing the locations of the poles in a 16th order Butterworth filter.

```
PlotPoles[ButterworthPoles[16]];
```

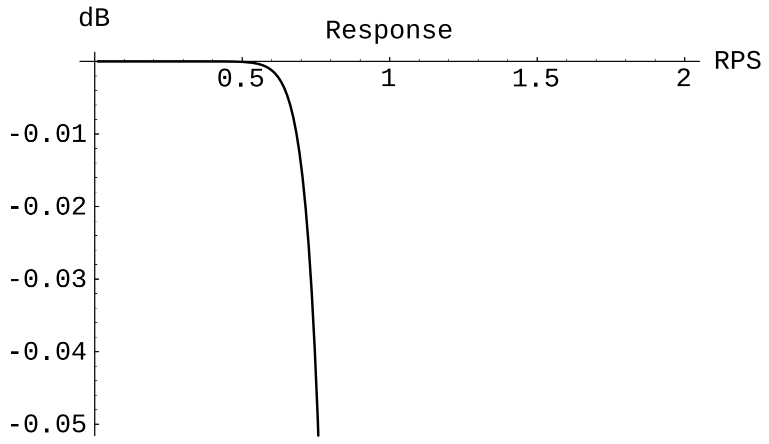


The polynomial resulting from a eighth order Butterworth filter is

```
PolynomialFromRoots[ButterworthPoles[8]]//N
```

```
(0.19509 - 0.980785 I + s) (0.19509 + 0.980785 I + s)
  (0.55557 - 0.83147 I + s) (0.55557 + 0.83147 I + s)
  (0.83147 - 0.55557 I + s) (0.83147 + 0.55557 I + s)
  (0.980785 - 0.19509 I + s) (0.980785 + 0.19509 I + s)
```

```
ContinuousFreqResponseRadians[
    ContinuousAdjustGain[1/
        PolynomialFromRoots[ButterworthPoles[8]],0],2];
```



Note that at the corner frequency (1 rps) this filter has a gain of 3 dB.

```
ContinuousFilterDb[ContinuousAdjustGain[
    PolynomialFromRoots[ButterworthPoles[8]],0],
    1/2/Pi] //N
```

```
3.0103
```

If we want to have a different gain then we need to move the roots slightly to give us the desired gain at the corner frequency. This correction factor is included in the following function which takes as arguments the order of the desired filter and the gain (in dB) at 1 rps.

```
ButterworthLp[n_,amax_] :=
    Block[{poles,gain},
        poles = Sqrt[10^(amax/10)-1]^(-1/n) *
            ButterworthPoles[n]//N;
        gain = ContinuousFilterGain[
                    PolynomialFromRoots[poles],0]//N;
        Return[{gain, {}, poles}]]
```

An example of this shown below for an eighth order Butterworth filter with a gain of 1 dB at the corner frequency.

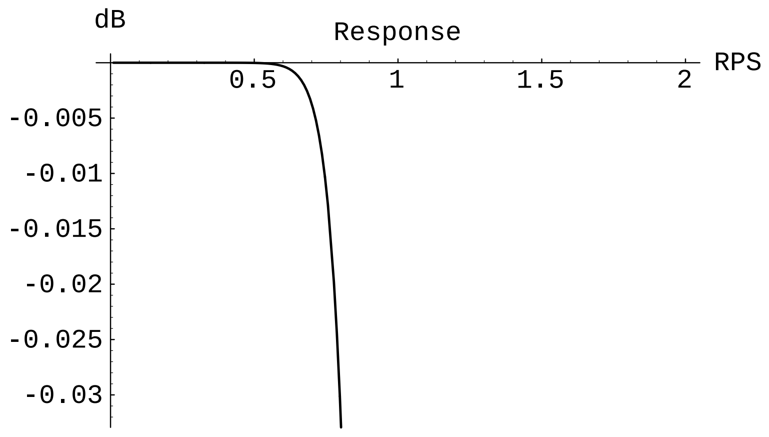
```
ButterworthLp[8,1]
```

```
                          -20
{1.96523 - 2.71051 10    I, {},
  {-0.212282 + 1.06721 I, -0.604527 + 0.904738 I,
   -0.904738 + 0.604527 I, -1.06721 + 0.212282 I,
   -1.06721 - 0.212282 I, -0.904738 - 0.604527 I,
   -0.604527 - 0.904738 I, -0.212282 - 1.06721 I}}
```

```
FilterFromGZP[ButterworthLp[8,1]]
```

```
(1.96523 - 2.71051 10^-20 I)/
  ((0.212282 - 1.06721 I + s) (0.212282 + 1.06721 I + s)
    (0.604527 - 0.904738 I + s)
    (0.604527 + 0.904738 I + s)
    (0.904738 - 0.604527 I + s)
    (0.904738 + 0.604527 I + s)
    (1.06721 - 0.212282 I + s) (1.06721 + 0.212282 I + s)
    )
```

```
ContinuousFreqResponseRadians[
    FilterFromGZP[ButterworthLp[8,1]],2];
```



▫ 1.3.3 - Chebychev Polynomials

The Chebychev polynomials are an alternative to the Butterworth filters described above. The poles of a Butterworth low pass filter are arrayed so that the filter's response is flat through most of its passband and as the frequency approaches the corner frequency the gain quickly falls off. In some cases this characteristic is an advantage because the gain between DC and the corner frequency is nearly flat. In addition if a Butterworth filter is designed that has a given tolerance in the passband then it will easily meet the specification at low frequencies and only at the corner frequency will it approach the maximum specified passband loss.

It turns out that filters with a much smaller variation in gain in the passband can be designed using the Chebychev polynomials. Chebychev filters no longer have a flat response in the passband but like Butterworth filters the passband error can be made arbitrarily small.

The Chebychvev polynomials are described by the following recursive relationship. A technique that *Mathematica* calls dynamic programming is used to remember lower order Chebychev polynomials so that they do not need to be recalculated. (I think caching is a better name.)

```
ChebychevPoly[0] := 1
ChebychevPoly[1] := s
```

```
ChebychevPoly[n_] := ChebychevPoly[n] =
    Simplify[2 s ChebychevPoly[n-1] - ChebychevPoly[n-2]]
```

```
ChebychevPoly[6]
```

```
          2       4       6
-1 + 18 s  - 48 s  + 32 s
```

The poles of these function are given by the following expression. This expression is a function of the desired order (n) and the error (e) in the passband.

```
Chebychev[n_,e_] :=
    Table[Sin[Pi/2(1+2k)/n]Sinh[ArcSinh[1/e]/n] +
            I Cos[Pi/2(1+2k)/n]Cosh[ArcSinh[1/e]/n],
        {k,n,2n-1}]
```

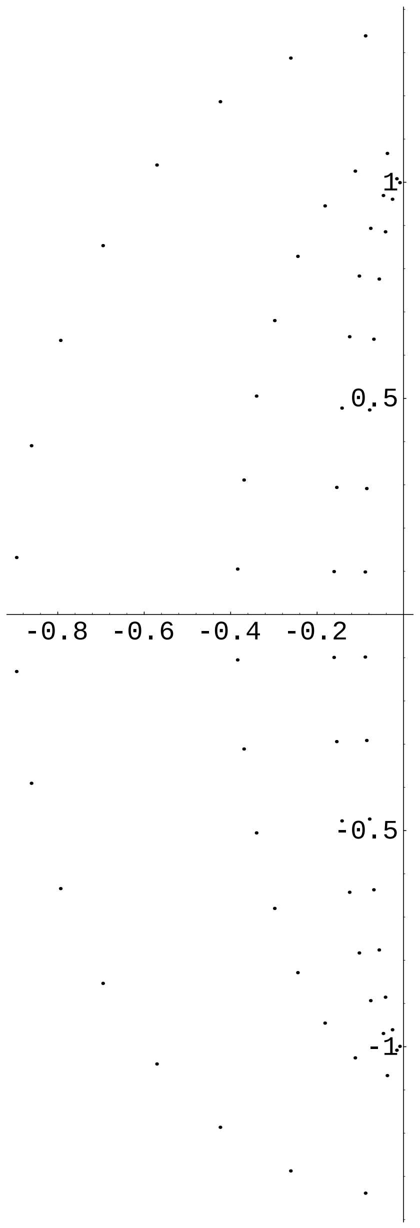
```
ChebychevPoles[n_, amax_] :=
    Chebychev[n,Sqrt[10^(amax/10)-1]]
```

```
ChebychevPoles[6,2]//N
```

```
{-0.0469732 - 0.981705 I, -0.128333 - 0.718658 I,
  -0.175306 - 0.263047 I, -0.175306 + 0.263047 I,
  -0.128333 + 0.718658 I, -0.0469732 + 0.981705 I}
```

As can be seen from the pole plot below, the roots of a Chebychev polynomial fall on an ellipse. This plot shows the roots as the maximum error in the passband is varied from 10^{-10} (the ones that look most like a circle) and a passband error of 1 dB.

```
PlotPoles[Flatten[Map[N[ChebychevPoles[16,#]]&,
                                    {10^-10,10^-4,.1,1}]]];
```

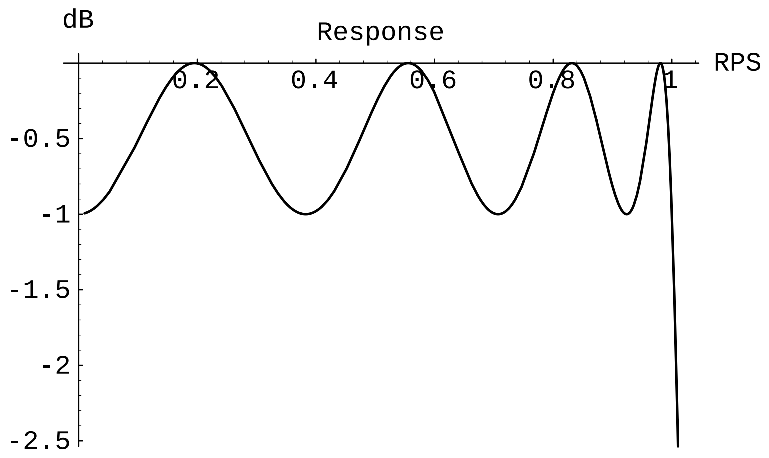


The next function computes a Chebychev low pass filter and returns a list with the gain, zeros (none for a low pass filter) and poles. This list can then be passed to the filter transform routines to realize other types of filters (bandpass, bandreject and highpass.) In this filter design function the gain at the corner frequency (1 radian per second) is adjusted so that it has a loss of amax. As will be seen in the plots to follow this will set the maximum gain of the filter (at the peaks in the passband) to 0 dB.

```
ChebychevLp[n_,amax_] :=
    Block[{poles,gain},
        poles = ChebychevPoles[n,amax]//N;
        gain = ContinuousFilterGain[
                    PolynomialFromRoots[poles],
                    1/(2 Pi)]//N;
        gain = 10^(-amax/20) * gain;
        Return[{gain, {}, poles}]]
```

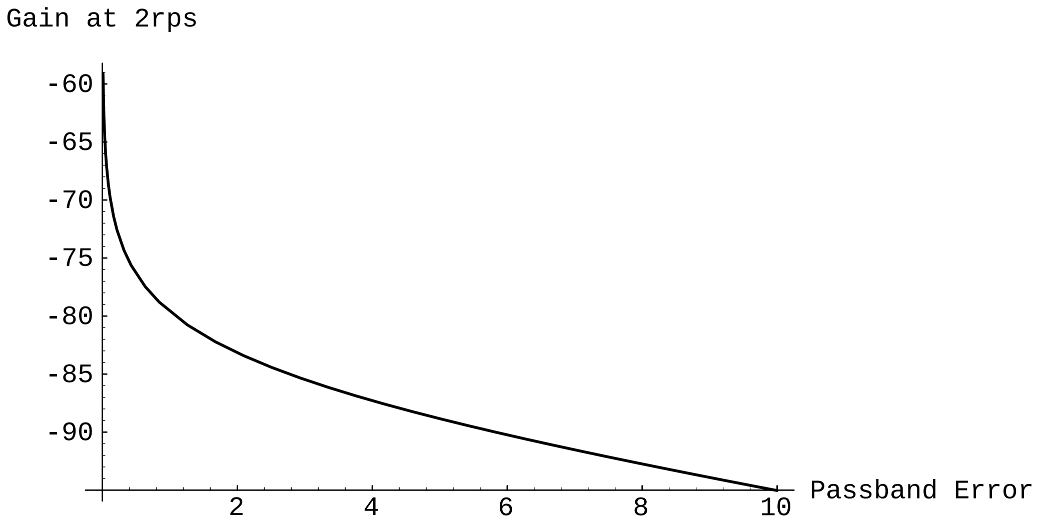
The plot below shows the response of a eighth order Chebychev low pass filter with a passband error of 1 dB.

```
ContinuousFreqResponseRadians[
    FilterFromGZP[ChebychevLp[8,1]],1.02];
```



