APP NOTE

Application Note 3

Digital Audio Parametric Equalizer

Highlights
Digital-IIR Equalizer Transform
32-Bit Floating Point Implementation
LEQ², LEQ, BEQ, HEQ, HEQ² Filters

■ Design Objective
Parametric Equalizer: Five Filter Functions
Cut/Boost: ±15dB
Frequency: 30Hz - 300Hz
Bell Width: 0.33 - 3.0 Octave
Control Resolution: 256 Steps (8-Bit)

Parametric equalizers allow for all of the parameters of an equalization filter to be adjusted independently. Typically this includes three parameters: cut/boost gain (dB), frequency (Hz), and Width (Octave) for bell/Bandpass equalizers. This type of control is often used in professional audio equipment where more sophisticated and powerful equalization is demanded.

There are three basic forms of equalization configurations: Lowpass Shelving, Highpass Shelving, and Bandpass Bell. In FilterShop these are known as LEQ, HEQ, and BEQ respectively.

For this digital IIR class of filters, we have added two additional types: LEQ² and HEQ². Since we will already be working with a single biquadratic IIR section that can produce 2nd order polynomials, and the LEQ/HEQ types are only 1st order polys, we can generate a squared version of each that fits within the 2nd order structure with no additional difficulty. These filters have a slightly steeper slope rate then the 1st order forms which can sometimes be desirable.

The user control interface for a digital equalizer could be physical controls such as pots(encoders) and switches, or a full computer based software graphical user interface. In either case we will assume that the control resolution is provided as 8 bit codes ranging from 0 to 255. This provides 256 steps of resolution for each Gain, Frequency, and Width control. This yields about 0.1dB steps for the Gain.

■ Fixed Equalizers

It is worthwhile to briefly discuss and contrast the requirements of a fixed equalizer before examining the special design requirements of a parametric.

The implementation of *fixed* IIR equalizers is relatively simple, especially when using the *Digital-IIR: Equalizer Transform* in this program. Since the frequency and/or Q of the filter is fixed, with only the gain adjustable, the IIR coefficients can be precomputed for each and every gain position.

The coefficient sets are then stored in memory for each gain position and used as a *lookup table*. The controller or DSP does not need to perform complex calculations. It only needs to extract six numbers from the table based on the gain encoder position, and load these into the proper DSP memory locations. Using the *Make Table* feature in the Equalizer Transform dialog, the coefficient values for all of the gain encoder positions can be generated at once.

Thus fixed equalizers can be readily implemented using a lookup table containing precomputed IIR coefficients. For example, using 8 bit resolution for the gain parameter 256 sets of coefficients are needed. If each of the six coefficients are stored with 24 bit precision, the total memory usage is only 4608 bytes. Since the B0 coefficient is always unity, this can even be reduced further down to 3840 bytes.

The above exercise demonstrates that most fixed equalizers can be stored in less then 4000 bytes of memory, even in a small 4K EPROM. The controller CPU requirement for fixed equalizers is thus minimal, and 8 or 16 bit CPUs are typical.

The situation becomes more memory intensive for multiband graphic equalizers. Each band in a graphic equalizer is fixed, but there are many bands, each at a different frequency. For example, a 1/3 octave graphic equalizer with 32 bands would require 128K of coefficient memory.

It is still certainly practical to implement a multiband equalizer using the brute force lookup table method. However, in these cases the alternative approach of direct coefficient computation becomes far more efficient. The total memory for both the code and data to implement such a method would probably be 1/10 of the previous value. But, the computations are very difficult with fixed point processing. Floating point math becomes almost a necessity. This method is discussed in the following section.

■ Parametric Equalizers

The implementation of *parametric* IIR equalizers using complete lookup tables is almost impossible. Storing the coefficients for all of the possible control combinations would demand enormous resources. For example, assuming each of the three Gain, Frequency, and Width parameters use 8 bit precision, the number of possible permutations is 16,777,216. If each of the five coefficients are again stored in 24 bit precision, the total memory requirement would be 250MB.

Parametric equalizers virtually demand that the host controller CPU and/or DSP perform computations for the IIR coefficients. This is usually handled in one of two ways, depending on the availability of floating point math:

(a) If only fixed point math is available, you will probably need to utilize a linear interpolation method along with a partial lookup table. Since the lookup table does not contain all of the possible combinations, interpolation is used to approximate the other coefficients between the table points. While this only requires simple arithmetic, determining the best set of table values can be difficult and time consuming. Many interpolation points must be checked to verify that the proper response is obtained. In some cases the interpolated IIR coefficients may produce unstable filters which oscillate.

