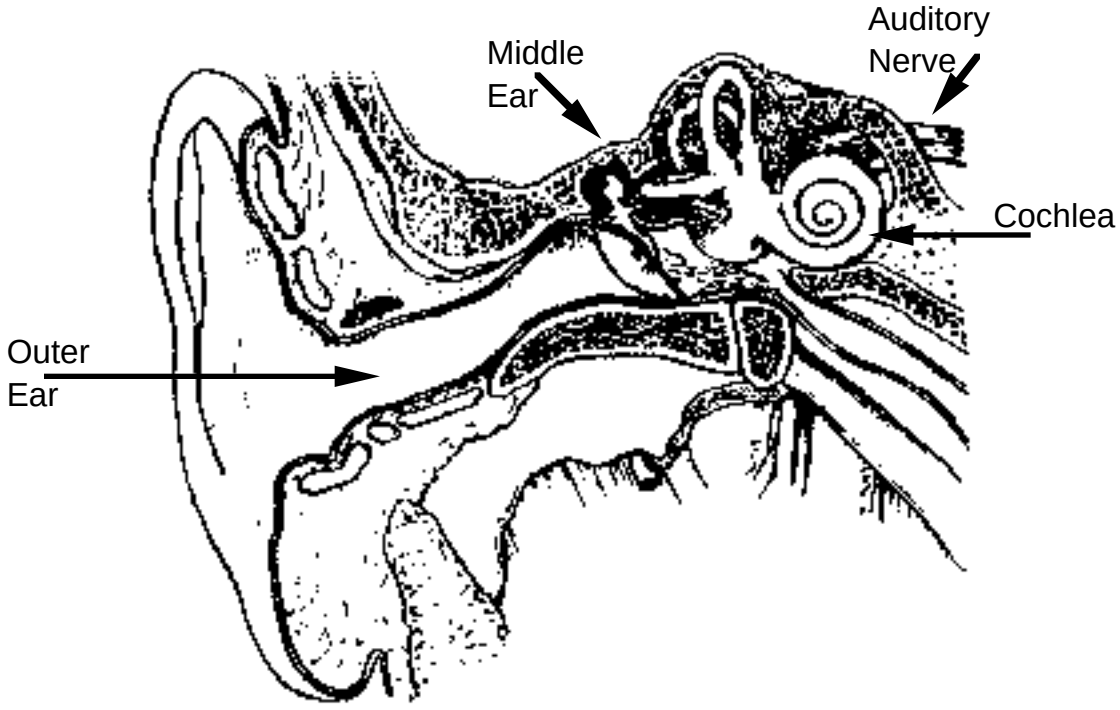
Lyon's Cochlear Model

MalcolmSlaney
AdvancedTechnologyGroup
AppleTechnicalReport#13
Copyright © 1988
AppleComputer,Inc
malcolm@apple.com



From *Tissues and Organs: A Text-Atlas of Scanning Electron Microscopy*, by Richard Kessel and Randy H. Kardon, Copyright © 1979, W. H. Freeman and Company. Reproduced with permission.

1 - Introduction

1.1 - About this report

This technical report describes the implementation of a model of the cochlea developed by Richard Lyon. Automatic speech recognition is a difficult problem and by studying the human cochlea we will gain a better understanding of how humans perceive speech. Hopefully this knowledge will help us to design better speech recognition systems.

1.2 - About *Mathematica*

The words you are now reading were formatted using a revolutionary symbolic math program called *Mathematica*. The resulting document is called a notebook. *Mathematica* is a symbolic math package developed by Wolfram Research Institute and is available on many popular computers. This report is available as a conventional paper document but more interestingly is also available as an electronic document. Much like a scientist's notebook, this report is a living, breathing document. Readers are encouraged to play with this notebook, modify the models and extend it to apply to their own research.

Mathematica is used here to describe the cochlear model for two reasons. First, it provides a portable way to describe the characteristics of the computations. This will allow more people to understand the important characteristics of the model than if the model were described in a conventional programming language. Secondly, and perhaps more importantly, because *Mathematica* is a complete symbolic math package it allows the reader to explore the mathematics of the model. Thus if the reader is unsure about a concept it is possible to play with the equations that are confusing.

The *Mathematica* system is described in a book written by Stephen Wolfram called "*Mathematica*: A System for Doing Mathematics by Computer." Like many symbolic math packages (for example Macsyma) a model is built using textual equations. These equations can then be manipulated algebraically or actually used for computation. In this

report *Mathematica* is mostly used for its computational ability and its excellent graphics.

One characteristic of a *Mathematica* notebook is that details of the model can be hidden in closed cells. Like an outline, this notebook is organized into sections and subsections. A section of this notebook is used to describe one part of the model (for example the filter design software) and subsections are used to state the equations, show graphical examples and perhaps include some test code. Most readers of this report are probably not interested in the test code and these subsections can be closed so as to not clutter the report. See the *Mathematica* help screens for information on opening and closing sections of a notebook.

The information in this report is being published not only on paper but also on Macintosh® 3 1/2" floppy disk. By publishing this material electronically we hope to give other scientists and engineers better acccess to this information. Not only do we hope that readers will understand the model better but we hope they will be able to better apply it to their own work.

1.3 - How to use this report

This report can be read two ways. The paper copy of this notebook can be read like a normal report. Readers who are not familiar with the *Mathematica* notation used to write this report can ignore the equations. We have written this report so that most of the material is explained in text and figures. While equations (and programs) are entered into *Mathematica* with linear text the basic principles should be evident for those readers who have questions about the details.

We hope that most readers will be able to access a copy of *Mathematica* and read the electronic version of this report. A notebook reader is provided on the floppy disk to allow readers to browse through the document but the real power of the notebook comes from interacting with the equations defined here. Learning is not a one way process and the material will be better understood if you, the reader, can interact with the material as I, the author, did while writing this report.

The best way to interact with this notebook is to read the description, study the examples and then modify an example to see how different parameters give different results. For example an appendix to this report describes digital filtering and provides functions to design first and second order filters. Much can be learned about digital filtering by combining these filters and studying the resulting frequency response or pole-zero plots.

Readers might also want to modify this model to better fit their own experience or ideas. For example this notebook describes a relatively simple model of the effects of the outer and middle ears on the sound. A reader might be interested in providing a better model or removing the outer and middle ear filters completely and studying the change in response. As another example, this report describes a simple Automatic Gain Control (AGC) to compensate for the large range of sounds produced by humans. This notebook explores several variations on the basic AGC but readers might want to try their own.

1.4 - Prerequisites

This report was written for readers with some knowledge of signal processing. The filters in this notebook are described using the Laplace and Z transforms. While we never actually calculate a Laplace or Z transform, readers who are comfortable with these tranformations will get the most out of this notebook. More information about digital signal processing can be found in [Oppenheim75].

An appendix to this report defines several functions that are used to design continuous and digital filters. In the rest of this report the names used for these functions should be self explanatory. For readers of the electronic version of this document, more information is available about any function by selecting the function name and picking "About the selection" from the menu.

1.5 - About This Notebook

The purpose of this notebook is to describe the design and implementation of a model of sound propagation in the human cochlea. Two versions of the cochlear model are described here. In the model originally published a combination of a cascade filter bank and a parallel set of resonators were used. Later it was realized that these resonators could be folded into the cascade filter bank to build a cascade-only version of the model. It is important to realize that the behaviour of these two models are identical; the change only affects the computational efficiency.

This report first describes the philosophy of this cochlea model and the two different implementation techniques are described and illustrated. The model of the cochlea and its implementation as a cascade filter bank are then described. Finally the Automatic Gain Control (AGC) used in this model is explored.

2 - The Ear Model

2.1 - Philosophy

The cochlear model to be described here was first sketched by Richard Lyon [Lyon82 and Lyon85] based on work described elsewhere [Schroeder73 and Zweig76]. These papers should be consulted for more information about the theory of the cochlea on which this model is based. The purpose of this notebook is to describe a number of the details that have not been previously published. The information in this report should be sufficient to allow the reader to implement their own version of this cochlear model.

This model describes the propagation of sound in the inner ear and the conversion of the acoustical energy into neural representations. We do not describe the effect on the sound as it enters the ear and travels down the ear canal. This is commonly thought to consist of a simple linear filter of the sound and thus to be relatively unimportant for speech recognition. The middle ear couples the energy that is traveling in the ear canal through the ear drum and a series of bones into the fluid filled chambers of the cochlea. The middle ear is also thought to provide some automatic gain control (AGC) via the stapedial reflex, but we have chosen not to model this mechanism [Pickels82, p21].

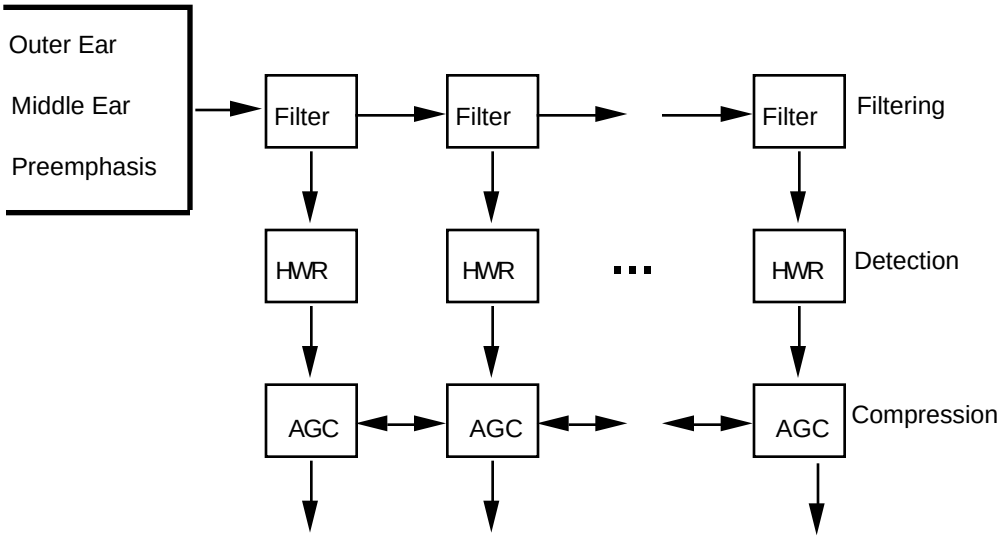
This model does not try to literally describe each structure in the cochlea but instead models the cochlea as a "black box." Sound entering the cochlea via the oval window is converted into nerve firings that then travel up the auditory nerve into the brain. The output of this model is a vector proportional to the firing rate of neurons at each point in the cochlea.

While many of the structures in this model (such as half-wave rectification or automatic gain control) are present in the cochlea, we have implemented our model differently to make the computations easier. Hopefully the results or our model are similar to the real cochlea. A more accurate description of the cochlea would model the propagation of pressure waves in two dimensional or three dimensional ducts, replace the AGC described later in this report with structures similar to the

outer hair cells and finally would assume a continuous-time analog implementation. This type of model is described in other works [Lyon88a and Lyon88b].

■ 2.2 - Overview

The cochlear model described by Lyon [Lyon82] combines a series of filters that model the traveling pressure waves with Half Wave Rectifiers (HWR) to detect the energy in the signal and several stages of Automatic Gain Control (AGC). This structure is shown in the figure below.



Sound that enters the outer and the middle ear is passed through the oval window into the cochlea. Once in the cochlear duct the the pressure wave propagates down the basilar membrane. The stiffness of the basilar membrane varies smoothly over its length and at any one point will resonate most strongly with a pressure wave of a particular frequency. At each stage of the cochlea some of this motion is sensed by the hair cells. It is these cells that convert the mechanical signals which in turn cause stimulation of the neurons which commicate with higher levels in the brain.

An important characteristic of the cochlea is that energy in the acoustitic wave is separated by frequency and each point in the cochlea will respond best to one frequency. In a sense the cochlea maps the frequency content of the signal into the spatial domain. The cochlea near its base (where the sound enters) is most sensitive to high frequency

sounds and as the wave travels down the cochlea lower and lower frequencies are sensed.

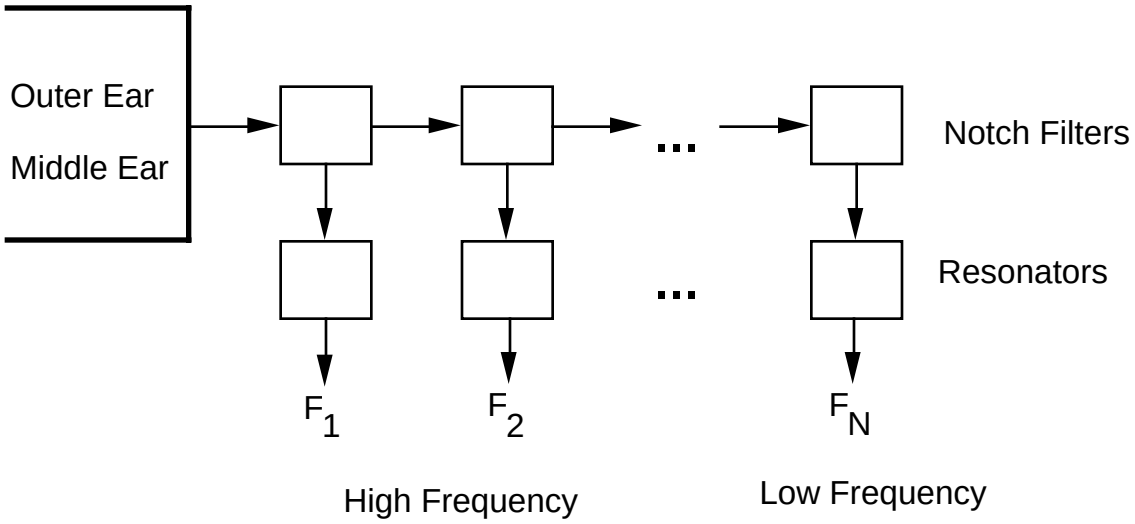
This notebook describes the filters and the AGC's shown in the picture above. The outputs of the AGC stage are positive signals that indicate the firing rate of the neurons leading to the brain. This notebook will first describe the characteristics of a filter stage. A number of *Mathematica* structures used to design digital filters are described in an Appendix. We will then describe the construction of each stage. Finally we will conclude with a discussion of the AGC used in this model.

Two versions of the filtering stage are described here. In the original formulation each filter stage was implemented as two separate filters. The sound passes through a cascade of filters to model the propagation down the cochlea and a parallel set of filters model the basilar membrane motion. For this reason the original model is known as the cascade-parallel formulation. Later it was discovered that the poles and zeros of the original two filters per stage could be rearranged and each stage implemented as a single second order filter. This is called the cascade-only filter bank.

■ 2.3 - The Cascade-Parallel Model

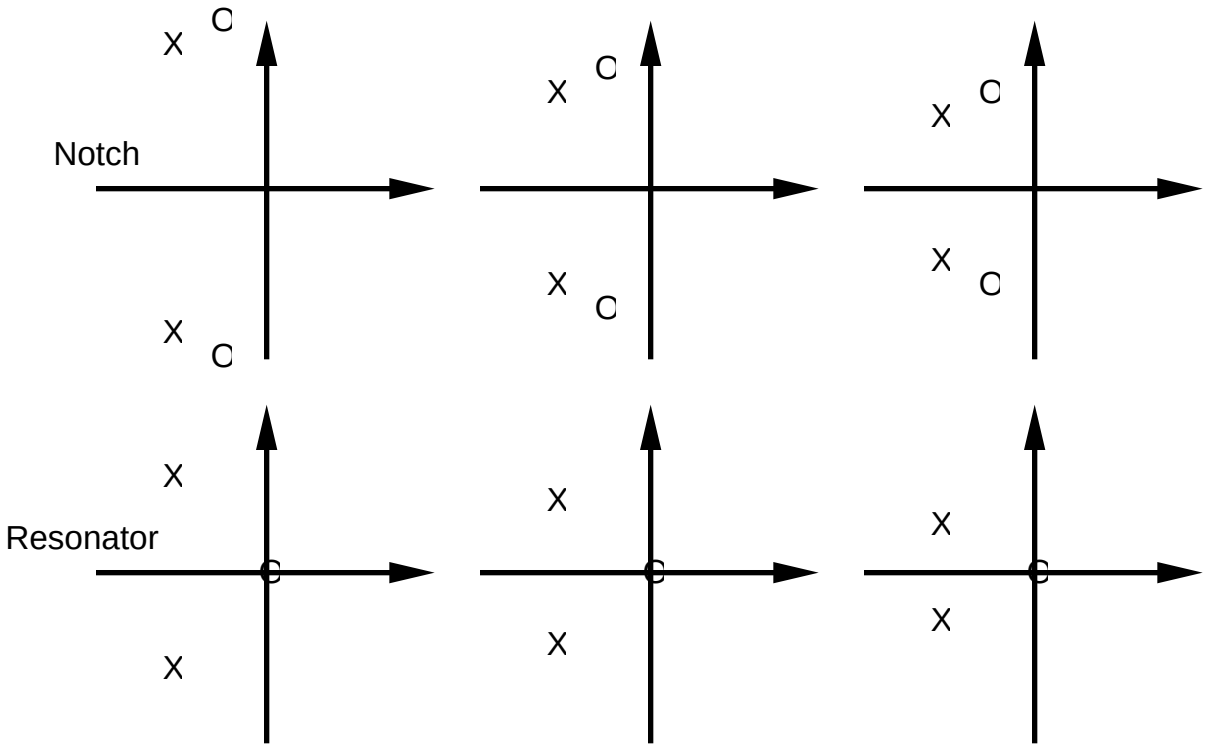
The cochlea is best modeled using a continuous differential equation. This is very difficult to implement on a digital computer so instead we split the cochlea into a large number of discrete sections. We can then model each section with a simple linear transfer function. For small enough sections the errors involved will be negligible and our digital implementation will be accurate. In this implementation of the model we have used approximately 80 stages.

