RESEARCH AND DEVELOPMENT OF A VST PLUG-IN FOR THE AUTONOMOUS POST PRODUCTION OF A STEREO PIANO RECORDING

MICHAEL FISHER

A THESIS SUBMITTED TO THE UNIVERSITY OF HUDDERSFIELD IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE BY RESEARCH

THE UNIVERSITY OF HUDDERSFIELD

JANUARY, 2012

ABSTRACT

This thesis describes the research and development of a VST Plugin capable of the autonomous post production of stereo piano recordings within the context of a popular music production. The use of software to make and manipulate music is the focus of the Music Technology industry. Software utilities do not attempt to replace the need for the creativity of technologists but instead allow them to perform increasingly complex methods of production. Automated mixing is a popular area of research and the prototype developed demonstrates that software is able to make creative decisions within production. The piano is a common instrument within popular music and was chosen to be the subject of the research due to the diverse range of sounds it produces. Development involved the implementation of algorithms for signal analysis with the Fast Fourier Transform, frequency domain transformation with digital filters and time domain transformation with dynamic range compression. These procedures were combined to allow the plugin to analyse a signal and calculate effects parameters in real time, resulting in the dynamic and autonomous application of equalisation and compression processing to a piano recording.

LIST OF FIGURES

DISC CONTENTS

/JUCE_VST.dll

- The VST Plugin DLL. It has been tested on the following system:
  - Cubase 5.1
  - Windows 7 Ultimate x86
  - Pentium Dual-Core CPU @ 2.2GHz
  - 3GB RAM
- To use it, add the following directory to the Cubase/Nuendo plugin paths:

  [DISC]/VST Development/JUCE_VST__FFT/Builds/VisualStudio2008/Release/

/Fisher M U0656660 - MSc Thesis - Jan 2012.pdf

- The electronic copy of the thesis

/Project

- /AES
  - The Poster, Teaser Slides and Paper Handout used when presenting the research at the 130th AES Convention 14th-16th May
- /Research
  - The two Visual Basic Prototypes created to explore signal analysis principles.
- /Source
  - All of the C++ source used, including external libraries and JUCE.
- /Testing
  - /AB Tests
    - The audio samples and plugin settings screenshots used to carry outthe AB tests demonstrating the use of polynomial regression on FFT spectra.
  - /Audio Samples
    - The audio samples used to carry out the testing.
  - /Data Outputs
    - The raw CSV data output by the plugin.
  - /Listening Test
    - The samples used with the online listening test.
  - /Recording Project
    - The Cubase 5 project used to record all audio samples.
- /VST Development
  - /JUCE_VST__FFT
    - The Visual Studio 2010 project used to develop the plugin.

/Thesis

- /Sections
  - The sections of the thesis as written using Microsoft Word 2010.
- /Videos
  - The videos of the plugin in operation referenced in the thesis. Please note that the audio is included to give an idea of what is being run through the plugin - it is not in sync.

COPYRIGHT NOTICE

SECTION I

BACKGROUND

I.1

Introduction

The initial idea informing this research was developed in 2009 during a research project carried out by the author to determine optimal recording and mixing techniques for a piano within the context of a modern popular music production (Fisher, 2010). The outcome was a set of guidelines covering microphone choice and placement and corrective mixing procedures for both grand and upright pianos. The research demonstrated that both subjective and objective criteria are used to evaluate the success of a production. The recording process was found to be mostly subjective – the criteria that constitute a successful piano recording vary between engineers and applications and are preferential, even if they are informed as much as possible by objective reasoning. In contrast it was discovered that the corrective post production procedures used to transform a piano for inclusion in a popular music production were fairly standard; that is the majority of sources used very similar techniques. A significant component of the similarities was the desire within modern popular music to achieve uniformity; perfect timing and pitching, low dynamic range and constant tonal properties. The criteria for mixing procedures were not so influenced by individuals and subjectivity, instead being determined collectively by the requirements of listeners of popular music and current trends within the production community. It is this objectivity and centralisation of criteria of success that lead to the question of whether the mixing process could be carried out autonomously by a piece of software.

Audio mixing techniques can be broadly categorised into creative and corrective processes. Creative processes are artistic in nature and include the use of delay based effects such as echo, reverb and chorus. Their purpose is to artistically transform the sound in order to enhance a composition. Corrective processes are used to remove undesirable characteristics from a recording, for example removing resonant frequencies from ringing drums, emphasising low frequencies in a bass guitar recording, or reducing the range of dynamic change over the length of a recording; their purpose is to correct a composition.

Mixing for popular music involves careful regulation of the tonal and dynamic elements of each instrument to give the engineer as much control as possible over each component of a mix. The tonality of an instrument can be controlled using equalisation (EQ); multiple digital filters connected in series, each passing, stopping, cutting or boosting a specified frequency range. EQ would be used to implement the above example of removing resonant frequencies from ringing drums by isolating the resonant band of frequencies and attenuating them. Dynamics can be controlled using Dynamic Range Compression (DRC), commonly referred to as compression. Compression in its most basic form controls the dynamic range of a piece of audio by attenuating a signal when it exceeds a set threshold of amplitude. The threshold and ratio of change in input level to change in output level can be set to control the amount of attenuation, as can the attack and release times controlling the rate of change of attenuation. It is these two effects which are most commonly used to perform corrective post-production, and are most commonly implemented using software plug-ins. This research aims to streamline the corrective mixing process for a piano by developing one quick to use plugin which autonomously applies digital effects.

Over the past two decades recording studios have been involved in a rapid transition from the use of analogue hardware to digital software; microprocessors now perform the tasks once performed by transistors and vacuum tubes. This developmental link with technology has had a profound effect on signal processing; analogue hardware provided a platform for the modelling of mathematical theory with circuitry, whereas digital software provides the ability to create software which almost perfectly implements mathematical theory.

It is these digital advantages that have led to a very large proportion of modern recording studios recording and mixing using a digital medium, such as a Digital Audio Workstation (DAW) software package. The basic function of a DAW is to simulate an entire analogue recording studio – a mixing desk, recording medium, routing facilities and audio processing capabilities. DAWs incorporate the use of "plug-ins" as inserts in the signal chain to implement a wide variety of real-time digital audio effects. There are several software companies that produce DAWs, the three most notable being Avid, Apple and Steinberg, with all three using a different format of plug-in. Stenberg produce the *Cubase* and *Nuendo* DAWs, both implementing their VST (Virtual Sound Technology) plug-in environment. The VST platform allows the implementation of real-time effects within a piece of software used extensively throughout the world to record and mix music, and is therefore an ideal environment for the proposed software.

The implementation of mixing processes such as equalisation and compression using VST plugins is achieved through the setting of parameters. For example, an EQ plugin allows the characteristics of each filter to be changed by setting the cut-off frequency and gain parameters, whilst a compression plugin will provide control of threshold level, ballistics timing and overall output. Most commonly an engineer will spend time setting the parameters of a plugin, which then remain static throughout playback of the music, however it can also be useful to have parameters change throughout playback to reduce or increase the impact of an effect. This can be achieved either by splitting the recording of an instrument into multiple tracks and processing each separately or through the use of automation. A DAW can be used to automate plugin parameters by graphing the value of a parameter against time:

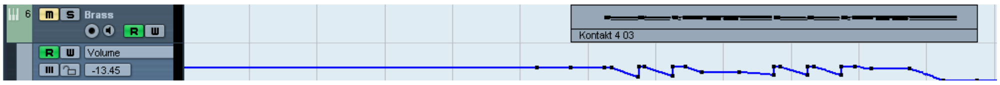


Figure I.1.1 - Volume automation in Steinberg Cubase 5

The graph created in Figure I.1.1 shows automation create by drawing lines using a mouse, which is a particularly time consuming process, as is making changes in the future. A quicker approach is the use of a physical interface connected to the DAW, for example via MIDI, which allows knobs, sliders and buttons to be mapped to particular parameters. This is a digital implementation of the technique of "riding" the faders of an analogue mixing desk to automate the volume of a particular instrument and introduces humanisation to the changes in parameters.

Automation is a useful tool when mixing piano recordings as it allows for different tonalities to be created throughout a performance. The wide range of frequencies and dynamics produced by a piano mean that mixing it for popular music mostly involves attenuating parts of the spectrum to allow it to interact well with other instruments. The desired amount of attenuation depends on how the piano is played, which will change over a performance and hence so will the amount of attenuation. Manually automating the EQ and compression effects used on a piano is extremely time consuming for example each parametric filter used will have at least three parameters to automate – cut-off frequency, gain and Q factor, and it is not uncommon to use three or four filters per instrument. Any future change of the automation of one parameter will then have a knock on effect on other parameters requiring them to be updated as well.

This research aims to develop a plugin which automatically applies EQ and compression to a piano recording resulting from experimentation into methods of changing effects parameters. The plugin will be capable of intelligently responding to the changing characteristics of a performance and update parameters to compensate. This will be achieved with the combination of signal analysis, compression and equalisation:

1. A frequency domain representation of the signal will be created using Fourier analysis and polynomial regression.
2. The amplitude bins of set frequency ranges will then be used as the input to individual envelope followers.
3. The gain reduction output of each envelope follower is then used to control the gain and cut-off frequency parameters of a parametric filter.

The design includes several novel contributions to the field of automated mixing;

- The use of polynomial regression to average and shape a frequency spectrum plot,
- The combination of compression and equalisation processes into a single effect,
- The ability to autonomously apply post production processing to a piano recording.

I.2

Post Production Systems

I.2.1 Signal Analsysis

Signal analysis tools help mixing engineers to make informed choices when mixing an individual instrument or assessing the overall quality of a mix. An example of their use is to provide a method of analysing the low frequency range of an instrument in a poor quality listening environment where low frequencies are not reliably reproduced (Senior, 2011); the plugin allows the user to see how the low frequencies are behaving. Another example is to compare the spectrums of two instruments which appear to be competing for space to inform how they may be equalised to better interact with one another.

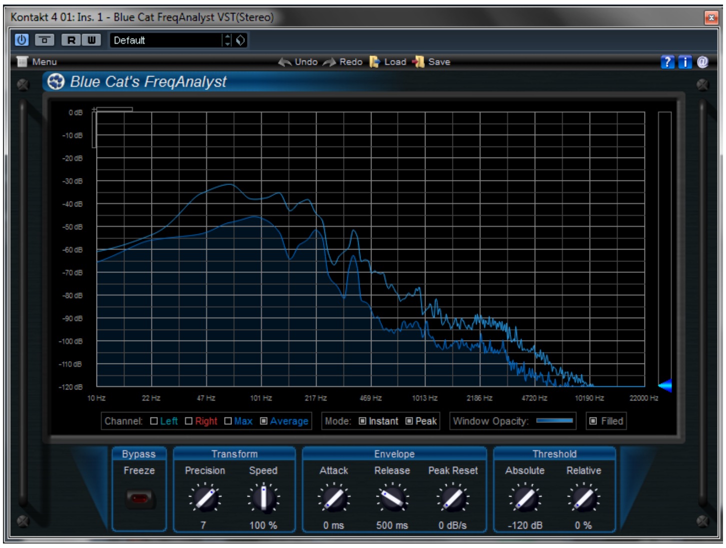


Figure I.2.1

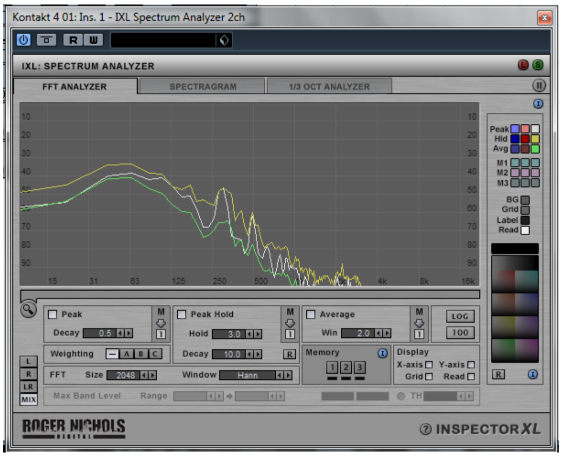


Figure I.2.2a

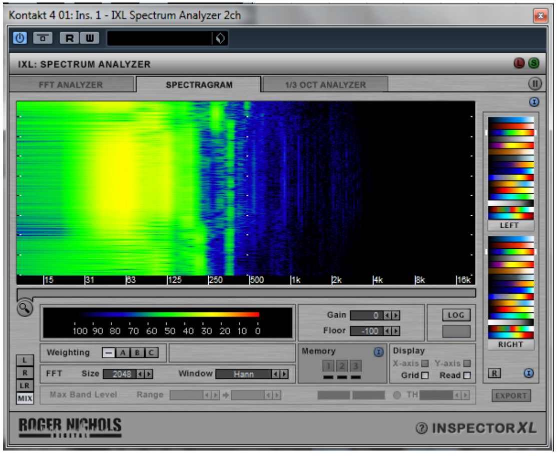


Figure I.2.2b

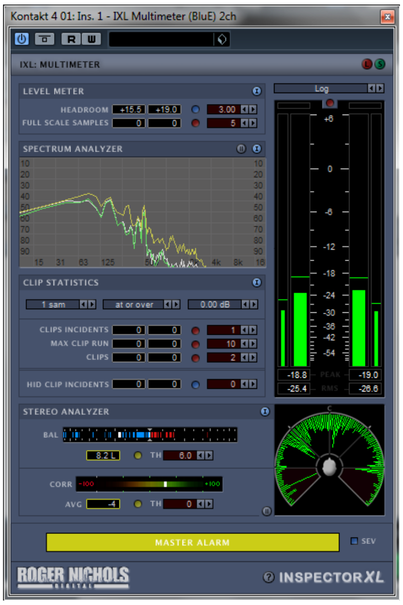


Figure I.2.3

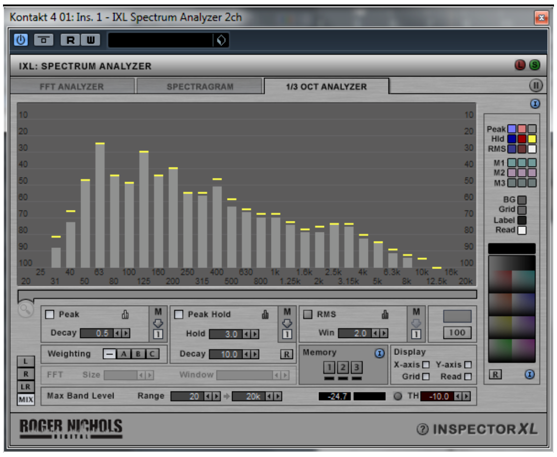


Figure I.2.2c

There a large number of plugins available offering signal analysis. Figure I.2.1 shows BlueCat Audio's FreqAnalyst, a simple FFT analysis tool. The plot displays either each stereo channel or a summed average of instant or peak level and the ballistics of levels can be set using the envelope control. This plugin demonstrates the basic functionality provided by spectral analysis plugins; a real-time FFT display with simple controls to transform it. Figure I.2.2 shows the more complex IXL Spectrum Analyzer, an FFT analyser which supports three different plot types; a basic FFT plot, a spectrogram and a third-octave analyser. Figure I.2.2a shows the basic FFT view which offers peak, peak hold and average plots. Figure I.2.2b shows the spectrogram plot; frequency is plotted on the x-axis, time on the y-axis and colour is used to denote amplitude as shown by the scale underneath the main plot. The plot scrolls upwards through time showing the last two to three seconds of audio within the window. Figure I.2.2c shows the third-octave analyser view which uses bars for each amplitude bin to display the spectrum. Yellow markers on the top of each bar show peak levels. Each view performs the same task; displaying spectral information over time to the user, and the view used is an individual preference. Figure I.2.3 shows the IXL Multimeter, a full signal analysis suite. Alongside the FFT plot statistics collected on the clipping behaviour of the signal are shown as well stereo analysis. The main stereo analysis tool uses colour to denote amplitude; blue for the left channel and red for the right, with colour brightness increasing with amplitude. The horizontal plot denotes the stereo image and the colour displays are used to inform the user of the volume of each part of the image. Another plot shows how the stereo image of signal will be perceived by a listener; a head is displayed at the centre and the horseshoe shape around it denotes physical hearing range. The green bars grow inward from the outer edge to show volume and hence where the listener would perceive the signal to be coming from.

EQ plugins are commonly parametric implementations and can be categorised into two main types; modern digital designs with a GUI providing a graphical display of the filters being applied, or outboard modelling designs with a knob layout UI mimicking that used on a mixing desk.