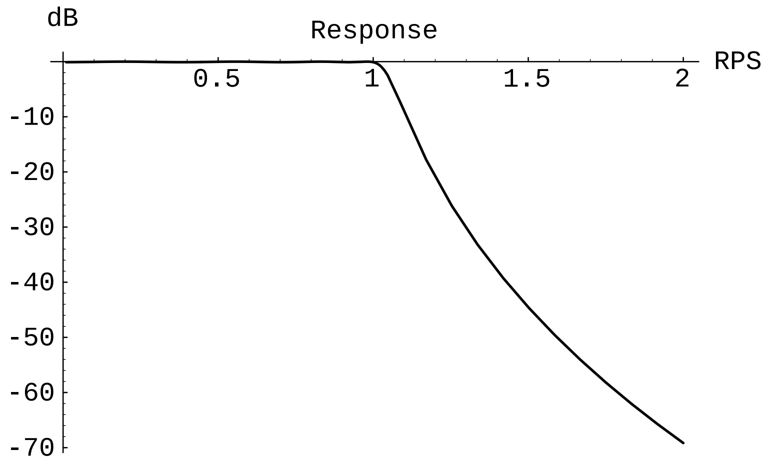
Chebychev filters can have an arbitrarily small error in the passband but this does not come for free. The plot below shows the gain at twice the corner frequency as a function of passband error. In each case an eighth order Chebychev low pass filter was designed.

```
Plot[ContinuousFilterDb[
        FilterFromGZP[ChebychevLp[8,e]],
        1/Pi],
    {e,.01,10},
    AxesLabel->{"Passband Error","Gain at 2rps"}];
```

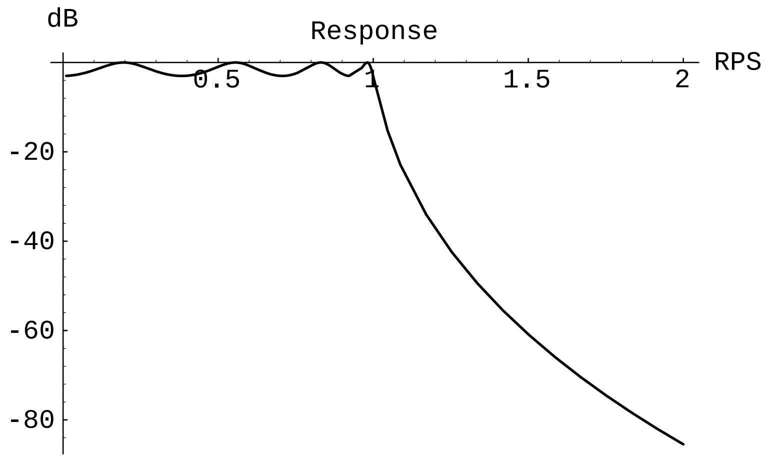


The two plots that follow show the actual frequency response of two of the filters used to make the plot above. The first plot is for a maximum passband error of 0.1 dB while the second plot is a Chebychev filter with an error of 3 dB.

```
ContinuousFreqResponseRadians[
    FilterFromGZP[ChebychevLp[8,.1]],
    2,{PlotRange->All}];
```



```
ContinuousFreqResponseRadians[
    FilterFromGZP[ChebychevLp[8,3]],
    2,{PlotRange->All}];
```



◻ 1.3.4 - Low Pass Filter Design

A low pass filter with a corner frequency of 1 rps can be transformed into a low pass filter with an arbitrary cutoff using a coordinate transformation. Using *Mathematica*, this transformation can be implemented two different ways. First we could simply replace s in the normalized filter with s/(2 Pi fp) where fp is the new center frequency in Hertz. This gives us

```
LpToLp[filter_, fp_] :=
    Simplify[filter/.s->s/(2 Pi fp)]
```

But it is hard to convince *Mathematica* to put this transformed function into the form of simple polynomials in s.

```
LpToLp[FilterFromGZP[ButterworthLp[3,1]],1000]
```

```
                           0.00156895 s   6.26288 10^-7 s^2
1.96523 / (1.96523 + ------------ + ----------------- +
                                Pi                2
                                                Pi

    1.25 10^-10 s^3
    ------------)
           3
         Pi
```

A better way to do this transformation is to use the gain, zero, pole (GZP) structure. The new **LpToLp** transformation function is given by

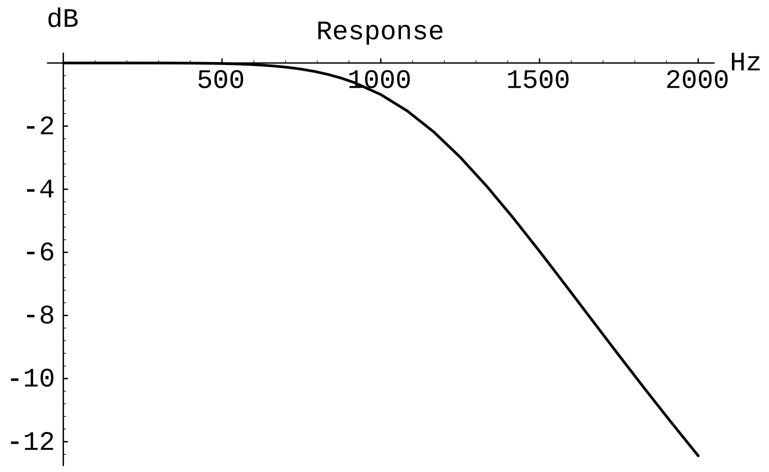
```
LpToLp[{gain_, zeros_, poles_}, fp_] :=
    {gain * (2 Pi fp) ^ (Length[poles]-Length[zeros]),
            zeros * 2 Pi fp,
            poles * 2 Pi fp}
```

```
FilterFromGZP[LpToLp[ButterworthLp[3,1],1000]]
```

```
(4.87475 10^11 + 0. I)/
  ((3935.08 - 6815.77 I + s) (3935.08 + 6815.77 I + s)
    (7870.17 + s))
```

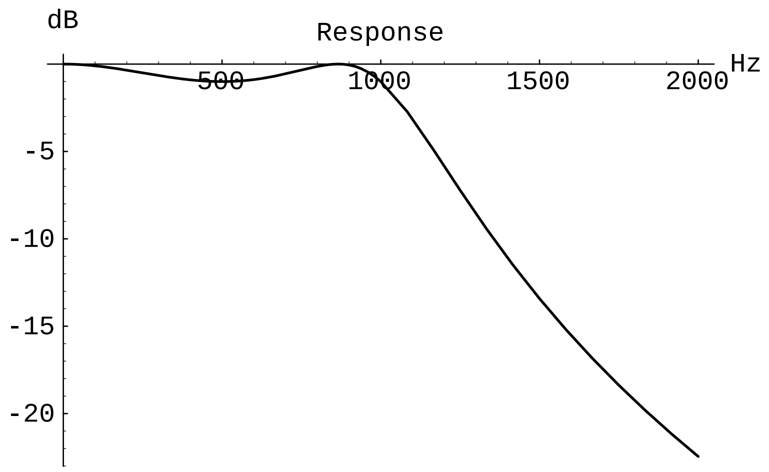
First we show the effect of this transformation on a simple 3rd order Butterworth filter with a passband error of 1 dB.

```
ContinuousFreqResponse[
    FilterFromGZP[LpToLp[ButterworthLp[3,1],1000]],
    2000,{PlotRange->All}];
```



Now we can compare the Butterworth response with a similar filter designed using the Chebychev polynomials. Note that by rearranging the positions of the poles we have designed a filter with a much sharper cutoff. The Butterworth filter has an attenuation of 12dB at 2kHz while the Chebychev filter is already down 20dB. (But in both cases the asymptotic loss for high frequencies will be a function of frequency to the sixth power or 36 dB per octave.)

```
ContinuousFreqResponse[
    FilterFromGZP[LpToLp[ChebychevLp[3,1],1000]],
    2000,{PlotRange->All}];
```



◻ 1.3.5 - Other Filter Transforms

Other filter transforms are also possible. This section describes transforms that make a low pass filter into a high pass, bandpass and band-reject. In each case we use the gain, zero, pole structure to keep track of the filter parameters.

High pass filters are designed by replacing s in the original normalized low pass filter transfer function with

$$s = \frac{wp}{s}$$

where wp is the desired high pass cutoff in radians per second. This transformation is done as follows. Not only is it is necessary to transform each pole into a zero and each zero into a pole, but also to adjust the overall filter gain.

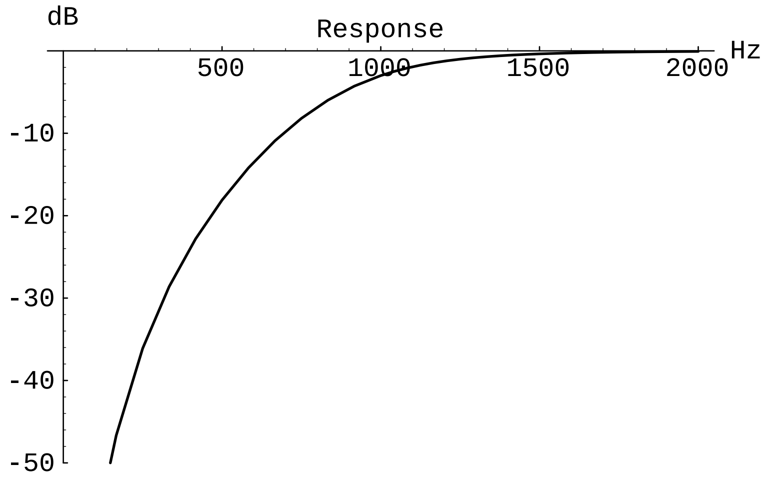
```
LpToHp[{gain_, zeros_, poles_}, fp_] :=
    Block[{RootDiff, ExcessPoles, ExcessZeros},
        RootDiff = Length[zeros]-Length[poles];
        If [ RootDiff > 0,
                    ExcessZeros = Table[0,{RootDiff}];
                    ExcessPoles = {},
                ExcessPoles = Table[0,{-RootDiff}];
                ExcessZeros = {}];
        {gain * Apply[Times,zeros]/Apply[Times,poles],
            Join[2 Pi fp / zeros,ExcessPoles],
            Join[2 Pi fp / poles,ExcessZeros]}]
```

Below we show the Butterworth and Chebychev high-pass filters that have been computed with this transform.