(b) If floating point math is available, the situation is considerably simplified. No lookup tables are required, and all IIR filter coefficients are computed directly. Transcendental functions are generally required, but these can be emulated in the software if the CPU/DSP only provides mul/div/add/sub operations.

Historically most of these designs have relied on fixed point processors since floating point devices were not available or too expensive. Indeed some fixed point DSP devices even provide lookup table interpolation instructions.

However, with the ever increasing power and capabilities offered in today's DSP and CPU devices, 32 bit floating point support has become relatively inexpensive and commonplace. This trend will obviously continue.

There are many advantages to floating point DSP including lower noise, virtual freedom from overload, and of course the ability to compute coefficients in real time. Most new designs will be increasingly implemented using 32 bit floating point processors, and therefore we will use this method for this example as well.

■ CPU/DSP Processing Options

Somebody will need to handle 32 bit floating point numbers. This could either be the CPU or DSP. At a minimum the math requirements are mul, div, add, sub, along with a few of the usual compare instructions <, =, >. Generally any 32 bit processor offering floating point support should meet this criteria.

Since the primary role of the DSP is to process real data at the required sample rate, the most obvious candidate for nomination as the coefficient calculator is the CPU. In this scenario the CPU must provide the 32 bit floating point capability, while the DSP could be anything. An example of suitable common industry devices might be:

Intel 80960KB	(32 bit CPU)
Mot 56000	(24 bit DSP)

However, with the increasing power being loaded into new DSPs everyday, they can now perform many different tasks. If a 32 bit floating point DSP is selected, the CPU can be almost anything. In this scenario the DSP would perform the required coefficient computations interleaved between its usual chores of data processing. This of course assumes that there are excess FLOPS available in the DSP, which is often the case. These are simple IIR 2nd order filters and produce very low utilization in most DSPs at audio sampling rates. An example of suitable common industry devices might be:

Microchip PIC16C74	(8 bit CPU)
TI TMS320C30	(32 bit DSP)

These devices are listed to illustrate the two basic alternatives. There are many other similar devices which could also be used in their place.

■ Encoder Translation

The source of control for the three parameters Gain, Frequency, and Width will be 8 bit values ranging from 0 to 255. In order to have a true center position, the number of steps must be odd. Since we have 256 steps, positions 0 and 1 will be assigned the same value, resulting in a range of 1 - 255 with a center position of 128.

The Gain control will have a linear scale in dB. However the Frequency and Width controls must have exponential scaling. The appropriate formulas to convert encoder position to parameter value would be:

Ao = 15.0* (Pos - 128)/127 (-15dB ... +15dB)
Fo = 30.0*Exp10(Log10(300.0/30.0)*(Pos-1)/254) (30Hz ... 300Hz)
Oct = 0.333*Exp10(Log10(3.0/0.333)*(Pos-1)/254) (0.333Oct ... 3.0Oct)

■ IIR_PEQ Example Source Code

Contained in the AppNote03 folder are two files IIR_PEQ.C and IIR_PEQ.H. These files contain an example of the necessary code to implement a IIR Parametric EQ. An emulation library for the required transcendental functions is also provided.

A full listing of the code would require too much space here, but the headers for the two exported functions are shown on the following page. The first function *IIR_EQ* provides an interface based on the user supplying actual values for the Gain, Frequency, Width parameters Ao, Fo, Oct. The second function *IIR_EQ_ENCODE* provides an interface which performs the required encoder translation based on the user supplying the raw encoder position data.

In both cases there are six variables used to return the values of the numerator and denominator IIR coefficients. Two parameters are also provided which select the type of filter function and method of transform.

The IIR_PEQ source code module can be incorporated into a CPU/DSP program to perform the coefficient calculations, and then the resulting coefficients can be loaded into the DSP as needed.

■ IIR_PEQ Response Examples

The following pages show some of the different response profiles produced from the source code for the LEQ², LEQ, BEQ, HEQ, and HEQ² filter functions. In all cases the Fo was 300Hz, and the intervals were 31. A frequency range of 20Hz-20kHz was used, and a sampling frequency of 48kHz was used.