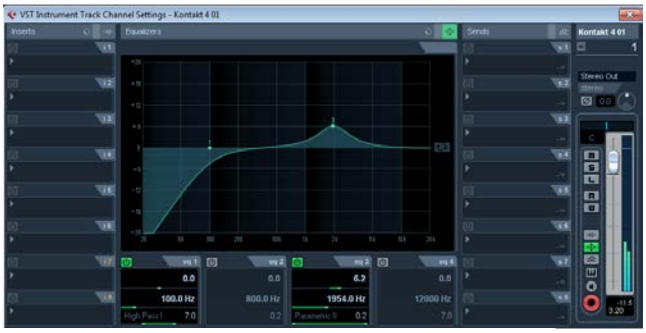


Figure I.2.4

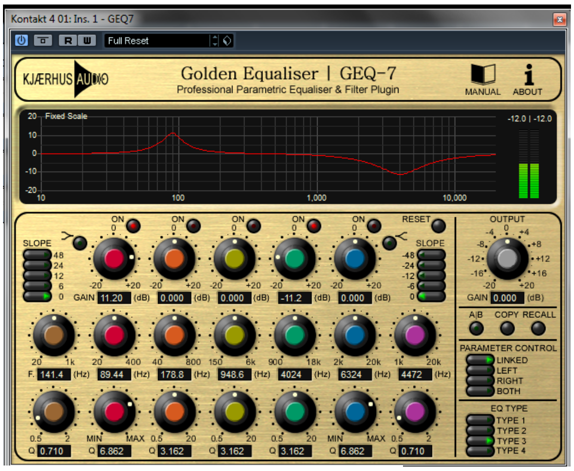


Figure I.2.5

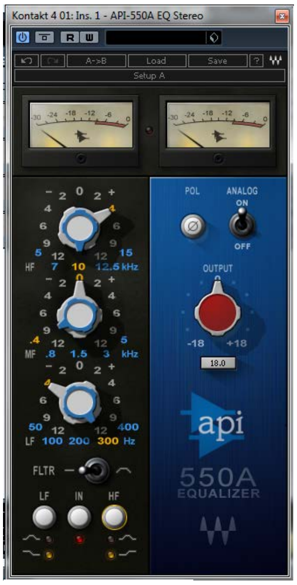


Figure I.2.6

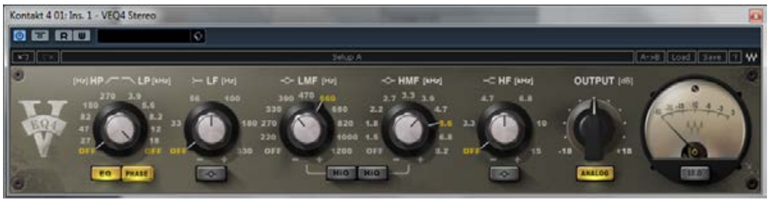


Figure I.2.7

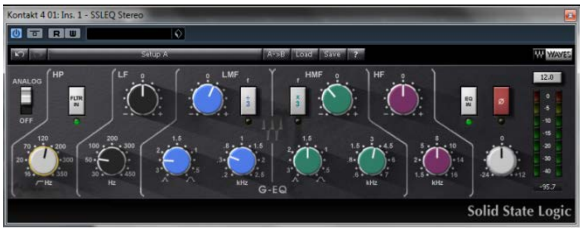


Figure I.2.8

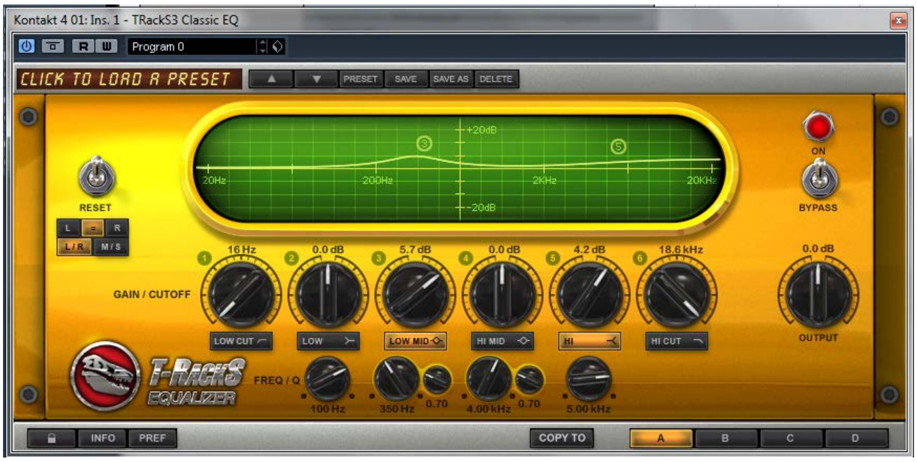


Figure I.2.9

Figure I.2.4 Steinberg - Cubase EQ

The stock equalizer provided with Steinberg Cubase. Parameters can be manipulated either with the graphical representation or the sliders underneath. Four bands are available and any filter type can be selected for each from high pass, low pass, high shelf, low shelf and parametric.

Figure I.2.5 Kjærhus Audio - Golden GEQ-7

Filter parameters are manipulated using the bank of knobs and the resulting contour displayed graphically. Rather than selectable bands seven filters are provided; low and high shelving filters and five parametric filters.

Figure I.2.6 Waves – API 550A

Filter parameters are controlled with double height knobs. The upper knob controls centre frequency and lower knob controls gain. Centre frequencies are selected from a limited set. Three bands are provided; 50Hz-400Hz, 400Hz-5kHz, 5kHz-15kHz. The low and high bands can be switched between parametric and shelving types.

Figure I.2.7 Waves – VEQ4

Similar to the API 550A but with more centre frequency options. Low pass and high pass filters can be set along with low and high shelving filters and low-mid and high-mid parametric filters.

Figure I.2.8 Waves – SSL EQ

Offers a model of the classic Solid State Logic mix bus EQ. Freely selectable centre frequencies are provided for the same filter set as the VEQ4 less the low pass filter. Polarity inversion is also available.

Figure I.2.9 T-RackS - T-RackS3 Classic

Offers the common filter set seen in these examples; low and high pass, low and high shelving and low-mid and high-mid parametric filters. Filter parameters are set using the graphical display and can be fine-tuned using the knobs below.

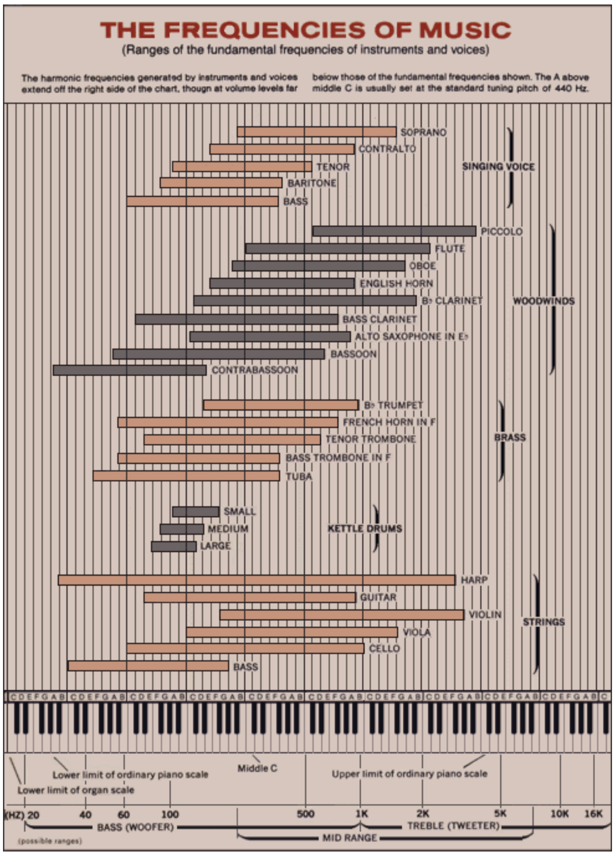


Figure I.2.10

The plugins offer very similar functionality; a set of filters connected in series covering the expected regions of the frequency spectrum. The plugins with knob based user interfaces usually have a vintage design as they are programmed to simulate a specific piece of analogue hardware, for example the API 500A. Hence why these plugins tend to restrict choice of cut-off frequency and gain; the analogue circuitry's frequency response is modelled with specific parameters and it is impractical to do this for every Hertz and Decibel.

The most common format seen in the examples above is to have several filters available operating in ranges that together cover the whole frequency spectrum but individually control significant bands; low frequencies below 300Hz, low-mid frequencies between 300Hz and 1kHz, high-mid frequencies between 2kHz and 7kHz and high frequencies above 8kHz. For most instruments the low frequencies would be cut to provide room in the mix for instruments that intentionally produce low frequencies, for example drums and bass guitars. This is almost always carried out with a high pass filter and/or a low shelving filter (Mellor, 1995). The low-mid frequencies contain the majority of fundamental frequency content for the common popular music instruments; guitars, pianos, keyboards and low vocal ranges. The high-mid frequencies contain the harmonic frequency content produced by these instruments and impact greatly on their tonal qualities, for example the presence of a vocal recording usually occupies 3-4kHz (Mellor, 1995). Different instruments have very different frequency content in these two bands, hence why filters covering the mid frequencies are usually parametric to allow the maximum amount of control (Mellor, 1995); frequency centre, frequency range and gain can be carefully controlled. The high frequencies contain the frequency content from cymbals and other percussion instruments such as tambourines and the higher harmonics of the whole mix. They can be treated like the low and mid frequency bands using shelving or parametric filters; the whole range can be attenuated if not relevant to the instrument to leave room for others, for example a bass drum, or specific frequencies can be controlled to add or remove “shine” and “brilliance”.

Instrument	Characteristic and Frequency Location
Kick Drum	Bottom or depth is usually found in the 60-80Hz region; Slap at 2.5kHz.
Snare Drum	Weight, fatness or body at about 240Hz; Bite at 2kHz; Crispness at 4-8kHz.
Hi-hat	'Gong' at 200Hz; Shimmer at 7.5-12kHz.

Cymbals	'Clunk' from 100-300Hz;
	Ringing overtones at 1-6kHz;
	Sizzle at 8-12kHz.

Rack Toms	Fullness around 240Hz; Attack at 5kHz.
Floor Toms	Fullness around 80-120Hz; Attack at 5kHz.
Congas	Resonance around 200-240Hz; Slap at 5kHz.

Bass Guitar	Bottom at 60-80Hz;
	Attack or 'pluck' at 700Hz to 1kHz;
	'Pop' at 2.5kHz.

Electric Guitar	Mains hum at 50Hz (UK) or 60Hz (US);
	Fullness at 240Hz;
	Bite at 2.5kHz.

Acoustic Guitar	Bottom or weight at 80-100Hz; Body around 240Hz; Clarity from 2-2.5kHz.
Electric Organ	Bottom from 80-120Hz; Presence at 2.5kHz.

Acoustic Piano	Bottom from 80-120Hz;
	Presence between 2.5 and 5kHz;
	Attack around 10kHz;
	'Shrillness' at 5-7.5kHz.

Horns	Fullness at 120-240Hz; Shrillness from 5-7kHz.

Brass	Warmth at 200-400Hz;
	'Honk' at 1-3.5kHz;
	'Rasp' at 6-8kHz;
	Shrillness at 8-12kHz.

Solo Trumpet & Sax	Warmth at 200-400Hz; Nasal tones at 1-3kHz.

Strings	Fullness at 200-300Hz; 'Scratch' (bow and string noise) from 7.5-10kHz.

Vocals	Fullness around 120Hz;
	'Boom' around 200-240Hz;
	Presence at 5kHz;
	Sibilance from 7.5-10kHz.

Figure I.2.11

Compression plugins can be categorised in the same was as EQ plugins; digital designs with graphical user interfaces and outboard modelling designs with vintage styled interfaces:

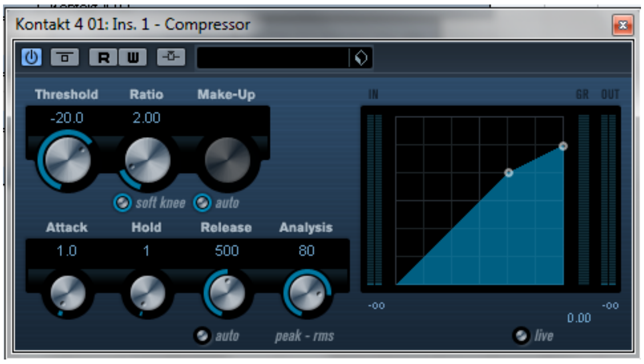


Figure I.2.12

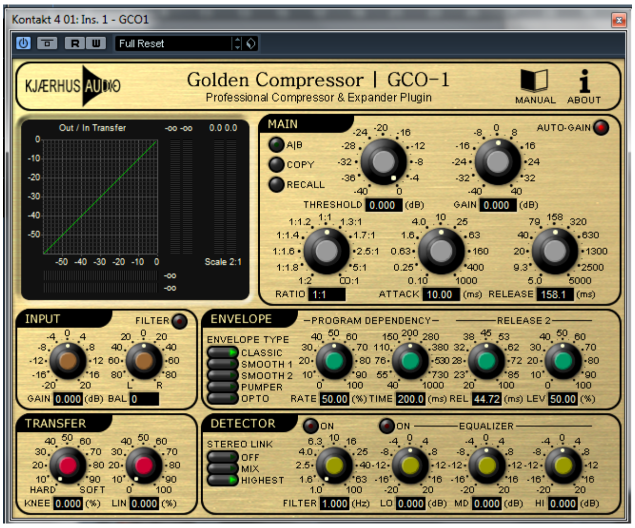


Figure I.2.13

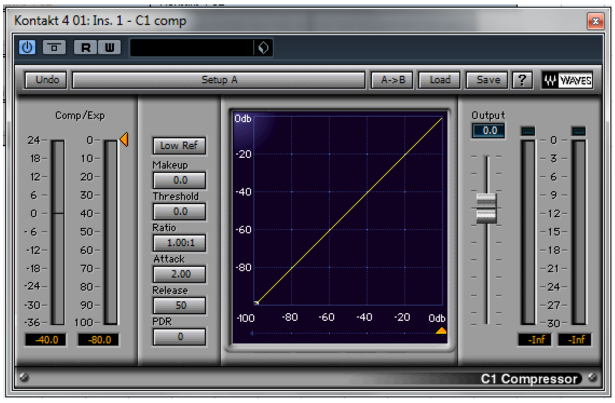


Figure I.2.14

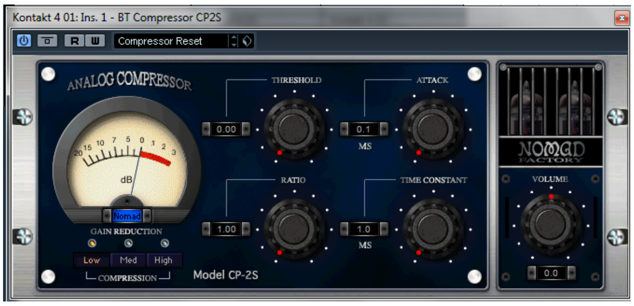


Figure I.2.16

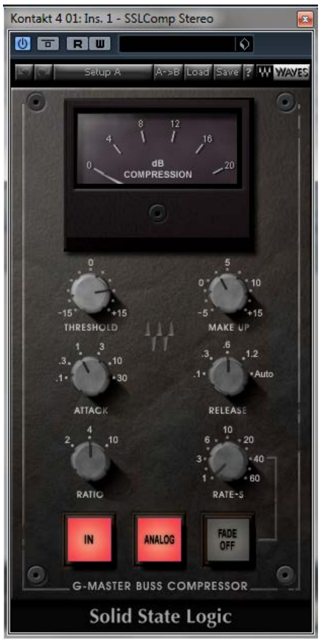


Figure I.2.15

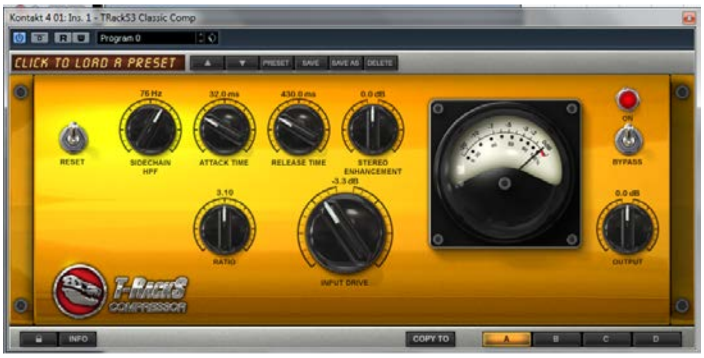


Figure I.2.17

Figure I.2.12 Steinberg – Cubase Compressor

The stock compression plugin supplied with Cubase. It offers basic digital compressor functionality; threshold and ratio control, attack and release timing and make-up gain. Extra features include hold time, automatic make-up gain, automatic release time, hard/soft knee control and blending between peak and rms analysis. A graphical display of the resulting compression curve is provided.

Figure I.2.13 Kjærhus Audio – Golden Compressor GCO-1

Basic functionality is combined with detailed control over analysis and the envelope. Level detection customisation includes stereo linking and equalisation. A choice of selectable envelopes based on transient response is supplied. Also included is hard/soft knee control and input gain level. Graphical display of compression curve is provided.

Figure I.2.14 Waves – C1 Compressor

Basic functionality with a very visual graphical user interface; threshold levels are set using a slider on the input level meter; the effect of current settings is determined by the joint compression curve/level meter dominating the user interface.

Figure I.2.15 Waves – SSL Compressor

A hardware modelling plugin simulating the classic Solid State Logic mix bus compressor. Threshold and make-up gain are freely chosen but ratio, attack time and release time are selected from limited options, although automatic release time is included. A basic level meter showing gain reduction is provided.