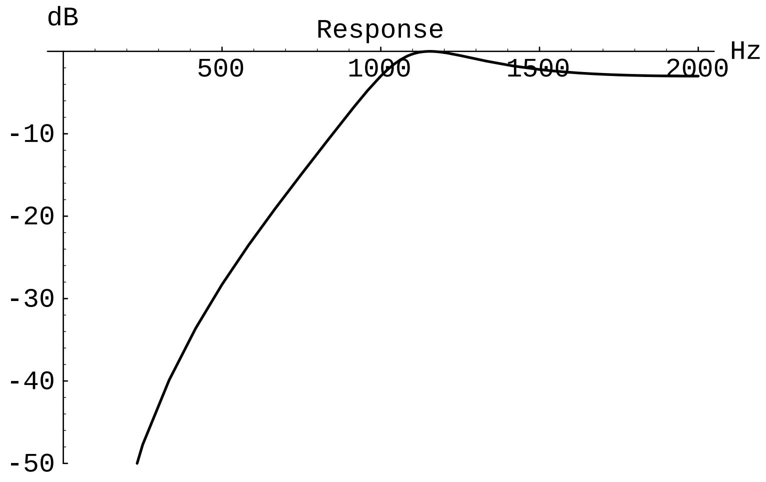
```
FilterFromGZP[LpToHp[ButterworthLp[3,3],1000]]
```

```
                 3
((-1. + 0. I) s )/
  ((3139.11 - 5437.09 I + s) (3139.11 + 5437.09 I + s)
    (6278.21 + s))
```

```
ContinuousFreqResponse[
    FilterFromGZP[LpToHp[ButterworthLp[3,3],1000]],
    2000,{PlotRange->{0,-50}}];
```



```
ContinuousFreqResponse[
    FilterFromGZP[LpToHp[ChebychevLp[3,3],1000]],
    2000,{PlotRange->{0,-50}}];
```



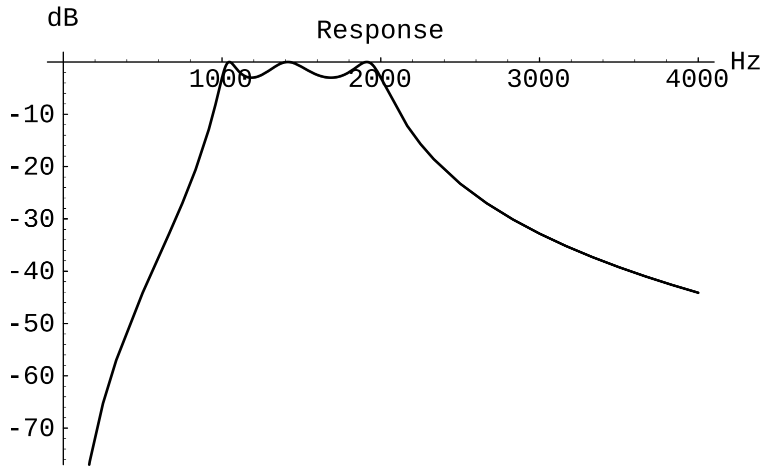
Bandpass filters are more difficult to design. A low pass filter is transformed into a bandpass by specifying the location of the two corner frequencies. We make this transform by subsituting the following expression for s into the normalized low pass filter

$$S = \frac{s^2 + w0^2}{B\ s}$$

In these expressions B is the difference (in radians) between the two edges of the passband and w0 is the geometric mean of the freqencies at the edges of the passband. The function **BpTransform** is used to transform a single root of the normalized filter into two new roots due to the substitution above.

```
BpTransform[roots_,w0_,B_] :=
    N[Flatten[Map[{B # / 2 + Sqrt[B^2#^2-4w0^2]/2,
                      B # / 2 - Sqrt[B^2#^2-4w0^2]/2}&,
                      roots]]]

BpTransform[ButterworthPoles[3],
             2Pi Sqrt[1000 2000],
             2Pi 1000]

{-1104.8 - 6450.39 I, -2036.79 + 11891.8 I,
  -3141.59 + 8311.87 I, -3141.59 - 8311.87 I,
  -1104.8 + 6450.39 I, -2036.79 - 11891.8 I}

LpToBp[{gain_, zeros_, poles_}, fp1_, fp2_] :=
    Block[{w0, B, RootDiff, ExcessPoles, ExcessZeros},
        w0 = 2 Pi Sqrt[fp1 fp2];
        B = 2 Pi (fp2 - fp1);
        RootDiff = Length[zeros] - Length[poles];
        If[RootDiff > 0,
                      ExcessZeros = RootDiff;
                      ExcessPoles = 0,
                  ExcessPoles = -RootDiff;
                  ExcessZeros = 0];
        {gain/B^RootDiff,
         Join[BpTransform[zeros,w0,B],
                  Table[0,{ExcessPoles}]],
         Join[BpTransform[poles,w0,B],
                  Table[0,{ExcessZeros}]]}]

LpToBp[ButterworthLp[3,3],1000,2000]//N

          11
{2.4864 10   + 0. I, {0, 0, 0},
  {-1105.33 - 6448.69 I, -2038.75 + 11894.4 I,
   -3144.08 + 8310.93 I, -3144.08 - 8310.93 I,
   -1105.33 + 6448.69 I, -2038.75 - 11894.4 I}}
```

```
ContinuousFreqResponse[
    FilterFromGZP[LpToBp[ChebychevLp[3,3],1000,2000]],
    4000];
```



We transform a low pass filter into a bandreject filter by specifying the width and the geometric mean frequency of the rejection band. We make this transform by subsituting the following expression for s into the normalized low pass filter

$$S = \frac{B\ s}{s^2 + w0^2}$$

Again in these expressions B is the difference (in radians) between the two edges of the passband and w0 is the geometric mean of the freqencies at the edges of the passband.

Two functions are defined here to make the transformation easier. First the function **BrTransform** is used to take a single root and apply the above transformation to create two new roots. Second the function **BrExtraRoots** is used provide the extra roots for either the numerator or the denominator due to the following term.

$$s^2 + w0^2$$

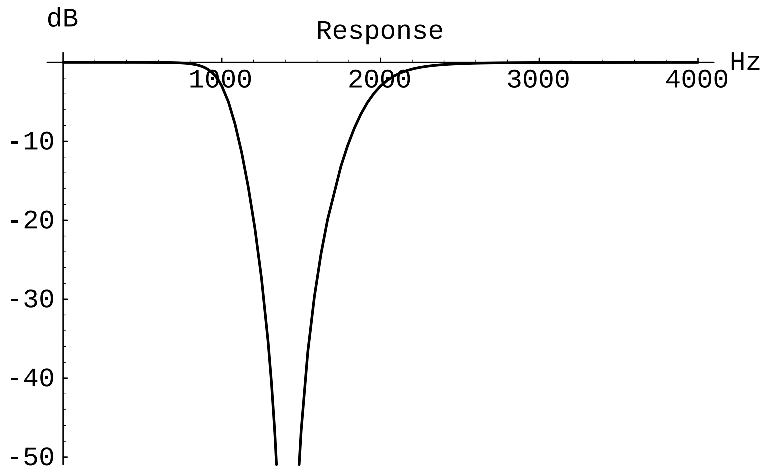
```
BrTransform[roots_,w0_,B_] :=
    N[Flatten[Map[{(B+Sqrt[B^2 - 4 #^2 w0^2])/(2#),
                   (B-Sqrt[B^2 - 4 #^2 w0^2])/(2#)}&,
                   roots]]]

BrExtraRoots[w0_, num_] :=
    Flatten[Table[{w0 I,-w0 I},{num}]]
```

```
LpToBr[{gain_, zeros_, poles_}, fp1_, fp2_] :=
    Block[{w0, B, RootDiff, ExcessPoles, ExcessZeros},
        w0 = 2 Pi Sqrt[fp1 fp2];
        B = 2 Pi (fp2 - fp1);
        RootDiff = Length[zeros] - Length[poles];
        If[RootDiff > 0,
                    ExcessZeros = RootDiff;
                    ExcessPoles = 0,
                ExcessPoles = -RootDiff;
                ExcessZeros = 0];
        {gain*Apply[Times,Map[-#&,zeros]]/
                Apply[Times,Map[-#&,poles]],
         Join[BrTransform[zeros,w0,B],
                BrExtraRoots[w0,ExcessPoles]],
         Join[BrTransform[poles,w0,B],
                BrExtraRoots[w0,ExcessZeros]]}]
```

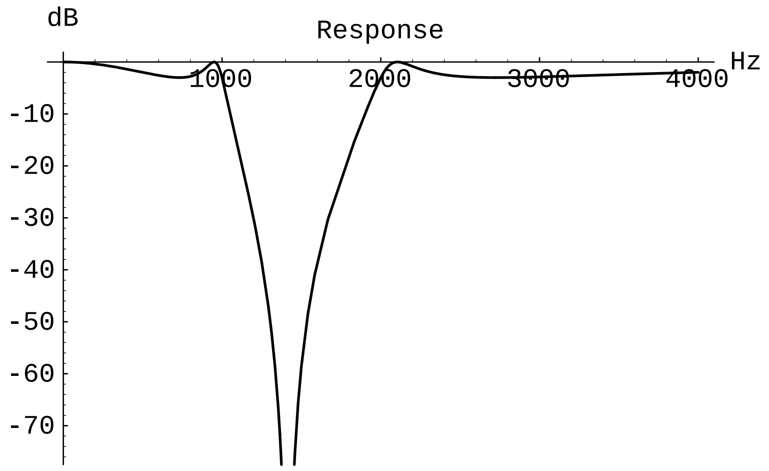
```
LpToBr[ButterworthLp[3,3],1000,2000]//N
```

```
{1. + 0. I, {8885.77 I, -8885.77 I, 8885.77 I,
   -8885.77 I, 8885.77 I, -8885.77 I},
  {-2034.83 - 11889.2 I, -1104.27 + 6452.08 I,
   -3139.11 - 8312.81 I, -3139.11 + 8312.81 I,
   -2034.83 + 11889.2 I, -1104.27 - 6452.08 I}}
```

```
ContinuousFreqResponse[
    FilterFromGZP[LpToBr[ButterworthLp[3,3],1000,2000]],4000];
```



```
ContinuousFreqResponse[
    FilterFromGZP[LpToBr[ChebychevLp[3,3],1000,2000]]
    ,4000];
```



▫ 1.3.6 - DeCarlo’s Test Cases

Most of the above sections came from the EE445 course at Purdue University taught by Prof. Ray DeCarlo. It seems only right to check the results of this notebook using his test cases.

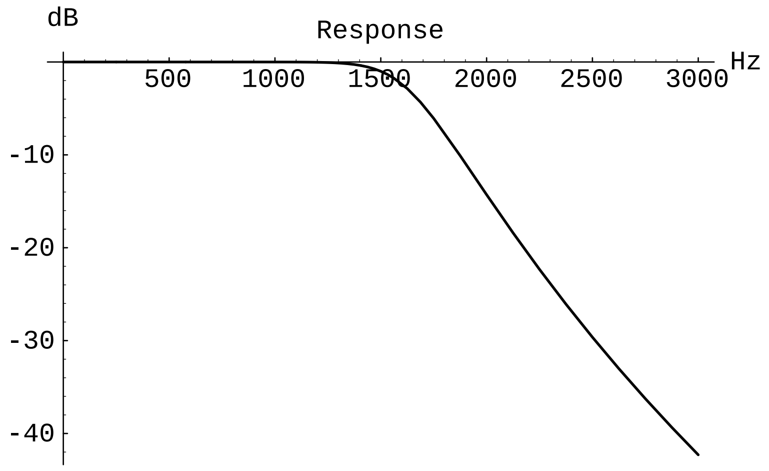
Design an 8th order Butterworth filter with Amax is 1 dB, whose DC gain is 1 (absolute) and whose cutoff frequency is 1500 Hz.

```
ButterworthLp[8,1]

                    -20
{1.96523 - 2.71051 10    I, {},
  {-0.212282 + 1.06721 I, -0.604527 + 0.904738 I,
   -0.904738 + 0.604527 I, -1.06721 + 0.212282 I,
   -1.06721 - 0.212282 I, -0.904738 - 0.604527 I,
   -0.604527 - 0.904738 I, -0.212282 - 1.06721 I}}
```

Now check the gain constant

```
dB[Abs[First[ButterworthLp[8,1]]]]//N

5.86825

LpToLp[ButterworthLp[8,1],1500]//N

          32            12
{1.22344 10   - 1.6874 10   I, {},
  {-2000.71 + 10058.2 I, -5697.53 + 8526.96 I,
   -8526.96 + 5697.53 I, -10058.2 + 2000.71 I,
   -10058.2 - 2000.71 I, -8526.96 - 5697.53 I,
   -5697.53 - 8526.96 I, -2000.71 - 10058.2 I}}

dB[Abs[First[LpToLp[ButterworthLp[8,1],1500]]]]//N

641.752
```

```
ContinuousFreqResponse[
    FilterFromGZP[LpToLp[ButterworthLp[8,1],1500]],
    3000,{PlotRange->All}];
```