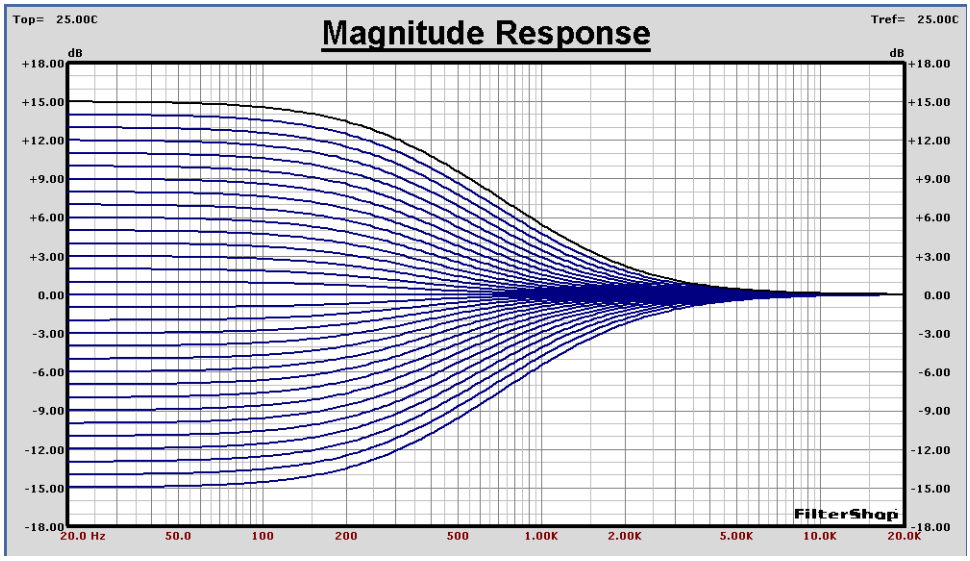
■ Summary

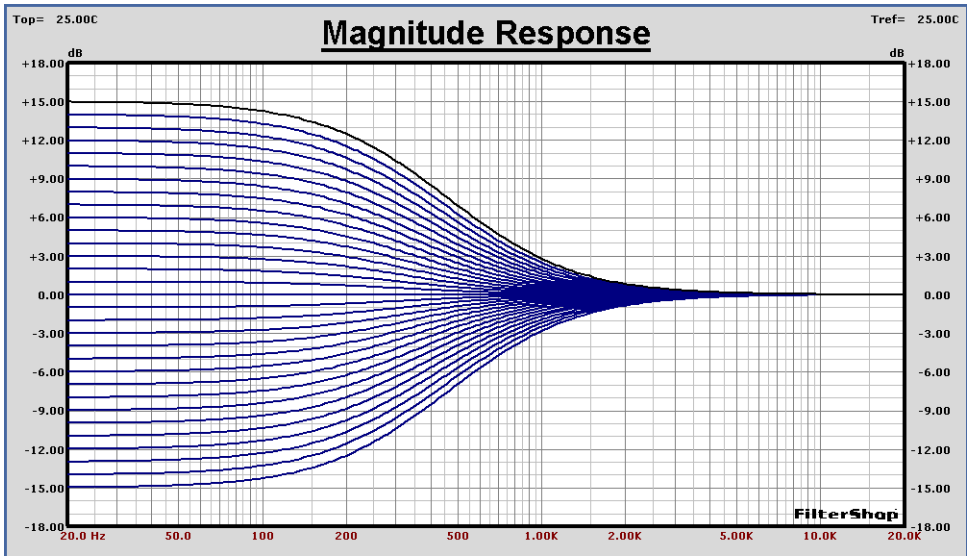
While creating a digital parametric equalizer requires code generation to compute the IIR coefficients, FilterShop is very useful for generating lookup tables and for checking interpolated coefficients or verifying the source code results.

This completes the digital parametric equalizer design.