Figure I.2.16 Nomad Factory – BT Compressor CP25

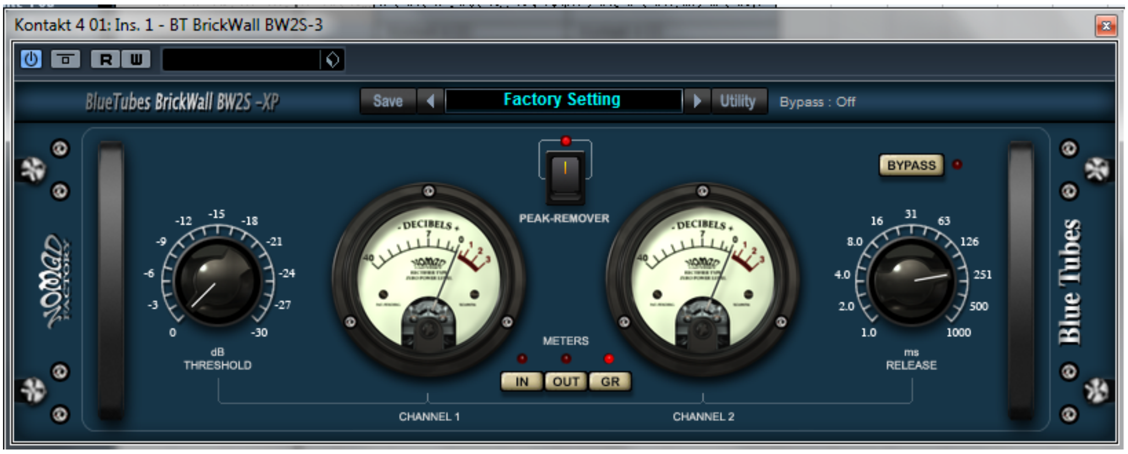
This plugin offers a model of a tube compressor. Threshold and ratio controls are provided as normal but ballistics are controlled with an attack time and time constant parameter; effectively a release time. A general level of compression is selectable between low, medium and high which is assumed to control envelope characteristics. Gain reduction meter supplied.

Figure I.2.17 T-RackS – T-RackS Classic Compressor

A modern digital compressor with a vintage styled GUI. A threshold parameter is not provided, instead it is kept at 0dB and an input drive control is used. Timing and ratio controls are provided as usual, as is a pre-level detection high pass filter. A gain reduction meter is provided.

Functionality is as common between the above compression examples as it is for the EQ examples. The broad categorisation of modern digital and vintage modelling reflects the two main desirable outputs of a compression plugin; careful control over parameters to apply a small amount of compression produce and a clean, modern sounding result, or the colouration and more severe compression provided by outboard hardware to produce a vintage sounding result

. Compression is used to control the transients of a recording to emphasise either the attack or body characteristics. The attack of a sound is emphasised by using long attack and release times that allow a transient through the plugin untouched but then attenuate the body of the sound, allowing make-up gain to boost the transient (Preve, 2008). By far the most common usage within modern popular music is the emphasis of attack, particularly on drums, to produce a "punchy" sound with exaggerates rhythm. The body of a sound is emphasised using short attack and release times and a threshold level set to be lower than the transient peak level and higher than the body average level; the transient is attenuated but the envelope falls back as the body passes through the plugin, allowing the make-up gain to increase the level of the body (Preve, 2008). Both of these implementations have the effect of shaping the characteristics of a recording as well as reducing its dynamic range. Compression is also widely used across an entire mix to reduce the dynamic range of a composition as a whole and hence increase its loudness (Robjohns, 1999). Loudness is an important quality of mainstream popular music as it increases the “energy” of a composition and also sounds better when listened to through headphones; it is increasingly common to test a mix with several sets of headphones as well as several sets of loudspeakers due to the recent boom in portable personal music players (Walker, 2007).

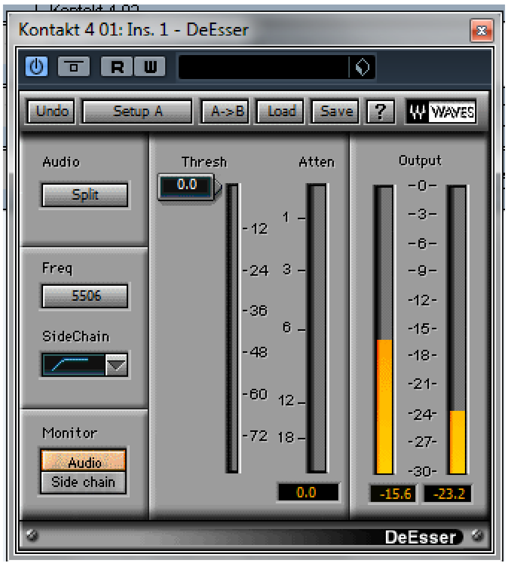


Kontakt 4 01: Ins. 1 - BT BrickWall BW2S-3

Factory Setting

Bypass : Off

Figure I.2.18



Kontakt 4 01: Ins. 1 - DeEsser

Audio: Split

Freq: 5506

Monitor: Audio

Thresh: 0.0

Figure I.2.19

The compression concept is also utilised in other forms; the two examples above show a brick wall limiter and de-esser. A brick wall limiter employs a theoretically infinite ratio to prevent a signal rising above a specified amplitude. The Nomad Factory BT BrickWall BW2S-3 shown in Figure I.2.18 provides a threshold parameter controlling the maximum peak amplitude and a release time parameter. Brick wall limiting is typically used to create the “pumping” or “breathing” sound associated with over-compression. A de-esser reduces the sibilant characteristics of a vocal recording by compressing the frequency band which typically contains the offending harmonics; between 4kHz and 10kHz (Senior, 2009). The Waves De-Esser shown in Figure I.2.19 allows control of the centre frequency and threshold.

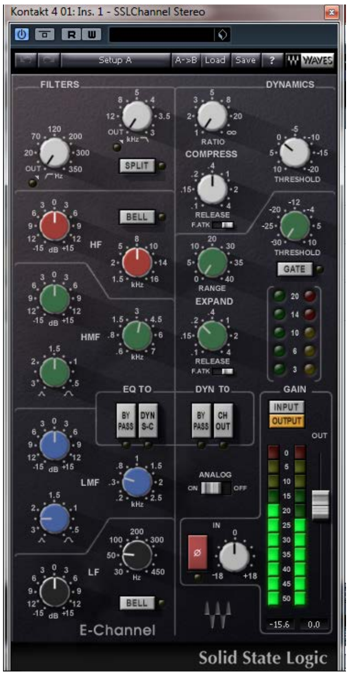


Figure I.2.20

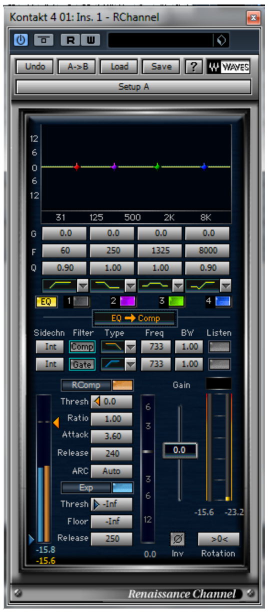


Figure I.2.21

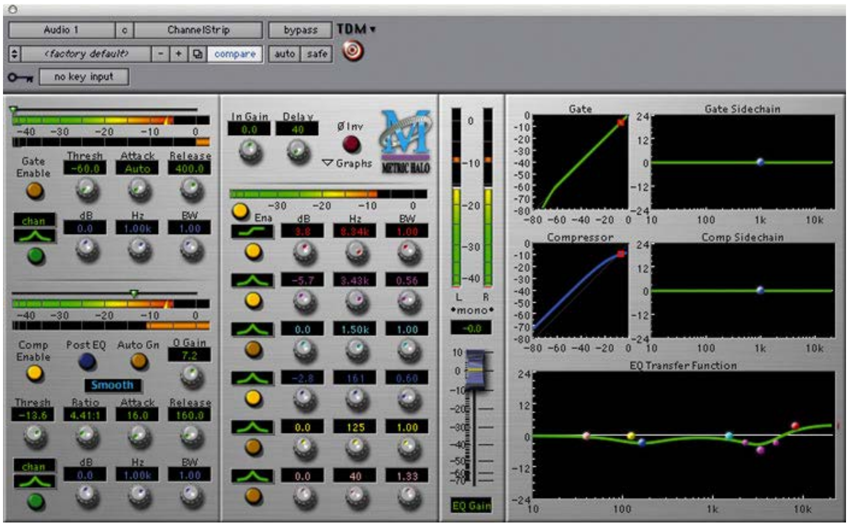
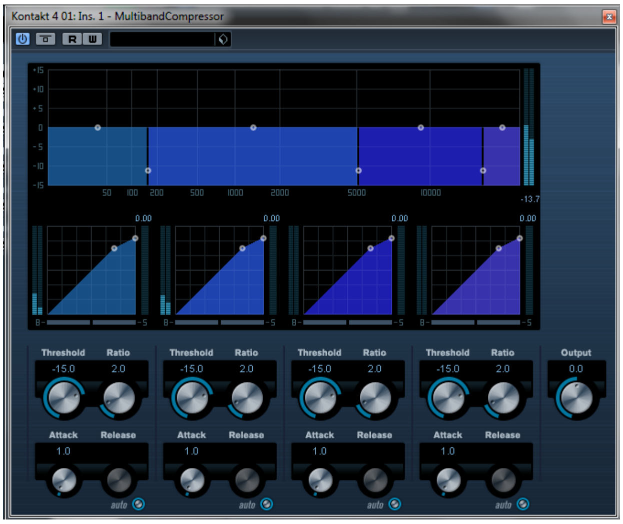


Figure I.2.22

The main purpose of a channel strip plugin is to reduce the number of individual plugins required to produce a desired result. They offer a reduced CPU load to employ all of the included effects than would be possible with one effect per plugin. CPU load is an important consideration when mixing a popular music composition as most tracks will require at least one of the processes provided by the channel strips shown above and most up to date DAWs have the capability to run hundreds of audio tracks at once; all of this real-time processing requires significant resources.

The effects included within a channel strip plugin commonly include noise gating, compression, equalisation, phase inversion and level metering. A noise gate mutes a signal when it falls below a particular threshold. An example of its use is on a close miked drum track to remove the spill from other drums; the spill will be at a lower level than the recorded drum and so can be easily removed. Figure I.2.20 shows the Waves SSL channel strip; another example of the modelling of prominent analogue hardware. The plugin incorporates the SSL EQ and SSL Compressor plugins described in section II.1.1 alongside a noise gate, phase inversion switch and input gain level. Figure I.2.21 shows the Waves RChannel plugin from the Renaissance range. This plugin provides a graphical display of the applied EQ and level meters for both pre and post dynamics processing. It allows for a choice of dynamics processors so any combination of two can be connected in series. Figure I.2.22 shows Metric Halo's ChannelStrip which employs interactive level meters for control of the gate and compressor parameters; thresholds are set with a slider on the meter and the gain reduction of each process is shown below. Graphical displays of the gating and compression curves are shown. The plugin provides six filters, the type of which can be chosen between high pass, low pass, shelving and peaking to allow full customisation. The filters can also be manipulated using the graphical display of the EQ contour.

Multiband compression allows separate control of the dynamics of individual frequency ranges. The incoming signal is filtered into specified frequency bands to which an individual compressor is applied; de-essing, as described in Section II.1, is an example of multiband compression. An example of its use is when processing a bass guitar; although the majority of the sound produced is confined to the low frequencies, there can be a significant amount in the high-mid range if the player has included “pops” and “slaps” to the performance, which are typical characteristics of a bass performance for genres such as funk music. Compressing the recording with a full bandwidth plugin would result in the less powerful high-mid frequencies being attenuated when the more powerful low frequencies force the signal above the set threshold (Walden, 2007). Multiband compression can be used in this situation to control the dynamics of the low frequencies without affecting the high-mid range.


Kontakt 4 01: Ins. 1 - MultibandCompressor

Threshold -15.0 Ratio 2.0 Attack 1.0 Release

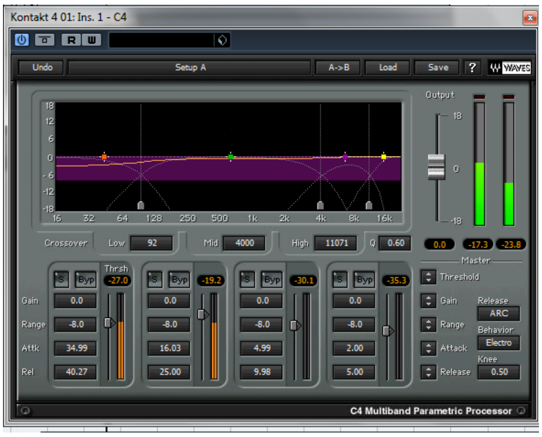
Threshold -15.0 Ratio 2.0 Attack 1.0 Release

Threshold -15.0 Ratio 2.0 Attack 1.0 Release

Threshold -15.0 Ratio 2.0 Attack 1.0 Release

Output 0.0

Figure I.2.23


Kontakt 4 01: Ins. 1 - C4

Crossover Low 92 Mid 4000 High 11071 Q 0.60

Thrsh -27.0 -19.2 -30.1 -35.3

Gain 0.0 0.0 0.0 0.0

Range -8.0 -8.0 -8.0 -8.0

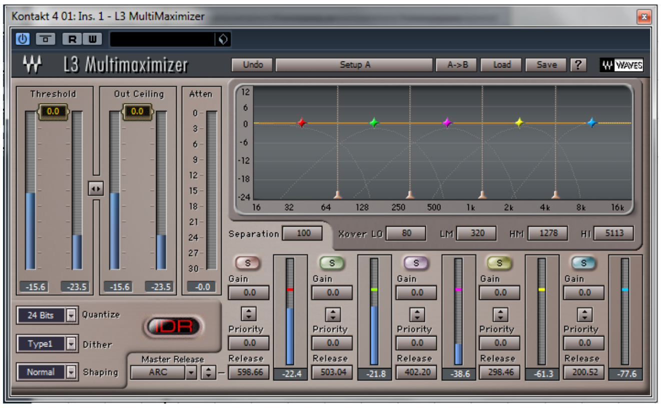
Attk 34.99 16.03 4.99 2.00

Rel 40.27 25.00 9.98 5.00

Master Release ARC Behavior Electro Knee 0.50

C4 Multiband Parametric Processor

Figure I.2.24


Kontakt 4 01: Ins. 1 - L3 MultiMaximizer

L3 Multimaximizer

Threshold 0.0 Out Ceiling 0.0

Separation 100 Xover LO 80 LM 320 HM 1278 HI 5113

Gain 0.0 0.0 0.0 0.0 0.0

Priority 0.0 0.0 0.0 0.0 0.0

Release 598.66 503.04 402.20 298.46 200.52

24 Bits Quantize

Type1 Dither

Normal Shaping

Master Release ARC

Figure I.2.25

Figure II.2.7 shows Steinberg’s MultibandCompressor, included as standard with their Cubase DAW. The plugin provides four bands of operation, the boundaries of which can be set by the user. Each band includes basic compression parameters, ratio, threshold, attack time and release time, and a graphical display of the compression curve. Make-up gain is applied manually to the summed processed bands. Figure II.2.8 shows the Waves C4 plugin which offers identical functionality as the Steinberg plugin, except the threshold level can be set with a slider on the band level meter. Figure II.2.9 show Waves’ L3 MultiMaximiser which employs multiband limiting. No ratio control is available and neither is an attack time parameter, though a release time can be set for each band. A feature unique to the LQ is the “separation” parameter which provides control over the blending of the frequency bands before processing.

Dynamic EQ is the newest and rarest of the effects described in this section. It allows for the automated control of individual filters within an EQ plugin based on the amplitude of the overall signal or a specific frequency range. It serves a similar purpose to multiband compression in that it allows for one area of the frequency spectrum to be transformed without affecting any other areas.

