Graphic Equalizer Plug-In Report

Peter Rowland-Jones[1]
[1]Derby University, Audio Engineering, Derbyshire, United Kingdom
p.rowlandjones1@derby.ac.uk

Introduction

This paper discusses the design and implementation of a 7 Band Graphic Equalizer[GEQ]. This completed using object orientated code design in C++ with Visual Studio [VS]. In conjunction with this, plug-in development was streamlined using RackAfx. The combination of the two allowing the production of high quality audio effects processing.

Influence has been taken from a similar plug-in found in the digital audio workstation [DAW] 'Reaper'. Within the DAW a 7 band GEQ was available that utilised the 'Robert Bristow Johnston' cookbook [RBJ Cookbook]. This allowing simplification of 2nd order bi-quadratic (Transposed Direct Form 2) architecture within the code. Previous (Jones, 2016) impulse response tests of the plug-in showed poor representation of parameter settings in terms of magnitude over frequency, reasons for this were assessed.

Contents

1. **Background** **Pg. 3**
2. **Design** **Pg.4-5**
3. **Implementation** **Pg.6-8**
4. **Analysis** **Pg.9-11**
5. **Conclusion** **Pg.12**
6. **References** **Pg.13**

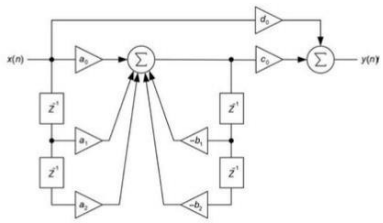
1 Background

A GEQ allows for manipulation of the frequency response of a given input signal. This being split over defined frequency ranges (bands) and allows for either a cut or boost in gain, at pre-defined frequencies. The Q for each band can either be non-constant, where as the gain increases, the bandwidth (-3dB) is proportional to a given frequency. Or constant, as gain increases, bandwidth remains independant of frequency. (Rane,D.B 2005)

The bi-quad algorithms from the RBJ cookbook allow for coefficients to be calculated with no pre-set integers. Allowing higher manipulation with code implementation. The RBJ cookbook also provides many different filter types, more than sufficient for use within a GEQ. (Johnston, RB 2016)

Making use of the peaking filter and High Pass filter types. These bands would then need to be cascaded to create the final design. (Pirkle, 2013)

Audio specific filters require a modification to the generic biquad structure that allows a mix between wet & dry signals for sample processing. Below in [Fig1.0] the addition of coefficients c0 and d0 allow this mixing. Within the plug-in, the biquad structure has been implemented in direct form II, allowing just 2 delay blocks per instance of class, lowering memory allocations in comparison to using direct form I.



[Fig 1.0]

The aims for the functionality of this GEQ were as follows,

1. Allow cut/boost of 7 individual peaking bands
2. Additional High Pass filter
3. Variable Q slider (For all bands)
4. Input/Output Controls (Volume, Mute, Bypass)
5. Visualisation of Frequency & Phase of output signal.

2 Design

To begin a one band peaking filter with VS & RackAFX was implemented. This allowing full view of sample flow, requiring a new RackAFX project and 1 slider (m_fGaindB). Once this was set, class abstraction was used to allow multiple instances of the filter.

Two constructors were implemented within the header file, one for generic instances of the class & one for setting the filter once. This then coupled with the destructor to free up memory allocation when not in use. The class itself employs 64 bit floating point (double) computation within the filter calculations , & 32 bit floating point (float) for in/out sample processing. This allowing high accuracy setting for the filter and adequate audio processing for the plug-ins use. (Redmon, 2012)

Each pre-defined variable within the .h file was then given a function within the .cpp file. This required the use of the maths functions within VS. It was found that these did not come pre-defined within VS & had to use [#_USE_MATH_DEFINES] and [#include <Maths.h>] within the .cpp file to allow the use of [M_PI] within the biquad algorithms.

Within the plug-in the constant Q method was used with a variable option that applies to all bands. This calculated with the equation found in fig.[2.0](Pirkle, 2013). The result being 2.04, the option of a slider for varying Q from 0.707 to 2.0 was given to allow smoother band integration if desired.