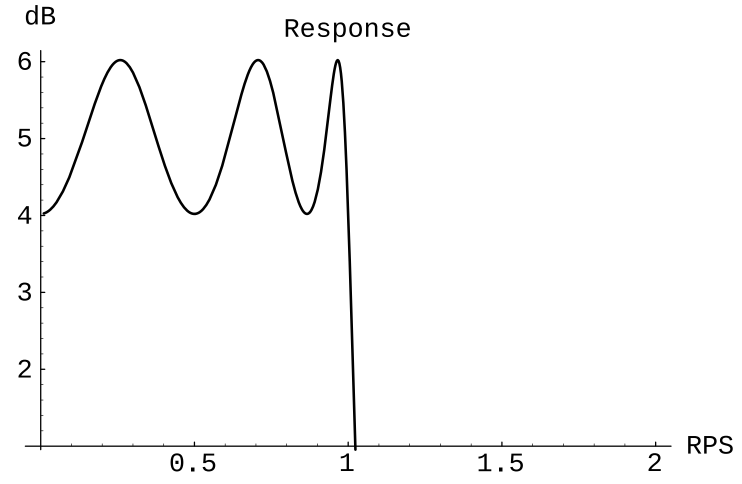
Design a sixth-order Chebychev filter whose Amax is 2 dB and whose DC gain is 1.588656 (absolute), that is the peak bandpass gain is 2 (absolute).

```
ChebychevLp[6,2]//N
```

```
{0.0205241 + 0.0353328 I, {},
  {-0.0469732 - 0.981705 I, -0.128333 - 0.718658 I,
   -0.175306 - 0.263047 I, -0.175306 + 0.263047 I,
   -0.128333 + 0.718658 I, -0.0469732 + 0.981705 I}}
```

Our filter design functions always normalize the filters so that the maximum gain is unity. We can get a DC gain of 1.5 by adjusting the gain of the filter at DC so that it is 1 (using the ContinuousAdjustGain function) and then multiplying by the appropriate gain constant.

```
ContinuousFreqResponseRadians[
    1.588656 *
    ContinuousAdjustGain[FilterFromGZP[ChebychevLp[6,2]],
                                0],2];
```



The following is a third-order Chebychev filter whose Amax is .5 dB and whose DC gain is 1 (absolute).

```
ChebychevLp[3,.5]
```

```
{-0.507168 + 0.504974 I, {},
  {-0.313228 - 1.02193 I, -0.626456,
   -0.313228 + 1.02193 I}}
```

```
dB[Abs[First[ChebychevLp[3,.5]]]]
```

```
-2.90546
```

The next example is a third-order Butterworth filter whose Amax is 3 dB and whose DC gain is 1.0 (absolute.)

```
ButterworthLp[3,3]
```

```
{1.00238 + 0. I, {}, {-0.500396 + 0.866711 I, -1.00079,
   -0.500396 - 0.866711 I}}
```

```
dB[Abs[First[ButterworthLp[3,3]]]]//N
```

```
0.0206244
```

The following is the third order Butterworth filter transformed into a highpass filter with a corner frequency of 1000Hz.

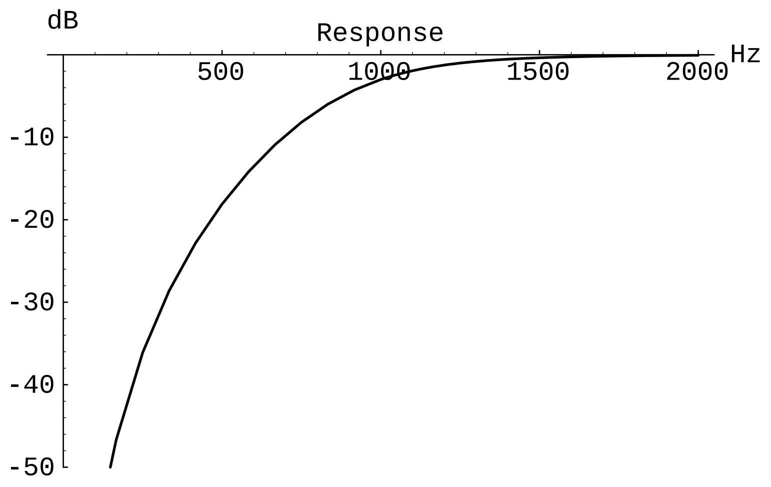
```
LpToHp[ButterworthLp[3,3],1000]//N
```

```
{-1. + 0. I, {0, 0, 0},
  {-3139.11 - 5437.09 I, -6278.21, -3139.11 + 5437.09 I}}
```

```
dB[Abs[First[LpToHp[ButterworthLp[3,3],1000]]]]//N
```

```
0.
```

```
ContinuousFreqResponse[
    FilterFromGZP[LpToHp[ButterworthLp[3,3],1000]],
    2000,{PlotRange->{-50,0}}];
```



The next example is the normalized third order Butterworth above transformed into a bandpass filter with corner frequencies of 1000 and 2000 Hz.

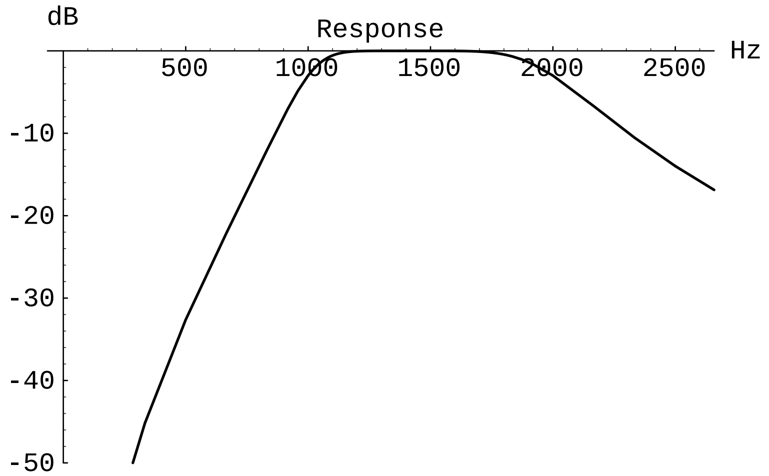
```
LpToBp[ButterworthLp[3,3],1000,2000]//N
```

```
{2.4864 10^11 + 0. I, {0, 0, 0},
  {-1105.33 - 6448.69 I, -2038.75 + 11894.4 I,
   -3144.08 + 8310.93 I, -3144.08 - 8310.93 I,
   -1105.33 + 6448.69 I, -2038.75 - 11894.4 I}}
```

```
dB[Abs[First[LpToBp[ButterworthLp[3,3],1000,2000]]]]//N
```

```
227.911
```

```
ContinuousFreqResponse[
    FilterFromGZP[LpToBp[ButterworthLp[3,3],1000,2000]],
    4000,{PlotRange->{-50,0}}];
```



The following example is the normalized third order Butterworth above transformed into a bandreject filter with corner frequencies of 1000 and 2000 Hz.

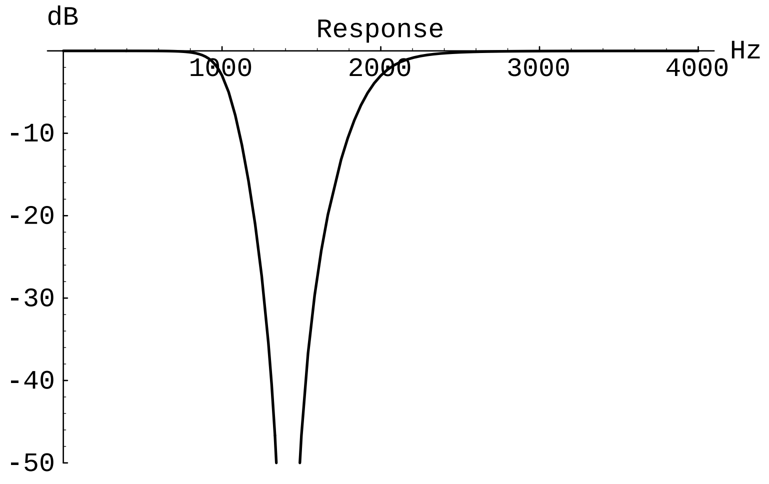
```
LpToBr[ButterworthLp[3,3],1000,2000]//N
```

```
{1. + 0. I, {8885.77 I, -8885.77 I, 8885.77 I,
   -8885.77 I, 8885.77 I, -8885.77 I},
  {-2034.83 - 11889.2 I, -1104.27 + 6452.08 I,
   -3139.11 - 8312.81 I, -3139.11 + 8312.81 I,
   -2034.83 + 11889.2 I, -1104.27 - 6452.08 I}}
```

```
dB[Abs[First[LpToBr[ButterworthLp[3,3],1000,2000]]]]//N
```

```
0.
```

```
ContinuousFreqResponse[
    FilterFromGZP[LpToBr[ButterworthLp[3,3],1000,2000]],
    4000,{PlotRange->{-50,0}}];
```



2. - Discrete Time Filter Design

This section of the notebook describes how to calculate first and second order digital filters. Discrete time filters are defined by their response in the z-domain. This section first presents some simple routines for describing polynomials in the z-domain and describes how to design first and second order filters. The frequency response and impulse response of these filters are then shown.

2.1 - Polynomial Evaluation Utilities

This section of the report defines some *Mathematica* functions to make it easier to work with filter functions in the Z-domain. These functions allow us to define a new polynomial given a list of its coefficients (**MakePoly**), evaluate a polynomial (**FilterEval**), and find the zeros and roots of ratios of polynomials (**RationalZeros** and **RationalPoles**).

In this notebook polynomials in z will be represented as lists of polynomial coefficients. The coefficients of a polynomial are listed from the initial constant to the coefficient multiplying the highest power. All coefficients past the end of the list are defined to be zero. The **PolyCoeff** function is used to pick out the n'th coefficient of a polynomial.

```
PolyCoeff[coeffs_, n_] :=
    If [ n < Length[coeffs],
        coeffs[[n+1]],
        0]
```

Given a list of coefficients, **MakePoly** returns a *Mathematica* expression that represents the polynomial. The use of z for the polynomial variable is arbitrary and was chosen to make the polynomials look good when printed.

```
MakePoly[coeffs_List] :=
    Block[{i},
        PolyCoeff[coeffs,0] +
        Sum[PolyCoeff[coeffs,i] z^i,
            { i, 1, Length[coeffs]}]]
```

FilterEval will return the value of a polynomial at a given point by substituting the desired value into the polynomial. Note, this function will work on all z-transforms.

```
FilterEval[poly_, x_] := (poly)/. z->x
```

For example the following is a third order polynomial and the result of evaluating it at z=1.

```
zPoly = MakePoly[{1,4,2,1}]
```

$1 + 4\ z + 2\ z^2 + z^3$

```
FilterEval[zPoly,1]
```

```
8
```

Likewise we can define a rational function by writing the ratio of two **MakePoly**'s and then evaluate it using **FilterEval**.

```
ratPoly = MakePoly[{1,-1,-1,1}] / MakePoly[{3,2,1}]
```

```
          2    3
1 - z - z  + z
---------------
            2
 3 + 2 z + z
```

```
FilterEval[ratPoly,2]
```

```
3
--
11
```

Finally we define two functions, **RationalPoles** and **RationalZeros**, that return a list of the poles and zeros of a rational function. The function **GetSolution** is used to pick apart the stylized expression that *Mathematica* uses to show the roots of an equation.

```
GetSolution[x_] :=
    If [ x == {},
         {},
         x[[1]][[2]]]

RationalPoles[rat_] :=
    Block[{roots},
         roots = Solve[FilterEval[Denominator[rat],x] == 0,x];
         Map[GetSolution, roots]]

RationalZeros[rat_] :=
    Block[{roots},
         roots = Solve[FilterEval[Numerator[rat],x] == 0,x];
         Map[GetSolution, roots]]
```

Using the previously defined rational polynomial (rat), we can find the poles and zeros of this filter:

```
RationalPoles[ratPoly]
```

```
           3/2          3/2
 -2 + I 2      -2 - I 2
{------------, ------------}
      2             2
```

```
RationalZeros[ratPoly]
```

```
{1, 1, -1}
```

■ 2.2 - Digital Filter Design Equations

This section describes some general first and second order filter design functions.