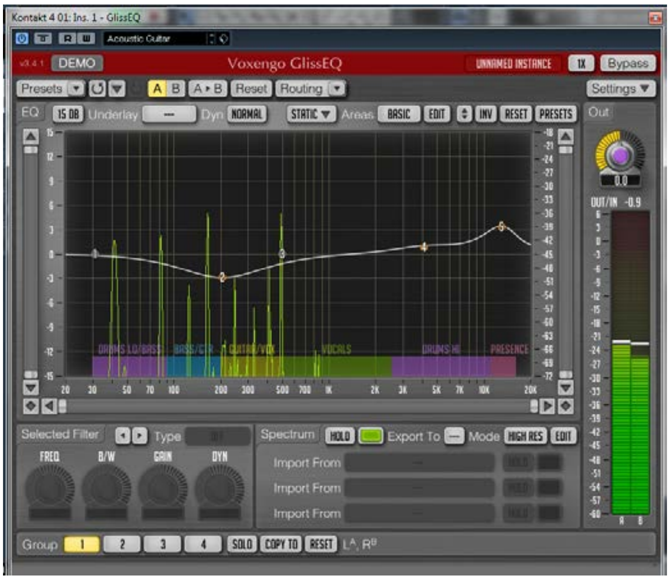


Figure I.2.26

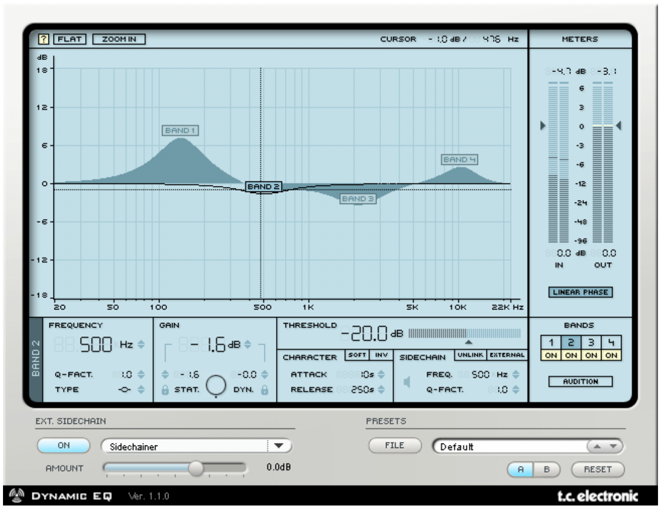


Figure I.2.27

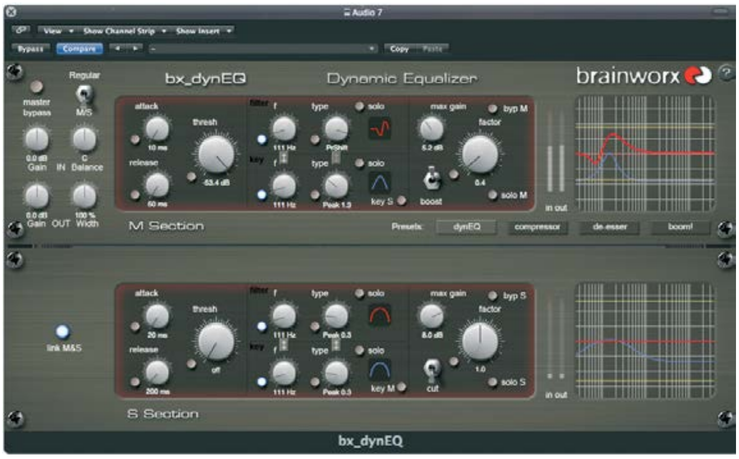


Figure I.2.28

Figure II.2.10 shows Voxengo’s GlissEQ plugin which offers five EQ bands, each of which has cut-off frequency, gain and bandwidth parameters as expected. Included within each band is the “Dyn” parameter which is used to control the amount the dynamic qualities of the incoming signal affect the EQ parameters (Voxengo GlissEQ Overview). The plugin includes spectral analysis which allows the user to view the effect the plugin has in real-time. Figure II.2.11 shows T.C. Electronic’s Dynamic EQ plugin. It employs four frequency bands which can implement static or dynamic EQ and each band’s filter can be selected between parametric and shelving types. Each band provides compression parameters to control the dynamic response of each filter. The input to each compressor can be set to be any frequency range; the high frequency content of the signal can be used to control the low frequency content (T.C. Electronic Dynamic EQ Overview). Figure II.2.12 shows the Brainwork BX_dynEQ plugin which takes the functionality of the previous two examples a step further by offering mid-side processing. The plugin is capable of splitting the incoming signal into its sum and difference components, otherwise known as the mid and side components. This allows the plugin to treat the centre and edges of the stereo image differently, including the ability to compress the edges of the stereo image using the centre as a side-chain input (White, 2009). The frequencies within the mid and side components to which the compressor responds can be set by the user alongside standard compression parameters.

I.2.7 Evaluation

The research into existing products can be used to demonstrate in what ways the proposed plugin is similar and different to current processes, and hence the novel contribution of this research.

The plugin combines several commonly used processes to create a mixing process that would be very time consuming to recreate with traditional plugins. The only method of recreating its capabilities using the basic plugins above would be to use separate EQ and compression plugins or a channel strip, and manually automate all parameters using critical listening and some form of signal analysis for assistance.

The multiband compressors and dynamic equalizers are a step up from the basic plugins and show how processes can be combined to form more complex effects. The proposed plugin has similarities to both processes; it compresses the signal by frequency band and automatically controls EQ parameters. It differs from multiband compression in that the signal is not filtered in order to be analysed and processed; signal analysis and regression are used to determine band amplitude and dynamic range compression is applied via filters. It is closet to dynamic EQ – both use dynamic processing to control EQ parameters, but again is unique in its use of frequency domain signal analysis to determine effects parameters.

This research aims to take the existing processes described throughout this section and combine them in a way that produces a novel contribution to the field of automated mixing by experimenting with different methods of interpreting signal analysis data and its use to apply corrective mixing, and to produce a plugin which can autonomously mix a piano recording for a popular music production.

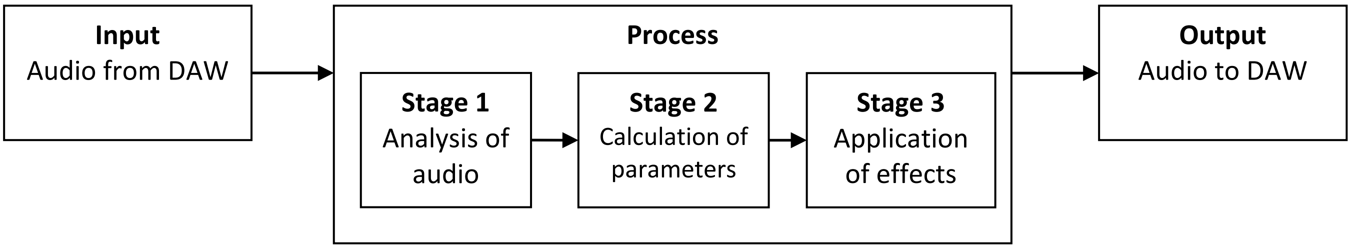
SECTION 3

DEVELOPMENT

II.1

Formal Specification

This section provides an overview of the system. The developed plugin can be described as system whose input is an stream of audio samples from a DAW, which are analysed and processed by an algorithm, and outputted back to the DAW. The audio processing algorithm has as its input the results of signal analysis which are interpreted to derive effect parameters and output a processed stream of samples:



Three instances of the algorithm are active within the plugin, each processing a specified frequency band. The three frequency bands to be effected were derived from the previous work into piano recording and mixing carried out by the author.

Low frequencies – 30Hz – 150Hz

As explained in II.1 this band is commonly attenuated within a popular music production to allow room for other instruments. It also contains noise from the pedals of a piano and air movement within the recording environment. The content of the band produced by the piano is “boomy” and resonant, and its attenuation produces a much clearer overall tone.

Low-mid frequencies – 300Hz – 700Hz

This band contains the “body” of piano tone and so it best treated with a peaking filter with a wide Q and relatively low attenuation. Its characteristics are “ringing” and resonant, and so like the low frequencies its attenuation produces a much clearer overall tone.

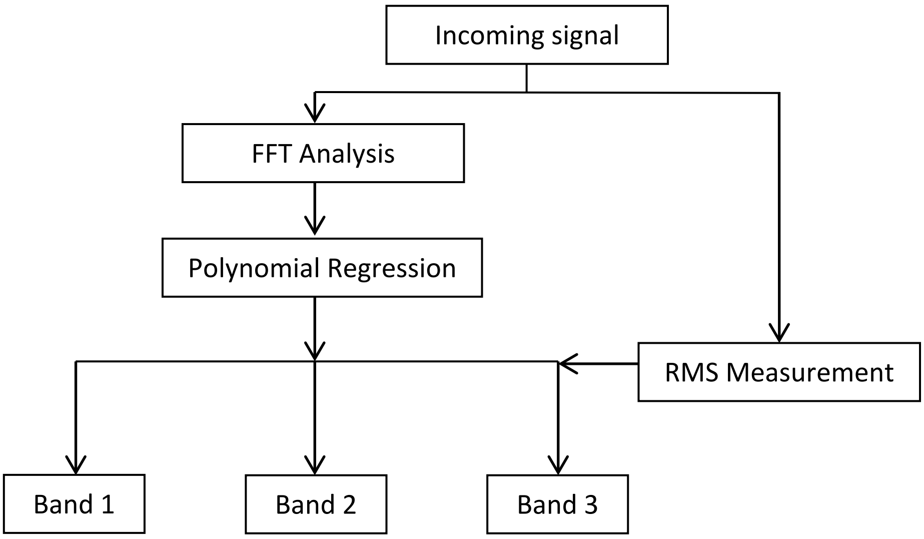
High-mid frequencies – 3kHz – 5kHz

This band mostly contains harmonics, but most importantly the “attack” characteristic of notes. Boosting it produces clearer note attack because of this, but also increases the brightness of a recording due to harmonic content.

Generally speaking the equalisation of a piano recording aims to reduce “muddiness” and boost brightness. This produces a clear, crisp tone conducive to a popular music production. The three bands described above cover nearly all of the audible frequency spectrum up to 5kHz. The frequencies above 5kHz are relatively low powered and so do not have much of an impact on the dynamic qualities of a recording, meaning that regulation of the three bands with dynamic processing will efficiently control the overall dynamic range of the audio.

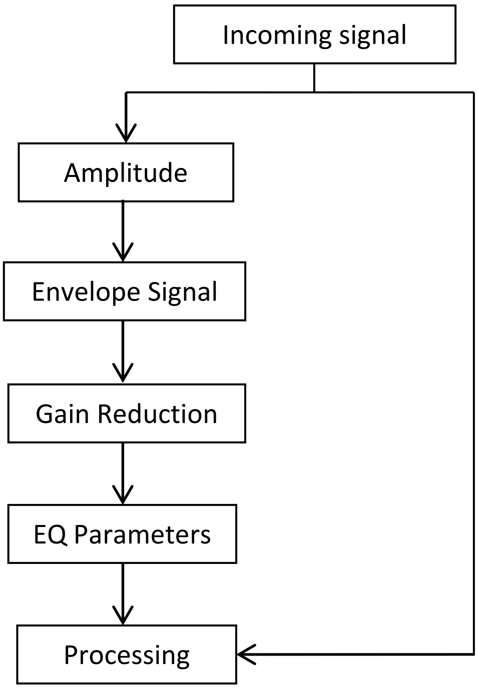
The signal analysis section analyses windows of samples and produces average amplitudes of the three frequency bands to be processed:

1. FFT Analysis
    a. A frequency domain representation of the signal is estimated using FFT analysis. This produces a set of frequency and amplitude pairs representing the energy content of the window.
2. Polynomial Regression
    a. Regression is used to smooth the variance of each amplitude value over time by deriving a curve of best fit for the spectrum.
3. Band Average
    a. Each amplitude value within each of the three desired frequency bands is averaged to produce a signal level for that band.
4. RMS Measurement
    a. A final smoothing stage scales each band level using the RMS level of the entire signal.



The audio processing section uses the band levels calculated through signal analysis to calculate parameters for three instances of a custom effect and apply them. For each of the three frequency bands:

1. Envelope Signal
    a. An envelope signal is calculated using the smoothed band level and predefined attack and release times.
2. Gain Reduction
    a. A gain reduction value is calculated using the envelope and predefined threshold.
3. EQ Parameters
    a. Gain and cut-off frequency values are calculated using the gain reduction value.
4. Processing
    a. The filter is applied to the signal.



As the three bands encompass the most important content produced by a piano, their regulation via dynamics processing also has a large impact on overall dynamic range.

II.2

System Design

Fourier Analysis is a branch of mathematics that is extremely useful in digital signal processing, first published by James Fourier in 1808 (Fourier, 1808). At its core is the idea than any periodic signal can be expressed as the sum of an infinite number of sine and cosine waves, known as the Fourier series, (Croft and Davidson, 2008, p.1139), with the process known as a Fourier transform. In 1965 J.W. Cooley and John Tukey rediscovered the Fast Fourier Transform algorithm, a re-working of the Discrete Fourier Transform (a solution for applying Fourier analysis to discrete rather than continuous functions), which computes the transform at speed (Cooley and Tukey, 1965). Many variations of the FFT exist, but all involve some kind of decimation (Jackson, 1996, p.213).

The purpose of the FFT algorithm within this research is to provide a means to analyse the tonal qualities of a piano recording. In order to do this, the software must be able to determine the average amplitude over time of different frequency bands, taking those amplitudes from an FFT trace. As shown in the videos* and three screenshots below, an FFT trace is a jagged line which moves seemingly randomly over time.

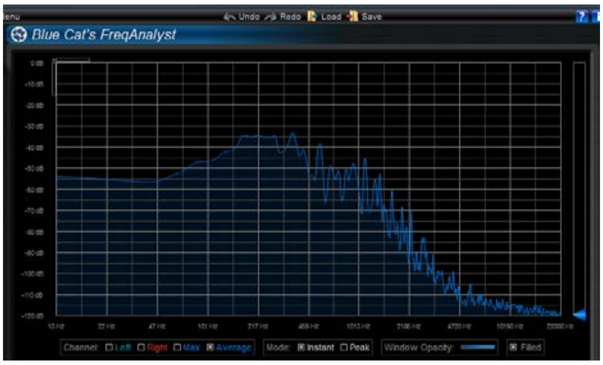


Figure II.2.1a

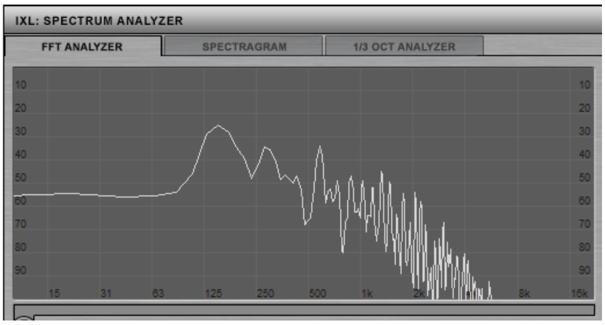


Figure II.2.1b

*The videos can be found on the DVD under /Thesis/Videos/FFT Plugins

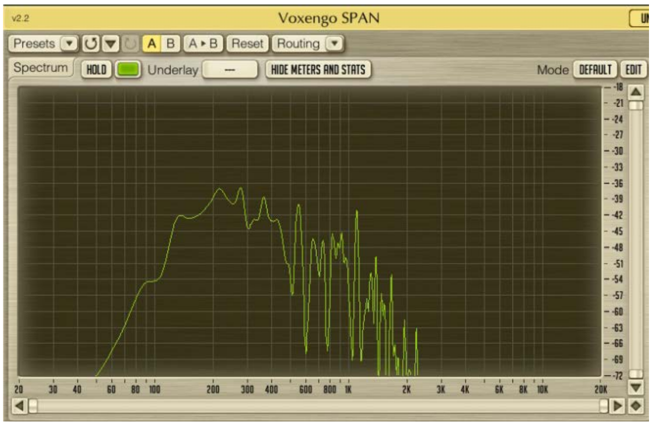


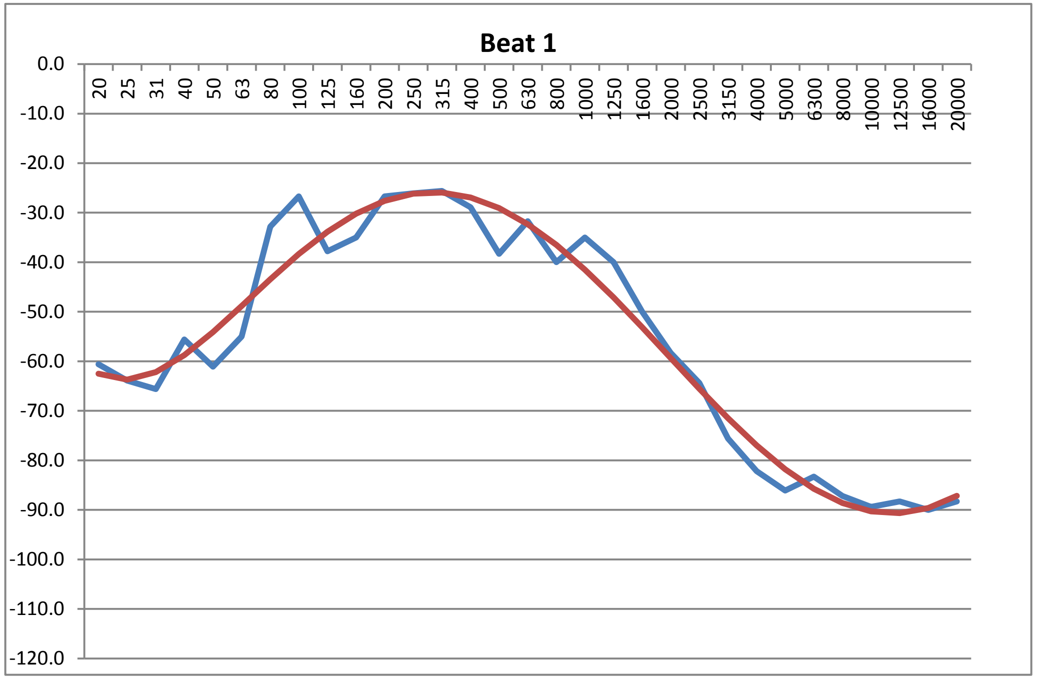
Figure II.2.1c

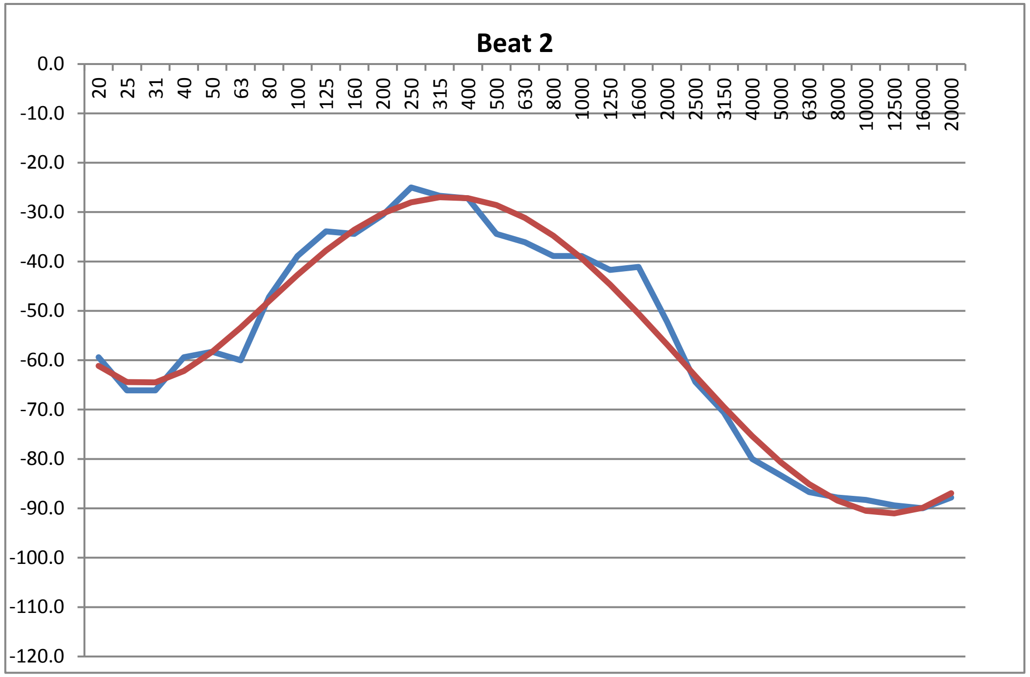
For early implementations of the plugin the amplitude of each desired frequency band was calculated by averaging a number of bins from an FFT trace. However, the sharp movements in the trace led to sharp changes in effects parameters. To avoid always having to use long attack and release times to smooth these sharp changes, a different method of calculating frequency band amplitude was required. In order to pursue the assertion that frequency domain signal analysis could be used for this purpose, Microsoft Excel was used to explore the usefulness of regression in smoothing the trace. Specifically, if the frequency domain representation was a smooth plotted line rather than jagged, would this produce band amplitudes with lower variances, and hence be more representative of the actual volume of a band over time.