[Fig 2.0]

$$N = \frac{\text{Number of modules}}{10}$$

$$Q = \frac{\sqrt{2^N}}{2^N - 1}$$

The centre frequencies for each band were calculated using the International Organization for standardization equation. [Fig 2.1] (Pirkle, 2013) This requiring the centre frequency to be 1000Hz, with each further frequency referenced to it. This could be edited in future to allow for an extra user-defined variable in place of ‘number of modules’ then defining the centre frequencies for all automatically.

[Fig 2.1]

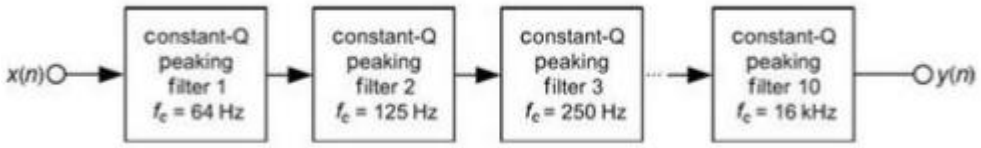
$$q = \frac{\text{Number of bands}}{10}$$

$$f_n = 1000 * 2^{n/q}$$

$$n = 0, \pm 1, \pm 2, \pm 3, \text{etc} \ldots$$

When using the bi-quad algorithms in VS, changing of coefficients in multiple bands can be computationally heavy. Considering this, best practice was to only complete this when parameters changed. These had to be independent of each other and remember their current state. The Direct Form 2 architecture of the biquad allowed good floating point characteristics and a lower amount of memory locations for delay blocks (Redmon, 2012).

The summing of individual outputs could be done in either cascade or parallel. The cascade option is commonly used within graphic equalizer digital signal processing[Fig 2.2](Pirkle, 2013) During the implementation of the design it became clear why. The inline process function allowed faster sample processing and simpler routing at the output. (Pirkle, 2013)



[Fig 2.2]

When looking to implement the parallel topology of this EQ, the filter type would have to be changed to the bandpass & include a gain value. Adding in the additional gain variable proved problematic so was not implemented. Although parallel would require referencing the input sample 8 times. Which in comparison with cascade, seemed costly to the CPU and general plug-in efficiency.

3 Implementation

To begin, the main class header file contained all variable/class instances that were to be used within the plug in, this then allowing their function variables to be altered throughout. [Fig 3.0]

```
// Add your code here: ------------
// Create instance of class
float m_fVolumedB;
float m_fVolumedB_IN;
Biquad Band1;
Biquad Band2;
Biquad Band3;
Biquad Band4;
Biquad Band5;
Biquad Band6;
Biquad Band7;
Biquad HPF;

// END OF USER CODE ---------------
```

[Fig 3.0]

Within the main .cpp file each band began with a defined set of variables to ensure it was ready to be used [Fig 3.1]. If not set, a default constructor (lowpass) was set in the class header file as a precaution. The master gain control required cooking due to the slider values being sent to the compiler being between 0-1. This value was converted to linear gain with a new variable name and referenced to the slider. Allowing representation in dB. This also takes place within the 'K' variable in the biquad class for cooking of slider values for gain of each band.

```
73  bool __stdcall Coneband::initialize()
74  {
75      // Add your code here
76      Band1.setType(bq_type_peak);
77      Band1.setPeakGain(m_fGaindB);
78      Band1.setFc(50);
79      Band1.setQ(0.707);
80      Band1.setSamplingRate(m_nSampleRate);
```

[Fig 3.1]

The sampling Rate variable is also assigned to each class instance in the 'prepare for play function'. This meant that each time the plug-in was created or audio was passed to the input buffer, sample rate would be set and each band updated. This used because the new value is required within the 'tan' functions for the calculation of biquad coefficients for frequency. This ensured the plug-in had higher flexibility for use in different systems but at a cost of using more processing power due to sending the same value 8 times. (HPF included). [Fig 3.2]

```
bool __stdcall Coneband::prepareForPlay()
{
    // Add your code here
    Band1.setSamplingRate(m_nSampleRate);
    Band1.setSamplingRate(m_nSampleRate);
    Band1.setSamplingRate(m_nSampleRate);
    Band1.setSamplingRate(m_nSampleRate);
    Band1.setSamplingRate(m_nSampleRate);
    Band1.setSamplingRate(m_nSampleRate);
    Band1.setSamplingRate(m_nSampleRate);
    HPF.setSamplingRate(m_nSampleRate);

    return true;
}
```