The cochlea model described by Lyon [Lyon82] combines a series of notch filters that model the traveling pressure waves with resonators to model the conversion of pressure waves into basilar membrane motion or velocity. The combination of notch filters and resonators used to model the cochlea are shown below.

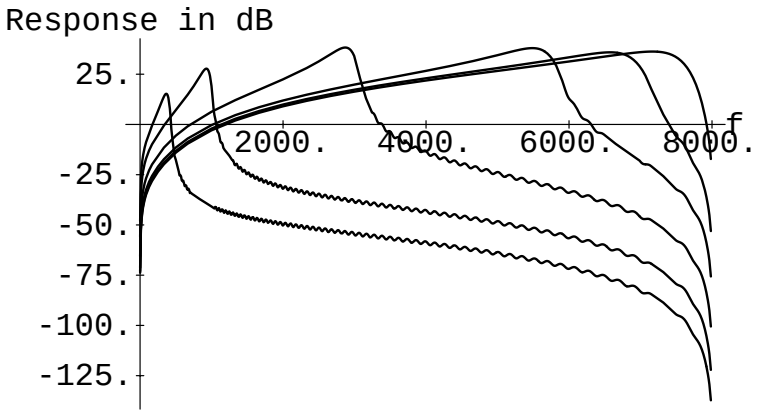


At each point in the cochlea the acoustic wave is filtered by a notch filter. Each notch filter operates at successfully lower frequencies so the net effect is to gradually low pass filter the acoustic energy. An additional resonator (or bandpass filter) picks out a small range of the traveling energy and models the conversion into basilar membrane motion. It is this motion of the basilar membrane that is detected by the inner hair cells. In the work to follow a combination of a notch and a resonator is called a stage.

The filters used in this model have pole-zero plots as shown below. Sound travels down the line of notch filters, at each stage getting filtered at lower and lower frequencies. At each stage a resonator (or bandpass filter) senses the output. The following plots show the locations of the poles and zeros in the s-plane for several of the notch and resonator filters.



The frequency domain responses of the 1, 4, 10, 30, 60 and 75'th stages are shown below. The curves with their peaks at the right represent the response of the low numbered stages while later stages respond best to low frequencies. These curves include the effects of a simple model of the outer and middle ears.



2.4 - Parameters of Cascade-Parallel Model

We will first describe the parameters of the original (cascade - parallel) cochlea model. To make the description simpler we will assume there is no sampling and express the filters in the Laplace domain. The bandwidth of each ear filter is a function of its center frequency. At high frequencies the bandwidth is approximately equal to the center frequency divided by a constant (**EarQ**). At lower frequncies the bandwidth approaches a constant given by **EarBreakFreq/EarQ**.

```
EarBandwidth[cf_] := Sqrt[cf^2 + EarBreakFreq^2]/EarQ
```

where

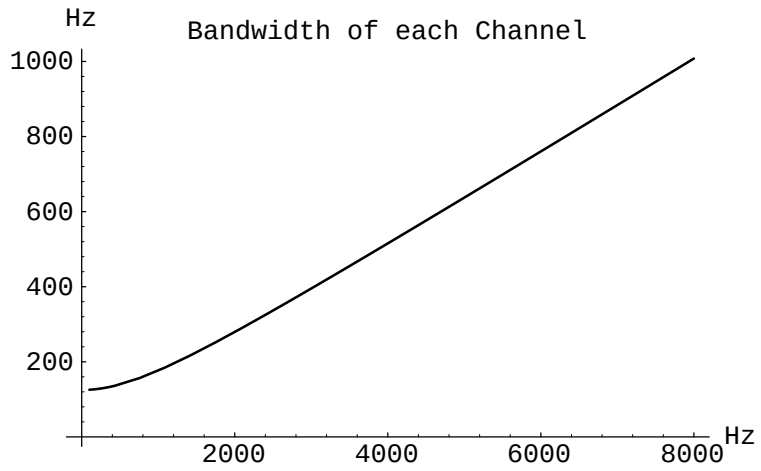
```
EarBreakFreq = 1000.0;
```

```
EarQ = 8;
```

```
Limit[EarBandwidth[f],f->0]
```

```
125.
```

The bandwidth of the ear filter as a function of its center frequency looks like this:



Successive ear filter stages are overlapped by a fraction of their bandwidth. This parameter is arbitrary but smaller numbers lead to more computations. We currently overlap 4 stages within the bandpass region of any one filter.

```
EarStepFactor = .25;
```

We model each section of the cochlea with a second order section. We use a pole at the center frequency to give a slight peak to the filter's response at its center frequency. A zero is placed in the response slightly above the center frequency to provide the band rejection.

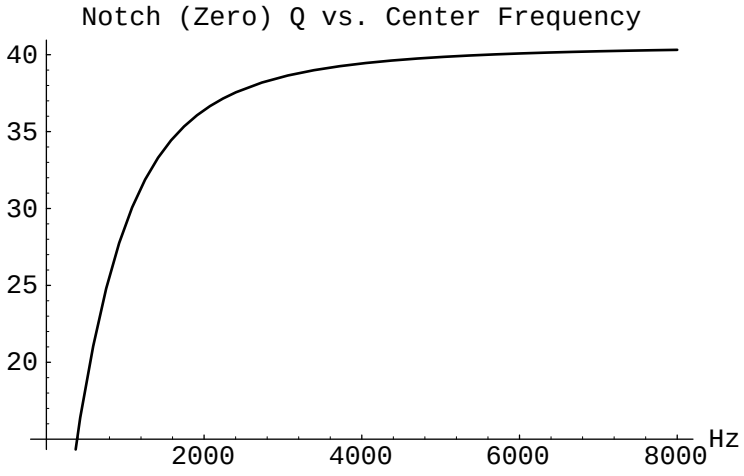
The center frequency and quality factor (q) of the zeros are given by:

```
OriginalZeroCF[cf_] := cf +
                       EarBandwidth[cf]*EarStepFactor*
                                        OriginalEarZeroOffset

OriginalEarZeroOffset = 0.5;
```

Note, that **OriginalEarZeroOffset** is a factor that determines how far the zero is offset from the center frequency of the filter stage. The offset is a function of the center frequency change (**EarBandwidth[cf] * EarStepFactor**) and models the fact that the response of the filter is slightly lower above the notch than it is below.

```
OriginalZeroQ[cf_] := OriginalEarSharpness*
                                  OriginalZeroCF[cf]/
                                  EarBandwidth[cf]

OriginalEarSharpness = 5.0;
```

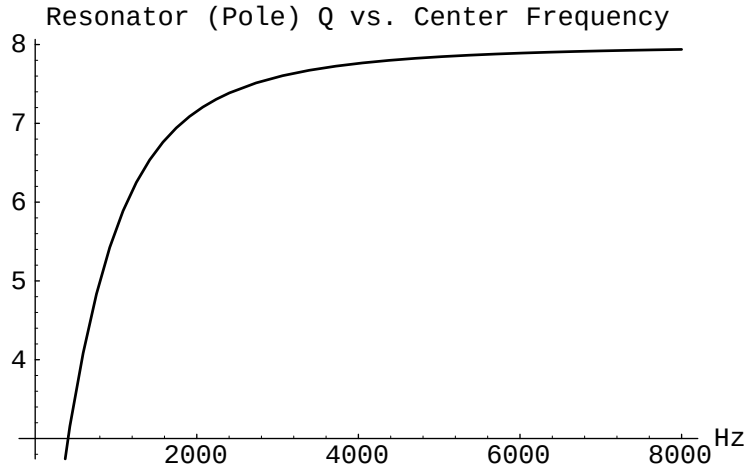
OriginalEarSharpness is a parameter that in effect sets how much sharper the notch (zero) is than the resonator (pole.)



The poles (resonators) in the response are centered at the center frequency of each stage but have a lower bandwidth or **q**.

```
OriginalPoleCF[cf_] := cf
```

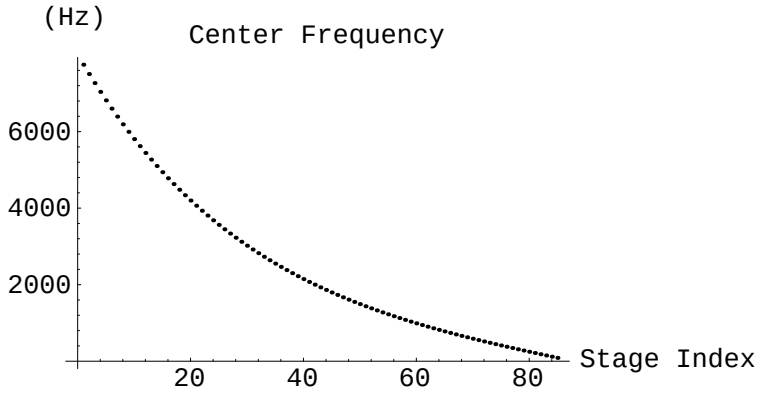
```
OriginalPoleQ[cf_] := cf / EarBandwidth[cf]
```



Each section or stage of the discrete cochlea is numbered starting at the base. Those stages that are closest to the base (low indices) are sensitive to the highest frequencies and stages with higher indices respond to lower frequencies. The center frequency of each discrete ear filter is defined by the following recursive relationship: [Note, **Block** is a *Mathematica* function that allocates storage for a list of variables, just **cf** in this case, and then executes the statements that follow.]

```
OriginalEarChannelCF[index_] :=
    OriginalEarChannelCF[index] =
    If [ index <= 0,
            8000,
            Block[{cf},
                cf = OriginalEarChannelCF[index-1];
                cf - EarStepFactor EarBandwidth[cf]]]
```

Starting at an arbitrary high frequency (8000 Hz for **index = 0**) we step down in frequency by **EarStepFactor** times the bandwidth of the filter at the previous frequency.



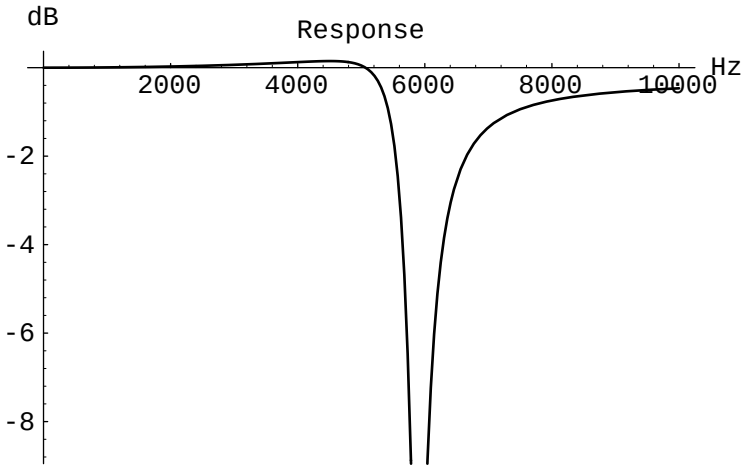
■ 2.5 - Cascade-Parallel Filter Design

We can now design the actual filters used in the original ear model. Each notch filter is a combination of a pair of poles (with low **q**) and a pair of zeros (with a higher **q** and higher center frequency.) To simplify the explanation we will illustrate the model by showing the continuous time filters. [Note the function **ContinuousSecondOrderFilter** and other basic filter design functions are defined in the Appendix to this report.]

```
OriginalNotch[index_] :=
    Block[{cf,zerof,polef},
        cf = OriginalEarChannelCF[index];
        zerof = ContinuousSecondOrderFilter[
                                    OriginalZeroCF[cf],
                                    OriginalZeroQ[cf]];
        polef = ContinuousSecondOrderFilter[
                                    OriginalPoleCF[cf],
                                    OriginalPoleQ[cf]];
        N[ContinuousAdjustGain[zerof/polef,0]]]
```

For example this is the response to the 10'th notch filter.

```
onp = ContinuousFreqResponse[OriginalNotch[10],10000];
```

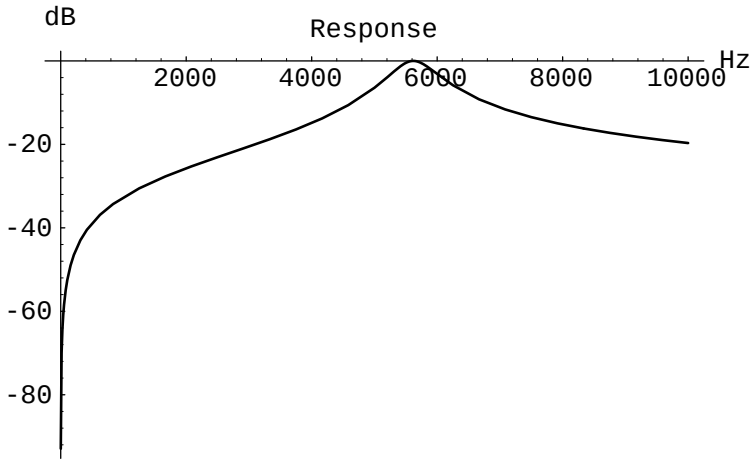


A resonator converts pressure in the cochlea into basilar membrane motion. We represent the resonator as a combination of a zero at DC (implemented as a differentiator) and a pair of poles. The poles (or bandpass filter) are slightly below the center frequency of the stage so that they emphasize the frequencies near the cutoff frequency. To make the computations easier the poles of this stage are actually placed at the same location as the poles in the resonator of the following stage. The reasons for this will be described in the next section. In the latest model we combine the poles in adjacent stages to reduce the computational effort.

```
OriginalResonator[index_] :=
    Block[{cfplus1,zerof,polef},
        cfplus1 = OriginalEarChannelCF[index+1];
        zerof = s;
        polef = ContinuousSecondOrderFilter[
                                    OriginalPoleCF[cfplus1],
                                    OriginalPoleQ[cfplus1]];
        N[ContinuousAdjustGain[zerof/polef,cfplus1]]]
```

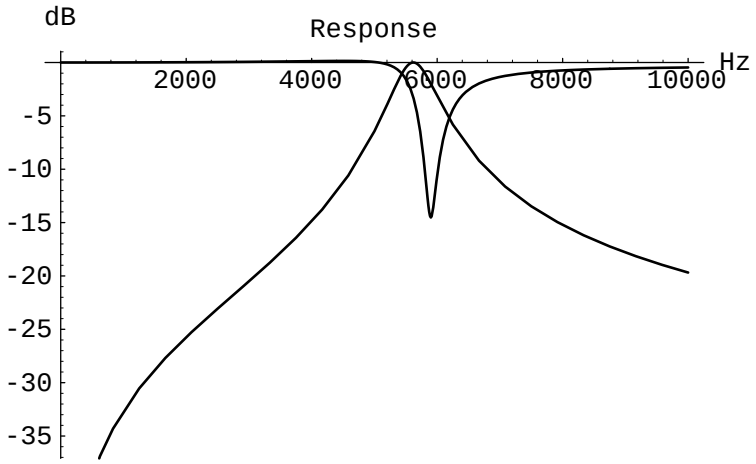
This is the response of the 10'th resonator. The sharp cutoff at DC is caused by the differentiator.

```
orp = ContinuousFreqResponse[OriginalResonator[10],
            10000];
```



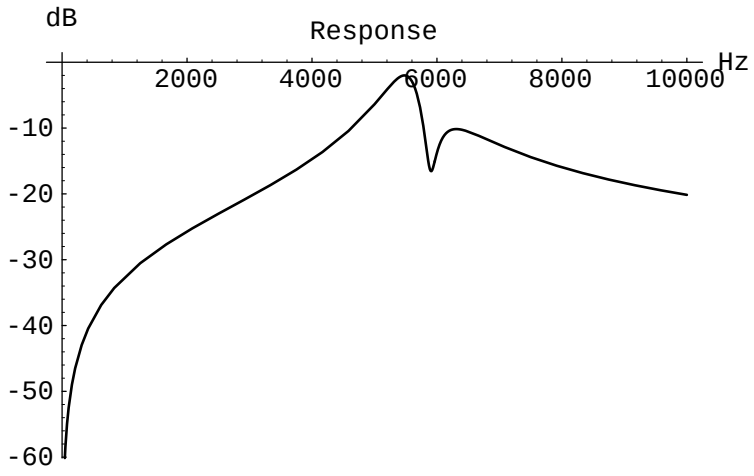
We can overlay these two last plots to show that the resonator peaks at the lower edge of the notch filter.

```
Show[onp,orp];
```



Finally the response of the combined notch and the resonance filters is shown below (for the 10'th filter stage.)