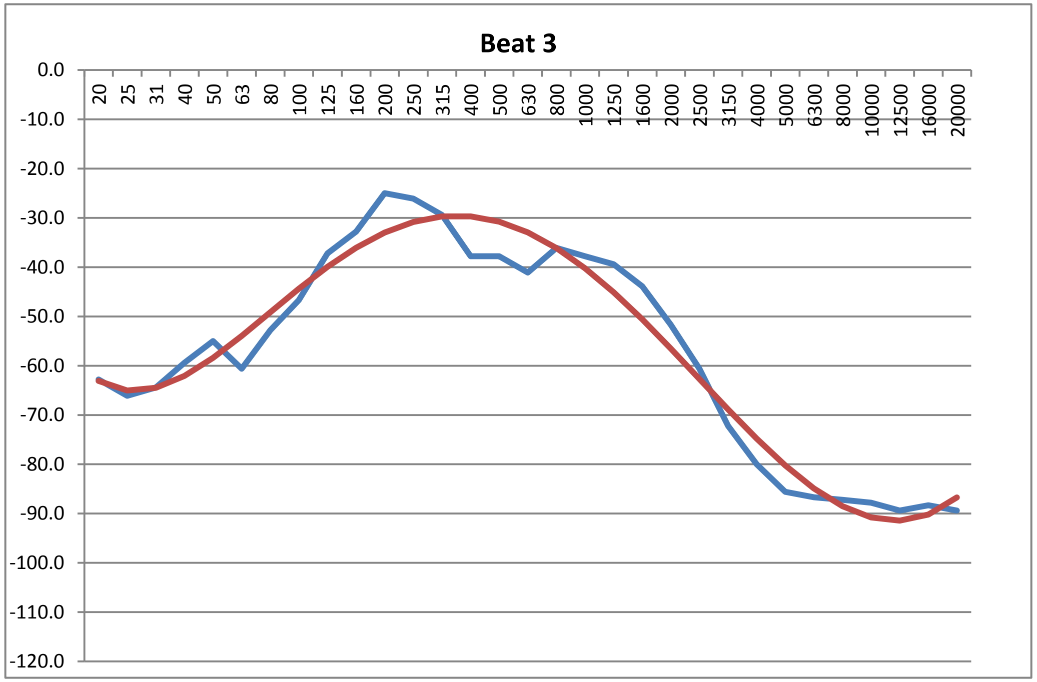
Polynomial regression is a form of curve fitting which derives a polynomial to represent the non-linear relationship between two variables. The least squares process is used by most software implementations to achieve this (see V.2). To determine whether it was a viable option for smoothing an FFT trace, a piano recording was made and analysed in Cubase 5 using the 1/3 Octave Analyser section of the IXL Spectral Analyser and data copied by hand into Microsoft Excel to be regressed. Traces were recorded at four beats of a bar to allow the same point of a recoding to be easily captured multiple times for data gathering purposes. Accompanying Figure II.2.1 are some videos* that demonstrate the movement of the regressed trace over time. Both results give a visual representation of the averaging effect that regression has on the FFT trace, and it was on this basis that it was decided to implement regression within the plugin to solve the problem of calculating frequency band amplitudes directly from the FFT trace.

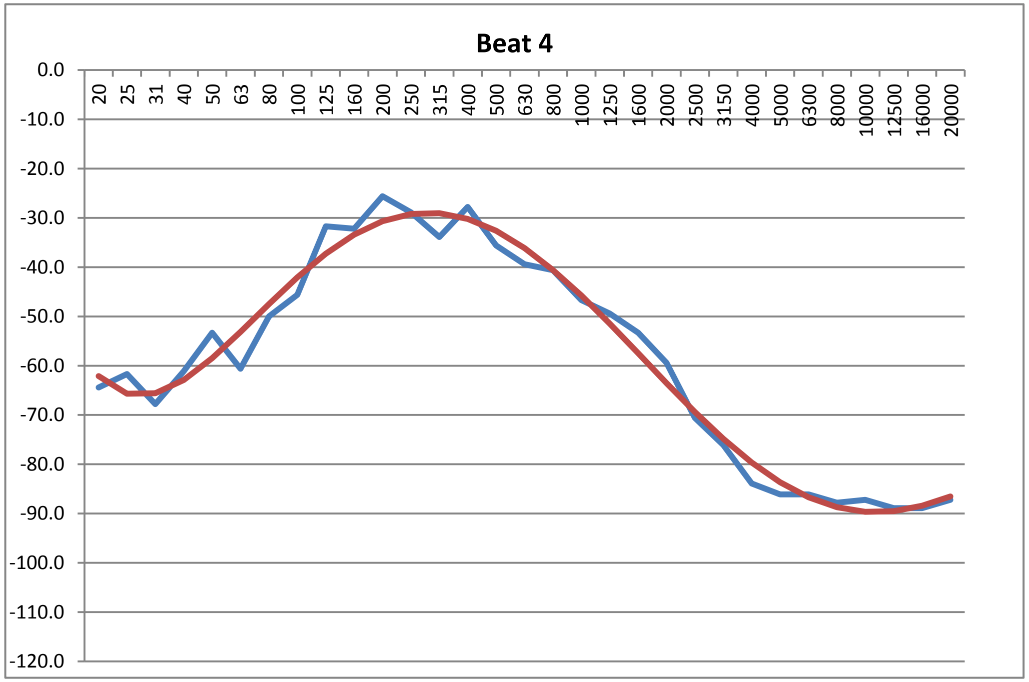
To implement polynomial regression within the VST framework an external library, ALGLIB, was used (ALGLIB, 2012). ALGLIB was chosen as it was the only open source implementation of regression in C++ that could be found. It also includes a comprehensive set of FFT functions, keeping the process of linking external resources to the project as simple as possible and also keeping the size of resulting plugin smaller than if a separate library was used to calculate the FFT.

Aural tests were carried out to determine whether using a regressed FFT trace to calculate frequency band amplitudes reduced the effect of the sharp changes in bin amplitudes on effect parameter changes. The regressed trace produced much smoother changes in amplitude and as a result more musically pleasing results.


Beat 1
0.0
-10.0
-20.0
-30.0
-40.0
-50.0
-60.0
-70.0
-80.0
-90.0
-100.0
-110.0
-120.0
20
25
31
40
50
63
80
100
125
160
200
250
315
400
500
630
800
1000
1250
1600
2000
2500
3150
4000
5000
6300
8000
10000
12500
16000
20000


Beat 2
0.0
-10.0
-20.0
-30.0
-40.0
-50.0
-60.0
-70.0
-80.0
-90.0
-100.0
-110.0
-120.0
20
25
31
40
50
63
80
100
125
160
200
250
315
400
500
630
800
1000
1250
1600
2000
2500
3150
4000
5000
6300
8000
10000
12500
16000
20000


Beat 3
0.0
-10.0
-20.0
-30.0
-40.0
-50.0
-60.0
-70.0
-80.0
-90.0
-100.0
-110.0
-120.0
20
25
31
40
50
63
80
100
125
160
200
250
315
400
500
630
800
1000
1250
1600
2000
2500
3150
4000
5000
6300
8000
10000
12500
16000
20000


Beat 4
0.0
-10.0
-20.0
-30.0
-40.0
-50.0
-60.0
-70.0
-80.0
-90.0
-100.0
-110.0
-120.0
20
25
31
40
50
63
80
100
125
160
200
250
315
400
500
630
800
1000
1250
1600
2000
2500
3150
4000
5000
6300
8000
10000
12500
16000
20000

The initial implementation of regression within the plugin used 1024 bins from an FFT with logical size 2048 samples – the symmetrical nature of the resulting trace means that only the first half of bins are used. The regression process is complex and the algorithm proved to be too slow when used with this many samples. To feed this many values into the regression algorithm would be hugely costly in efficiency terms; an accurate curve can be achieved with far less values. Even more inefficient would be to attempt to plot four traces on the GUI (pre and post FFT and regression traces) of 1024 samples each. A method of reducing the number of bins passed around the plugin was required. The simplest option would have been to take, for example, one sample for every five from the FFT output, resulting in an array of 204 bins. However as all spectrum are plotted on a logarithmic frequency axis, it was decided to implement a logarithmic method of taking bins from the FFT output. This is useful because the vast majority of sound produced by any musical instrument occurs below 10 kHz; above that are harmonics of lower frequencies. For a piano the point at which frequency content can be ignored is higher than for other instruments due to the extended range of notes that can be played, but it is still unnecessary to be analysing amplitudes up to 20 kHz. With an FFT resolution value of 512 or higher stuttering is introduced to the track within Cubase; the plugin cannot compute the regression and draw the traces quickly enough and uses so much processing power trying to that the audio buffers cannot be maintained. It was logical therefore to have the value as low as possible in order to have CPU overheads at a minimum. To find this minimum the FFT and regression traces were observed at different resolutions and visual observation used to determine if the value was acceptable; the value at which the regression curve was smooth i.e. had no sections of straight lines, and moved significantly less than the FFT trace was used.

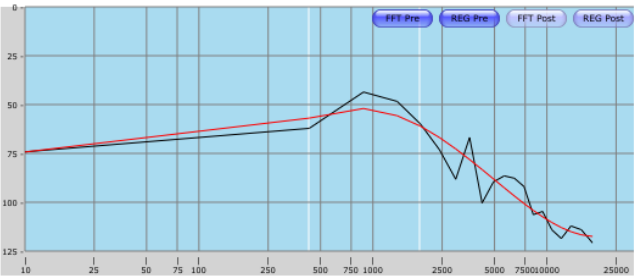


Figure II.2.3a

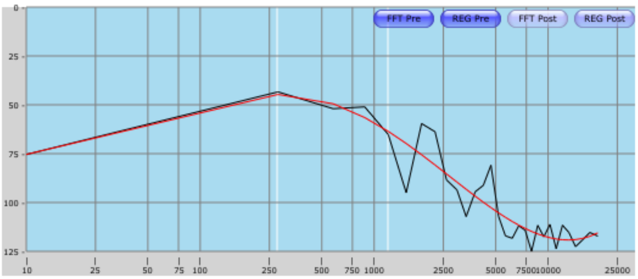


Figure II.2.3b

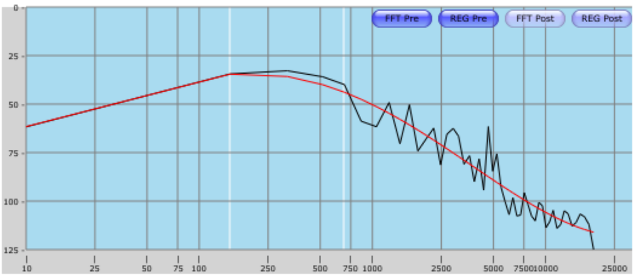


Figure II.2.3c

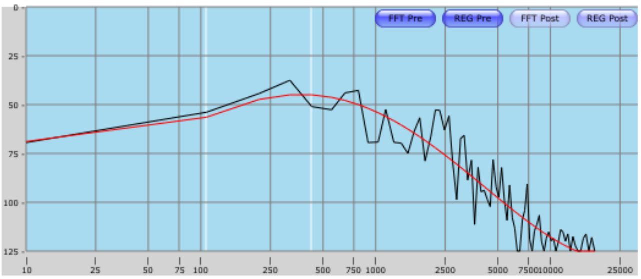


Figure II.2.3d

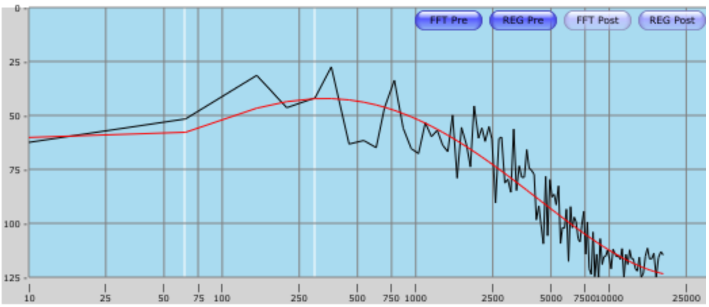


Figure II.2.3e

Figures II.2.3 (a), (b), (c), (d) and (e) show resolutions of 20, 30 50 , 75 and 100 respectively and were implemented using a base ten logarithm to extract bins from the FFT output, however there is still a significant difference in the distance between points at each end of the spectrum. This can be improved by using a higher base logarithm; the graph below shows how the severity of a logarithmic curve increases with its base.

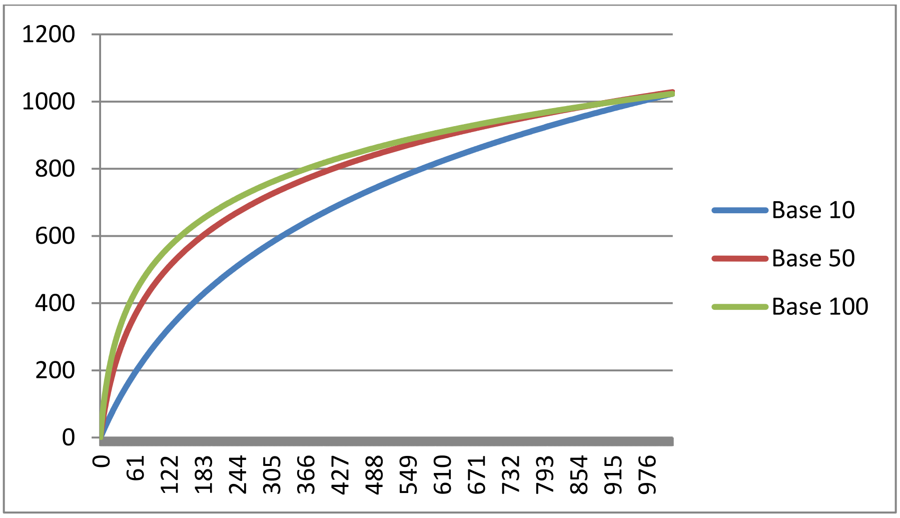


Figure II.2.4

Higher bases are implemented using the formula:

$$\log_b x = \frac{\log_a x}{\log_a b}$$

So the value of the base 50 logarithm of *x* is given by:

$$\log_{50} x = \frac{\log_{10} x}{\log_{10} 50}$$

In order to compute the curve above the input must be scaled to match the range of the relevant logarithm:

$$i_{out} = \frac{\log_{10}\left[(L_B - 1)\frac{i_{in}}{N} + 1\right]}{\log_{10} L_B} N$$

The input index must be scaled first to be between zero and one which is achieved by division by its maximum value, *N*, and then scaled between one and the logarithm base, L_B. Therefore, an input of zero is first scaled to zero then scaled to one giving a logarithm of zero. An input of *N* is first scaled to one then scaled to L_B giving a logarithm of one. The output of this is then multiplied by *N* to re-scale the value back to the index range. This is more easily demonstrated in the table below, which uses 1000 as the value of *N* and 50 as the value of L_B:

i_{in}	$\frac{i_{in}}{N}$	$(L_B - 1)\frac{i_{in}}{N} + 1$	$\frac{\log_{10}\left[(L_B - 1)\frac{i_{in}}{N} + 1\right]}{\log_{10} L_B}$	$\frac{\log_{10}\left[(L_B - 1)\frac{i_{in}}{N} + 1\right]}{\log_{10} L_B}N$
0	0	1	0	0
250	0.25	13.25	0.66	679.31
500	0.5	25.5	0.83	851.41
750	0.75	37.75	0.93	954.54
1000	1	50	1	1000