[Fig 3.2]

Within the ‘userInterfaceChange’ function, links to RackAFX parameters were made [Fig 3.3]. The switch yard allowed the plugin to jump to the the relevant case number, update it to the new assigned value and break out of the chain.

```
bool __stdcall Coneband::userInterfaceChange(int nControlIndex)
{
    // decode the control index, or delete the switch and use brute force calls
    switch(nControlIndex)
    {
        case 0:
        {
            Band1.setPeakGain(m_fGaindB);
            break;
        }
        case 1:
        {
            m_fVolumedB = pow(10.0, m_fMastVol / 20);
            break;
        }
```

[Fig 3.3]

The process VST audio function was used for in/out sample processing. This creating an array buffer of samples with pointers to each. This reference then used within the plug-in. ‘If’ statements were used for the pointers at the input buffers to assess mono/stereo input, allowing the use of both.

Within this function are the basic plug-in controls such as, mute, bypass & phase. These contained within the ‘while loop’ concerning if there are input samples. An ‘if, else’ statement was used to allow the mute and bypass function to be added. These linked to RackAFX controls using logic statements. The bypass allows the input pointer to reference straight to the output allowing the processing of the plug in to be skipped. The main cascasde of samples takes place by calling the process function of each band and feeding them into each other to create the output pointer.[Fig3.4]

```
//!!!!!!!!!!!!!!!!!Multiplying by master vol at the last bracket
//////// Left channel processing
*pOutputL = HPF.process(Band1.process(Band2.process(Band3.process
```

[Fig 3.4]

The phase switch required multiplying the output by a co-efficient of -1.0(Pirkle 20130. This implemented after the ‘else’ part of the statement allowing the output pointer to be referenced after all processing had been completed and before the pointers incremented.

Within the header file for the biquad class, enumerated variables were used to define filter type [Fig 3.5]. Allowing clearer & simpler reference, rather than multiple classes for each. Each type then being defined within the .cpp file for use in the calcBiquad function.

```
// create BiQuad class, and create Enumerated var
enum {
    bq_type_lowpass = 0,
    bq_type_highpass,
    bq_type_peak,
    bq_type_bandpass,
};
```

[Fig 3.5]

The inline ‘process’ function allowed use of a function in the .h file for streamlined processing[Fig 3.6]. This function takes the input sample, applies the current coefficient values & updates the delay blocks for the next sample & returns an output sample.

```
//take the input float samples and process t
inline float Biquad::process(float in) {
    double out = in * a0 + z1;
    z1 = in * a1 + z2 - b1 * out;
    z2 = in * a2 - b2 * out;
    return out;
}
```

[Fig 3.6]

The implementation of the biquad equations are based on the RBJ cookbook & an example found online (Redmon, 2012). This gave a good example on implementation within c++.The design equations are based on that of a second order peaking filter with constant Q. The gain values from RackAfx are cooked using linear gain conversion and applied as the V variable. The K variable then normalises the defined frequency (Radians) to be used within the function.

A switch yard was then used depending on the defined filter type at initialisation of the class instance[Fig 3.7]. These new values are used during the Biquad::process function and reflect the new changes made within the plugin to gain.

```
void Biquad::calcBiquad(void) {
    float Fstemp = Fs;//temporary value stored in samp
    double norm;
    double V = pow(10, fabs(peakGain) / 20.0);
    double K = tan((M_PI * Fc/Fstemp)/2);
    switch (this->type) {
    case bq_type_lowpass:
        norm = 1 / (1 + K / Q + K * K);
        a0 = K * K * norm;
        a1 = 2 * a0;
        a2 = a0;
        b1 = 2 * (K * K - 1) * norm;
        b2 = (1 - K / Q + K * K) * norm;
        break;

    case bq_type_highpass:
        norm = 1 / (1 + K / Q + K * K);
        a0 = 1 * norm;
        a1 = -2 * a0;
        a2 = a0;
        b1 = 2 * (K * K - 1) * norm;
        b2 = (1 - K / Q + K * K) * norm;
        break;
```

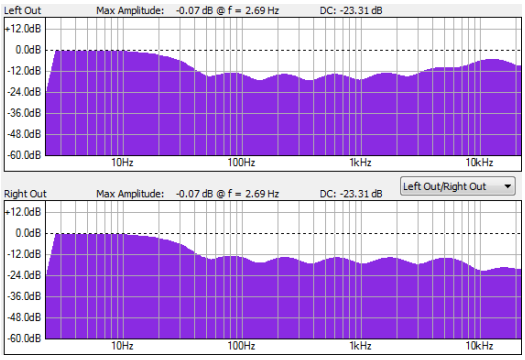
[Fig 3.7]

Below is the final GUI interface for the plugin[Fig 3.8]. Unfortunately a bug was found when using in reaper causing the output meter level to not function properly.