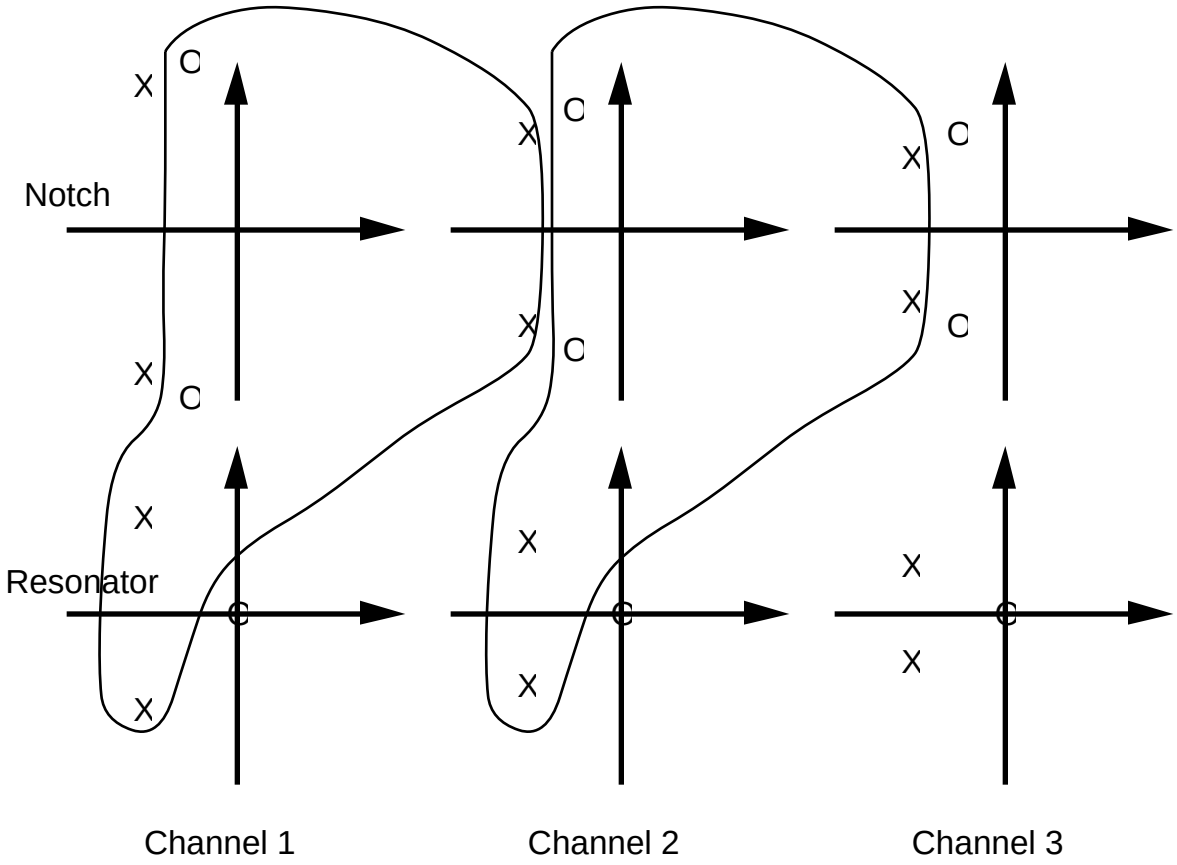
```
ContinuousFreqResponse[OriginalResonator[10] *
                       OriginalNotch[10], 10000];
```



■ 2.6 - The Cascade-Only Model

The latest version of the ear model combines the notch and resonator filters of each stage into a single filter. This gives a cascade-only representation of the ear model instead of the cascade-parallel version described above. Rearranging the filters does not change the the net result of the filtering; its only effect is to reduce the computational effort needed to model the cochlea.

The pole-zero plots of the original model are shown again below. We have circled the poles and zeros that are combined to form each stage of the cascade-only filter bank. First note that every resonator includes a zero at DC. Since all the filters shown here are linear we can move the differentiator into an initial preemphasis stage.



A more important optimization is possible if the poles in each notch filter are combined with the poles in the previous resonator filter. This optimization is shown above. The zeros from each notch filter and the poles from a resonator and the next notch filter are rearranged into a single filter. This is possible because the locations of the poles in the resonator filters were chosen to be at the same location as the poles in the succeeding notch filter.

The new cascade-only ear filter has an initial stage that combines the effects of the outer and middle ears, the differentiator (zero at DC) that was originally part of the resonator and a pair of poles from what was the first stage. Each succeeding stage is a pair of poles and a pair of zeros and the output of each stage is input to not only the next stage in the cascade but also the detection blocks (HWR).

The cascade-only filter structure has slightly different parameters. We repeat the following equations with several small changes. Like before the center frequency of each stage is described by the location of the poles. This means the center frequency of the associated zero is an extra

channel higher than the pole frequency therefore **EarZeroOffset** changes from 0.5 to 1.5.

```
CascadeZeroCF[cf_] := cf +
                      EarBandwidth[cf]*EarStepFactor*
                                   EarZeroOffset

EarZeroOffset = 1.5;

CascadeZeroQ[cf_] := EarSharpness*CascadeZeroCF[cf]/
                                EarBandwidth[cf]

EarSharpness = 5.0;

CascadePoleCF[cf_] := cf

CascadePoleQ[cf_] := cf / EarBandwidth[cf]
```

We define a new version of **EarChannelCF** that is based on a sampled version of the ear model. Now the maximum channel frequency is a function of the Nyquist rate (**fs**/2) and the location of the zeros in the first stage of filtering.

```
MaximumEarCF[fs_] :=
    Block[{topf},
             topf = fs / 2.0;
             topf - (CascadeZeroCF[topf]-topf) +
                 EarBandwidth[topf] EarStepFactor]
```

Starting at the **MaximumEarCF** (**index = 0**) the center frequency of each channel decreases by **EarStepFactor** of the bandwidth at the previous stage. [Note that since the **EarChannelCF** function is used everywhere in the ear model we have used a *Mathematica* technique known as dynamic programming (or caching) to remember previously calculated values in an array. Later, if we want to find the center frequency of an already computed filter section, we only need to look in the array.]

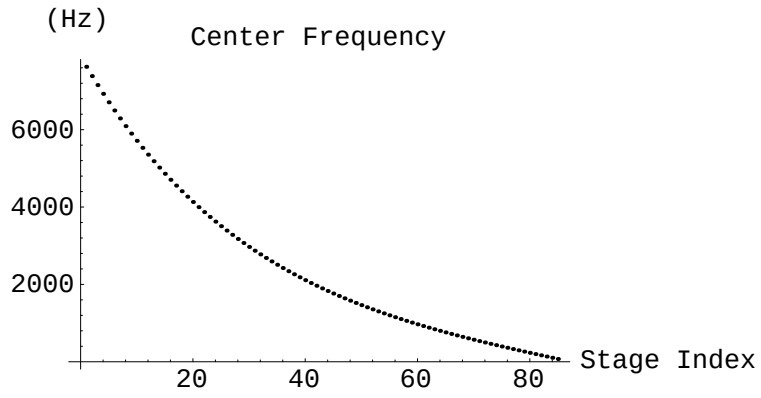
```
EarChannelCF[index_, fs_] :=
    EarChannelCF[index,fs] =
        If [ index <= 0,
                 MaximumEarCF[fs],
                 Block[{cf},
                     cf = EarChannelCF[index-1, fs];
                     cf - EarStepFactor EarBandwidth[cf]]]
```

Note that this expression is a function of the sampling frequency (**fs**). To prevent aliasing it is important that the zero in the first stage be below the Nyquist rate.

```
CascadeZeroCF[EarChannelCF[1,16000]]
```

```
7986.52
```

The center frequency of each stage in the cascade-only filter bank is shown below.



☐ Test Code

The following is some test code to show that the *Mathematica* definitions above are working correctly.

```
samplingfrequency=16000
```

```
16000
```

```
cf=EarChannelCF[1,samplingfrequency]
```

```
7625.99
```

```
EarBandwidth[cf]
```

```
961.409
```

```
CascadeZeroCF[cf]
```

```
7986.52
```

```
CascadeZeroQ[cf]
```

```
41.5355
```

CascadePoleCF[cf]

7625.99

CascadePoleQ[cf]

7.93209

3 - The Cochlea Filter Bank

This section describes the cascade of filters that define the ear model. As described above there is an initial preemphasis stage followed by the cascade of ear filter stages. This section will define the preemphasis stage, each of the cochlea stages and then show their combined frequency response. In the remainder of this report we will describe the filters in the z-domain. Again, see the Appendix of this report for the definitions of the filter design functions used here.

3.1 - Preemphasis Stage

Our model of the ear uses a simple preemphasis filter to roughly model the effects of the outer and middle ear. This is followed by a differentiator and a high frequency compensator that are common to all of the stages. Finally, there is a string of second order filters to model each section of the cochlea.

The outer and middle ear add a slight high pass response to the system. This initial high pass filter also helps to normalize (or whiten) the input to the inner ear and makes it easier to display the resulting output. The outer and middle ears are modelled here with a high pass filter with a corner frequency of 300 Hz.

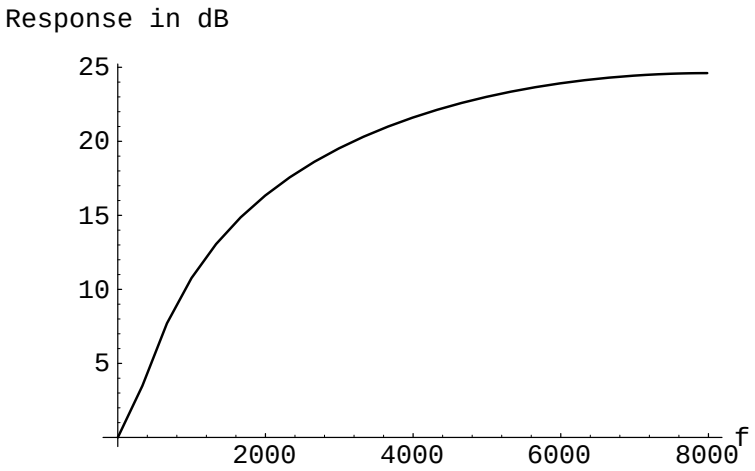
```
EarPremphCorner = 300.0;
```

```
OuterMiddleEarFilter[fs_] :=
    MakeFilter[FirstOrderFromCorner[EarPremphCorner,fs],
                MakePoly[{1}],
                fs, 0, 1.0]
```

```
OuterMiddleEarFilter[16000]//N
```

```
8.99808 (1. - 0.888865 z)
```

```
FreqResponse[OuterMiddleEarFilter[16000],16000];
```



Each stage of the cascade-parallel filter bank described in [Lyon82] uses a differentiator to convert pressure waves into basilar membrane motion. Each differentiator is adjusted so that at the center frequency of the stage the differentiator has unity gain. In this model we have "factored" this differentiator out of each stage and it appears just once before the ear cascade (a term of the form **1-z**). Later when we define the stages of the cascade we will adjust the gain of each stage so that the differeniator has unity gain at its center frequency. In addition we combine the differentiator with a zero at the Nyquist rate (**1+z**) to compensate for the close spacing of the poles near z=-1 for high frequencies. This combined filter looks like:

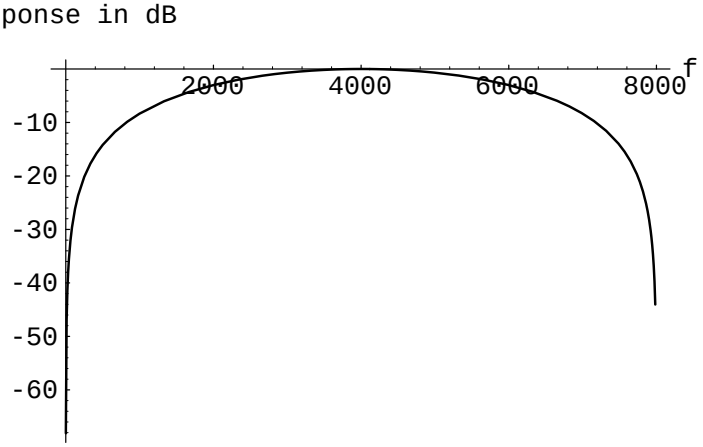
```
Compensator[fs_] := MakeFilter[MakePoly[{1,0,-1}],
                               MakePoly[{1}],
                               fs,
                               fs/4,
                               1.0]
```

```
Compensator[16000]
```

```
          2
0.5 (1 - z )
```

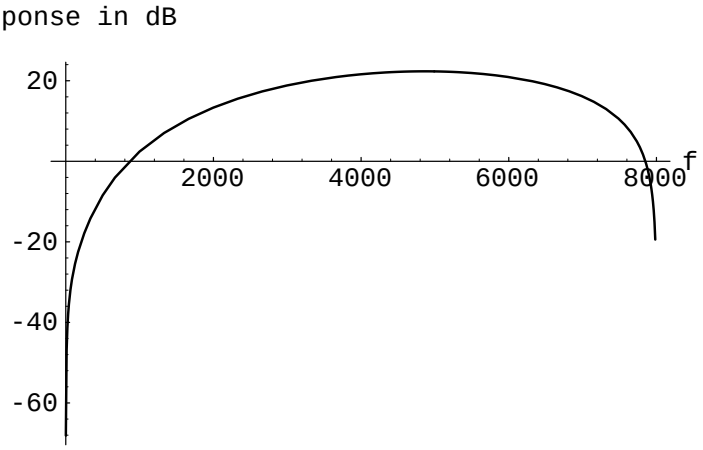
In the frequency domain the response looks like:

```
FreqResponse[Compensator[16000],16000];
```



Combining the outer/middle ear filter and the compensator the total response is:

```
FreqResponse[Compensator[16000]
             OuterMiddleEarFilter[16000], 16000];
```



Finally we can combine these initial stages with the first two poles of the ear filters. The first stage of the cascade is a combination of the outer/middle ear filter, the compensator and a pole pair.

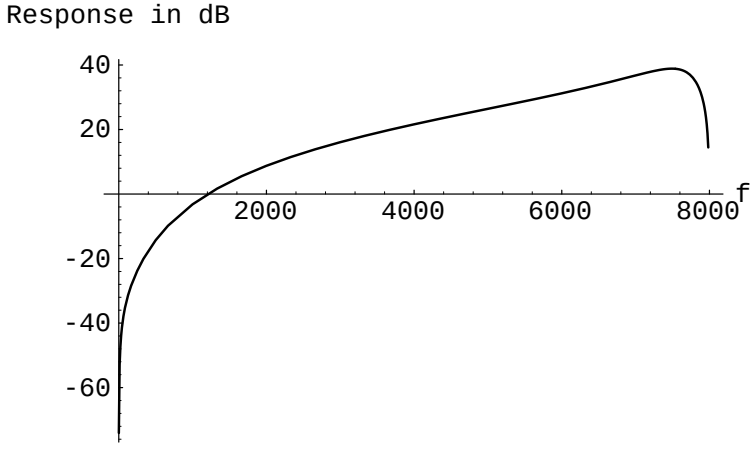
```
EarFrontFilter[fs_] :=
     (OuterMiddleEarFilter[fs]
      Compensator[fs]
      MakeFilter[MakePoly[{1}],
                 SecondOrderFromCenterQ[
                         CascadePoleCF[MaximumEarCF[fs]],
                         CascadePoleQ[MaximumEarCF[fs]],
                         fs],
                 fs,
                 fs/4,
                 1.0])
```

The combined initial filters and first two poles of the ear filter have the response shown below.

```
EarFrontFilter[16000]//N
```

$$\frac{(-1.45178 + 7.39389\,I)\,(1. - 0.888865\,z)\,(1. - 1.\,z)^2}{0.677314 + 1.64344\,z + z^2}$$

```
FreqResponse[EarFrontFilter[16000],16000];
```



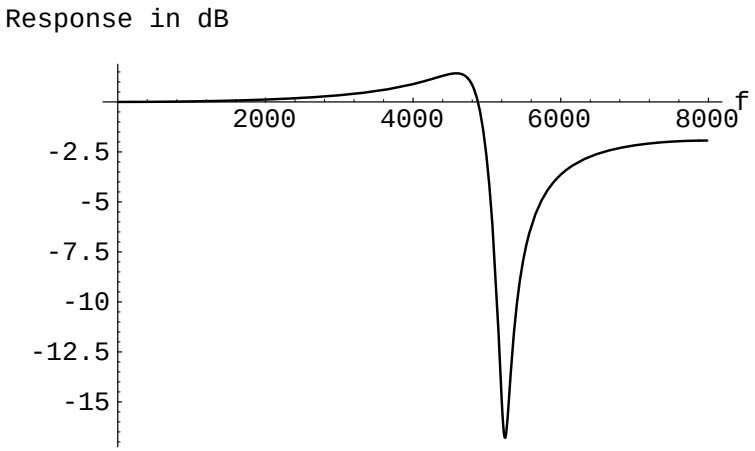
■ 3.2 - The Stage Filters

□ Definitions

The stages of the filter bank that simulate each section of the cochlea are each a combination of two poles and two zeros. Recall that a pair of poles with a broad response is combined with a pair of zeros at a slightly higher frequency.

```
EarStageFilter[cf_, fs_, dcgain_] :=
    MakeFilter[
        SecondOrderFromCenterQ[CascadeZeroCF[cf],
                               CascadeZeroQ[cf],
                               fs],
        SecondOrderFromCenterQ[CascadePoleCF[cf],
                               CascadePoleQ[cf],
                               fs],
        fs,
        0.0,
        dcgain]
```

This filter has the following response (shown here for the filter section centered at 5000Hz.) Note that there is a slight peak in the response before the notch due to the poles.



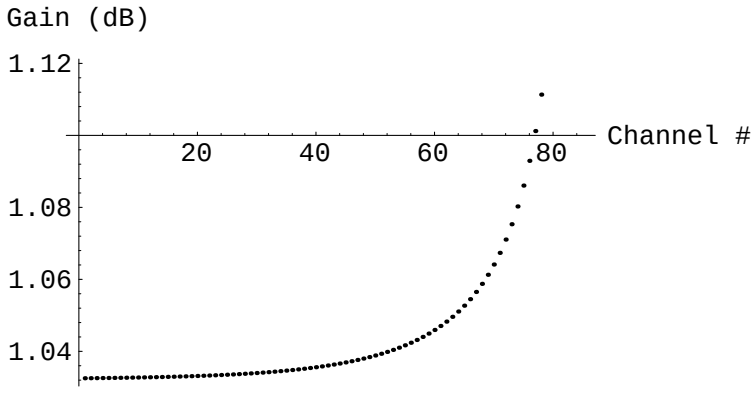
We set the gain of each stage in the ear filter to compensate for the gain provided by the differentiator in the preemphasis stage. The gain of an ideal differentiator is proportional to frequency. Preceeding all channels of the ear filter with a single differentiator will cause the lower frequency channels (high index) to have a much lower output then the preceeding stages. While within a single channel we still want to add a term that is proportional to frequency we want to adjust the differentiator at each stage so it has unity gain at the center frequency.