Using the impulse-invariant technique, a one pole (first-order lowpass) filter is defined by the following transfer function (in the z domain):

```
1 / MakePoly[{-epsilon,1}]
```

```
      1
 ___________
-epsilon + z
```

This filter has a time constant of tau if epsilon is given by

```
EpsilonFromTauFS[tau_, fs_] := E^(-1 / tau / fs)
```

Note that epsilon is a function of not only the time constant (tau) but also the sampling frequency (fs) of the digital filter. A first-order filter with time constant tau is given by (high pass in this case):

```
FirstOrderFromTau[tau_, fs_] :=
    MakePoly[{-EpsilonFromTauFS[tau, fs],1}]
```

A first order filter with a time constant (tau) of 1 ms is given by

```
FirstOrderFromTau[0.001, 16000]
```

```
-0.939413 + z
```

Likewise, a first-order filter with a corner frequency of f is described by :

```
FirstOrderFromCorner[f_, fs_] :=
    MakePoly[{1,-E^(-2 Pi f / fs)}]
```

A second order filter is defined by its center frequency (f), quality factor (q) and sampling frequency (fs):

```
SecondOrderFromCenterQ[f_, q_, fs_] :=
    Block[{cft, rho, theta},
        cft := f/fs;
        rho := E^(-Pi cft / q);
        theta := 2 Pi cft Sqrt[1 - 1/(4 q ^ 2)];
        MakePoly[{rho^2, -2 rho Cos[theta], 1}]]
```

The function **FilterGain** evaluates the filter transfer function (z transform) at the frequency f by making the substitution

z -> E ^ (I 2 Pi f / fs)

where fs is the sampling interval.

```
FilterGain[filter_,f_,fs_] :=
    FilterEval[filter,E^(I 2 Pi f / fs)]
```

Likewise the functions **FilterMag**, **FilterPhase** and **FilterDb** compute the magnitude, phase and gain in dB of a digital filter.

```
FilterMag[filter_,f_,fs_] :=
    Abs[FilterGain[filter,f,fs]]

FilterPhase[filter_,f_,fs_] :=
    Arg[FilterGain[filter,f,fs]]

FilterDb[filter_,f_,fs_] :=
    Db[FilterMag[filter,f,fs]]
```

The function **AdjustGain** adjusts the gain of a filter so that it has unit gain at any desired frequency.

```
AdjustGain[filter_,f_,fs_] :=
    filter/FilterGain[filter,f,fs]
```

We define the **MakeFilter** function to design a filter with a given feedback (poles) and feedforward (zeros) response. The resulting filter is just the ratio of the two polynomials. In addition a frequency and gain are specified so that the resulting filter can be normalized to have any desired gain at a specified frequency.

```
MakeFilter[forward_, feedback_, fs_, f_, gain_] :=
    gain AdjustGain[forward/feedback,f,fs]
```

The frequency response of a filter is plotted in *Mathematica* using the following function.

```
FreqResponse[coeffs_, fs_, opts_:{}] :=
    Block[{func,f},
    func = N[dB[Abs[FilterGain[coeffs,f,fs]]]];
    Plot[func,
             {f,1,fs/2-fs/1000},
             AxesLabel->{"  Hz","dB"},
             PlotLabel->"Response",
             opts]]
```

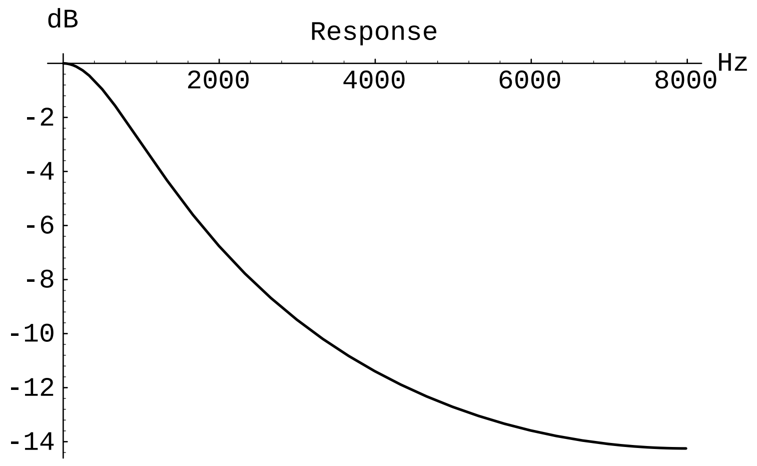
Likewise we define a similar function to plot the phase of the filter.

```
PhaseResponse[coeffs_, fs_] :=
    Block[{theFunction,f},
    theFunction = N[180/Pi FilterPhase[coeffs,f,fs]];
    Plot[theFunction,
             {f,1,fs/2-fs/1000},
             AxesLabel->{"  f","Phase (Degrees)"}]]
```

For example we define a first order low pass filter with a 3 dB point at 1000 Hz (the frequency response would look better with a logarithmic frequency scale):

```
firstFilt = MakeFilter[1,FirstOrderFromCorner[1000,16000],
                     16000,0,1]//N

  0.324768
______________
1. - 0.675232 z
```

```
FreqResponse[firstFilt,16000];
```



The gain at the corner frequency is given by the following *Mathematica* expression:

```
20 Log[ 10, Abs[N[FilterGain[firstFilt,1000,16000]]]]
```

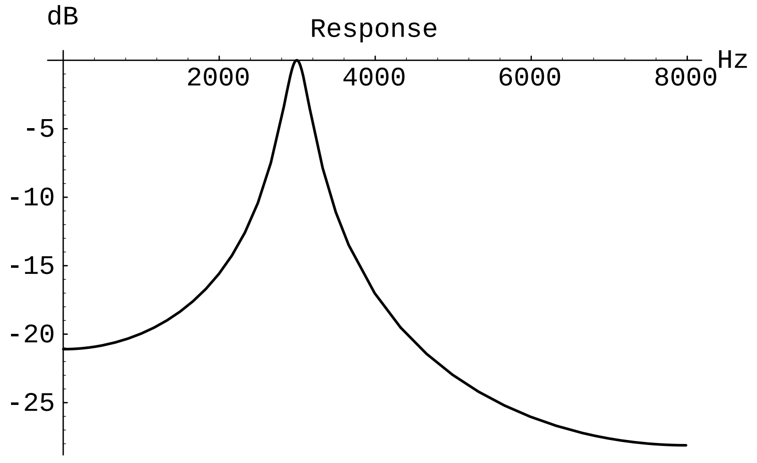
```
-2.95485
```

A second order bandpass filter with center frequency 3000 and a q of 10 is given by

```
secondFilt = MakeFilter[1,
                 SecondOrderFromCenterQ[3000,10,16000],
                 16000,3000,1]//N
```

$$\frac{-0.0953624 + 0.0380781\, I}{0.888865 - 0.724151\, z + z^2}$$

```
FreqResponse[secondFilt,16000];
```

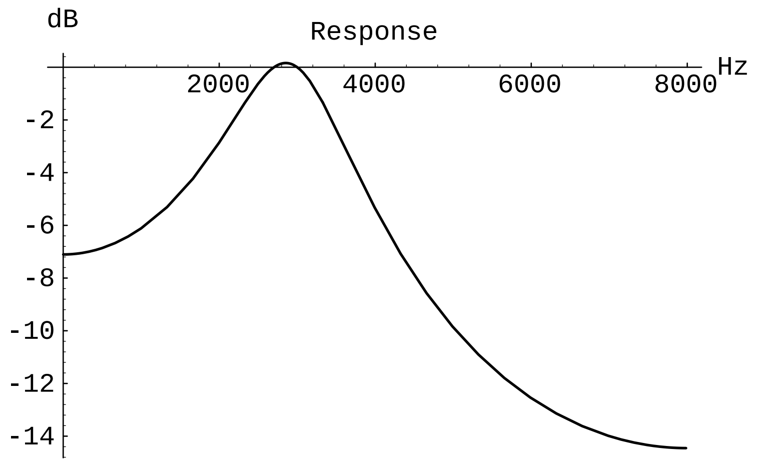


A less peaky response is possible using a lower q.

```
secondFilt = MakeFilter[1,
                  SecondOrderFromCenterQ[3000,2,16000],
                  16000,3000,1]//N
```

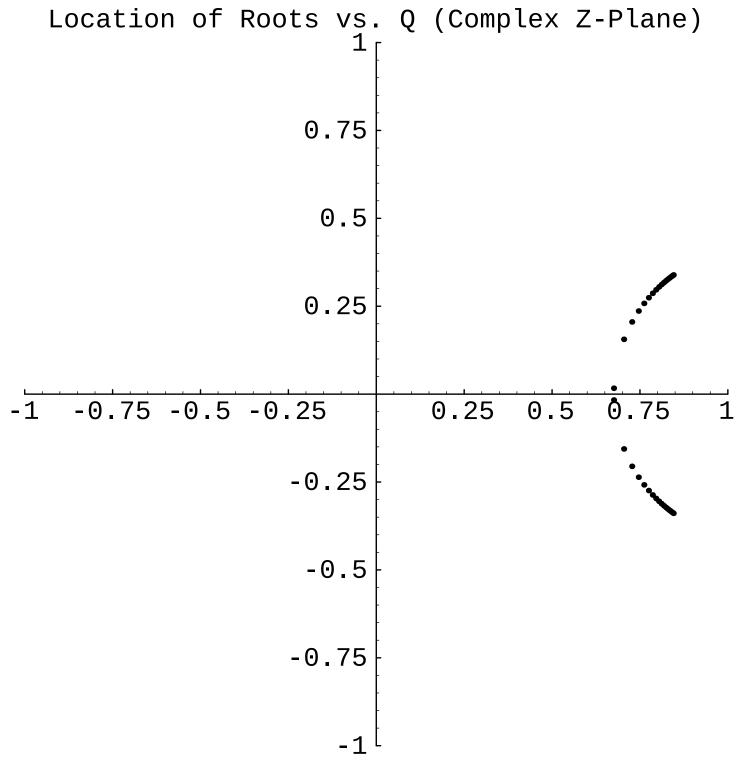
```
  -0.389971 + 0.133203 I
-------------------------
                        2
0.554855 - 0.621189 z + z
```

```
FreqResponse[secondFilt,16000];
```

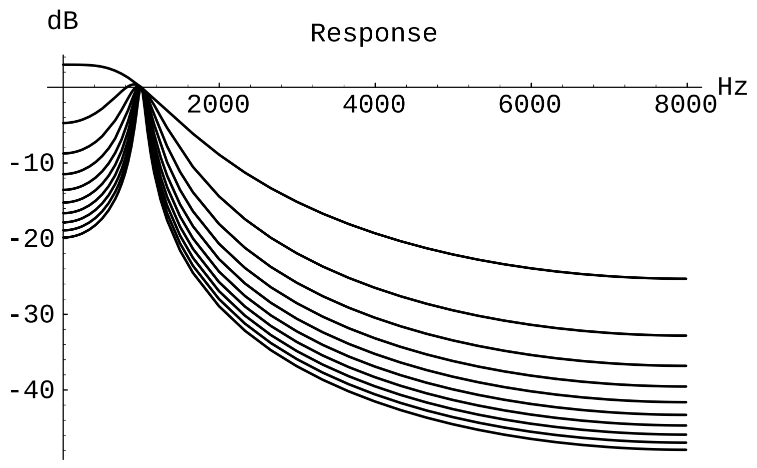


The poles and zeros in the complex z-plane of a stable discrete time filter are inside the unit circle. In general the resonant frequency of the root is proportional to the angle of the root from the positive real axis. Roots that are close to the unit circle will have a high q. The plot below shows the roots of a second order section with resonant frequency of 1000 Hz and a sampling frequency of 16000 Hz as the q varies from 0.5 to 2.2.

```
ListPlot[Map[{Re[#],Im[#]}&,
    Flatten[Table[
        RationalZeros[
            N[SecondOrderFromCenterQ[1000,q,16000]]],
        {q,.501,2.2,.1}]]],
        PlotRange->{{-1,1},{-1,1}},
        AspectRatio->1,
        PlotLabel->"Location of Roots vs. Q (Complex Z-Plane)"];
```



The following plot shows the frequency response of a number of second order discrete bandpass filters with a center frequency of 1000 Hz and a sampling frequency of 16000. The flattest curve is the frequency response of a second order bandpass filter with a q of 0.7 while the sharpest (narrowest) curve is the frequency response when the q is 10.