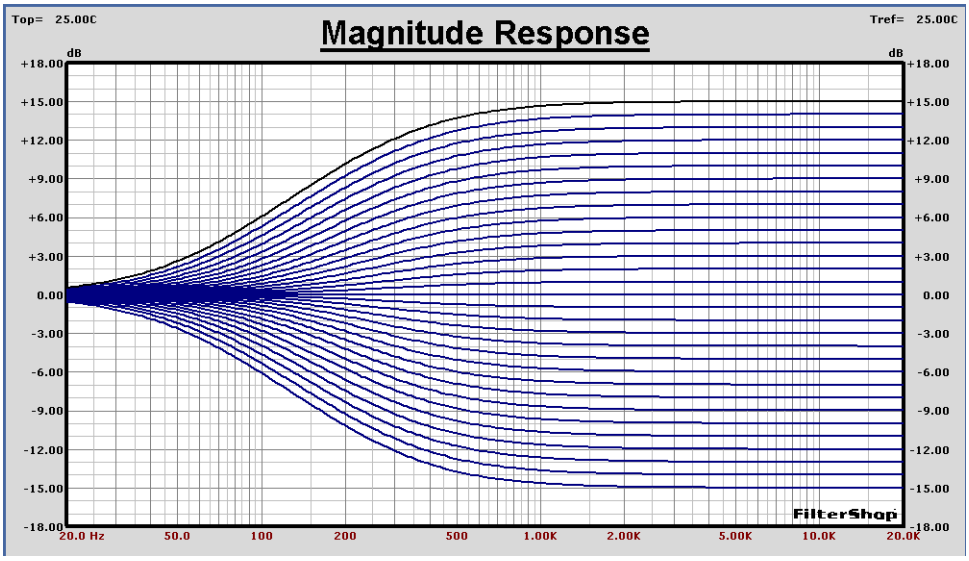
```
//  ================================================
// * =====  Main IIR Parametric EQ Routine     ====== *
// * Input:                                          *
// * FilterType: 1=LEQ², 2=LEQ, 3=BEQ, 4=HEQ, 5=HEQ²  *
// * MethodType: 1=Matched-Z, 2=Bilinear              *
// *         Fs: sampling frequency (Hz)              *
// *         Fo: corner/center frequency (Hz)         *
// *         Ao: cut/boost gain (dB)                  *
// *        Oct: width of Bandpass (Octaves)          *
// * Output:                                         *
// *     A0,A1,A2 IIR numer coefficients              *
// *     B0,B1,B2 IIR denom coefficients              *
//  ================================================
extern void  IIR_EQ(int          FilterType,
                    int          MethodType,
                    RealType     Ao,
                    RealType     Fo,
                    RealType     Oct,
                    RealType     Fs,
                    RealType    *A0,
                    RealType    *A1,
                    RealType    *A2,
                    RealType    *B0,
                    RealType    *B1,
                    RealType    *B2);

//  ================================================
// * =====  Encoder IIR Parametric EQ Routine  ====== *
// * Input:                                          *
// * FilterType: 1=LEQ², 2=LEQ, 3=BEQ, 4=HEQ, 5=HEQ²  *
// * MethodType: 1=Matched-Z, 2=Bilinear              *
// *         Fs: sampling frequency (Hz)              *
// *         PosAo: Encoder Position gain (dB)        *
// *         PosFo: Encoder Position frequency (Hz)   *
// *         PosOct:Encoder Position width (Octaves)  *
// * (Encoder position values range from 0...255)     *
// * (Position 128 is center, and 0,1 are same value) *
// * Output:                                         *
// *     A0,A1,A2 IIR numer coefficients              *
// *     B0,B1,B2 IIR denom coefficients              *
//  ================================================
extern void  IIR_EQ_ENCODE( int         FilterType,
                            int         MethodType,
                            int         PosAo,
                            int         PosFo,
                            int         PosOct,
                            RealType    Fs,
                            RealType    *A0,
                            RealType    *A1,
                            RealType    *A2,
                            RealType    *B0,
                            RealType    *B1,
                            RealType    *B2);
```

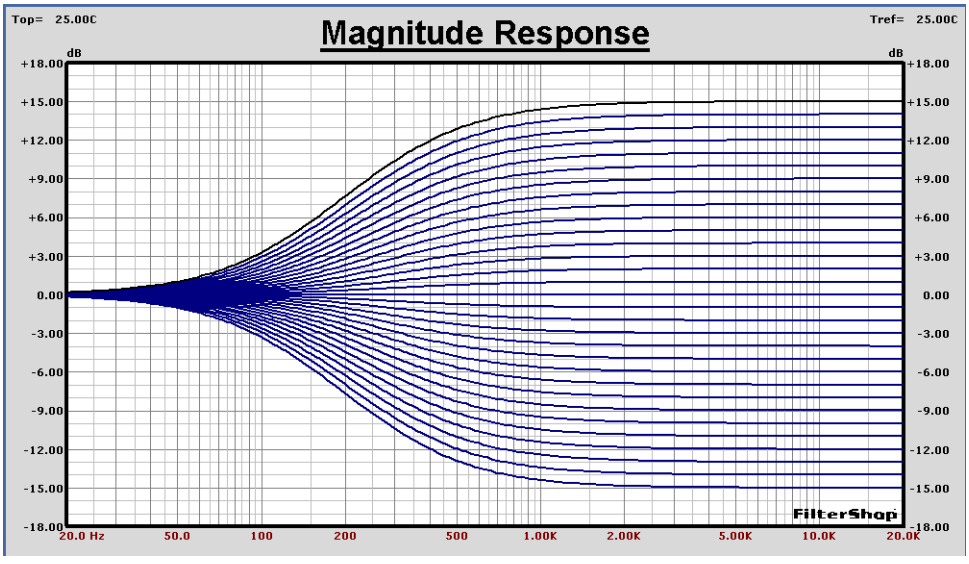
LEQ and LEQ² Filter Functions, Fo=300Hz
Fs=48kHz, Transform=Matched-Z