To compensate for the single differentiator we adjust the gain of each stage of the cascade filter by dividing its response by its center frequency (**cf**). But since all of the filter stages are in series we must

also remove the effect of the gain from the previous section. Thus the gain at each stage is given by:

```
EarFilterGain[i_,fs_] := EarChannelCF[i-1,fs]/
                         EarChannelCF[i,fs]
```

When the differentiator is combined with each stage of the ear filter (by multiplying their responses) the net effect of this gain term is to normalize the differentiator so that it has unity gain. The gain at each stage is shown below.



To produce the final filter function we combine the response for **EarFrontFilter** (section 0) with those defined by the **EarStageFilter** function.

Again we use what *Mathematica* refers to as dynamic programming (caching) to save the resulting filters. It is important to remember that this function implicitly depends on the values of a number of constants (for example **EarSharpness** and **EarQ**) and if any of these parameters are changed then the *Mathematica* function **Clear[EarFilter]** must be used to invalidate the cache.

```
Clear[EarFilter]
```

```
EarFilter[i_,fs_] :=
    EarFilter[i,fs] =
    If [ i == 0,
        N[EarFrontFilter[fs]],
        Block[{cf, stagegain},
                cf = EarChannelCF[i,fs];
                stagegain = EarFilterGain[i,fs];
                If [ CascadePoleQ[cf] < 0.5 ,
                    0,
                    N[EarStageFilter[cf,fs,stagegain]]]]]
```

The filter calculated using the function **SecondOrderFromCenterQ** will have real roots when the value of **q** is less than 0.5. We arbitrarily stop the cascade structure when this happens. Solving the equation **CascadePoleQ[f] == .5** we find this is true for all stages with a center frequency less than 63hz.

```
Solve[CascadePoleQ[f] == .5,f]
```

```
{{f -> 62.6224}}
```

```
EarChannelCF[85,16000]
```

```
71.9869
```

```
EarChannelCF[86,16000]
```

```
40.656
```

□ Test Cases

These test cases are used to verify the consistency between this *Mathematica* model and the C and Lisp versions.

```
EpsilonFromTauFS[5/cf,fs]
```

```
 -1525.2/fs
E
```

```
FirstOrderFromTau[5/cf,fs]//N
```

```
      -1.
---------------- + z
       1525.2/fs
2.71828
```

```
FirstOrderFromCorner[cf/4,fs]//N
```

```
           1. z
1. - ------------------
              11978.9/fs
     2.71828
```

```
SecondOrderFromCenterQ[cf/4,2,fs]//N
```

```
                                      11598.5
                            2. z Cos[-------]
       -5989.44/fs    2              fs
2.71828            + z  - ------------------
                                    2994.72/fs
                            2.71828
```

```
EarPreCoeffs[16000]//N
```

```
EarPreCoeffs[16000.]
```

```
EarFrontCoeffs[16000]//N
```

```
EarFrontCoeffs[16000.]
```

```
EarCascadeCoeffs[cf,16000,1.032525595]//N
```

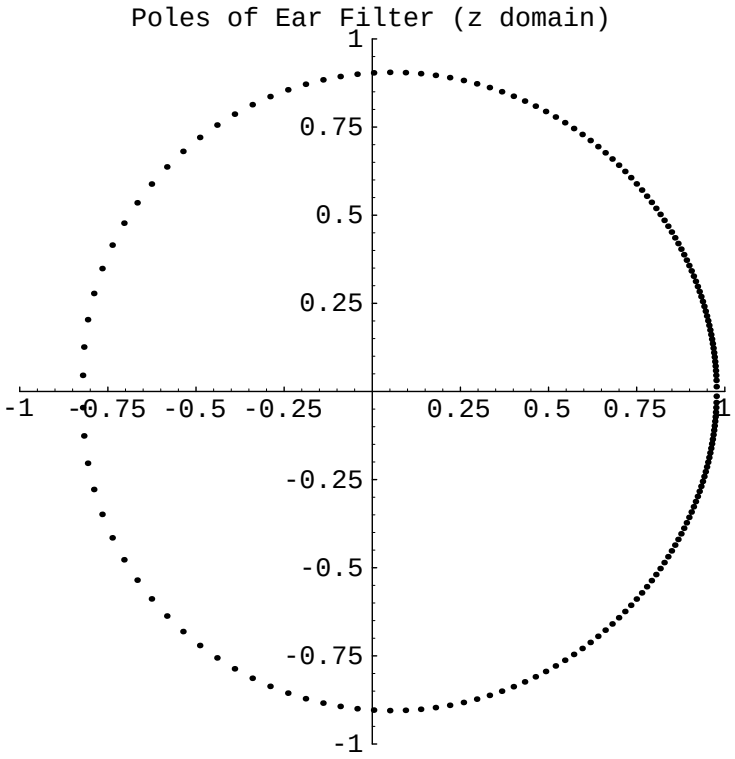
```
EarCascadeCoeffs[7625.99, 16000., 1.03253]
```

☐ Graphics

Using the following functions we can plot the poles and zeros of all of these filters.

```
PlotPoles[fs_] :=
    ListPlot[Map[{Re[#],Im[#]}&,
                 Flatten[Table[RationalPoles[
                                   EarFilter[i,16000]],
                               {i,0,85}]]],
             PlotRange->{{-1,1},{-1,1}},
             AspectRatio->1,
             PlotLabel->"Poles of Ear Filter (z domain)"]
```

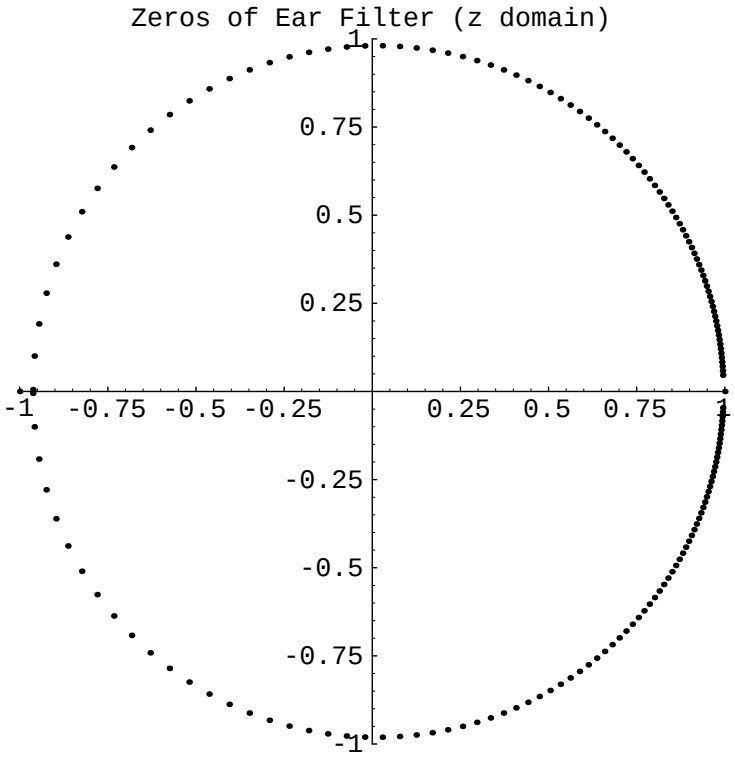
```
PlotPoles[16000];
```



This plot shows the poles of the filter bank going from high frequencies (near the base of the cochlea) around the unit circle to those of the low frequencies. Recall in the z-domain that frequency is mapped into a position on the unit circle. DC is at 1 on the real axis, the Nyquist frequency is at -1 and all other frequencies are on the unit circle between these two points. In addition the distance the pole (or zero) is from the unit circle is proportional to the quality factor of the filter. The poles in the cochlear filter have a lower q and thus are farther from the unit circle then the zeros shown below.

```
PlotZeros[fs_] :=
    ListPlot[Map[{Re[#],Im[#]}&,
                 Flatten[Table[RationalZeros[
                                      EarFilter[i,16000]],
                               {i,0,85}]]],
             PlotRange->{{-1,1},{-1,1}},
             AspectRatio->1,
             PlotLabel->"Zeros of Ear Filter (z domain)"]
```

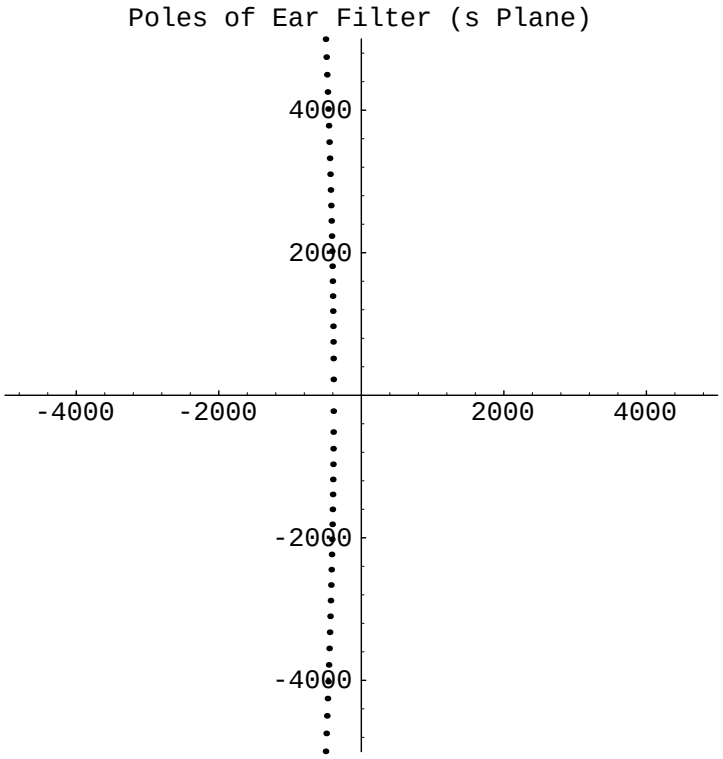
```
PlotZeros[16000];
```



In the s-plane the poles look like (using the substitution **z = exp[sT]**):

```
PlotSPoles[fs_,smax_] :=
    ListPlot[Map[{Re[#],Im[#]}&,
                 Log[Flatten[Table[RationalPoles[EarFilter[i,16000]
                                   {i,0,85}]]]] fs,
             PlotLabel->"Poles of Ear Filter (s Plane)",
             PlotRange->{{-smax,smax},{-smax,smax}},
             AspectRatio->1,
             Axes->{0,0}]
```

```
PlotSPoles[16000,5000];
```



■ 3.3 - The Cascade Filters

We can now define a cascade response. At each position in the cochlea the response is the product of the response of this section and all preceeding sections. In *Mathematica* this is written:

```
Clear[CascadeFilter]

CascadeFilter[index_, fs_] :=
    CascadeFilter[index, fs] =
        Product[EarFilter[i,fs],
                {i, 0, index}]
```

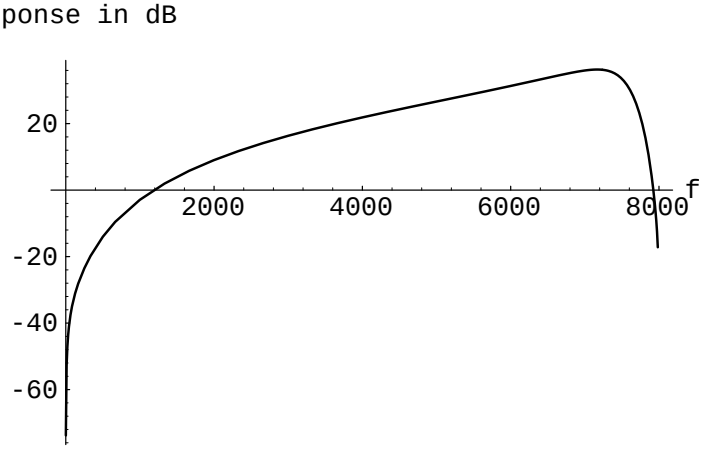
Since each term in this product is a second order function the cascade filter quickly becomes very high order. It is tempting to wrap **ExpandNumerator** and **ExpandDenominator** functions around the **Product** in this defintion but the numerical errors involved become large. For low index numbers we can display the results which look like:

CascadeFilter[1,16000]

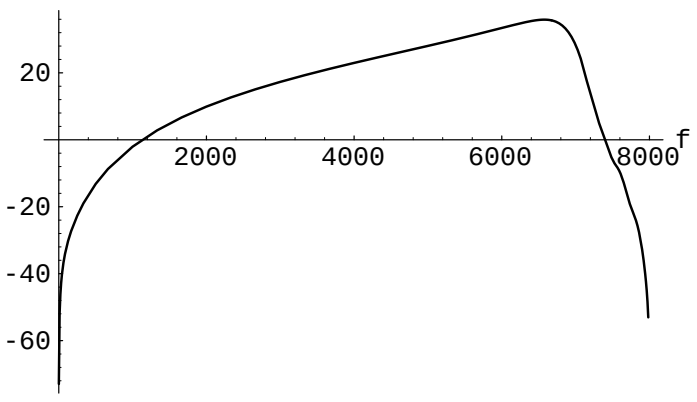
```
                                                                    2
((-1.29244 + 6.58239 I) (1. - 0.888865 z) (1. - 1. z ) /

                                  2
    (0.927271 + 1.92587 z + z )) /

                                 2
  ((0.685543 + 1.63665 z + z ) /

                                  2
    (0.677314 + 1.64344 z + z ))
```

The frequency responses at a number of different places in the cochlea are shown below. In each case the maximum of the response is near the center frequency of the last stage of the filter.

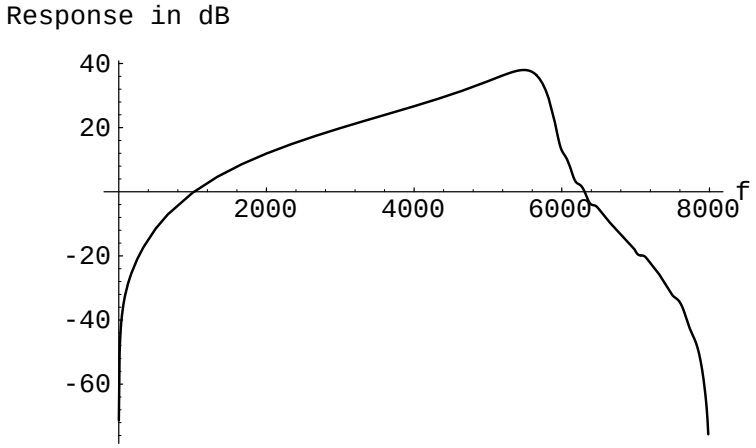
FreqResponse[CascadeFilter[1,16000],16000];



FreqResponse[CascadeFilter[4,16000],16000];

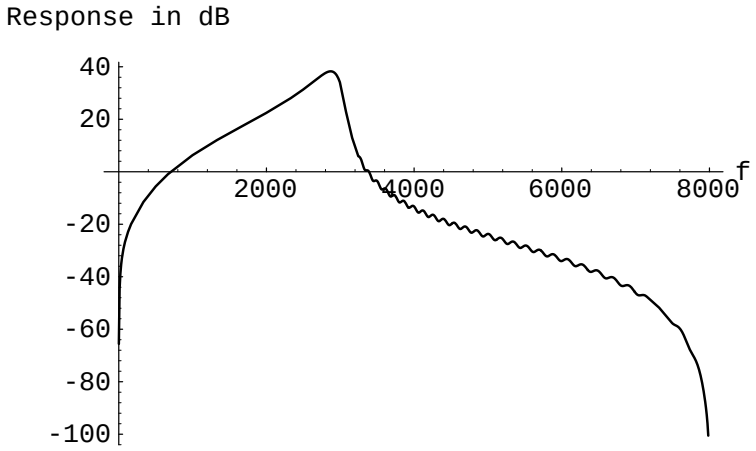


```
FreqResponse[CascadeFilter[10,16000],16000];
```

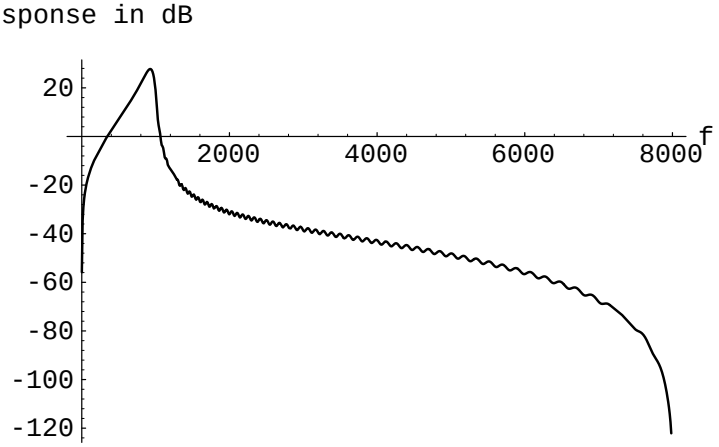


Note the response curves for later stages (higher indices) take a long time to compute. The 30'th stage, for example, is actually the product of 31 second order sections or a 62nd order filter.