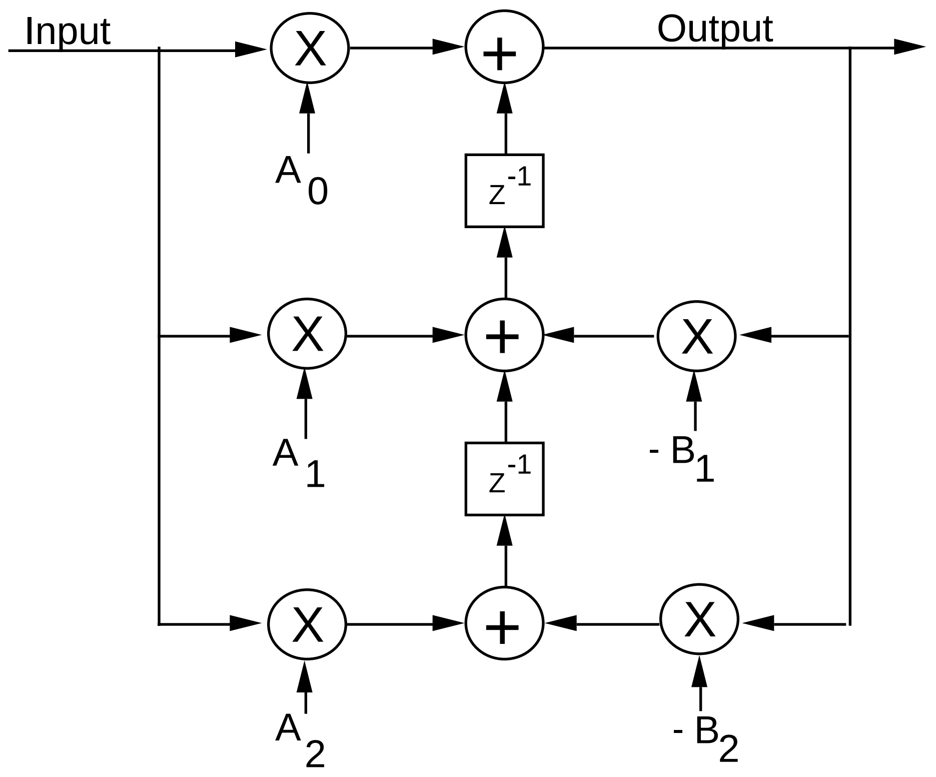


■ 2.3 - Time Domain Implementation

The following signal flow graph shows how the filter

$$\frac{A0 + \frac{A1}{z} + \frac{A2}{z^2}}{1 + \frac{B1}{z} + \frac{B2}{z^2}} = \frac{A0\ z^2 + A1\ z + A2}{z^2 + B1\ z0 + B2}$$

can be computed. Each stage of the ear cascade model discussed in this report is implemented this way.



The following function is used to evaluate the inverse z-transform of a ratio of two polynomials in z.

```
FindImpulseResponse[filter_,maxn_] :=
    Block[{num, denom, numorder, denomorder, response,
            theorder, inverseFilt, new},
            new = Expand[Numerator[filter]] /
                         Expand[Denominator[filter]];
            theorder = Abs[Exponent[Numerator[new],z]-
                         Exponent[Denominator[new],z]];
            inverseFilt = Expand[new/.z->1/z];
            num = Expand[Numerator[inverseFilt]];
            denom = Expand[Denominator[inverseFilt]];
            inverseFilt = Simplify[Expand[z^theorder num]/
                                 Expand[z^theorder denom]];
            num = Expand[Numerator[inverseFilt]];
            denom = Expand[Denominator[inverseFilt]];
            numorder = Exponent[num,z];
            denomorder = Exponent[denom,z];
            response = Table[0,{maxn}];
            Do [ response[[i+1]] =
                    (Coefficient[num,z,i] -
                     Sum[Coefficient[denom,z,j] *
                             response[[i-j+1]],
                        {j,1,Min[i,denomorder]}])/
                     Coefficient[denom,z,0],
                {i, 0, maxn-1}];
            Return[response]]
```

This is the impulse response for a simple low pass filter.

```
FindImpulseResponse[z/(z-.9),10]
```

```
{1, 0.9, 0.81, 0.729, 0.6561, 0.59049, 0.531441,
  0.478297, 0.430467, 0.38742}
```

■ 2.4 - Digital Test Code

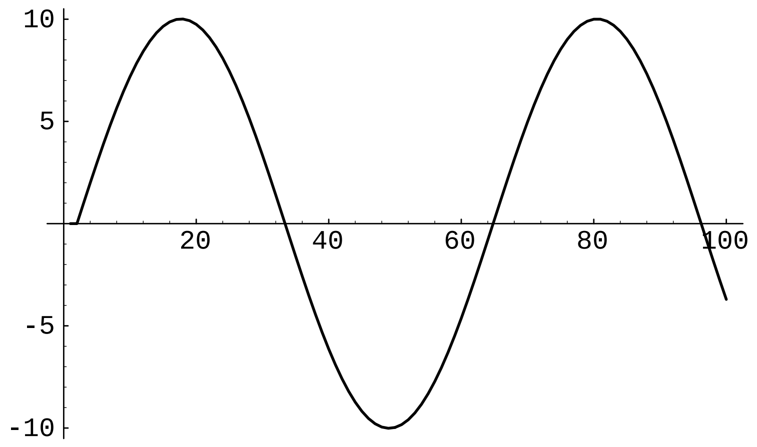
The following examples are used to verify the digital filter code.

```
Clear[a,b]
Simplify[FindImpulseResponse[1/(1-a z)/(1-b z),5]]
```

```
{0, 0, 1/(a b), (a + b)/(a^2 b^2), (a^2 + a b + b^2)/(a^3 b^3)}
```

The next example is the impulse response for a perfect oscillator with a pair of poles on the unit circle.

```
ListPlot[FindImpulseResponse[1/(1-1.99z+z*z),100],
              PlotJoined->True];
```



A second order filter with finite q will have an impulse response that is a decaying sinusoid. The following example shows the impulse response to a second order filter with a center frequency of 1000 Hz, sampling frequency of 8000 Hz and a q of 10.

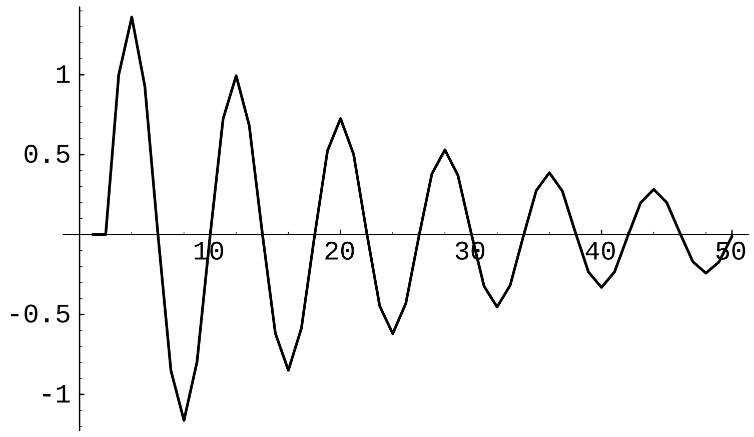
```
aFilter = SecondOrderFromCenterQ[1000,10,8000] //N
```

```
0.924465 - 1.36109 z + z^2
```

```
RationalPoles[1/aFilter]
```

```
{0.680544 - 0.679209 I, 0.680544 + 0.679209 I}
```

```
ListPlot[FindImpulseResponse[1/aFilter,50],
         PlotJoined->True];
```



■ 2.5 - Analog to Digital Filter Conversion

Filters can be transformed from the analog domain into the digitial domains using a number of methods. The simplest is known as the standard-z transform or the impulse invariant technique. A more accurate approach, the bilinear transform, is discussed in the second half of this section

The basic procedure of the impulse-invariant technique is to reduce the filter into a number of first order sections using the partial fractions expansion. Each first order section (in the s-domain) can then be transformed into its digital equivalent by mapping the first order pole in the s-plane into a single pole in the z-plane. Complex poles lead to complex filter coefficients but real valued filter coefficients are possible by combining complex conjugate pairs.

The function **GroupRoots** is used to group complex conjugate roots into common lists. This routine works by keeping a list of active roots (roots). Each iteration through the loop it takes the first root off of the list (current) and then sorts the remaining roots by how close they are to the conjugate of the current root. If the difference in magnitude between the complex conjugate of the current root and the first item in the sorted list is less than 0.0001% of the magnitude of the root then we have a complex conjugate pair. This routine returns a list of isolated roots and complex conjugate pairs. (Each item in the returned list is a list of either one or two element lists.)

```
GroupRoots[listofroots_] :=
    Block[{groups, current, roots},
        groups = {};
        roots = listofroots;
        While[Length[roots]>0,
            current = Conjugate[First[roots]];
            roots = Sort[Rest[roots],
                        Abs[#1-current]<=Abs[#2-current]&];
            If [Length[roots]>0 &&
                Abs[First[roots]-current]<Abs[current/10^6],
                    Block[{},
                        AppendTo[groups,
                                {Conjugate[current],
                                    First[roots]}];
                        roots = Rest[roots]],
                AppendTo[groups,List[Conjugate[current]]]];
        ];
        Return[groups];
    ]
```

```
GroupRoots[Last[LpToBr[ChebychevLp[7,3],1000,2000]]]
```

```
{{-122.44 + 12636.4 I, -122.44 - 12636.4 I},
  {-555.427 - 13663.2 I, -555.427 + 13663.2 I},
  {-2819.07 + 17784.2 I, -2819.07 - 17784.2 I},
  {-60.5376 - 6247.79 I, -60.5376 + 6247.79 I},
  {-234.529 + 5769.27 I, -234.529 - 5769.27 I},
  {-686.512 - 4330.89 I, -686.512 + 4330.89 I},
  {-1643.86}, {-48031.3}}
```

The partial fractions expansion is done by calculating the residue due to each pole in the filter's s-domain transfer function. Several functions are defined here to make that easier. **DeletePole** deletes a single pole from a list of poles by removing the pole closest to the given pole. **EvaluateGZP** finds the value of a filter (expressed in terms of its gain, zeros and poles) at a given point. Finally, **FindResidue** combines all of these function to compute the residue due to a pole of the transfer function.

```
DeletePole[pole_, poles_] :=
    Rest[Sort[poles, Abs[#1-pole]<=Abs[#2-pole]&]]
```

```
DeletePole[-1,{3,-1}]
```

```
{3}
```

```
EvaluateGZP[gain_, zeros_, poles_, w_] :=
    gain * Apply[Times,Map[(w-#)&,zeros]]/
            Apply[Times,Map[(w-#)&,poles]]
```

```
FilterFromGZP[{2,{2},{3,-1}}]
```

```
   2. (-2. + s)
__________________
(-3. + s) (1. + s)
```

```
EvaluateGZP[2,{2},{3,-1},4]
```

```
4/5
```

```
FilterFromGZP[{2,{2},{3,-1}}]/.s->4//N
```

```
0.8
```

```
FindResidue[pole_, gain_, zeros_, poles_] :=
    EvaluateGZP[gain, zeros, DeletePole[pole,poles], pole]
```

```
FindResidue[-1,2,{2},{3,-1}]
```

```
3/2
```