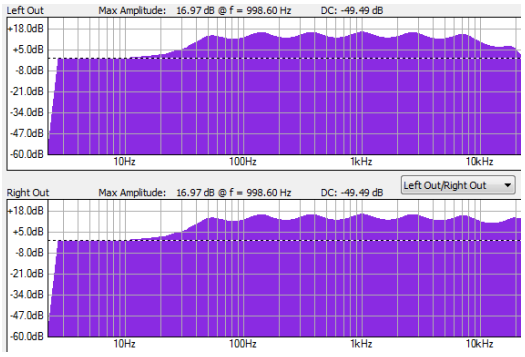
[Fig 3.8]

4 Analysis

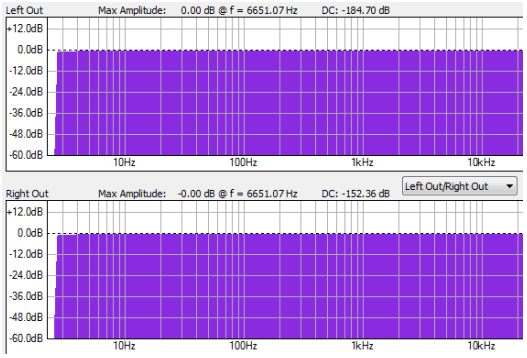
In Fig 4.1 the frequency analyser tool in RackAFX was used to take an impulse response of the system to plot the magnitude response of the plug in. All bands were set to maximum gain (+/- 12dB) to assess the effectiveness of the cascade and constant Q. The resulting plots are consistent with that of a constant Q graphic EQ. This shown with the slight drop in magnitude towards higher frequencies due to less summation between bands. It is clear that there is a bug in the right hand channel which does not exhibit these characteristics. As is clear the combination of bandwidths results in a higher magnitude approaching 18dB when Q is set to 2.0 and 30dB with Q set at 0.707. This is one reason for the master gain control to allow compensation for band interaction. Below in [Fig 4.2] is a plot of all bands set to 0dB to show that the resulting flat magnitude response when the plug in is not being used.



[Fig 4.1 – (-12dB Gain – Q -2)]

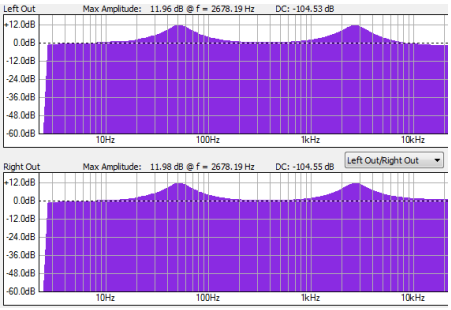


[Fig 4.1 – (12dB Gain – Q -2)]

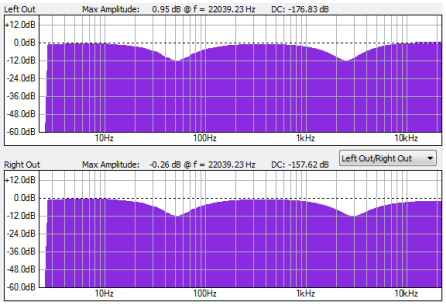


[Fig 4.2 – Flat Magnitude – Gain -0dB]

In [Fig 4.3] a test was completed to check that the constant Q was functioning properly. As is visible the bandwidth of each is the same regardless of frequency. This indicating a constant Q across all frequencies rather than a -3dB point skirt Q.