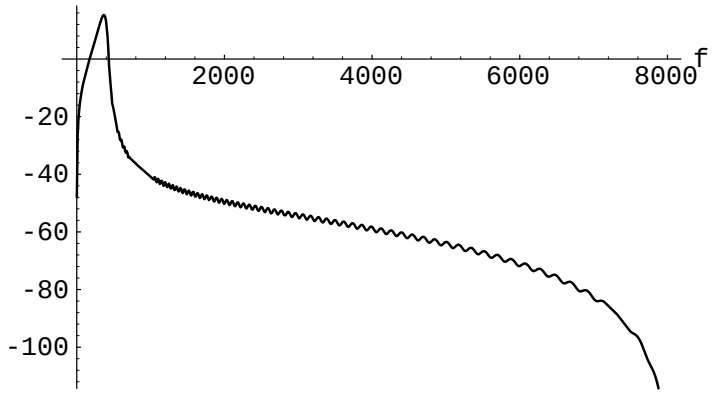
```
FreqResponse[CascadeFilter[30,16000],16000];
```



```
FreqResponse[CascadeFilter[60,16000],16000];
```

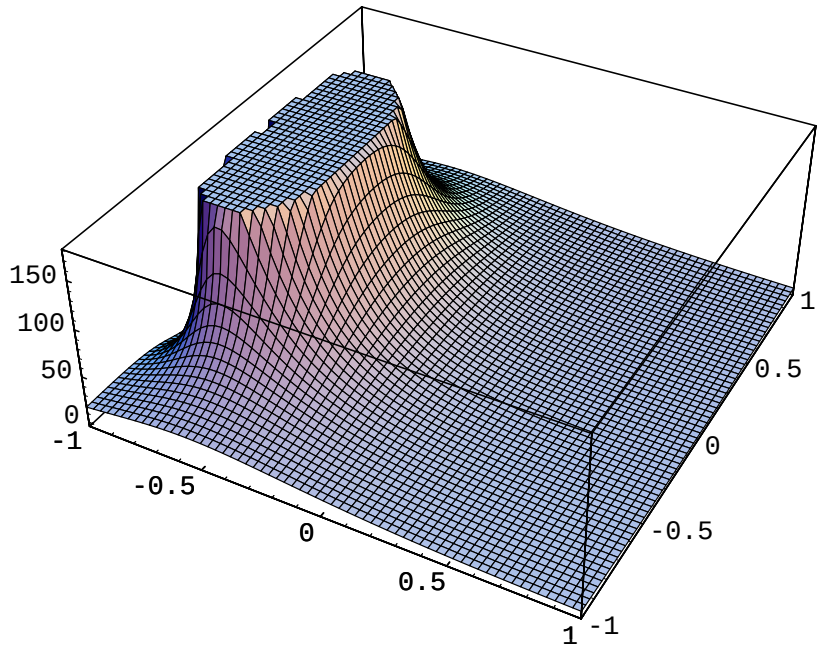



```
FreqResponse[CascadeFilter[75,16000],16000];
```



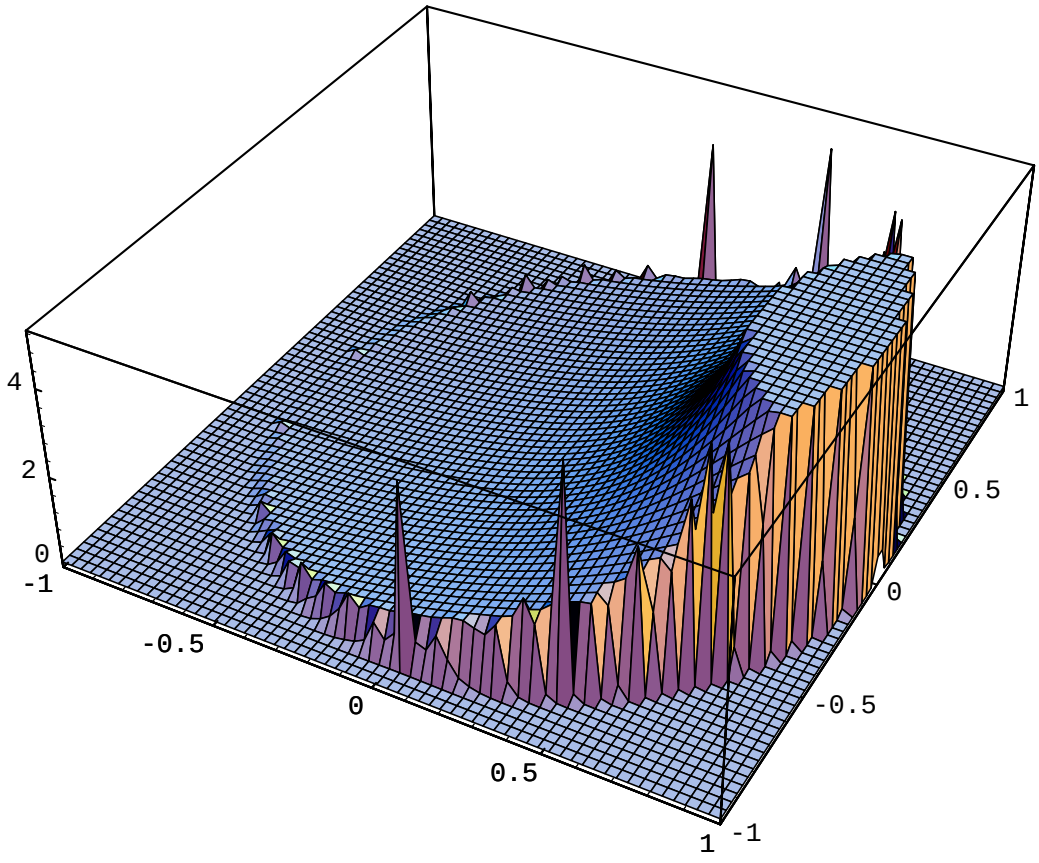
We can evaluate the response of the fifth cascaded stage on the z-plane using the expression below. Note that the first five stages includes the first 10 poles plus 10 zeros with a higher q. The zeros lead to the sharp dropoff.

```
Plot3D[Abs[FilterEval[CascadeFilter[5,16000],
                          x + I y]],
        {x,-1,1}, {y,-1,1},
        PlotPoints->64,
        Lighting->True];
```

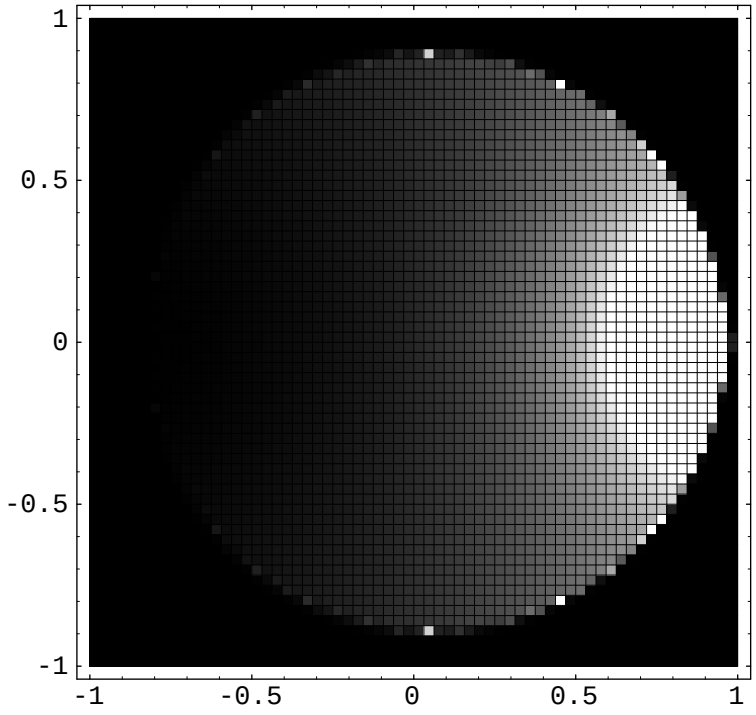


The next plot shows the total response for all 85 stages of the ear filter. The zeros with their high q lead to a low response outside the unit circle. This plot does not have enough resolution to show each of the poles. Thus the four spikes are places where *Mathematica* happened to sample especially close to a pole. The response is generally higher near DC because the end of the cochlea is especially sensitive to low frequencies.

```
Plot3D[Abs[FilterEval[CascadeFilter[85,16000],
                      x + I y]],
       {x,-1,1}, {y,-1,1},
       PlotPoints->64,
       Lighting->True];
```



```
DensityPlot[Abs[FilterEval[CascadeFilter[85,16000],
                           x + I y]],
        {x,-1,1}, {y,-1,1},
        PlotPoints->64];
```



4 - The Automatic Gain Control (AGC)

4.1 - The Simple AGC

The output of each filter stage is a bandpass representation of the original audio signal. Each of these bandpass signals is passed through a half-wave rectifier and then through four stages of AGC. The half-wave rectifier models the detection nonlinearity of the hair cells, providing a non-negative output that can be used to represent neural response. The AGC stages described below depend on their inputs being rectified.

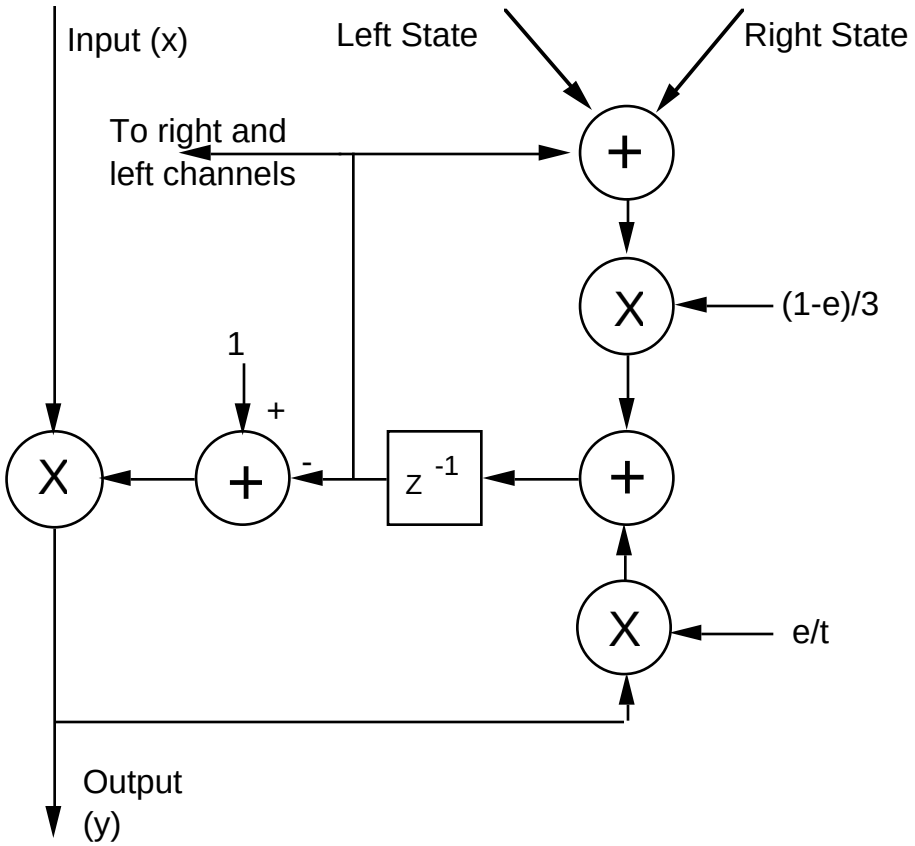
Each stage uses a different time constant to simulate the different adaptation times in the ear. The cochlear model as currently implemented uses four stages of AGC. Each AGC in turn reduces the level of the channel, but with shorter time constants. The **target** values and time constants that are currently used are:

First AGC:	.0032	640ms
Second AGC:	.0016	160ms
Third AGC:	.0008	40ms
Fourth AGC:	.0004	10ms

The significance of these figures will be explained shortly.

The half-wave rectification provides a crude energy measure in the signal. Each AGC stage is implemented as a variable gain which tries to keep the output of the AGC stage from exceeding a fixed level. In general the gain will be between zero and one. To model the masking effects of the ear, each stage of the AGC combines the bandpass outputs from the current channel plus its nearest neighbors. Since all channels are coupled one channel can affect all channels in the filter bank although the effect will decay exponentially with distance.

The resulting AGC is shown below. The letter **e** in the drawing is the same as the **epsilon** variable in the equations to follow. Likewise the letter **t** is substituted for **target**.



The purpose of the AGC is to attenuate the incoming signal so that on average it remains below the **target** value. The loop with a feedback gain of **(1-epsilon)/3** (the **state** equation below) represents a simple low pass filter with a time constant related to the **epsilon** parameter. A longer time constant means that the AGC takes longer to respond to the input. The division by 3 in this expression in effect takes the average of the left, right and current channels. The **targe**t parameter is used to scale the input to the loop filter, **y**. In the long run the **state** equation will track the value of the output of the AGC divided by the value of **target**.

Assuming the left, middle and right inputs are equal, the following equations define the response of the AGC.

```
Gain[i_] := 1 - State[i-1]

State[i_,y_,epsilon_,target_] :=
    State[i,y,epsilon,target] =
        epsilon*y/target + 3*State[i-1]*((1-epsilon)/3)
```

In the long term the output of the **state** loop (for constant input **y** in all channels) will be equal to the output value divided by the **target**. This is found using *Mathematica* by solving the following equation.

```
Solve[state == epsilon y/target + state(1 - epsilon),
        state]
```

```
                y
{{state -> ------}}
           target
```

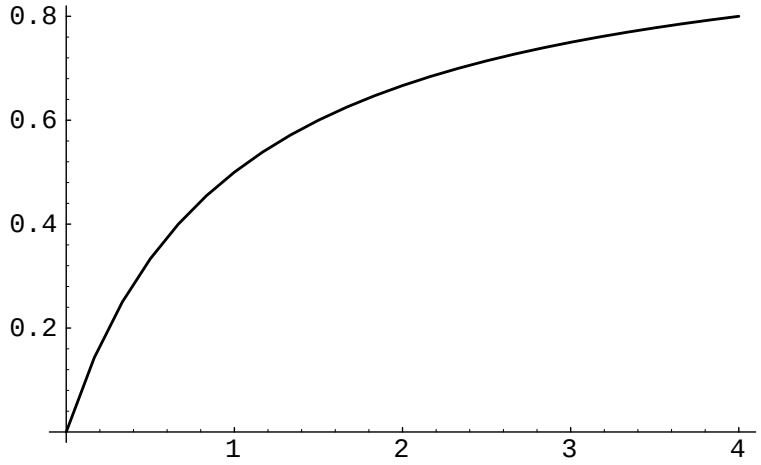
The output of the AGC is found by replacing the **state** in the gain equation above with the steady state solution. Thus the steady state output of the AGC is

```
Solve[y == x ( 1 - y / target),y]
```

```
         target x
{{y -> ----------}}
       target + x
```

If the **target** is equal to one then the following curve shows the output of the AGC versus input values. Note the limit of this function for large input values is equal to **target** (one in this case.)

```
AGCFun[x_,target_] = N[target x / (target + x)];

Plot[AGCFun[x,1],{x,0,4}];
```

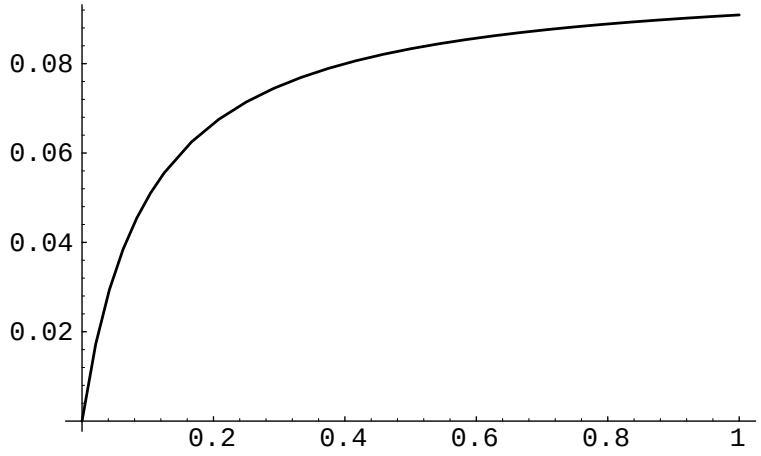



```
Limit[AGCFun[x,target],x->Infinity]
```

```
target
```

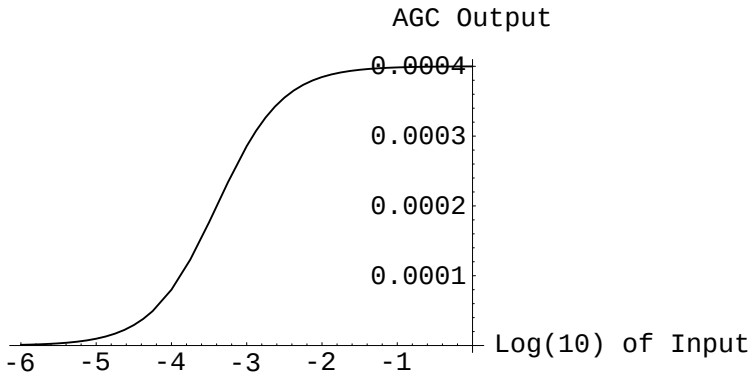
The output of the AGC scales. With a **target** of .1 the following transfer function is measured.

```
Plot[AGCFun[x,.1],{x,0,1}];
```



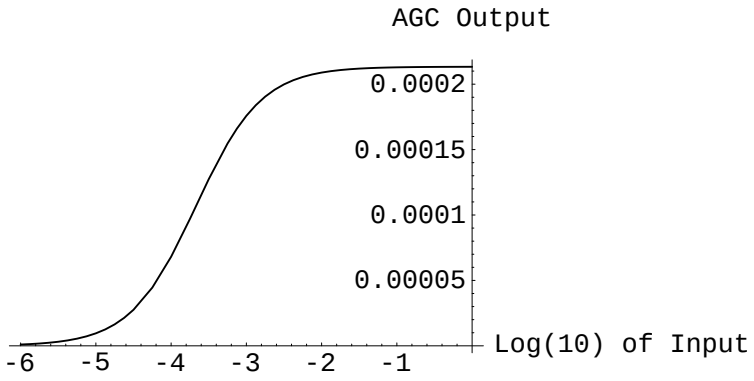
Plotting the resoponse on a logarithmic axis we see the output of the AGC approach the target value (.0004 this time.)

```
Plot[AGCFun[10^x,.0004],{x,-6,0},
        AxesLabel->{"Log(10) of Input","AGC Output"}];
```



Using actual targets described above for four cascaded stages, here is the overall response (again on a logarithmic scale):

```
Plot[AGCFun[AGCFun[AGCFun[AGCFun[10^x,.0032],
                                 .0016],
                          .0008],
                 .0004],
     {x,-6,0},
     AxesLabel->{"Log(10) of Input","AGC Output"}];
```



The impulse response of the AGC is uninteresting not only because the AGC is a nonlinear filter but also because all impulses are passed through unchanged. This is because the gain reduction occurs after the impulse has ended. Instead we will demonstrate the behaviour of the AGC using step inputs. The following function simulates the AGC response for step inputs of arbitrary height (**input**). The result of this function is a list of output values for the AGC at each step in time.