However a calculation must also be made to “reverse” the logarithmic curve; the graph and table above show curves that would take more values from the high end of the spectrum than low and the opposite is required:

$$i_{out} = 1 - \frac{\log_{10}\left[(L_B - 1)\left(1 - \frac{i_{in}}{N}\right) + 1\right]}{\log_{10} L_B}N$$

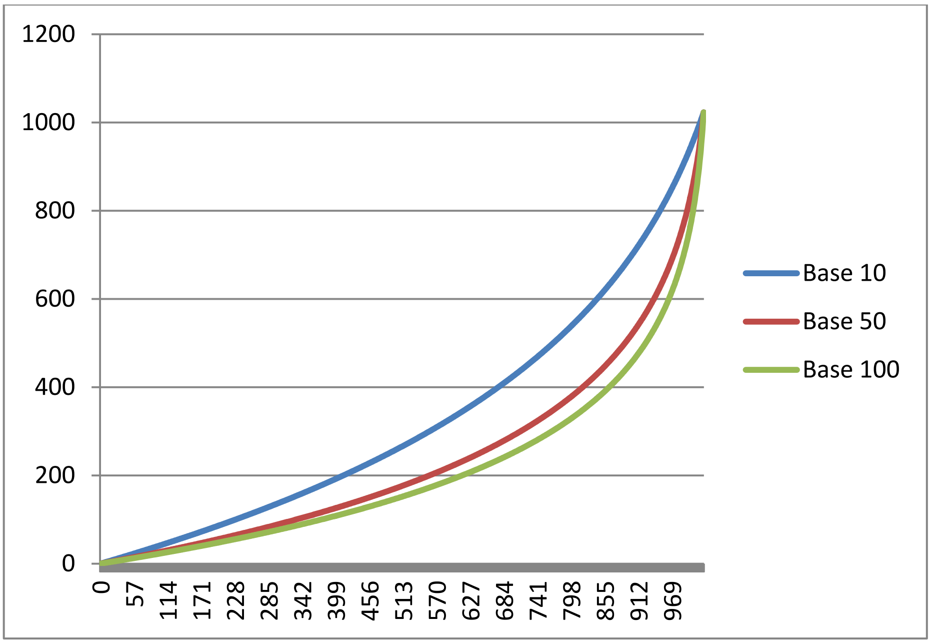


Figure II.2.5

More experimentation was carried out to determine a suitable logarithm base. Starting at ten the value was increased until the distance between points on the trace had evened out. More importantly the value was used to reduce the size of the straight line at the beginning of the trace before the curve begins, caused by the base ten logarithmic frequency scale on the trace which displays the 10 Hz to 100 Hz frequency range in the same physical distance as the 1 kHz to 10 kHz range. A higher base value forces more bins to be taken from the low frequency range, reducing the distant between points. Figure II.2.6 (a), (b), (c), (d), (e) and (f) show *LOGBASE* values of 10, 50, 100, 150, 200 and 250 respectively.

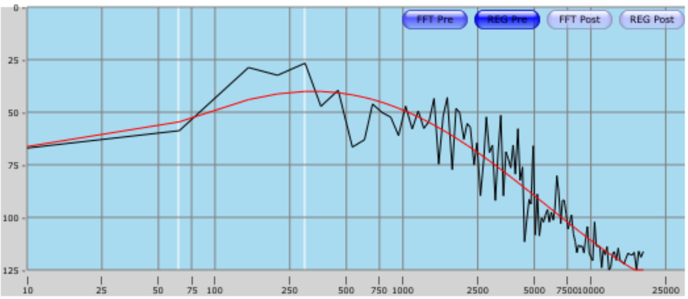


Figure II.2.6a

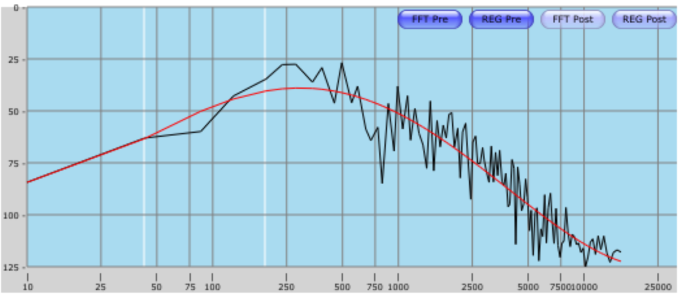


Figure II.2.6b

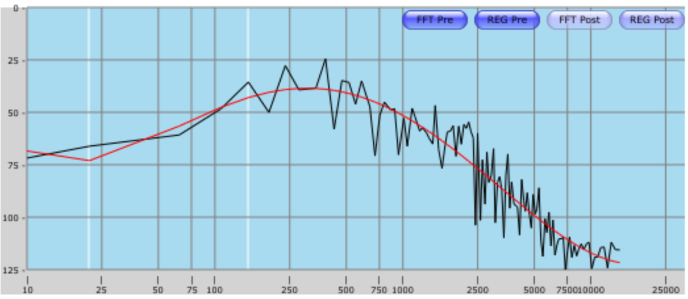


Figure II.2.6c

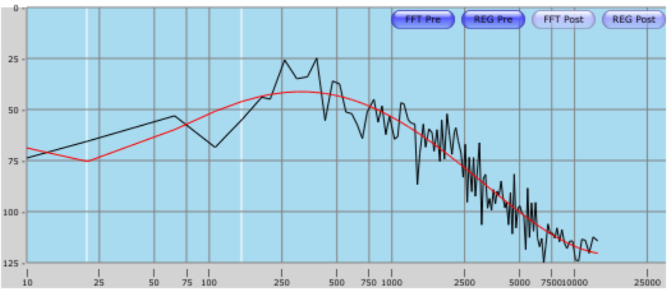


Figure II.2.6d

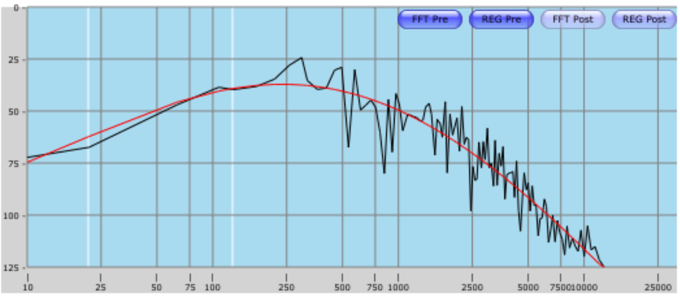


Figure II.2.6e

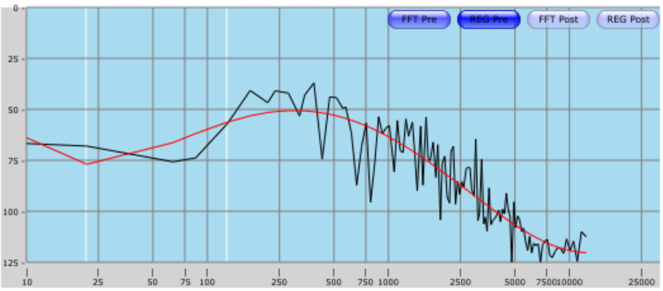


Figure II.2.6f

At 250 the increased logarithm base has shrunk the sharp point where the straight line meets the curve from around 75 Hz to around 20 Hz and has had a significant effect on the spread of points across the trace; at the low end the points are far closer together and though it is difficult to see at the high end points are further apart. The logarithm base and resolution values allow the plugin to operate more efficiently by reducing the amount of data to processed using the FFT resolution, and allowing what is processed to be more relevant. A description of the signal path through the plugins codebase follows, which can be cross referenced with the source code provided in the appendix.

The *FFTALG* class handles the interface between the ALGLIB FFT routines and the plugin. Two arrays were used to send and receive data from the FFT, both of ALGLIB data types; *real_1d_array in* contains the real valued input and *complex_1d_array out* contains the complex valued output. The *Init()* functions defines these and two sample arrays, *Lc* and *Rc*, with size *N*. The *Do()* functions begins by populating the *in* array with stereo to mono converted samples from the *Lc* and *Rc* arrays. These channel arrays are populated outside of the class in the *processBlock()* function within *Juce_vstAudioProcessor*. The FFT is carried out using the *fftr1d()* function, returning to the array, *X*. The first half of this array is then converted from complex to real values and then to Decibels, resulting in the returned array, *out*.

In the *Juce_vstAudioProcessor* header file two *FFTALG* members *FFTpre* and *FFTpost* are declared along with two output pointers, *FFTpre_out* and *FFTpost_out*, and two index integers, *fftpre_index* and *fftpost_index*. The two pointers *Lx* and *Rx* are also declared to carry the audio samples around the function. In the main file the two *FFTALG* members are initialised in the *Juce_vstAudioProcessor* constructor using the *Init()* function. This function takes the *N*, *FFTRES* and *LOGBASE* values, corresponding to the FFT size, FFT resolution and output logarithm base, which are declared as static constants in the header file. The integer *winType* is then set using the *wintype* class declared in the *FFTALG* header file. This value is used as an index to indicate the desired windowing function. Finally the indexes *fftpre_index* and *fftpost_index* are set to 0 and the output pointers *FFTpre_out* and *FFTpost_out* are defined with size *FFTRES* and each element is set to zero.

In the *processBlock()* function the current buffer of samples is transferred to the *Lx* and *Rx* pointers and the index variables reset to zero if they have reached *N*. The index variables are used in the main *for* loop to ensure that *N* samples are passed to the FFT functions no matter what the size of the buffer; the *Do()* function is called when the input pointers have been populated with *N* samples. Therefore for buffer sizes smaller than *N* an FFT will be carried out every few buffers, for buffer sizes larger than *N* an FFT is carried out every buffer and the remaining samples are discarded. This is achieved within the *for* loop with an if statement comparing the relevant index variable to *N*; if the index is less than *N* the current stereo samples are added to the *Lc* and *Rc* pointer members of the *FFTALG* class and the index variable is incremented, if the index is equal to *N* the FFT is carried out first by applying the chosen windowing function to the samples with the *Window()* function, and then by transferring the resulting frequency bins to the relevant output pointer. When the FFT is completed the index variable to reset to zero. Extra fail safes protect this function in the form of a check at the beginning of the *processBlock()* function that resets the index variables to zero should they be at *N*, and the if statement within the loop checks to see if the index variable is greater than or equal to *N*.

The *Polyfit* class serves as the interface between the ALGLIB regression routines and the plugin. The class requires the *ap.h* and *interpolation.h* ALGLIB header files be included; *ap.h* is the general ALGLIB header containing definitions of custom data types and *interpolation.h* contains the functions and data types specific to curve fitting. To use these functions and data types the class was written within the *alglib* namespace. The class contains two functions, *Init()* and *Solve()*. *Init()* takes the value *degree* as an argument and denotes the degree of the resulting polynomial; the highest power to which a term is raised. The higher this value the more turning points present in the curve and hence it is more accurate, but more costly to calculate. The function assigns the desired value to variable *m* of type *ae_int_t*, a ALGLIB custom integer type, incremented by one. The incremented value is the number of basis functions used by the least squares solver to carry out the regression. The *Solve()* function receives two pointers, *_x* and *_y*, which form the two dimensional input to the regression algorithm. *_x* contains the x-values of the Cartesian co-ordinates in pixel units used to plot the traces on the GUI. These values are used rather than array indexes as they reflect the logarithmic nature of the frequency axis and the method used to extract bins from the FFT output, and are calculated using the dimensions of the plotting canvas and the *LogScales* class. *_y* contains the amplitude values calculated by the FFT. The third argument is *FFTRES*, referred to within classes as *L*, and is used to define the size of*real_1d_array*s *x* and *y*, another ALGLIB custom data type for holding real valued one dimensional series of data. A for loop populates the *real_1d_array*s with the data contained in *_x* and *_y*. The regression itself is carried out by the *polynomialfit()* function, the arguments of which are show below, taken from the comments in the header file:

```
INPUT PARAMETERS:
    X   -   points, array[0..N-1].
    Y   -   function values, array[0..N-1].
    M   -   number of basis functions (= polynomial_degree + 1), M>=1

OUTPUT PARAMETERS:
    Info-   same format as in LSFitLinearW() subroutine:
            * Info>0    task is solved
            * Info<=0   an error occured:
                        -4 means inconvergence of internal SVD
    P   -   interpolant in barycentric form.
    Rep -   report, same format as in LSFitLinearW() subroutine.
            Following fields are set:
            * RMSError      rms error on the (X,Y).
            * AvgError      average error on the (X,Y).
            * AvgRelError   average relative error on the non-zero Y
            * MaxError      maximum error
```

The input parameters are as described above, with *N* being defined as the size of *x* and *y*, set using the *setLength()* function. The output parameters are defined in the header file as *ae_int_t info*, *barycentricinterpolant p* and *polynomialfitreport rep*. *info* is returned as an integer indicating the success of the algorithm with a value other than zero indicating successful completion. *rep* is of ALBLIG data type *polynomialfitreport* and contains several measures of success including values of RMS and average error of the regression model versus the original function. *p* is of ALBLIG data type *barycentricinterpolant* and contains the resulting polynomial in barycentric form – a homogenous co-ordinate system (Weisstein, 1999). The output parameters are submitted to the function as call-by-reference arguments allowing them to be altered by the function itself without returning values (Savitch, 2010). The process has computational complexity *$O(NM^2)$*.

The *polynomialbar2pow()* function converts the regression from barycentric from to power basis. The power basis form represents the polynomial in terms of its coefficients using the standard form:

$$y = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

The *polynomialbar2pow()* function returns a *real_1d_array*, identified by the class as *polyco*, containing these coefficients in reverse order; $a_0,\ldots,a_4$. As described in the comments above the function has computational complexity $O(N^2)$. The final part of the *Solve()* function populates and returns the *curve* pointer with the y values of the Cartesian co-ordinates of the polynomial using the power basis representation of the function and the polynomial form shown above. Each value of input *x* is put through a loop which accumulates the result of the value raised to a power and multiplied by the relevant coefficient.

The *Juce_vstAudioProcessor* header file contains declarations for the two *Polyfit* members *REGpre* and *REGpost* as well as the input pointer *fftX* and output pointers *REGpre_out* and *REGpost_out*. The *Plot* class member *fft* provides the x-values for the regression input; *Plot* is a class used by *Juce_vstAudioProcessorEditor* to calculate the co-ordinates of the GUI traces and draw them. The *Juce_vstAudioProcessor* constructor is used to initialise the *fft* instance and define *fftX*, instantiate the *REGpre* and *REGpost* members with the polynomial degree of four, and set the size of the output pointers as *FFTRES*. The implementation of the *Solve()* function is then very simple; at the point when the FFT is carried out in the *processBlock()* function the *REGpre_out* and *REGpost_out* pointers are equated to the result of the *Solve()* function which takes the *fftX* pointer as x-values, the *FFTpre_out* or *FFTpost_out* pointer as y-values and *FFTRES* as the array sizes.

The result of these processes are four pointers to arrays; the *FFTpre_out* and *FFTpost_out* arrays containing amplitude data resulting from the FFT algorithm, and the *REGpre_out* and *REGpost_out* pointers containing amplitude data as a polynomial representing the regression of the FFT data. These arrays are then carried over to the central processing section of the plugin where EQ filter parameters are calculated and the filters implemented.

The implementation of EQ within the plugin was achieved with a simple binomial difference equation and intermediate variables:

$$y_n = \frac{b_0}{a_0}x_n + \frac{b_1}{a_0}x_{n-1} + \frac{b_2}{a_0}x_{n-2} - \frac{a_1}{a_0}y_{n-1} - \frac{a_2}{a_0}y_{n-2}$$

$$A = \sqrt{10^{\frac{G}{20}}}$$

$$\omega 0 = 2\pi \frac{F}{f_s}$$

$$\alpha_{Q\ Filter} = \frac{\sin \omega 0}{2Q}$$

$$\alpha_{Shelving\ Filter} = \frac{\sin \omega 0}{2}\sqrt{\left(A + \frac{1}{A}\right)\left(\frac{1}{Q} - 1\right) + 2}$$

A class named *EQ* was written to implement the Bristow-Johnson equations and formulae. The class contains three functions; *Init()*, *setCoefficients()*, and *Do()*. The *Init()* function receives the desired filter type and sample rate as arguments. The filter type is defined using the first enumeration in the header file which declares three constants; *LowShelf*, *Parametric*, and *HighShelf*. This enumeration allows for the constants to be used in switch statements within the class and also provides an elegant method of indicating the filter type within *Juce_vstAudioProcessor* - *EQ::LowShelf* for example. The variable *n* is defined as the maximum sample delay in the system corresponding to the highest power in the z-transform of the difference equation and the order of the filter. Two pointers to arrays, *x* and *y*, which hold the current and delayed input samples are defined as two dimensional arrays to hold the current and two delayed stereo samples – three sets of two values – and all elements are set to zero. The *setCoefficients()* function receives the filter type and calculates the difference equation coefficients accordingly by first calculating the intermediate variables. The variables *cosw0*, *sinw0* and *sq* are defined to reduce the number of mathematical functions that need to be called whilst audio is passing through the plugin; in a basic EQ unit the filter parameters are set before playback and rarely changed during, within the plugin the parameters will be changing all the time. With the coefficients calculated the *Do()* function very simply implements the difference equation to perform the equalisation. It receives the current input sample and an integer denoting the current channel being processed as arguments. The current input sample is written to the *n*th *x* array element and the current output sample is written to the *n*th *y* array element using the delayed *n-1*th and *n-2*th delayed samples from each. Samples are then moved through the input and output arrays and the latest output sample is returned.