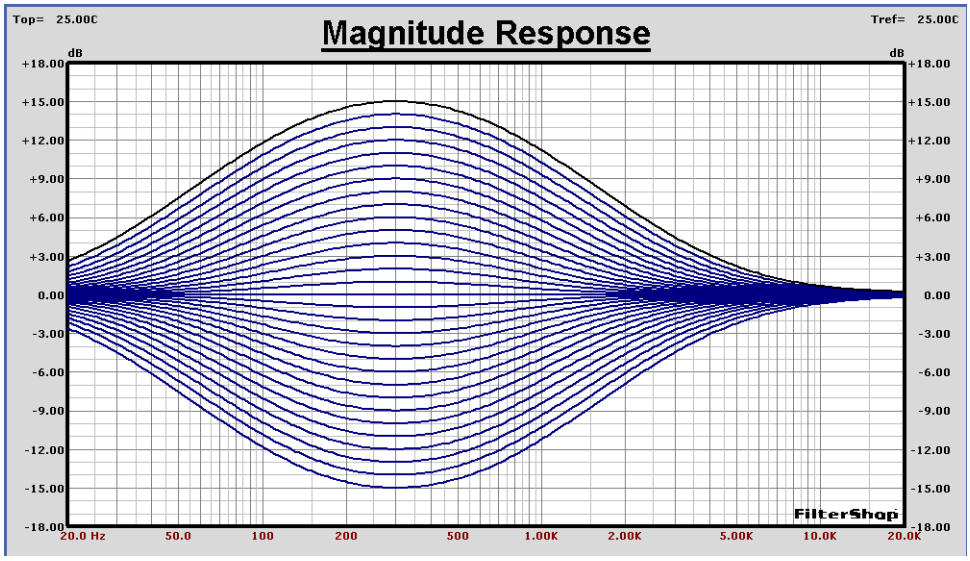


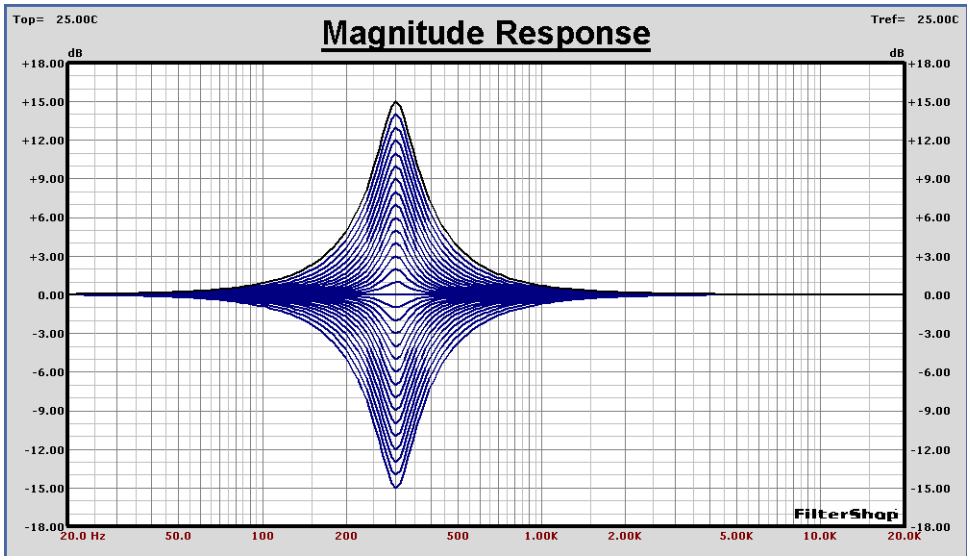
HEQ and HEQ² Filter Functions, Fo=300Hz
Fs=48kHz, Transform=Matched-Z





BEQ Filter Functions, Fo=300Hz, Oct=3.0 & 0.333
Fs=48kHz, Transform=Matched-Z





BEQ Filter Functions, Ao=+15dB, Fo=30Hz...300Hz, Oct=3.0...0.333
Fs=48kHz, Transform=Matched-Z