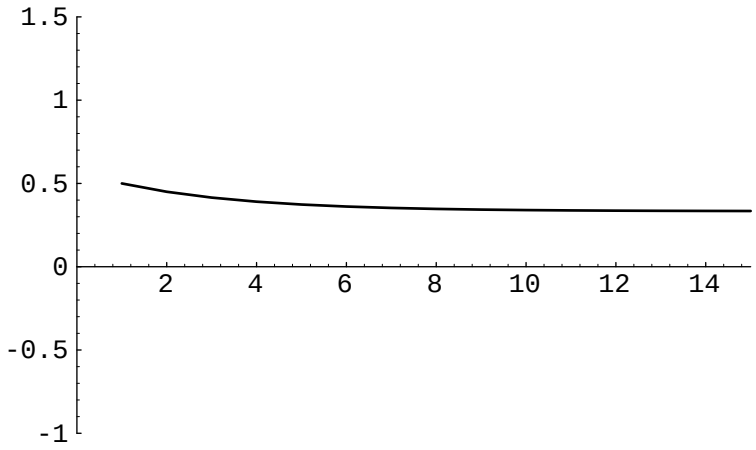
```
AGCResponse[input_, target_, epsilon_, count_] :=
    Block[{response,state,eps,eps1},
        state = 0;
        eps = epsilon/target;
        eps1 = 1 - epsilon;
        response = Table[0,{i,1,count}];
        Do [response[[i]] = N[(1-state)*input];
            state = response[[i]]*eps + state*eps1,
            {i,1,count+1}];
        response];
```

This leads to the following behavior for small inputs (we will use a **target** value of 1 and an **epsilon** of .2 for most illustrations .)

```
PlotResponse[responselist_] :=
    ListPlot[responselist,
                PlotRange->{{0,Length[responselist]},
                    {-1,1+Max[responselist]}},
                Axes->{0,0},
                PlotJoined->True];
```

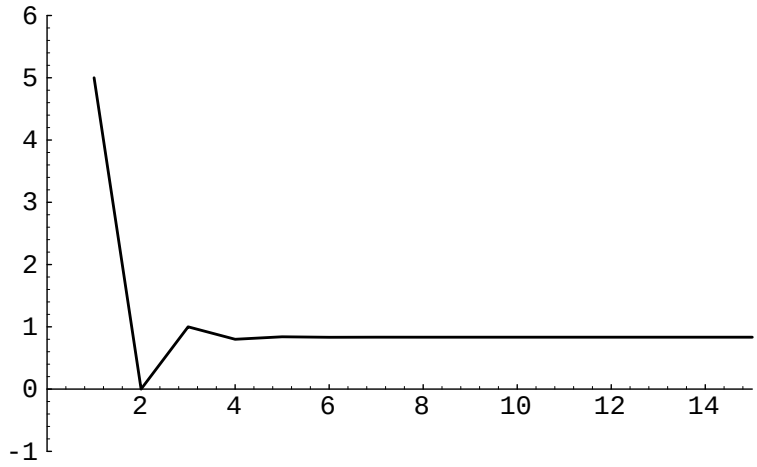
For a step **input** of height .5, a **target** value of 1 and an **epsilon** of .2 the following plot shows the first 15 outputs of this AGC.

```
PlotResponse[AGCResponse[.5,1,.2,15]];
```



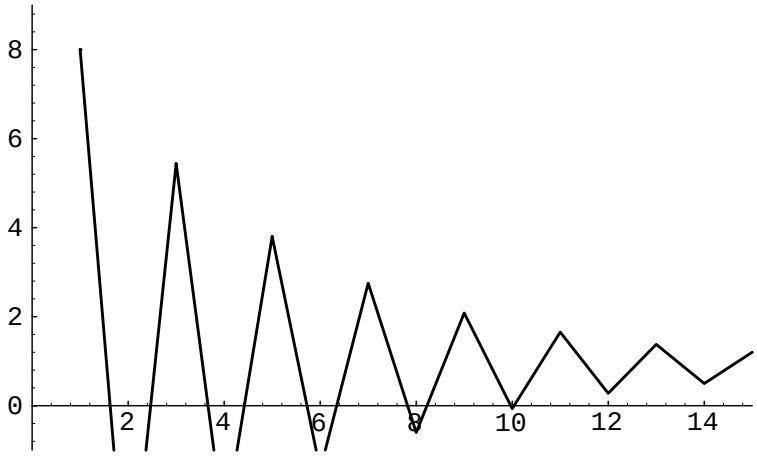
For an input that is 10 times larger the response is shown below.

```
PlotResponse[AGCResponse[5,1,.2,15]];
```



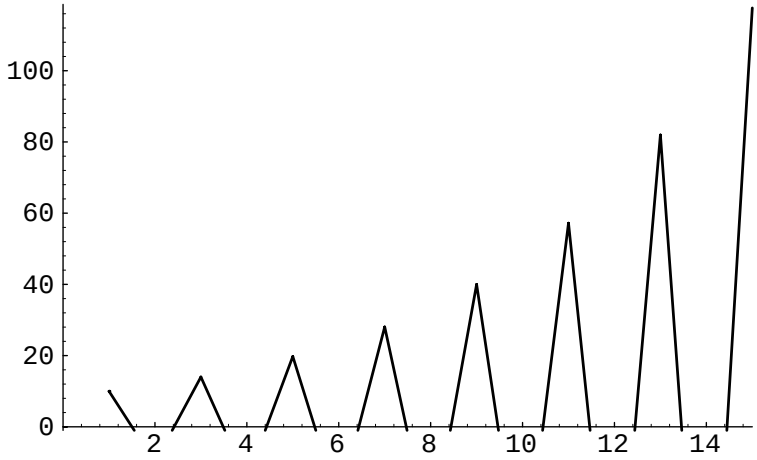
One problem with this AGC is that it initially overcorrects by reducing the gain too much. For large enough inputs the gain can go negative, leading to highly oscillatory behaviour.

```
PlotResponse[AGCResponse[8,1,.2,15]];
```



For even larger outputs this AGC is unstable.

```
PlotResponse[AGCResponse[10,1,.2,15]];
```



■ 4.2 - AGC Improvements

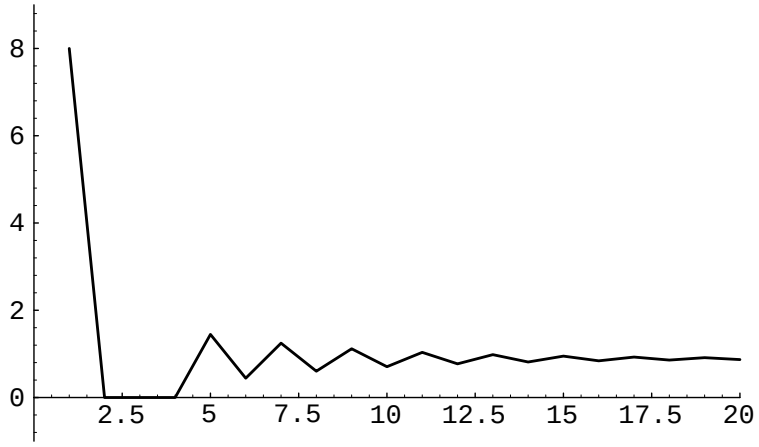
We will consider three different approaches to keep the AGC stable. First, if the variable gain is followed by a half wave rectification then this will prevent the **state** loop and the AGC stages that follow from ever seeing a negative input. This is described by:

```
Hwr[x_] := If [ x < 0, 0, x]
```

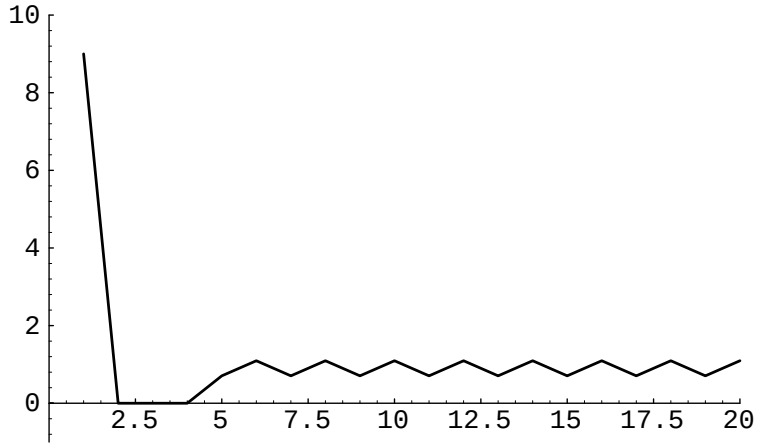
```
AGCResponse1[input_, target_, epsilon_, count_] :=
    Block[{response,state,eps,eps1},
        state = 0;
        eps = epsilon/target;
        eps1 = 1 - epsilon;
        response = Table[0,{i,1,count}];
        Do [response[[i]] = Hwr[(1-state)*input];
            state = response[[i]]*eps + state*eps1,
            {i,1,count+1}];
        response]
```

Note that this AGC does not blow up like **AGCResponse** above but its output is not very well behaved. For large enough inputs (and short enough time constants) the output will eventually oscillate between zero and a large number with a value between the input and the target.

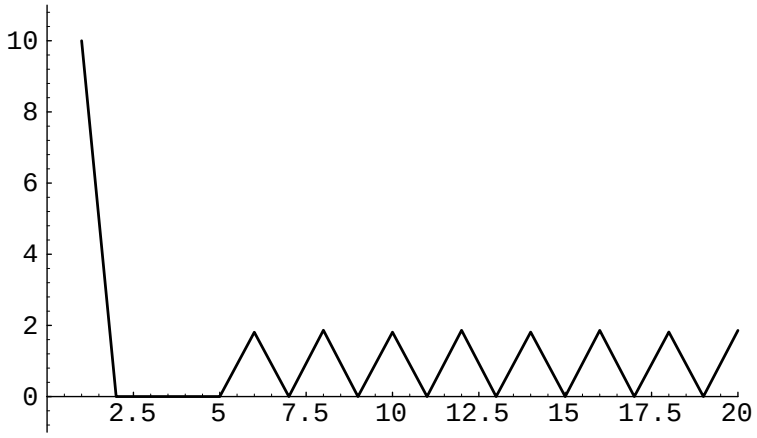
```
PlotResponse[AGCResponse1[8,1,.2,20]];
```




```
PlotResponse[AGCResponse1[9,1,.2,20]];
```



```
PlotResponse[AGCResponse1[10,1,.2,20]];
```

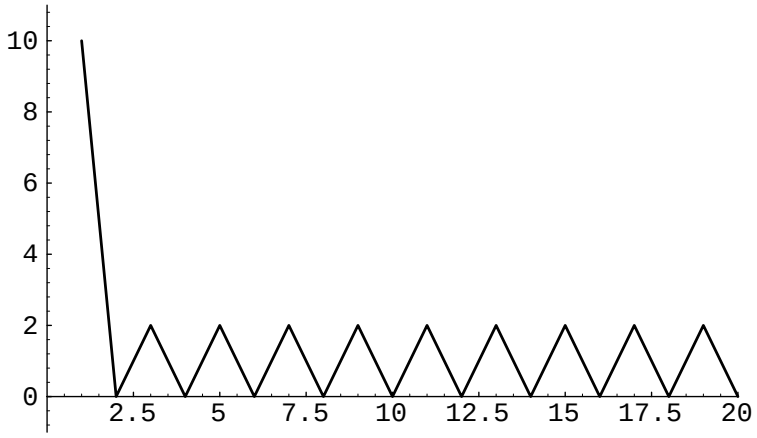


A second approach is to limit the value of the **state** variable to be less than one. This keeps the gain from ever going negative.

```
LimitValue[x_,value_] := If [ x < value, x, value]

AGCResponse2[input_, target_, epsilon_, count_] :=
    Block[{response,state,eps,eps1},
        state = 0;
        eps = epsilon/target;
        eps1 = 1 - epsilon;
        response = Table[0,{i,1,count}];
        Do [response[[i]] = (1-state)*input;
            state = LimitValue[response[[i]]*eps +
                                state*eps1,
                            1.0],
            {i,1,count+1}];
        response]
```

Again the response does not blow up, but it is still oscillatory.

```
PlotResponse[AGCResponse2[10,1,.2,20]];
```

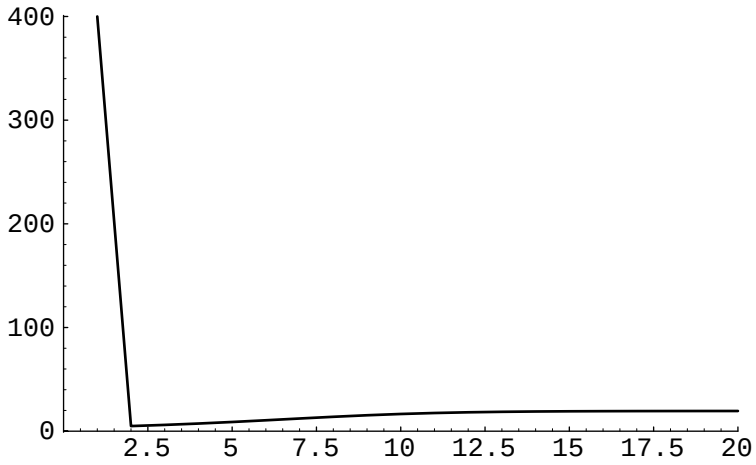


Finally, a third approach is to use a soft limiter so that the gain approaches but never quite reaches zero. In this case we have replaced the **state** variable in the gain equation with **state/(1+state)**. This function slowly approaches one as the **state** variable gets large and thus prevents the AGC from shutting off completely.

```
AGCResponse3[input_, target_, epsilon_, count_] :=
    Block[{response,state,eps,eps1},
        state = 0;
        eps = epsilon/target;
        eps1 = 1 - epsilon;
        response = Table[0,{i,1,count}];
        Do [response[[i]] = (1-state/(1+state))*input;
            state = response[[i]]*eps + state*eps1,
            {i,1,count+1}];
        response]
```

The response to this AGC is now more desireable, even for very large inputs, though the output is no longer kept below target for large inputs.

```
PlotResponse[AGCResponse3[400,1,.2,20]];
```



5 - Conclusions

This report has described the implementation of one of Richard Lyon's models of the cochlea (or inner ear.) While we hope this model accurately dscribes the total mechanical effects of the cochlea it makes no attempt to accurately model each individual component. However, we do not think this distinction is important when building speech recognizers. Seperate models of the hair cells and the neurons will be published later.

I wish to acknowledge the help of Richard Lyon, Steve Milne, Robert Hon (all at Apple) and Steve Skienna (now at the State University of New York, Stony Brook) for their help with this notebook. This model was first defined and implemented while Richard Lyon was a member of the Schlumberger Palo Alto Research Laboratory.

6 - Usage

7 - References

Lyon82 - R. F. Lyon, "A computational model of filtering, detection and compression in the cochlea," in *Proc. of the IEEE Int. Conf. Acoust., Speech, Signal Processing*, Paris, France, May 1982.

Lyon85 - R. F. Lyon and N. Lauritzen, "Processing speech with the multi-serial signal processor," in *Proc. of the IEEE Int. Conf. Acoust., Speech, Signal Processing*, Tampa, FL, Mar. 1985.

Lyon88a - R. F. Lyon and C. Mead, "An analog electronic cochlea," in *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-36, pp. 1119-1134, July 1988.

Lyon88b - R. F. Lyon and C. Mead, "Cochlear Hydrodynamics Demystified," California Institute of Technology, Computer Science Department Technical Report, 1988.

Oppenheim75 - Alan V. Oppenheim and Ronald W. Schafer, *Digital Signal Processing,* Prentice Hall, Englewood Cliffs, NJ, 1975.

Pickels82 - J. O. Pickels, *An Introduction to the Physiology of Hearing*, Academic Press, London, 1982.

Schroeder73 - M. R. Schroeder, "An integrable model for the basilar membrane," JASA 53, pp. 429-434, 1973.