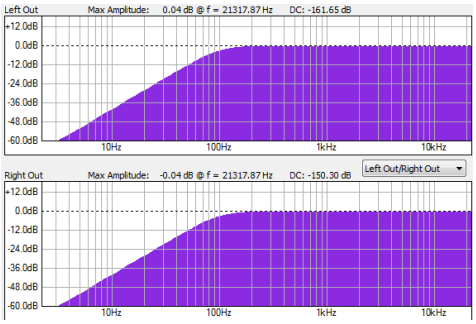


[Fig 4.3 – Constant Q Boost]



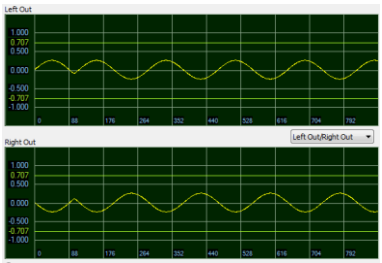
[Fig 4.3 – Constant Q Cut]

To test that the high pass filter was implemented correctly, its frequency was set to 100Hz, the resulting magnitude plot shows the cutoff (-3dB) point to be at 100hz. This is also indicative that the filter type function works as expected. [Fig 4.4]



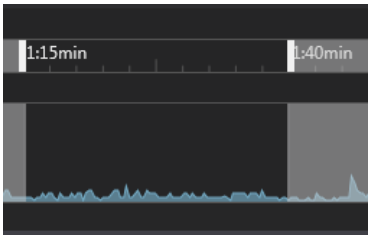
[Fig 4.4 HPF 100hz]

Using the oscillator tool in RackAFX, the phase switch was tested[Fig 4.5]. To do this only one channel was checked while pushing the left channel to represent the input. Here you can see a phase shift of 180 degrees when implemented. The first two peaks may be due to a bug causing a click when toggling the switch.



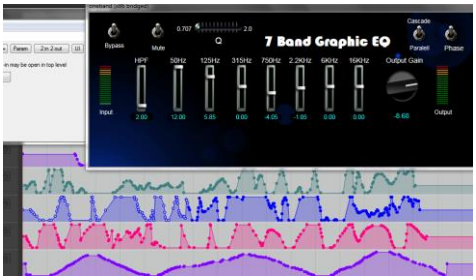
[Fig 4.5 Phase Switch]

A diagnostics test of the plug-in was completed to assess cpu usage of the current code. Fig 4.6 shows a 30second clip tracking overall CPU processor usage while the plugin was running. With a spike of 15% at 1:24seonds. This was when using the volume control, which may be due to changing multiple Biquad.process variables in real-time causing a bottleneck.



[Fig 4.6 – CPU Usage]

When using the EQ in reaper it was noticed that when applying extreme parameter changes with automation, bugs occurred [Fig 4.7]; This due to zipper noise. To overcome this a de-zipper class could be implemented using a low pass filter to smooth the changing of parameter values.



[Fig 4.7 Automation Test]

5 Conclusion

Overall the plug-in has achieved majority of the objectives first set. Allowing manipulation of 7 bands plus a high pass filter, a gain range of 24dB for each & variable Q range. Overall efficiency of the plugin in terms of CPU usage remains relatively low, due to inline processing, cascade topologies & class development. The quality of the audio processing is clean, aside from the zipper noise when used at extremes; which although unlikely for a graphic EQ, is still a key bug that needs fixing. Future work would be to develop the biquad calculations to ensure exact values are met, fix the parallel function to allow gain changes of the bandpass filters. The addition of a frequency analyser within the plug-in would require a larger GUI and is not as necessary as first thought. Additional options would also be to add an all pass filter type to achieve linear phase across the audible frequency range, & adding another set of sliders to allow full stereo processing rather than requiring two instances of the plug-in.

References

1. Bristow Johnston, R. (2016). Cookbook formulae for audio EQ biquad filter coefficients. Available: http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt. Last accessed 12/01/2017.
2. Jones, P (2016). CW1 Graphic EQ Investigation, Derby University.
3. Pirkle, W (2013). Designing Audio Effect Plug-ins in C++. Burlington: Focal Press. 18,189-196.
4. Pirkle, W. (2016). Rack AFX . Available: http://www.willpirkle.com. Last accessed 06/01/2017.
5. Rane, D,B. (2005). Constant-Q Graphic Equalizers. Available: http://www.rane.com/note101.html. Last accessed 05/01/2017.
6. Redmon, N. (2012). Biquad C++ Sourcecode. Available: http://www.earlevel.com/main/2012/11/26/biquad-c-source-code/. Last accessed 10/01/2017.