The function **PartialFractions** transforms a filter function (in gain, zeros, poles form) into a sum of first order rational polynomials. This is test code for the standard-z transform code that follows.

```
PartialExpand[polelist_, gain_, zeros_, poles_] :=
    If [Length[polelist] == 1,
        FindResidue[First[polelist],gain,zeros,poles]/
            (s-First[polelist]),
        FindResidue[First[polelist],gain,zeros,poles]/
            (s-First[polelist]) +
         FindResidue[Last[polelist],gain,zeros,poles]/
            (s-Last[polelist])]
```

```
PartialExpand[{-1},2,{2},{3,-1}]
```

```
    3
---------
2 (1 + s)
```

```
PartialFractions[{gain_,zeros_,poles_}] :=
    Apply[Plus,Map[PartialExpand[#,gain,zeros,poles]&,
                        GroupRoots[poles]]]
```

```
PartialFractions[{2,{2},{3,-1}}]
```

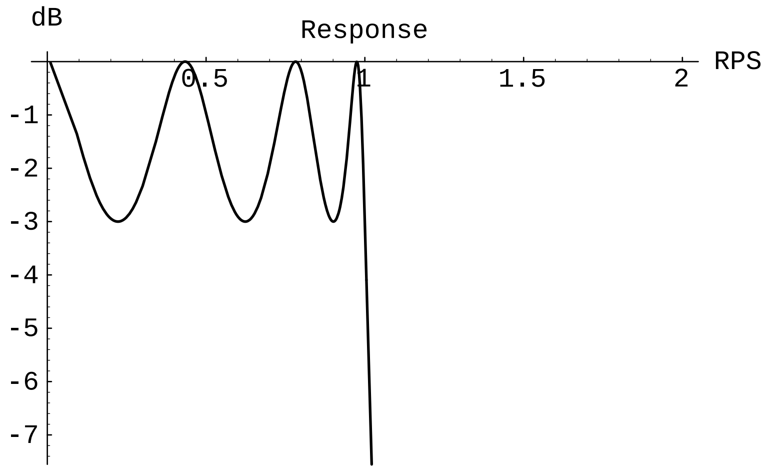
```
    1            3
---------- + ---------
2 (-3 + s)   2 (1 + s)
```

A seventh order Chebychev polynomial is a more difficult function to transform into partial fractions form. We plot the resulting transfer function to insure that it is equivalent to the original gain, zero, pole form.

```
PartialFractions[ChebychevLp[7,3]]//N
```

```
  0.0186432 - 0.021997 I       0.0272631 + 0.00938959 I
------------------------- + ------------------------- +
0.0281456 + 0.982696 I + s   0.0281456 - 0.982696 I + s
  -0.0634134 + 0.0515983 I
  ------------------------- +
  0.0788623 + 0.788061 I + s
  -0.0808722 - 0.0119726 I
  ------------------------- +
  0.0788623 - 0.788061 I + s
   0.120281 - 0.00874408 I      0.107875 - 0.0539153 I
  ------------------------- + ------------------------- +
  0.113959 - 0.437341 I + s   0.113959 + 0.437341 I + s
  -0.129777 + 0.0356411 I
  -----------------------
       0.126485 + s
```

```
ContinuousFreqResponseRadians[
    PartialFractions[ChebychevLp[7,3]]//N,2];
```



Using the impulse-invariant technique, a simple pole in the s-domain is transformed into the z-domain using the substitution

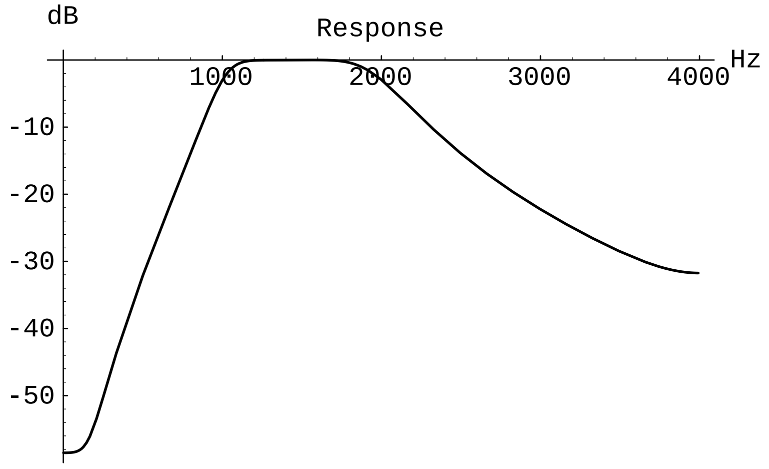
```
  r          T r
-----  ->  -------
s - p        T p
            E
        1 - ----
             z
```

The r term represents the residue corresponding to the pole at p in the original s-domain filter and T is the sampling interval in the new digital filter. The resulting digital filter has the same impulse response (within the limits due to aliasing) as the original continuous filter. Note that when a pair of complex conjugate poles are transformed the two complex first order filters can be combined into a single second order filter with real coefficients.

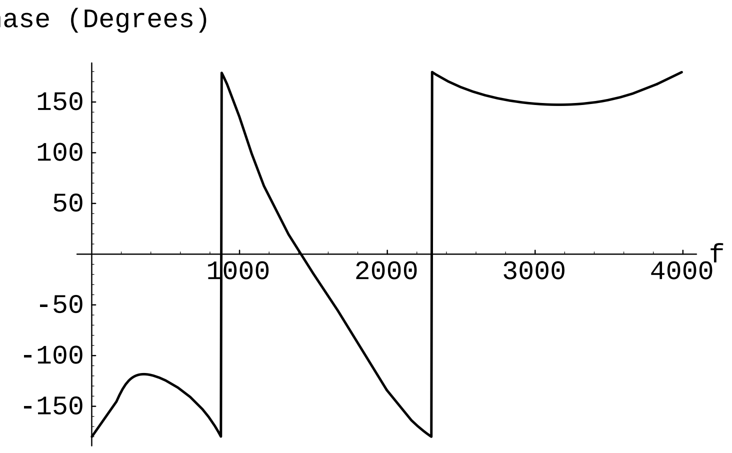
```
StandardZExpand[polelist_, gain_, zeros_, poles_,T_] :=
    If [Length[polelist] == 1,
        T FindResidue[First[polelist],gain,zeros,poles]/
            (1-E^(T First[polelist]) z^-1),
        Block[{r1,r2,p1,p2},
            p1 = First[polelist];
            p2 = Last[polelist];
            r1 = T FindResidue[p1,gain,zeros,poles];
            r2 = T FindResidue[p2,gain,zeros,poles];
            zp1 = E^(T p1);
            zp2 = E^(T p2);
            (r1 + r2 - (r1 zp2 + r2 zp1)/z)/
                (1 - (zp1+zp2)/z + zp1 zp2 / z^2)]]

StandardZTransform[{gain_,zeros_,poles_},T_] :=
    Chop[Apply[Plus,Map[StandardZExpand[#,gain,zeros,poles,T]&,
                        GroupRoots[poles]]]]

StandardZTransform[LpToBp[ButterworthLp[3,3],1000,2000],
                   1/8000.]//N
```

```
   0.78602 - 0.442098              -0.308588 + 0.125444
             --------                          --------
                z                                 z
------------------------ + ----------------------- +
     0.455655   0.684739        0.75856   1.20597
1. + -------- - --------   1. + ------- - -------
        2          z               2         z
       z                          z

    -0.477432 + 0.307017
                --------
                   z
  ------------------------
       0.600684   0.130048
  1. + -------- - --------
          2          z
         z
```

```
FreqResponse[
    StandardZTransform[LpToBp[ButterworthLp[3,3],
                                   1000,2000],
                        1/8000.]//N,
    8000];
```



```
PhaseResponse[
    StandardZTransform[LpToBp[ButterworthLp[3,3],
                                1000,2000],
                         1/8000.]//N,
    8000];
```



The bilinear z-transform is an alternative to the impulse invariant, or standard-z, transform. The impulse-invariant technique endeavors to match the impulse response of the original analog filter. But there is no guarantee that the impulse response is bandlimited. In fact, there is usually significant energy above the Nyquist frequency which aliases and corrupts the frequency response. Thus the impulse response will be “correct” but the frequency response is wrong.

The bilinear transform applies a conformal mapping to the original Laplace transform. This conformal mapping compresses the entire jω axis into the region between DC and F_S, thus it is necessary to adjust the design parameters of the original continuous filter design so that the transformed poles will be at the proper location. The filter design process is more complicated but the frequeny domain results are more accurate.

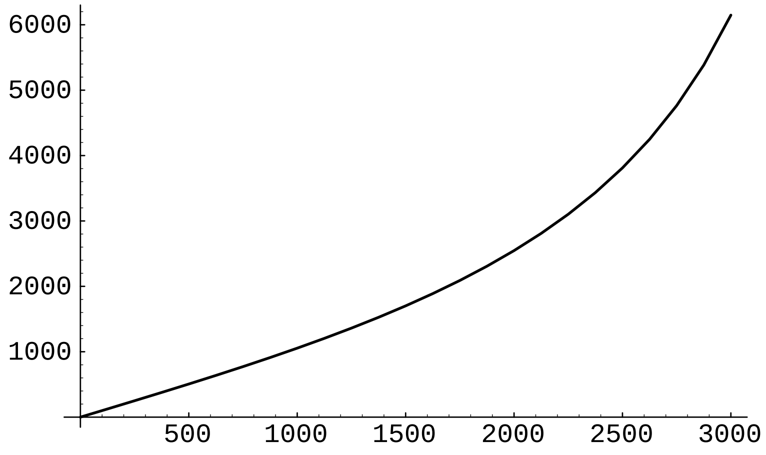
The bilinear z-transform changes a continuous domain filter into the z-domain by using the transformation

$$s \to \frac{2}{T} \frac{1 - z^{-1}}{1 + z^{-1}}$$

The following function warps frequencies in the continuous domain into the bilinear-z domain. Note that the bilinear transform maps frequencies in the sampled filter between 0 and the Nyquist rate into the entire frequency range from 0 to infinity. When designing filters that are to be transformed using the bilinear transform, it is necessary to first warp the corner frequencies using this transformation and then design the continuous filter. The **BinlinearPreWarp** is a function of the desired frequency in the digital domain and the sampling interval of the digitial filter.

```
BilinearPreWarp[f_,fs_] :=
    fs / Pi Tan [ Pi f/fs ]
```

```
Plot[BilinearPreWarp[f,8000],{f,0,3000}];
```



```
ButterworthLp[3,3]
```

```
{1.00238 + 0. I, {}, {-0.500396 + 0.866711 I, -1.00079,
   -0.500396 - 0.866711 I}}
```

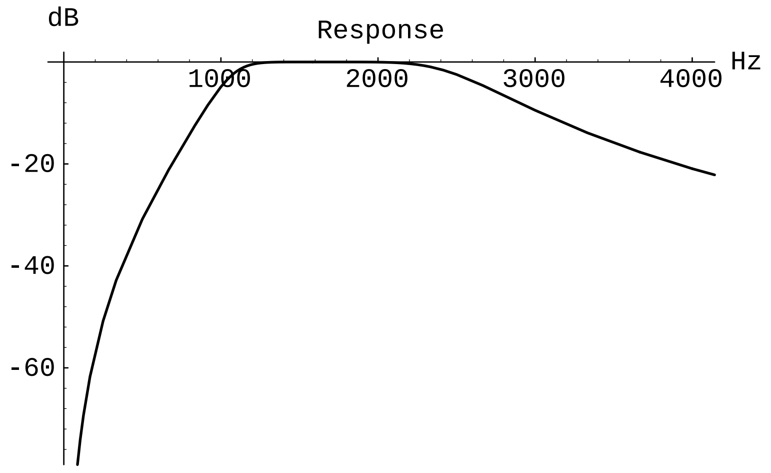
The following 3rd order filter will be used as an example for the rest of the bilinear-z transform functions. A third order polynomial is first transformed into a bandpass filter, but the passband frequencies have been prewarped.

```
ASampleGZP = LpToBp[ButterworthLp[3,3],
                       BilinearPreWarp[1000,8000],
                       BilinearPreWarp[2000,8000]]//N
```

```
           11
{8.25295 10   + 0. I, {0, 0, 0},
  {-1467.47 - 6792.22 I, -3222.53 + 14915.5 I,
   -4690. + 9167.47 I, -4690. - 9167.47 I,
   -1467.47 + 6792.22 I, -3222.53 - 14915.5 I}}
```