Wolfram88 - Stephen Wolfram, *Mathematica*™: *A System for doing Mathematics by Computer,* Addison Wesley, Redwood City, CA, 1988.

Zweig76 - G. Zweig, R. Lipes and J. R. Pierce, "The cochlear compromise," JASA 59, pp. 975-982, 1976.

Appendix - Filter Design

This appendix defines several *Mathematica* functions for signal processing applications. Mostly we concentrate on first and second order filters since the cochlear model is defined this way. In the first section we will describe continuous time filters and then we will describe 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 we use the word "Continuous" in the names of the continuous domain filters.

A.1 - Continuous Time Filter Design

A.1.1 - Roots of the filters

Continuous time filters are described by giving the filter's response as a function of complex frequency **s**. The following functions are used to evaluate the complex response of a filter for a real frequency (in cycles per second), its magnitude, phase and the response in decibels. A filter's response function is evaluated along the imaginary axis by making the substitution **s->I 2 Pi f** (or **j 2 Pi f** in conventional EE notation.) In the expressions that immediately follow **filter** can be an arbitrary function of the complex frequency s.

```
ContinuousFilterEval[filter_, f_] :=
        filter /. s->f;

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]]
```

Finally, we use the following function to display the frequency response of a continuous filter. (We start the plot at 1Hz to avoid any problems with filters that have a zero at DC.)

```
ContinuousFreqResponse[filter_, maxf_] :=
    Block[{response},
            response = N[ContinuousFilterDb[filter,f]];
            Plot[response,{f,1,maxf},
            AxesLabel->{" Hz", "dB"},
            PlotLabel->"Response"]];
```

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

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

A second order filter is described by its resonant frequency (**f**) and its quality factor (**q**). The 3dB 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.

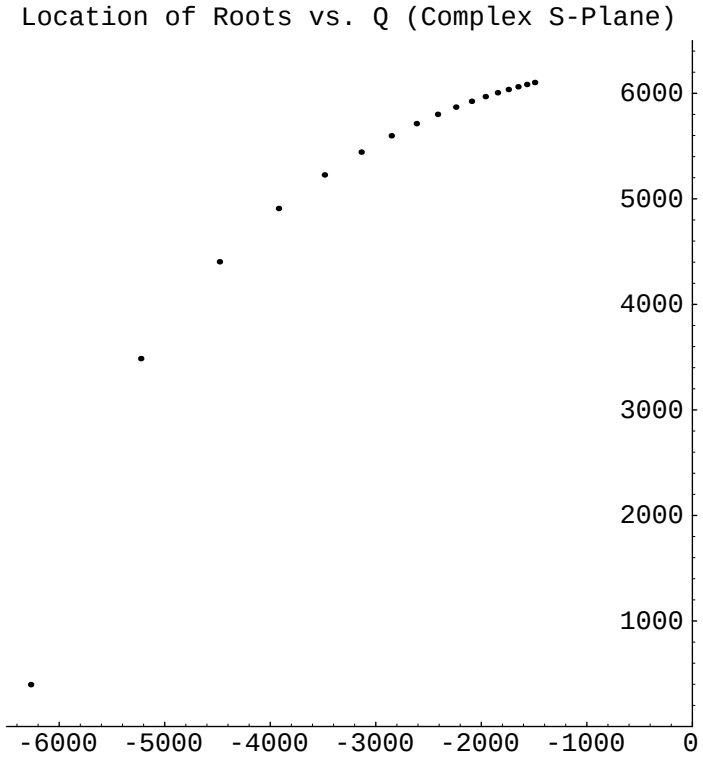
```
ContinuousSecondOrderRoot[f_,q_] := -2 Pi f/q/2 +
                                I 2 Pi f Sqrt[1-1/(2q)^2]
```

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 measured 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. 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 (.501) and thus a broad response in the frequency domain.

The radius of this circle is 2 Pi times the resonant frequency (1000Hz). As the resonance frequency is increased this plot will trace out a circle with a larger radius. In all cases a filter with real coefficients will

have a second set of roots that are complex conjugates of the ones shown below. Also for the filter to be stable all of the roots must be in the left half of the complex plane.



We can 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 3dB 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)
```

□ A.1.2 - Test Code

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])]
```

$$\left(-\left(\frac{f}{f\sqrt{1-\frac{1}{2q^2}}+\frac{f}{2q}}\right)+\frac{2q}{f}\right)q$$

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

$$\left(-\left(\frac{f}{f\sqrt{1-\frac{1}{2q^2}}-\frac{f}{2q}}\right)+\frac{2q}{f}\right)q$$

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]]
```

$$\frac{f}{q}$$

```
ContinuousSecondOrderRoot[1000,10]
```

```
-100 Pi + 100 I Sqrt[399] Pi
```

```
N[Abs[%]/ ( 2 Pi ) ]
```

```
1000.
```

□ A.1.3 - Continuous Filter Design

Now we can actually create a filter. A bandpass filter (or resonator) is defined by placing the roots of a second order polynomial in the denominator. This time we normalize the gain of the filter so at its peak (at the NaturalResonance frequency) it has unity gain.

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

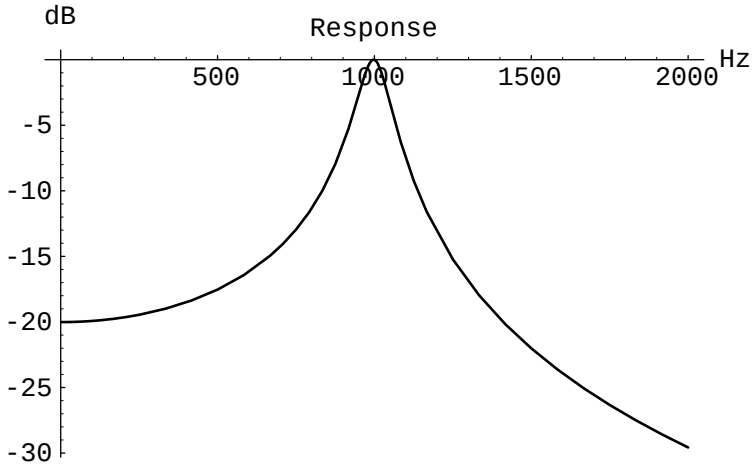
A typical bandpass filter (or resonator) with a center frequency of 1000Hz and a q of 10 looks like this.

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

$$\frac{197392. + 3.93796\ 10^{6}\ I}{3.94784\ 10^{7} + 628.319\ s + s^{2}}$$

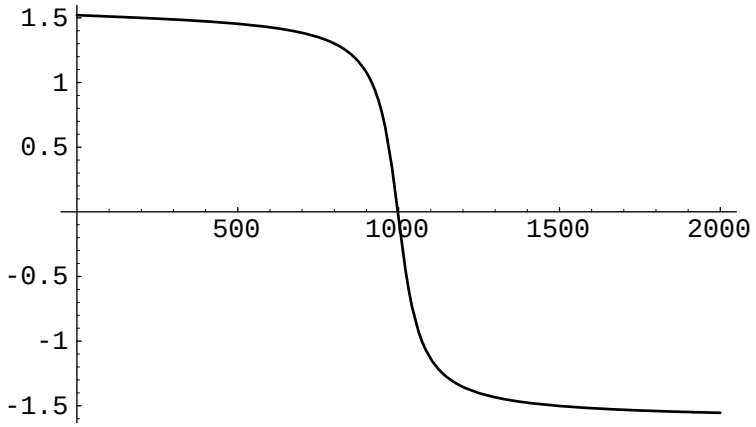
The gain of this bandpass filter is shown below.

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

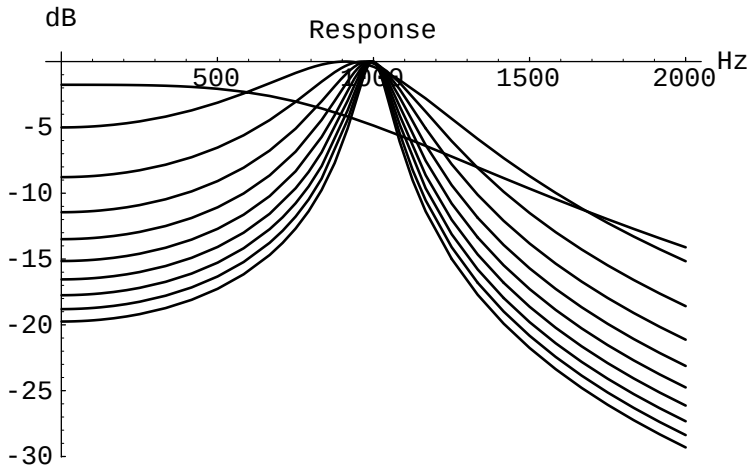


Finally, we can also show how the phase of this bandpass filter shifts as the frequency passes through the resonant frequency. The phase is equal to zero near the center frequency since we have normalized the gain of the filter so that it is equal to one at the resonance frequency.

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



The plot below shows the frequency response of second order bandpass filters with a center frequency of 1000Hz as the quality factor (**q**) is varied. The flattest curve is the frequency response of a filter with a **q** of .7 while the narrowest (sharpest) 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]
```

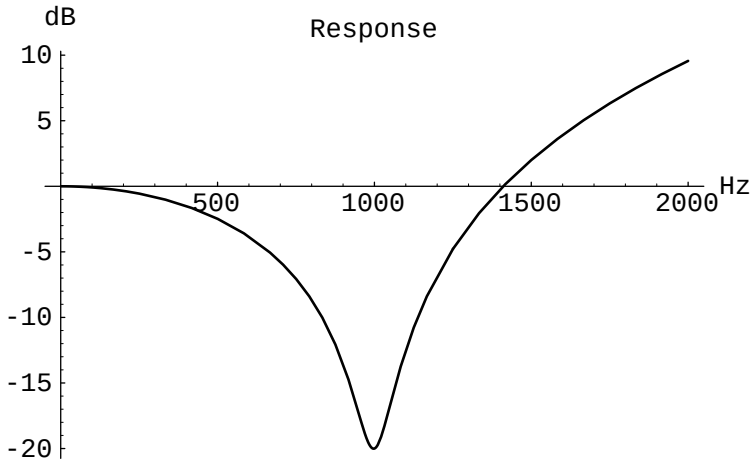
A filter with a notch centered at 1000 Hz and with a **q** of 10 is described by the following equation.

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

$$\frac{2.53303 \times 10^{-8}}{3.94784 \times 10^{7} + 628.319\, s + s^2}$$

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

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



□ A.1.4 - More Test Code

The following function prints some interesting information about a filter function.

```
FilterCheck[filter_,f_,q_] :=
        Block[{center,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.

```
a=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.

```
f2=FilterSearch[a,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.

```
f1=FilterSearch[a,N[Sqrt[2]/2],500,1000]
```

```
625.202
```

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

```
685.414
```

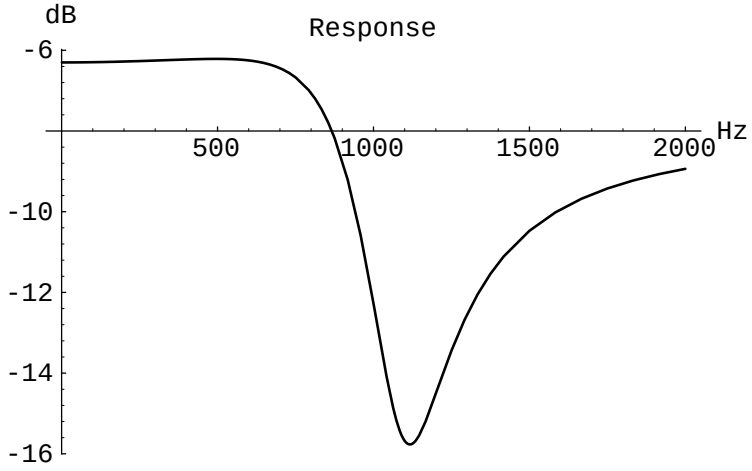
☐ A.1.5 - Playing Around

We can now combine notch and resonance filters to get more interesting responses. The following combination of a resonance with a q of 2 at 1000Hz and a notch with a q of 5 at 1100Hz is similar to the cascade-only ear filters described in the main part of this report.

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

$$\frac{(0.103306 + 0.386535\,I)\,(4.77689\,10^{7} + 1382.3\,s + s^{2})}{3.94784\,10^{7} + 3141.59\,s + s^{2}}$$

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

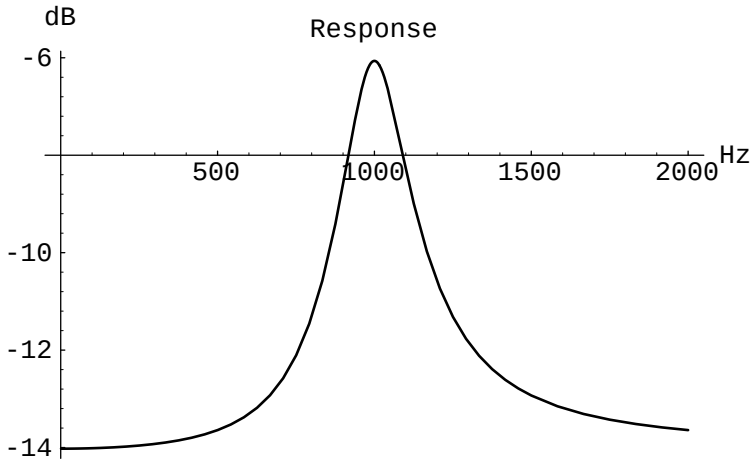


A second filter is shown below. We have combined a broad notch with a narrow resonance to give a bandpass filter.

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

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

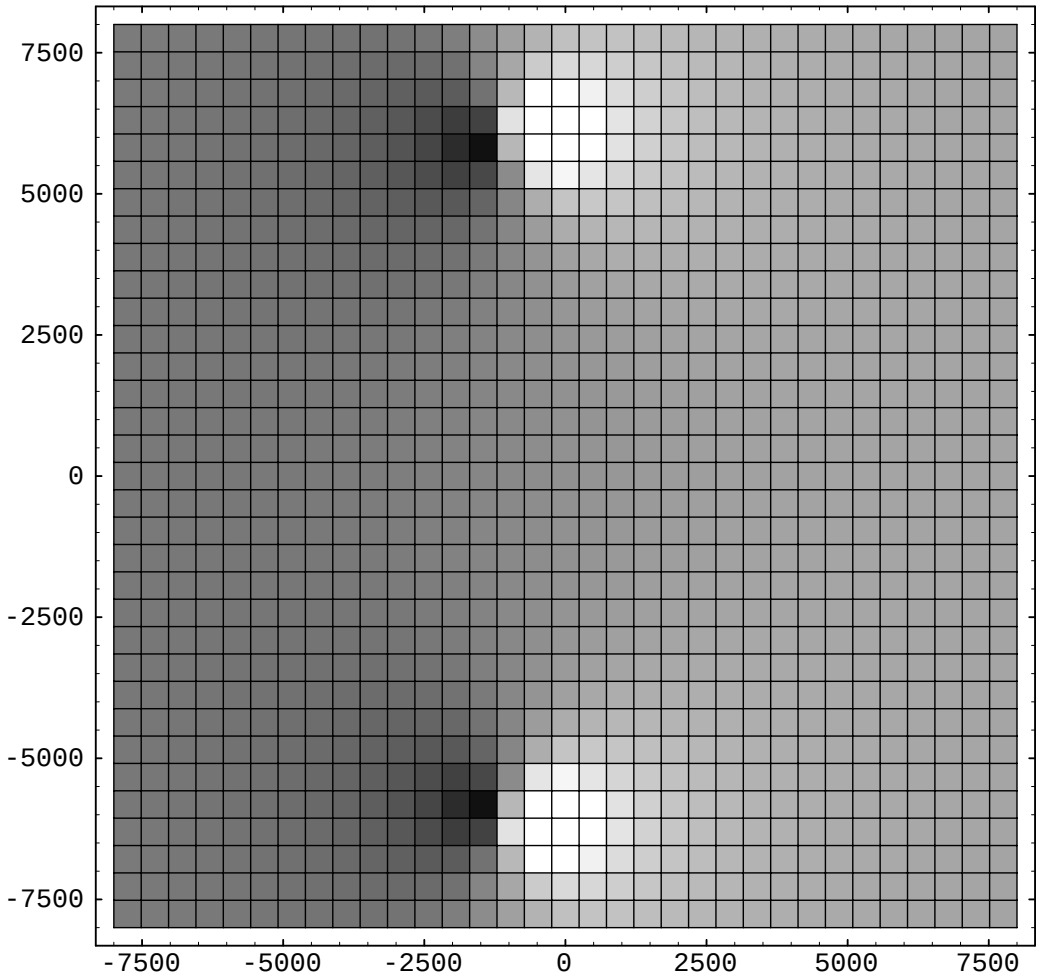
```
ContinuousFreqResponse[MakeContinuousResonance[1000,5]*
    MakeContinuousAntiResonance[1000,2]//N,2000];
```



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_] := Abs[(MakeContinuousResonance[1000,5] *
                    MakeContinuousAntiResonance[1000,2])
                        /.s->x]
```

```
DensityPlot[gg[x + I y],{x,-8000,8000},{y,-8000,8000},
            PlotPoints->33];
```



□ A.1.6 - Usage

■ A.2 - Discrete Time Filter Design

This section of the notebook describes how to calculate first and second order digital filters. Discrete time filters are described by their response in the z-domain. This section first describes 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.

A.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**). These function will be used in the next section when the ear filters are defined.

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**.