Compression was implemented within the plugin with the *Compression* class. The class contains four functions; *Init()*, *setParameters()*, *setAudio()*, *Do()*. The *Init()* function receives the compression type, buffer size and sample rate as arguments. The type variable is fed with either *Compress* or *Compand*, constants declared with an enumeration in the header file. Companion is an inverted form of compression; if the input signal if below the threshold level a gain boost is applied. The *Init()* function also defines the *look* and *width* variables, the lookahead time and rms window size respectively, by converting them from milliseconds to a number of samples using the sample rate.; as sample values they can be used as constraints within a loop. The values for these parameters are hard coded into the algorithm with a lookahead time of 3 ms and rms window size of 1 ms. The *setParameters()* function receives the classic compression parameters as arguments; threshold, slope, attack time and release time. The function defines the slope, attack and release coefficients by converting them from percentage and millisecond units respectively; slope is multiplied by 0.01 to scale it's range as zero to one, and the attack and release coefficients are calculated with an extra stage to convert the times from milliseconds to seconds to be compatible with the sample rate measured in Hertz. In order to allow the *Do()* function to be called with one input sample at a time the *setAudio()* function stores the current audio buffer within the class to be referenced by the *Do()* function. This means the loop through the buffer can be performed externally whilst still allowing the rms to be calculated within the class. The *setParameters()* function receives the stereo audio buffers and the averaged FFT band level calculated with those buffers as arguments. The *Do()* function then performs the compression or companision itself; the compansion algorithm differs from the compression algorithm only in the less than rather than greater than comparison between envelope and threshold and the result is a positive rather than negative gain.

The level input to the compression algorithm is derived from the FFT band level, however using this value alone would not have yielded very high quality results. The speed at which the gain reduction level can be changed should in theory be the same as the sample rate, however in reality this would only be required using attack and release times of zero, and without these coefficients the system would be performing waveshaping rather than dynamic range compression. Therefore the gain reduction and hence input level to the system must vary at least as quickly as the lowest attack and release times that can be expected, which within a digital audio application could be as low as 5 ms. The logical size of the FFT is 2048 samples, a window size of approximately 46 ms at a 44.1 kHz sample rate; the compressor would only be able to implement attack and release times above and return a new gain reduction value every 46 ms if the FFT band level was used as an input, as that is how often a new value is created. To counter this, the *Do()* function calculates an rms level and blends a small amount of this with the FFT band level converted to a linear amplitude. A ratio of 80% FFT band level to 20% rms level is used, causing fluctuations in the input level at a frequency which allows the compressor to vary the gain reduction as quickly as would be required by the application. A process of experimentation was used to defined the 4:1 ratio of FFT to rms level. In order that as much of the level as possible was defined by the FFT, an important characteristic of the plugin, the rms part of the ratio needed to be as low as possible. The amount of rms was increased in 5% intervals and for each value the transient response of the plugin at low attack and release times was analysed using critical listening; the first value that allowed the plugin to respond quickly enough was used.

This blended input level is fed into the compression algorithm which compares it to the current envelope level and defines the variable *theta* with the relevant attack or release coefficient. The envelope level is then converted into decibels and compared to the threshold, and a gain reduction value is calculated using the slope coefficient. A fail safe is added to the function between these two stages; a not equal comparison between the envelope level and itself which returns false if the variable does not contain a real valued number. This prevents the compressor responding to the envelope variable before it has been initialised and its value is "garbage", which results in noise at the output of the plugin.

The EQ and compression classes are combined and implemented with the *FBand* class. The class has a member instantiated for each frequency band which needs effecting; the low, low-mid and high-mid bands. The class is fed with all of the inputs described above and uses them to calculated EQ parameters and implement the filters. The class contains three functions; *Init()*, *setParams()* and *Do().* In the *Init()* function the *size*, *filterType*, *SR* and *compType* variables are defined and passed to the *EQ* and *Compression* classes as described above using their *Init()* functions with instances named *EQ* and *Comp* respectively. The *fft_frequency* and *fft_width* variables set the location and size of the frequency band itself; *fft_frequency* contains the index value of the start of the band and *fft_width* contains how many bins wide the band is. These indexes are in the zero to *FFTRES* range and are received in index form to allow for them to be easily used within loops that deal with the regression output array pointer. Finally the *Init()* function defines five pointers to contain output samples; *Lout* and *Rout* contain the output in stereo form, *out* contains the output in interleaved form and *Lcas* and *Rcas* are used to cascade filters where necessary to increase their orders. The *out* pointer is required because the function can only return one variable so could not return the *Lout* and *Rout* pointers.

The *setParams()* function links the compression output with the EQ parameters and therefore is a central component. The parameters used to control this process have been named "base" and "ratio" values for both frequency and gain. The base value is the minimum or maximum value, used when gain reduction is zero, and the ratio value controls the change in the parameter above or below the base value by scaling a non-zero gain reduction. An example below demonstrates this for cut-off frequency:

$$f_c = f_b + GRf_r$$

For the example of cut-off frequency f_c and f_b are measured in Hertz and f_r is measured in Hertz/dB; for every decibel of gain reduction the cut-off frequency is changed by the frequency ratio value. An inversion of the ratio values is required when the compression type is set to compression; the gain reduction value is negative and therefore a positive frequency ratio value would result in a negative product. For the resulting multiple to remain positive the frequency ratio must have its sign inverted; a negative gain reduction and negative frequency ratio have a positive product. These values' variables have the identifiers *f_base, f_ratio, g_base* and *g_ratio*.

The *Do()* function takes all of the above principles and coding implementations and returns effected audio. The function receives stereo pointers of audio samples, the regressed FFT trace and two Boolean values which determine whether or not the EQ and compression have been activated; functionality used during development and testing. Firstly the FFT band level is calculated by averaging regression bin amplitudes using the *fft_frequency* and *fft_width* variables. This value is then sent to the *Compression* class with the stereo audio buffers using the *setAudio()* function. Next a loop parses the audio buffers using the *size* variable. The current *i* value is used when calling the *Do()* function of *Compression* member *Comp*, which returns a gain reduction value. A conditional statement then checks to see if compression has been activated using the *DRCio* Boolean, and also checks if the current value of *i* is a multiple of three; the EQ algorithm uses a two sample delay to implement filters and so parameters cannot be allowed to vary more often than every three samples. The variables *F* and *G* are then defined using the base and ratio values described above and passed to the *EQ* member with the *setCoefficients()* function. The filters are then implemented with three conditional statements. The first checks if EQ has been activated with the *EQio* Boolean and whether the current filter type is Low Shelf; if so two shelving filters are connected in series using the *Lcas* and *Rcas* pointers in order to create a fourth order filter. Fourth order was decided based on a compromise between maximising roll off and reducing processing time; the more cascades used the longer it takes to apply the filter. Experimentation was carried out and engineering judgement determined that the roll-off was steep enough using a fourth order filter. The second conditional statement applies the other filter types with no cascading and the third applies no filtering when the *EQio* Boolean is set to false. Finally the two stereo audio buffers are interleaved to give the *out* output pointer which the function returns.

Within *Juce*_vstAudioProcessor the *FBand* members *LS, MP* and *HP* are declared corresponding to the low, low-mid and high-mid frequency bands respectively. Their names are abbreviations of the filters they employ; Low Shelf, Mid Parametric and High Parametric. Nine arrays are then declared which contain compression parameters and initialising EQ parameters; *F, G* and *Q* contain cut-off frequencies, gains and Q- factors of the filters, *T, S, A* and *R* contain threshold, slope, attack time and release time values for the compressors, and *EQio* and *DRCio* contain Boolean values which activate the EQ and compression functions. An enumeration is used to determine the three constants *_LS, _MP* and *_HP* which are used to refer to the relevant band when using these arrays. The *Init* Boolean is required as all of the *FBand* initialisation, and hence the *EQ* and *Compression* initialisation, needs to be carried out within the *processBlock()* function in order to provide access to the buffer size. This value is initialised to be false and at the start of each run of the *processBlock()* function is checked to see if *FBand* initialisation is necessary. Four functions calls per band are necessary to initialiase each member; the *FBand Init()* and *setParams()* functions, and the *EQ setCoefficients()* function and *Compression setParameters()* function which are accessed through the *FBand* member. The *setCoefficients()* and *setParameters()* function are supplied with the relevant parameter arrays described above as arguments. The values used for each band are tabled below:

	LS	MP	HP
EQ Type	Low Shelf	Parametric	Parametric
Compression Type	Compression	Compression	Compression
FFT Band Index	1	10	75
FFT band Index Width	3	9	25
FFT Band Start	22 Hz	345 Hz	4264 Hz
FFT Band End	129 Hz	711 Hz	8333 Hz
FFT Range	107 Hz	366 Hz	4069 Hz
Frequency Base	20 Hz	500 Hz	5000 Hz
Frequency Ratio	5 Hz/dB	0 Hz/dB	0 Hz/dB
Gain Base	-24 dB	0 dB	6 dB
Gain Ratio	1 arb	-1 arb	-1 arb
Threshold	-45 dB	-40 dB	-40 dB
Slope	50 %	50 %	40 %
Attack	20 ms	20 ms	10 ms
Release	200 ms	200 ms	80 ms

The following paragraphs give justification for the values shown in the above table. Unless specified otherwise the values were determined via extensive experimentation and a process of trial and error over the course of development. Aural analysis and informed judgement were used by the author to evaluate each incremental change in parameter, this is especially true of threshold values. The merits of these settings are determined by the testing section of this paper, which utilises statistical analysis to determine the effect of the plugin as it provides a quantitative and scientific testing methodology, which justifies the qualitative methods used to derive parameter values described below.

The low shelving band analyses amplitudes between 22 and 129 Hz; as discussed in section II this frequency range contains noise from the piano's pedals, noise from the recording environment and audio content which needs to be attenuated to make room for other instruments occupying the same spectral space such as the bass drum and bass guitar. A low shelf filter is used with a frequency base of 20 Hz and a gain base of -24 dB. As the level of the band increases the cut-off frequency of the filter increases 5 Hz for every decibel; the louder the piano plays the more frequency content is attenuated. The filter needs to be removing noise between 20 Hz and 50 Hz at all times to, as described in section II, leave room in the mix for other low frequency producing instruments, and sufficient attenuation is achieved with a cut-off frequency in the same margin. The gain reduction needs to be only 6 dB before the cut-off frequency is 50 Hz so the noise attenuation is achieved even at low piano volumes. As the piano is played louder it generates more frequency content below 150 Hz and this is countered by increasing the cut-off frequency of the filter so that the attenuation set by the gain parameter is achieved at higher frequencies. When this filter configuration was first tested aurally during development it was found that the movement in cut-off frequencies was audible at times. To counteract this a gain base of one was employed; for every decibel above the threshold the gain of the filter is increased by one decibel. Therefore as the cut-off frequency increases the gain is "ducked" in order to reduce the severity of the filter. This produces the amount of attenuation required and ensures the process is inaudible. A gentle slope of 50% (2:1 ratio) was used and attack and release times were set to be quite slow, at 20ms and 200 ms respectively, both as further measures for reducing the audibility of the parameter changes.

The mid parametric band operates within a frequency range of 345 to 711 Hz. As described in the alternative systems section this band contains frequencies which make the overall tone of the piano too rich and mask other instruments central to the production such as vocals. A frequency base of 500 Hz was used with a ratio of zero Hz/dB which keeps the cut-off frequency constant, along with a gain base of zero with a ratio of minus one. The cut-off frequency is kept constant because the filter needs to operate on a specific frequency spectrum. The filter acts as a simple compressor; as the level of the band surpasses the threshold attenuation is applied. The same slope and ballistics were used as for the shelving filter to again reduce the audibility of parameter changes by making them as smooth as possible.

The high parametric band functions between 4264 and 8333 Hz, a frequency band which, as discussed in the alternative systems section, contains note attack transients and frequencies which determine the brightness of the piano which need to be boosted. The range of the analysis band increases with frequency as low frequency content is more powerful than high frequency content, but also because the qualities that are altered in each band happen to occupy increasing frequency ranges; the bright frequencies produced by a piano occupy a large frequency range than the noise produced by the pedals. The filter is centred on 5 kHz and as for the mid parametric band this value is kept constant. The gain base is 6 dB and the ratio is -1; at rest the filter applies a 6 dB boost and as the band level increases the boost is reduced. This is because the volume of note attack frequencies increases with overall piano volume as would be expected, making the piano brighter when it is played harder and hence less boost is required. Shorter attack and release times were used because parameter changes are less audible in the higher frequency ranges, but more importantly to allow the filter to respond to transients quickly. To produce this responsiveness ballistics were set low enough to require a slight reduction in slope.

SECTION III

Evaluation

III.1

Testing

The purpose of the prototype is to the control the spectral and dynamic qualities of a piano recording in a way that imitates a mixing engineer in an efficient manner. A series of tests were employed to establish success covering three criteria:

- Control of spectral qualities,
- Control of dynamic qualities,
- Computational efficiency.

Both numeric and subjective testing processes were employed; four statistical tests and a listening test. The four statistical tests comprise two tests analysing the transformation of spectral characteristics and two analysing the transformation of dynamic characteristics. The listening test analyses the overall transformation performed by the plug-in. The tests compare the difference between a target audio sample and unprocessed and processed recorded audio samples. The final test assesses the computational efficiency of the plug-in; performance timers were inserted into the code to record the time taken to perform each task within the signal chain.

The audio tests required the gathering of various audio samples:

- **Target** samples
  - Recorded piano audio mixed with existing plugins and procedures by professionals to create the spectral and dynamic characteristics the plugin attempts to achieve
- **Original** samples
  - Recorded piano audio without any processing applied
- **Processed** samples
  - Recorded piano audio that has been processed by the plugin.

Three “target” samples were extracted from commercially successful songs produced by respected and professionally successful engineers in order to establish their credibility. A sample of the introduction of each song was taken where the piano is the only instrument playing. The audio tests compare mixed and unmixed samples to these targets. Attempts were made to procure unmixed versions of the target samples; however the high profile nature of the artists and songs themselves meant that this was impossible and recordings needed to be made. Comparing recordings to the targets reduces the validity of the numerical tests as the differences between the target, original and processed samples are not solely influenced by the plug-in. The listening test was carried out to provide a method of confirming numerical values; if a failed numerical test corresponded with a successful listening test it could be suggested that the failure was a result of numerical testing limitations. The listening test also gave an insight into the views of potential users of the plug-in - professionals and students within the Music Technology field.

The comparison to target samples provides an analysis of the operational ability of the developed plugin compared existing technologies.

Custom unprocessed “original” samples were made by re-recording the piano parts using three different microphones and configurations. Three configurations were used as it was the maximum number that could be recorded simultaneously with available equipment. Had it been possible three different pianos would have been recorded rather than using three different microphones and configurations, however only one piano and recording environment that produced professional quality recordings was available at the time of testing. It was decided that recordings of more limited tonal variety but of professional standard would be more useful for testing than low quality recordings with large tonal differences. Whilst this is a limitation, its impact on the validity of the testing is minimal because the three recording configurations resulted in three tonally different recordings, which was the intended result of using three pianos. A Steinway D was recorded in a large concert hall using three spaced stereo microphone techniques. The piano lid was fully open as this produces the brightest tone and also increases the directionality of the piano (Senior, 2008). Configuration 1 was a spaced AKG 451 pair with cardioid polar patterns positioned approximately six feet from the piano, configuration 2 was a Neumann U87 pair with omnidirectional polar patterns positioned above the strings, and configuration 3 was a Neumann U89 pair with omnidirectional polar patterns positioned above the hammers. Figure III.1.1 shows photographs of these configurations to provide a more detailed description. The recordings were completed using the Tascam US-800 audio interface to ensure all three configurations were recorded using the same pre-amps; each pre-amp colours a recording differently and would produce tonal changes impossible to measure, reducing the validity of tests analysing tone. The recording process resulted in nine audio samples, three performances with three recording configurations each.

The “processed” samples were created by processing the original samples with the prototype plugin.

In order to carry out the numeric tests various data streams needed to be exported from the plug-in in real-time into an analysable format. The testing required RMS levels, regressed FFT curves, compression input levels, EQ parameters and performance timers. Microsoft Excel was chosen as the software package to analyse the data with, primarily because it accommodates the importing of data in Comma Separated Values (CSV) format and secondarily because it supports the ability to reference cells contained in a different workbook, making it much easier to organise and present large amounts of data. To export the data from the plug-in in real-time the WriteToFile class was written (see Appendix V.1.8). The class utilises the streaming abilities of the base *ios* class through the *ofstream* data type to write data to a text file. The data is organised in CSV format to allow it to be easily imported into Microsoft Excel.

The listening test asked subjects to rate the original and processed samples based on their similarity to the target. A website was made to allow easy distribution of the test which provided streaming version of nine audio samples; the target and one original and processed configuration from each song. Using all three recording configurations was considered to be an overwhelming amount of data to analyse with critical listening. Subjects rated out of five based on the tonal and dynamic characteristics of the samples compared to the target; a successful result would have a higher score for the processed sample than the original.