Note, the passband is not between 1000 and 2000Hz in the resulting filter.

```
ContinuousFreqResponse[FilterFromGZP[ASampleGZP],8000];
```



BilinearExpand is a helper function for the BilinearZTransform function which does the real work.

```
BilinearExpand[root_,T_] :=
    (2/T + root)/(2/T - root)

BilinearZTransform[{gain_,zeros_,poles_},T_] :=
    Block[{RootDiff,ExtraZeros,ExtraPoles},
        RootDiff = Length[zeros] - Length[poles];
        If [RootDiff > 0,
                ExtraPoles = Table[-1,{RootDiff}];
                ExtraZeros = {},
            ExtraZeros = Table[-1,{-RootDiff}];
            ExtraPoles = {}];
        {gain * Apply[Times,Map[(2/T-#)&,zeros]] /
                Apply[Times,Map[(2/T-#)&,poles]],
         Join[Map[BilinearExpand[#,T]&,zeros],ExtraZeros],
         Join[Map[BilinearExpand[#,T]&,poles],ExtraPoles]}]
```

We can now apply the bilinear-z transform to the filter derived above.

```
aBilinearGZP = BilinearZTransform[ASampleGZP,1/8000]//N
```

```
                         -21
{0.0317453 + 1.2504 10    I, {1., 1., 1., -1., -1., -1.},
  {0.591357 - 0.618799 I, 0.0390923 + 0.806274 I,
   0.292825 + 0.572834 I, 0.292825 - 0.572834 I,
   0.591357 + 0.618799 I, 0.0390923 - 0.806274 I}}
```

Finally, the **SOSFromBilinearGZP** function is used to transform the GZP result of the bilinear-z transform into second-order sections. This is done by grabbing pairs of poles or zeros, preferably in complex-conjugate pairs, and creating a second-order biquadractic section from them. The **DropPair** and **GrabPair** functions are used to pick out pairs of roots from a list.

```
GrabPair[rootlist_] :=
    If [ Length[First[rootlist]] == 2,
            First[rootlist],
        Join[First[rootlist],
                If[Length[rootlist] > 1,
                    First[Rest[rootlist]],
                    {0}]]]

DropPair[rootlist_] :=
    If [ Length[First[rootlist]] == 2,
            Rest[rootlist],
        Rest[Rest[rootlist]]]
```

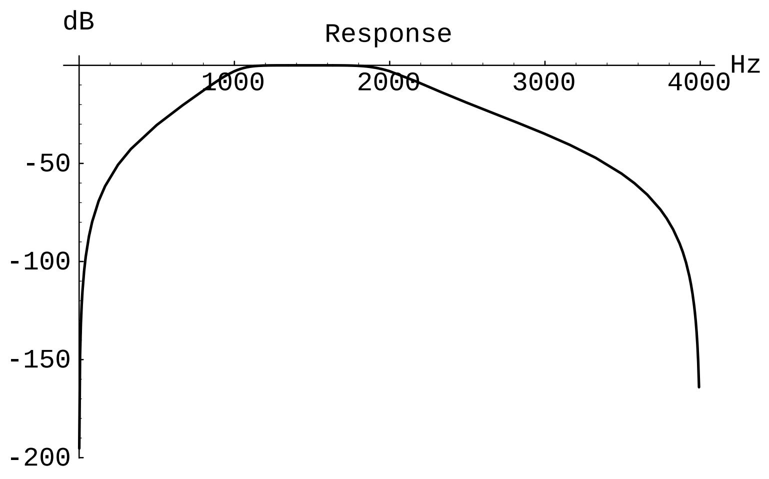
```
SosFromBilinearGZP[{gain_,zeros_,poles_}] :=
    Block[{pg,zg,pp,zp,filt},
        filt = gain;
        pg = Sort[GroupRoots[poles],Length[#1]>Length[#2]&];
        zg = Sort[GroupRoots[zeros],Length[#1]>Length[#2]&];
        While[Length[pg] > 0,
            pp = GrabPair[pg];
            pg = DropPair[pg];
            zp = GrabPair[zg];
            zg = DropPair[zg];
            filt = filt (z^-2 - Re[First[zp]+Last[zp]]z^-1 +
                                Re[First[zp]Last[zp]])/
                        (z^-2 - Re[First[pp]+Last[pp]]z^-1 +
                                Re[First[pp]Last[pp]]);
            ];
        Chop[filt]
        ]
```

The following shows the resulting biquadratic realization of the original 1000-2000Hz continuous bandpass filter. We can plot the frequency response to verify that the result is correct. Note, now the filter has the proper passband.

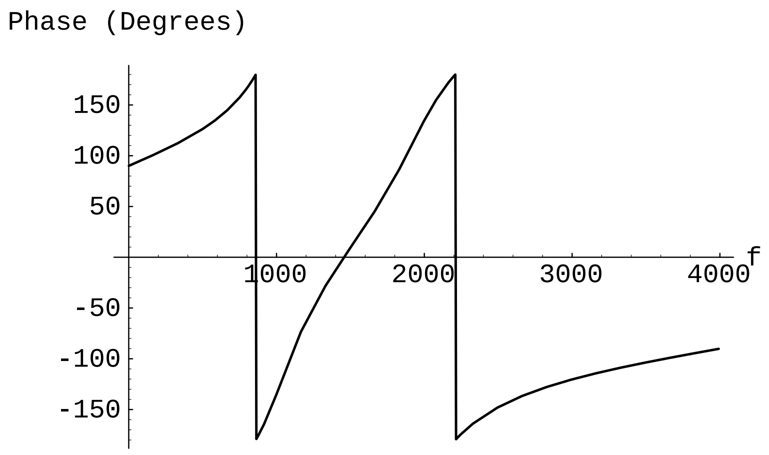
```
theSOSFilter = SosFromBilinearGZP[aBilinearGZP]
```

$$\frac{0.0317453\,(-1. + z^{-2})\,(1. + z^{-2} - 2./z)\,(1. + z^{-2} + 2./z)}{(0.732615 + z^{-2} - 1.18271/z)\,(0.413886 + z^{-2} - 0.585651/z)\,(0.651606 + z^{-2} - 0.0781846/z)}$$

```
FreqResponse[theSOSFilter,8000];
```



```
PhaseResponse[theSOSFilter,8000];
```



■ 2.6 - Digital Usage

```
FindImpulseResponse::usage = "FindImpulseResponse[filter,
maxn] returns an array of the first maxn samples of the
impulse response of the input filter.";

PolyCoeff::usage = "PolyCoeff[coeffs, n] returns the n'th
entry of a coefficient list.  Zero is returned for all
indices past the end of the list.";

MakePoly::usage = "MakePoly[coeflist] converts a list of
coefficients into a polynomial in z.";

FilterEval::uage = "FilterEval[filter,x] evaluates the
given filter at the argument x.";

RationalPoles::usage = "RationalPoles[rationalfunc] returns
the poles of a ratio of polynomials in z.";

RationalZeros::usage = "RationalZeros[rationalfunc] returns
the zeros of a ratio of polynomials in z.";

EpsilonFromTauFS::usage = "EpsilonFromTauFS[tau, fs]
returns the epsilon in the equation 1/(1-epsilon z) that
will give a time constant of tau with a sampling frequency
of fs.";

FirstOrderFromTau::usage = "FirstOrderFromTau[tau,fs]
returns a first order polynomial in z that represents a
digital filter with time constant tau and sampling
frequency fs.";

FirstOrderFromCorner::usage = "FirstOrderFromCorner[f,fs]
returns a first order polynomial in z that represents a
digital first with corner frequency f and sampling
frequency fs.";
```

```
SecondOrderFromCenterQ::usage = "SecondOrderFromCenterQ[f,
q,fs] returns a second order digital filter (represented
in the z-domain) with a center frequency of f, a quality
factor of q and a sampling frequency of fs.";

MakeFilter::usage = "MakeFilter[forward,feedback,fs,f,gain]
creates a filter by dividing a feedforward (zeros)
polynomial by the feedback (poles) polynomial.  The gain
of the resulting filter is adjusted so that it has the
desired gain at the frequency f.  The sampling interval is
fs.";

FilterGain::usage = "FilterGain[filter,f,fs] evaluates the
filter's gain (sampling interval of fs) at the frequency
f.";

FilterMag::usage = "FilterMag[filter,f,fs] evaluates the
magnitude of the filter's response (sampling interval of fs)
at the frequency f.";

FilterPhase::usage = "FilterPhase[filter,f,fs] evaluates the
filter's phase (sampling interval of fs) at the frequency
f.";

FilterDb::usage = "FilterDb[filter,f,fs] evaluates the
filter's gain in decibels (sampling interval of fs) at the
frequency f.";

FreqResponse::usage = "FreqResponse[filter,fs] plots the
frequency response of a digital filter.  The filter is
represented in the z domain (function of z) and fs is the
sampling interval.";

PhaseResponse::usage = "PhaseResponse[filter,fs] plots the
phase response of a digital filter.  The filter is
represented in the z domain (function of z) and fs is the
sampling interval.";

StandardZTransform::usage = "StandardZTransform[GZP, T]
transforms an analog filter in Gain-Zero-Poles (GZP) form
into a digital filter (with sample rate T) using the Standard
Z transform (impulse invariant).  This function transforms
each pair of complex conjugate poles into the equivalent
digital filter.  Note, this function fails if there are
multiple poles at one location.";

BilinearPreWarp::usage = "BilinearPreWarp[f, fs] prewarps a
a frequency (f) so that after the Bilinear Z-transform the
poles or zeros will be at the right spot.";

BilinearExpand::usage = "BilinearExpand[GZP, T] transforms
an analog filter described by the Gain-Zero-Poles (GZP)
structure into a digital filter with a sampling rate of T.
Note, this function fails if there are  multiple poles at
one location.";
```

```
SOSFromBilinearGZP::usage = "SOSFromBilinearGZP[GZP] rearranges
the poles and zeros of a digital filter so that complex
conjugate roots are combined.  The resulting filter will have
real coefficients and can be easily implemented.";
```

Acknowledgements

The author wishes to acknowledge many useful comments from Richard Lyon, Jim Kaiser, Ray DeCarlo, Daniel Naar, and Steve Shepard.

References

[Daryanani1976] Gobind Daryanani, *Principles of Active Network Synthesis and Design*, John Wiley and Sons, New York, 1976.

[Evans1992] Brian L. Evans and James H. McClellan, “Symbolic Analysis of Signals and Systems,” in S*ymbolic and Knowledge-Based Signal Processing*, Alan V. Oppenheim and S. Hamid Nawab, eds., Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Oppenheim1989] Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Slaney1988] Malcolm Slaney, “Lyon’s Cochlear Model,” Apple Technical Report #13, Apple Computer Corporate Library, Cupertino, CA 95014, 1988.

[Slaney1990] Malcolm Slaney, “Interactive Signal Processing Documents,” *IEEE ASSP Magazine*, 7 (1990), pp. 8-20.

[Smith1992] Julius O. Smith, “Rectangular, Hanning, and Hamming Window Transforms,” “The Kaiser Window,” “The Window Method for FIR Digital Filter Design,” *Mathematica* notebooks, CCRMA, Stanford University, 1993. [Available via anonymous ftp to ccrma-ftp.stanford.edu, files pub/DSP/GenHamming.ma.Z, pub/DSP/Kaiser.ma.Z, pub/DSP/WinFlt.ma.Z.]

[Slaney1992] Malcolm Slaney, “Interactive Signal Processing Documents,” in S*ymbolic and Knowledge-Based Signal Processing*, Alan V. Oppenheim and S. Hamid Nawab, eds., Prentice-Hall, Englewood Cliffs, NJ, 1992.

Index