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

```
                2    3
1 + 4 z + 2 z  + z
```

```
FilterEval[poly,1]
```

```
8
```

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

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

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

```
FilterEval[rat,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[rat]
```

```
 -2 - 2 I Sqrt[2]  -2 + 2 I Sqrt[2]
{----------------, ----------------}
        2                 2
```

```
RationalZeros[rat]
```

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

☐ A.2.2 - The Filter Design Equations

This section describes some general first and second order filter design functions. We will use these functions later to define the filters used in the ear model.

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 will have 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):

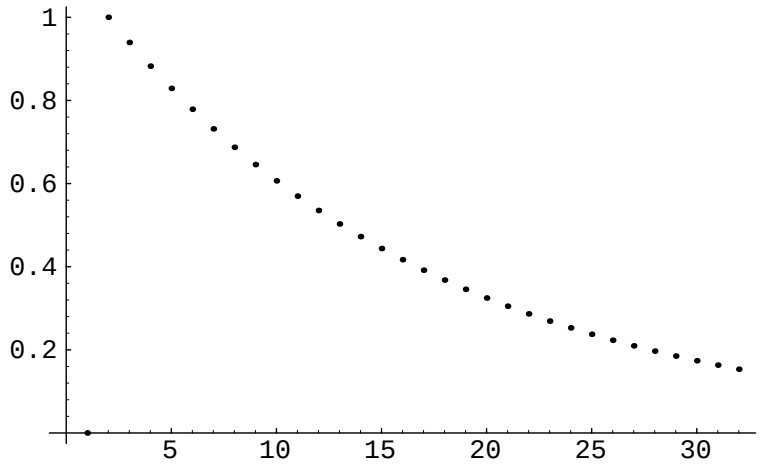
```
FirstOrderFromTau[tau_, fs_] :=
    MakePoly[{-EpsilonFromTauFS[tau, fs],1}]
```

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

```
fo = FirstOrderFromTau[0.001, 16000]
```

```
-0.939413 + z
```

```
ListPlot[FindImpulseResponse[1/fo,32]];
```



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 can be plotted in *Mathematica* using the following function.

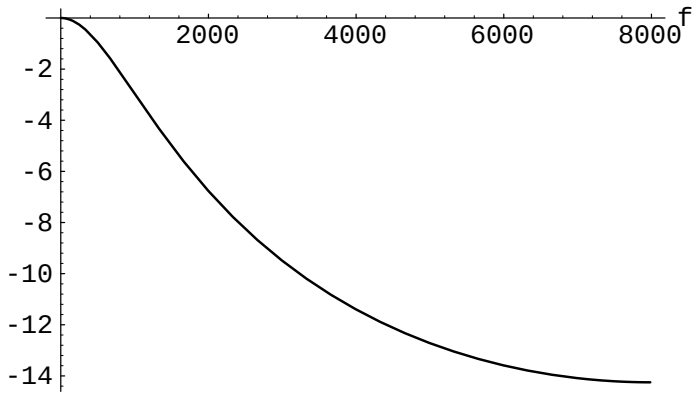
```
FreqResponse[coeffs_, fs_] :=
    Block[{function,f},
    function = N[dB[Abs[FilterGain[coeffs,f,fs]]]];
    Plot[function,
             {f,1,fs/2-fs/1000},
             AxesLabel->{"  f","Response in dB"}]]
```

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

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

```
   0.324768
--------------
1. - 0.675232 z
```

```
FreqResponse[first,16000];
```



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

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

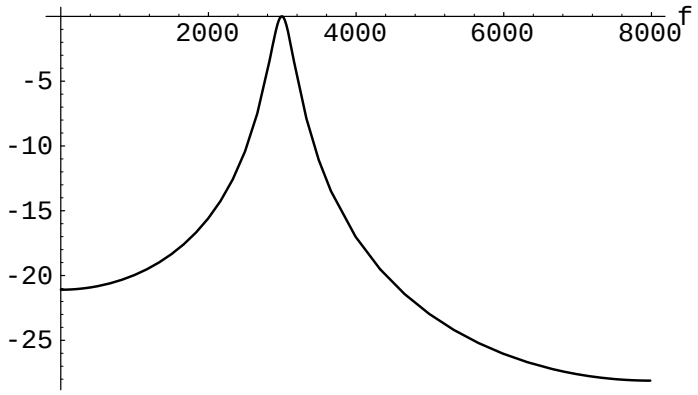
```
-2.95485
```

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

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

```
 -0.0953624 + 0.0380781 I
--------------------------
                        2
0.888865 - 0.724151 z + z
```

```
FreqResponse[second,16000];
```

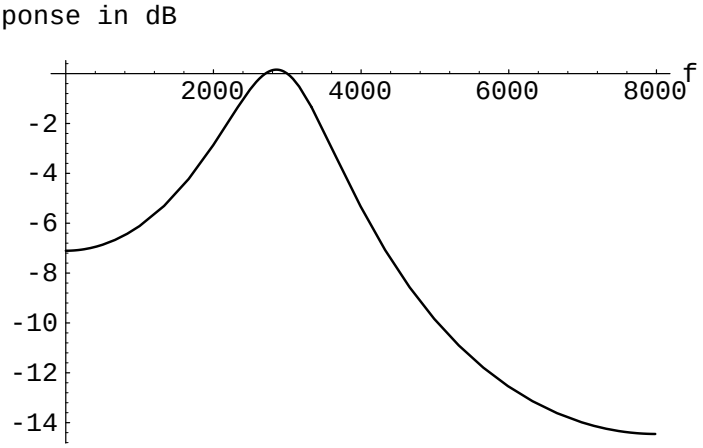


A less peaky response is possible using a lower **Q**.

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

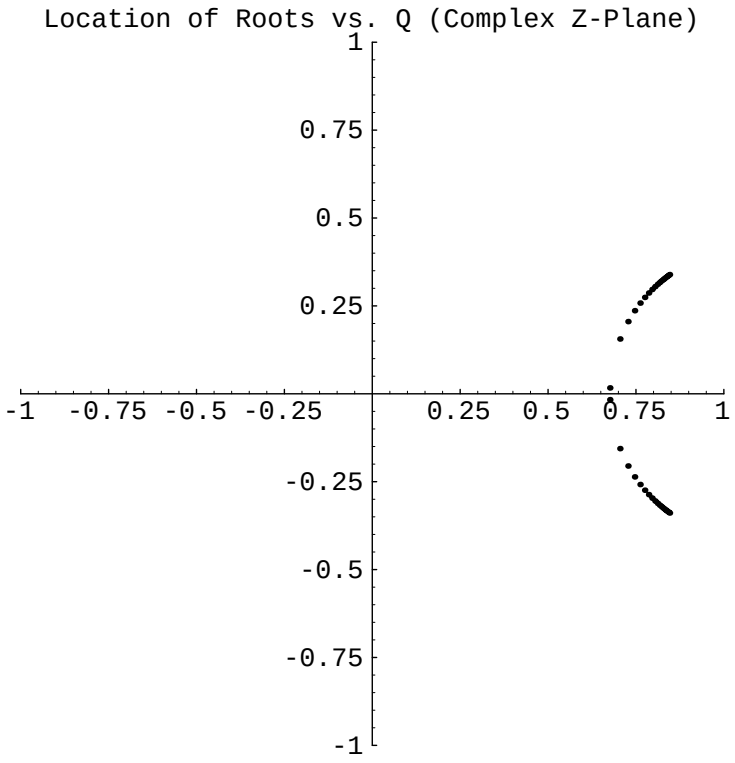
```
  -0.389971 + 0.133203 I
--------------------------
                         2
0.554855 - 0.621189 z + z
```

```
FreqResponse[second,16000];
```

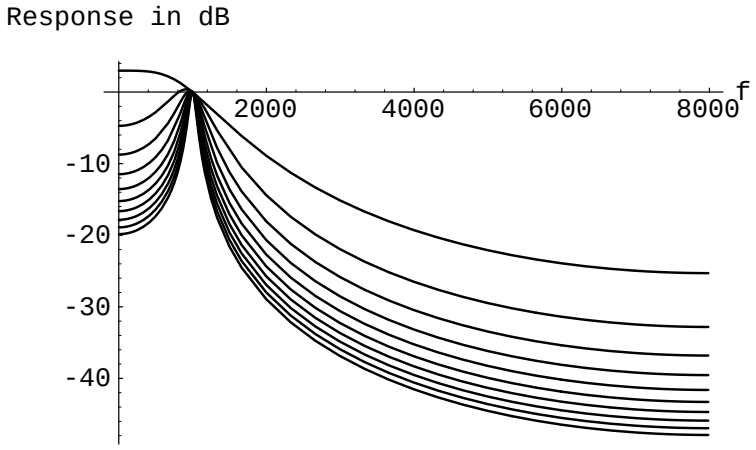


The poles and zeros in the complex z-plane of a stable discrete time filter are inside the unit circle. In general the resonant frequency of the root is proportional to the angle of the root from the positive real axis. Roots that are close to the unit circle will have a high **q**. The plot below shows the roots of a second order section with resonant frequency of 1000Hz and a sampling frequency of 16000Hz as the **q** varies from .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 1000Hz and a sampling frequency of 16000. The flattest curve is the frequency response of a second order bandpass filter with a **q** of .7 while the sharpest (narrowest) curve is the frequency response when the **q** is 10.

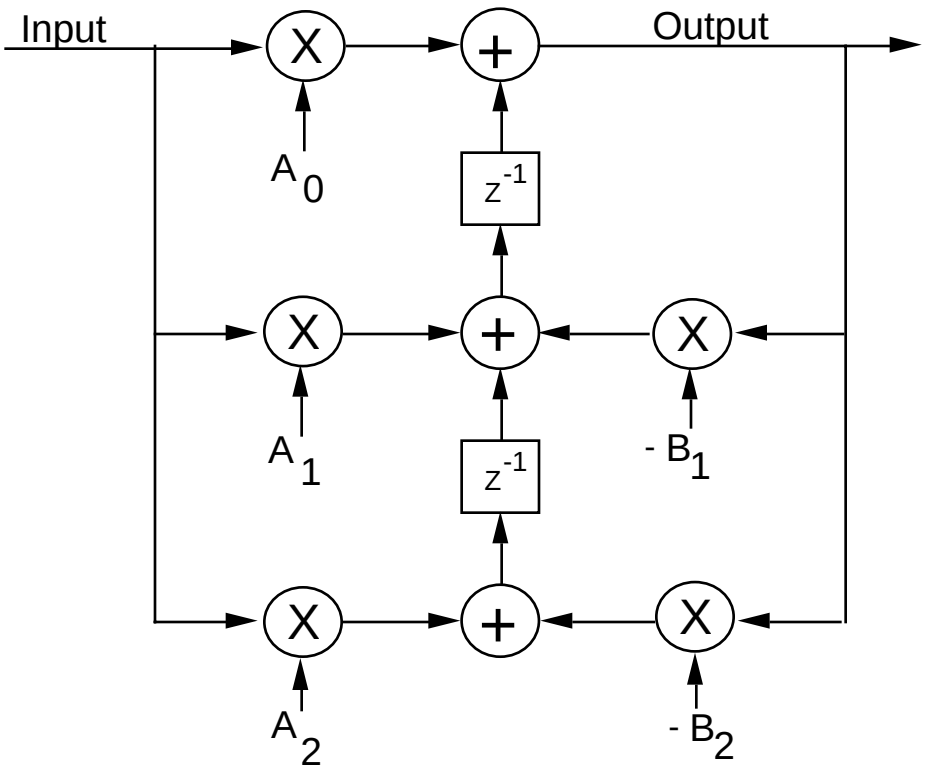


□ A.2.3 - Time Domain Implementation

The following signal flow graph shows how the filter

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

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



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

```
FindImpulseResponse[filter_,maxn_] :=
    Block[{num, denom, numorder, denomorder, response,
            order, inverse, new},
            new = Expand[Numerator[filter]] /
                        Expand[Denominator[filter]];
            order = Abs[Exponent[Numerator[new],z]-
                        Exponent[Denominator[new],z]];
            inverse = Expand[new/.z->1/z];
            num = Expand[Numerator[inverse]];
            denom = Expand[Denominator[inverse]];
            inverse = Simplify[Expand[z^order num]/
                                Expand[z^order denom]];
            num = Expand[Numerator[inverse]];
            denom = Expand[Denominator[inverse]];
            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}
```

□ A.2.4 - Test Code

The following examples are from page 35 of [Rabiner1975]

```
Simplify[Table[-a a^i/(b-a) + b b^i/(b-a),{i,0,4}]]
```

```
                14              14
{1, (3.1658 10    - 1.82236 10    I) /

                  7                 2
    ((3.94784 10  + 3141.59 s + s ) 

                  6             7               7
```

```
    (-4.9348 10  - 1.84643 10  I + 3.94784 10  b +

                        2
      3141.59 b s + b s )) +

                 2
                b
 ------------------------------,
                6             7
     -4.9348 10  - 1.84643 10  I
 b + --------------------------
                7               2
     3.94784 10  + 3141.59 s + s

           21             21
(4.92712 10   + 4.94613 10   I) /

              7               2 2
  ((3.94784 10  + 3141.59 s + s )

                6             7               7
    (-4.9348 10  - 1.84643 10  I + 3.94784 10  b +

                        2
      3141.59 b s + b s )) +

                 3
                b
 ------------------------------,
                6             7
     -4.9348 10  - 1.84643 10  I
 b + --------------------------
                7               2
     3.94784 10  + 3141.59 s + s

            28             29
(-6.70128 10   + 1.15384 10   I) /

              7               2 3
  ((3.94784 10  + 3141.59 s + s )

                6             7               7
    (-4.9348 10  - 1.84643 10  I + 3.94784 10  b +

                        2
      3141.59 b s + b s )) +
```

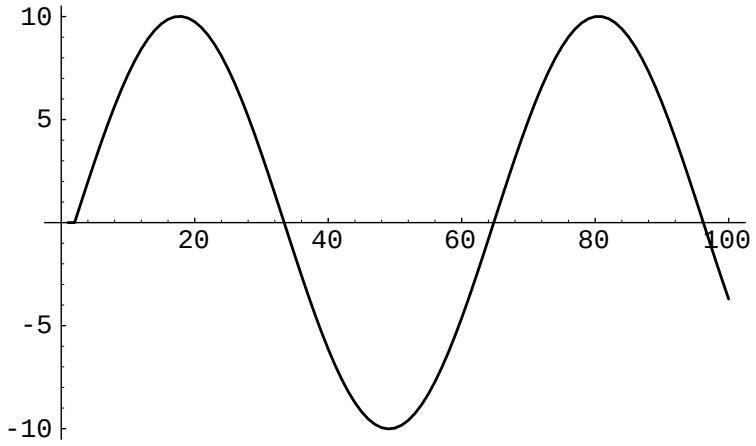
```
                  4
                 b
 ------------------------------,
                6             7
    -4.9348 10  - 1.84643 10  I
b + ------------------------------
                7                2
    3.94784 10  + 3141.59 s + s

           36             35
(2.46119 10   + 6.67948 10   I) /

                7                2 4
  ((3.94784 10  + 3141.59 s + s )

               6             7               7
    (4.9348 10  + 1.84643 10  I - 3.94784 10  b -

                        2
     3141.59 b s - b s )) +

                  5
                 b
 ------------------------------}
                6             7
    -4.9348 10  - 1.84643 10  I
b + ------------------------------
                7                2
    3.94784 10  + 3141.59 s + s
```

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

```
                    -8              -8
{0, 0, ((1.35095 10   - 5.05479 10   I)

                  7                 2
     (3.94784 10  + 3141.59 s + s )) / b,

                 -15              -15
   ((-2.37258 10    - 1.36575 10    I)

                  7                 2
     (3.94784 10  + 3141.59 s + s )

                 6              7                 7
     (4.9348 10  + 1.84643 10  I + 3.94784 10  b +

                        2      2
       3141.59 b s + b s )) / b ,

                -8              -8
   ((1.35095 10   - 5.05479 10   I)

                  7                 2
     (3.94784 10  + 3141.59 s + s )

                    -8              -8
     (((-1.35095 10   + 5.05479 10   I)

                     7                 2
          (3.94784 10  + 3141.59 s + s )) / b +

                   -15              -15
        ((-2.37258 10    - 1.36575 10    I)

                       6              7
          Power[4.9348 10  + 1.84643 10  I +

                     7                       2
            3.94784 10  b + 3141.59 b s + b s , 2]) /

          2
         b )) / b}
```

The first 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 1000Hz, sampling frequency of 8000Hz and a q of 10.

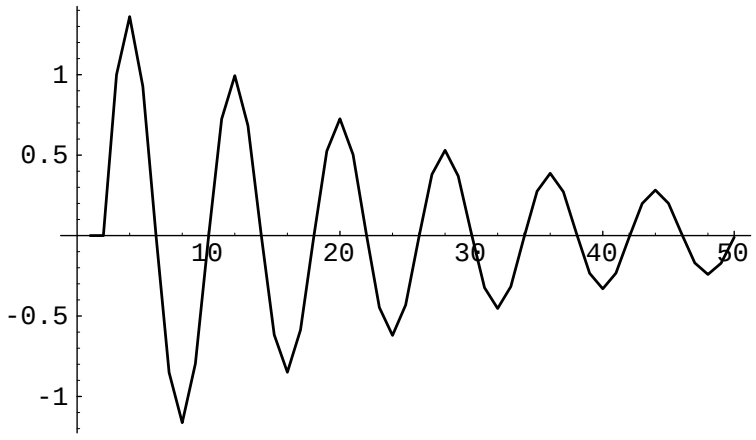
```
filter = SecondOrderFromCenterQ[1000,10,8000] //N
```

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

```
RationalPoles[1/filter]
```

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

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



A.2.5 - Usage