The performance timing was implemented using the high-resolution performance counter included in all versions of Microsoft Windows since Windows 2000. The counter is incremented at the clock speed of the processor and can be used to measure time to very high levels of accuracy; the number of clicks between two events is counted and divided by the click frequency to produce a value with a unit of time. Four sections were defined from the *processBlock()* function to be timed; the initialisation section, including *.Init()* functions and RMS calculations, the pre-effect FFT routine, effect implementation routine, and post-effect FFT routine. Although the total time would be the focus of the test it was relevant to the continuing development process to know which areas of the plugin performed efficiently and inefficiently.

Wav Filename	Ratings	Artist - Title	Source	Performance	Recording (Known details)	Mixing (Known details)
Target - 1	PCM Wave 16 bit 44.1 kHz	The Cinematic Orchestra – To Build a Home	The Cinematic Orchestra – "Ma Fleur", 2007, Ninja Tune	Patrick Watson (Discogs)	Recorded by Steve Hodge, location unknown (Discogs)	Mixed by Steve Hodge and Jason Swinscoe, location unknown (Discogs)
Target - 2	PCM Wave 16 bit 44.1 kHz	Coldplay – Trouble	Coldplay – "Parachutes", 2000, Parlophone	Chris Martin (Discogs)	Recorded by Ken Nelson at Parr Street Studios, Liverpool (Ken Nelson SOS article) • 2 mics set up, one bright one full • Full mic used in mix	Mixed by Michael Brauer at Quad Studios, New York (Michael Brauer SOS article)
Target - 3	PCM Wave 16 bit 44.1 kHz	Coldplay - Scientist	Coldplay – "Rush of Blood to the Head", 2002, Parlophone	Chris Martin (Discogs)	Album recorded by Ken Nelson at: • Studio 2, Mayfair Studios, London • Studio 3 Parr Street Studios, Liverpool • Studio 1, Air Studios, London (Discogs)	Mixed by Coldplay, Ken Nelson and Mark Phythian, unknown location. Mastered by George Marino, unknown location. (Discogs)

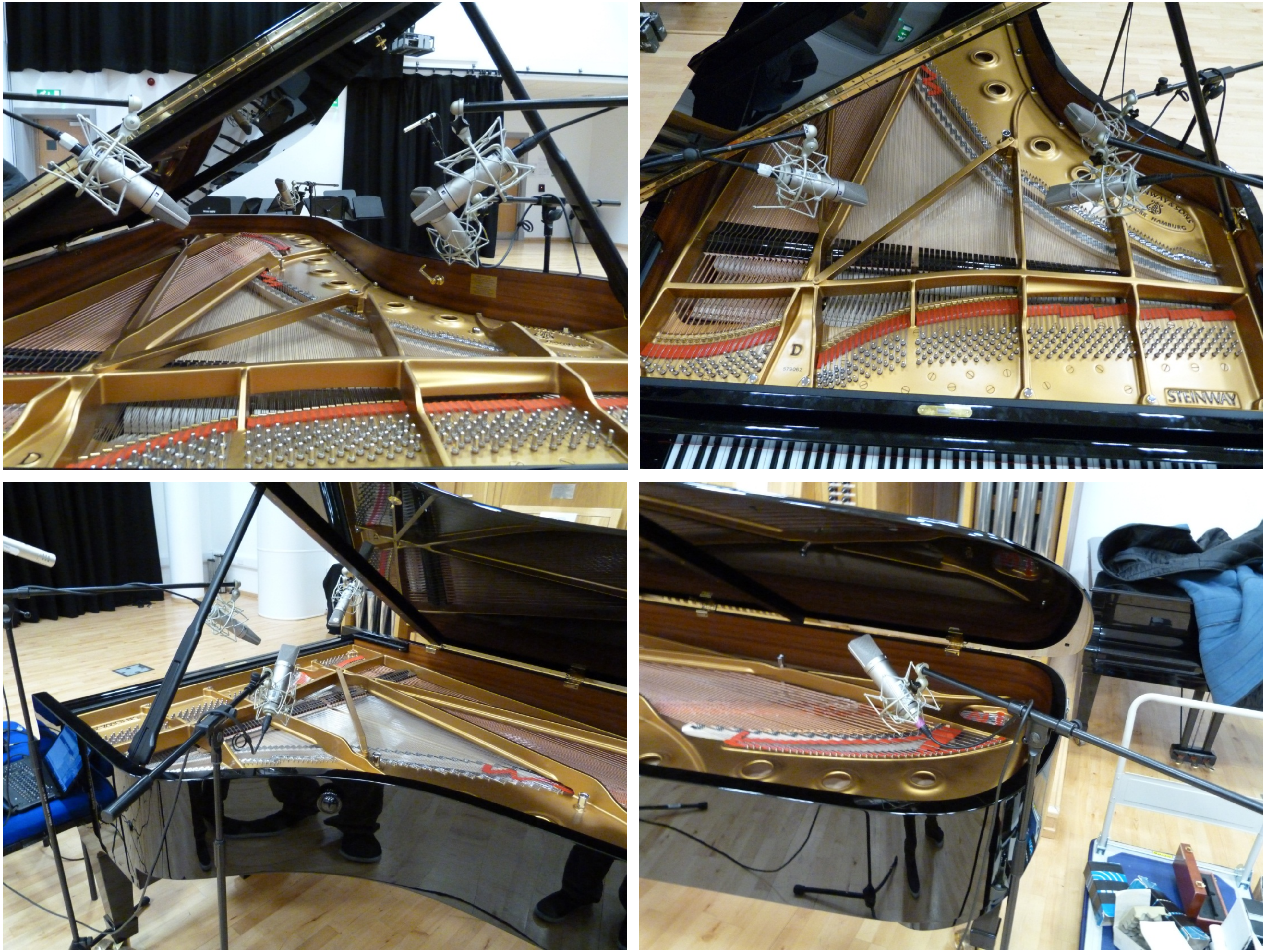


Figure III.1.1

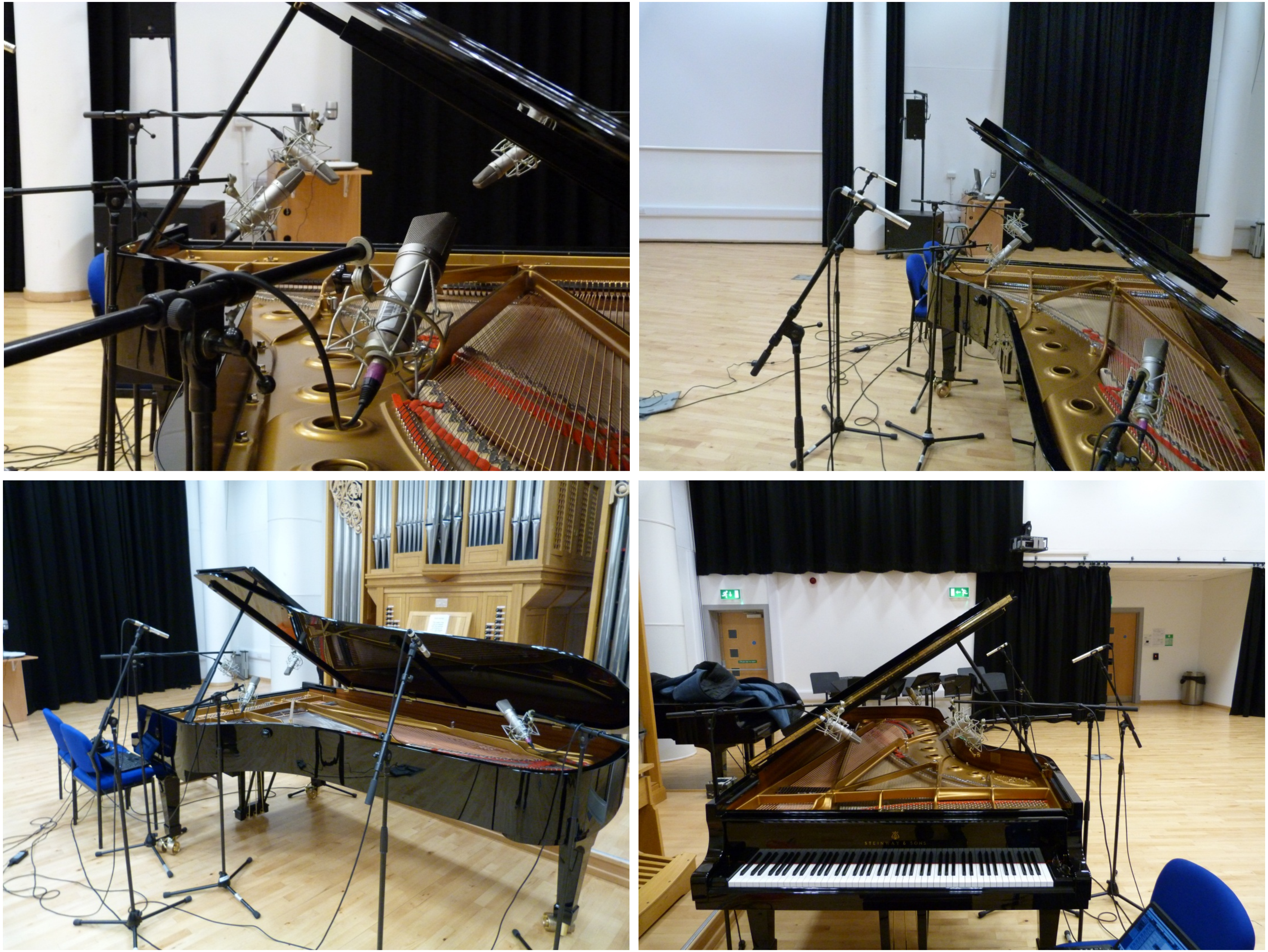


Figure III.1.1

The Sample Average Test was the first of the two numerical tests used to assess the ability of the plug-in to control spectral characteristics of piano recordings. The test utilised the regression curves in order to calculate average amplitude over the length of the sample at the critical frequencies discussed throughout this paper; 130 Hz, 500 Hz and 5 kHz. The average amplitude across the sample for these three bins was calculated for the target sample and all three original configurations, followed by the difference between each original configuration and the target. The same process was then carried out between the processed configurations and target. The target vs. original and target vs. processed results were then compared and two sets of results formulated; one set demonstrating whether the relevant positive or negative attenuation had been achieved at each frequency bin and the other set demonstrating whether or not the processed bins were more similar to the target bins than the original bins. This two result technique was used for all four tests, and for ease of communication in the paper, the first set of results will be called the basic results, and the second set the delta results. Expected values for basic results are a direction of change; a positive or negative value depending on the context. Expected values for delta results are negative values indicating that the processed sample is more similar to the target sample than the unprocessed sample.

Expected results

Basic results – negative values at the low and mid bands and positive values at the high band, indicating attenuation and boost respectively.

Delta results – negative values indicating a progression towards the target sample between original and processed samples.

III.1.3 Sample Transient Test

The second numerical test used to assess spectral control was the Sample Transient Test. Transients were analysed because the filters are operating at their most severe during periods of high amplitude due to the control of their parameters with compression. The test analyses whether or not the filters are behaving the way they should be when their parameters are at the maximum of their individual ranges. For this test the RMS data was used to select a transient from each song with two consecutive regression curve windows at the peak of that transient selected to be analysed. For example for Target – 1 the second transient, the A Major chord, was chosen as the transient. A plot of the RMS data was used to find the numBuffers value for the peak of the transient for each sample. The numBuffers value was then cross-referenced with the regression curve data to find the two adjacent curves created with windows of samples from the peak of the transient to be analysed. Two curves were used so that if one curve appeared to contain anomalies they could be confirmed by comparison with a second. The amplitudes at the critical frequencies 130 Hz, 500 Hz and 5 kHz were then recorded from each curve and the difference between each original and processed configuration and the target sample was calculated. Basic and delta results were then formulated in the same way as the Sample Average Test.

Expected results

Basic results – negative values at the low and mid bands and positive values at the high band, indicating attenuation and boost respectively.

Delta results – negative values indicating a progression towards the target sample between original and processed samples.

The RMS Variance Test is the first of the two numerical tests used to analyse the plug-in's ability to control the dynamics of a piano recording. It utilises the RMS data and two simple statistical measures of variance; standard deviation and average absolute deviation. Statistical variance indicates the dispersion, or range, of a data set. This can be used to measure the level of compression applied to audio as compressed audio has a lower dynamic range, and therefore lower variance, than uncompressed audio.

Let a set of *N* discrete samples be denoted x_i, where *i = 1,…,N*. The mean of the data, μ, is calculated:

$$\mu = \frac{\sum_{i=1}^{N} x_i}{N}$$

The deviation of each of sample, d_i, is then calculated as the distance between it and the mean:

$$d_i = x_i - \mu$$

The variance of the data set, σ^2, is then defined as the sum of the squares of the deviations, and the standard deviation, σ, the square root of the variance:

$$\sigma = \sqrt{\sum_{i=1}^{N} (x_i - \mu)^2}$$

Average absolute deviation is commonly used as an alternative to standard deviation and to maximise the reliability of the testing process it was decided to implement this second measure of variance to be able to rule out anomalies. Absolute average deviation is calculated very similarly to standard deviation, however it allows for the use of different measures of central tendency; either the mean or median. For the purposes of testing the mean will be used as the measure of central tendency, because within a digital audio system the median value is likely to be very close to zero due to the oscillating nature of the data.

As for the standard deviation the mean, µ, of the data set is first calculated. Then the absolute deviation, *Di*, of each data point is calculated:

$$Di = |x_i - \mu|$$

Average absolute deviation, D_μ, is then calculated as the average of the absolute deviations of the data set:

$$D_\mu = \frac{\sum_{i=1}^{N} |x_i - \mu|}{N}$$

The two measures of variance were applied to the RMS data for each of the original configurations and the target sample. The variances of the original configurations were then expressed as a percentage of the target sample, and the same process carried out with the processed configurations. The percentage value was calculated as follows:

$$\frac{Config}{Target}\% - 100$$

This equation results in a value of zero if the recorded sample and target values are equal, a positive value if the recorded sample value is larger and a negative value is the recorded sample value is smaller. Basic results were then formed by checking that there had been a reduction in variance between the original to target and processed target comparisons, and delta results formed by checking the processed variance were closer to that of the target than the original variances.

Expected results

Basic results – negative values indicating a reduction in dynamic range.

Delta results – negative values indicating a progression towards the target sample between original and processed samples.

III.1.5 FFT Band Variance Test

The FFT Band Variance test is the second of the two numerical tests used to analyse the effect of the plug-in on dynamic qualities of piano recordings. The test is very similar to the RMS Variance Test; standard deviation and average absolute deviation were applied to the levels for the low, mid and high frequency bands, and results expressed as percentages formulated in exactly the same way as the RMS Variance Test. The same frequency bands were used as for the sample average test; 120Hz, 500Hz and 5kHz. Where the RMS test shows overall change in dynamic range, the FFT Band test demonstrates change in dynamic range in the each of the critical frequency bands. Basic and delta results are then formed in the same way as the RMS Variance Test.

Expected results

Basic results – negative values indicating a reduction in dynamic range.

Delta results – negative values indicating a progression towards the target sample between original and processed samples.

III.1.6 Listening Test

Professionals and students within the Music Technology field were played the audio samples and asked to score each original and processed configuration on a scale of 0 to 5 based on its similarity to the target. A score of five meant the sample was very similar to the target and a score of zero meant the sample was not at all similar to the target. Questions were asked of the sample characteristics as a whole rather than of the three frequency bands and dynamic range to reduce the complexity of the test and encourage more subjects to take part. These questions produced delta results; basic results were not pursued as they could be reliably produced through numerical testing.

Expected results

Processed samples scoring higher than original samples to indicate a progression towards the target sample between original and processed samples.

IV.1.4 Performance Timing

To carry out the performance test timing data was exported from all nine of the Original samples at the four points described above at a rate of 1Hz. The values for each sample are then averaged and summed to form a total:

$$T_{total} = T_{Init} + T_{Pre-FFT} + T_{FX} + T_{Post-FFT}$$

$$T = \frac{\sum_{i=0}^{N} t_i}{N}$$

The total time is then compared to a target of 10ms, chosen as the largest tolerable delay to be introduced by the plugin.

Expected results

A total time to carry out the *processBlock()* function of 10ms or less

III.2

Results

Sample 31 – Original vs. Target

	Amplitude Values (dB)				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	150.11	149.63	149.75	150.44	
Mid (475 Hz)	143.54	143.12	143.49	145.11	
High (5.7 kHz)	94.35	89.60	79.41	85.14	
	Config vs. Target Difference (dB)				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		-0.48	-0.35	0.33	-0.17
Mid (475 Hz)		-0.42	-0.05	1.57	0.37
High (5.7 kHz)		-4.75	-14.94	-9.22	-9.64

Sample 31 – Process vs. Target

	Amplitude Values (dB)				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	150.11	146.69	146.46	146.50	
Mid (475 Hz)	143.54	142.40	142.49	142.29	
High (5.7 kHz)	94.35	94.77	85.10	90.09	
	Config vs. Target Difference (dB)				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		-3.42	-3.64	-3.61	-3.56
Mid (475 Hz)		-1.14	-1.05	-1.25	-1.15
High (5.7 kHz)		0.42	-9.25	-4.26	-4.36

Sample 2 – Original vs. Target

	Amplitude Values (dB)				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	136.52	147.53	147.20	146.56	
Mid (475 Hz)	145.07	147.17	146.70	146.19	
High (5.7 kHz)	100.30	95.95	89.51	91.97	
	Config vs. Target Difference (dB)				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		11.01	10.68	10.04	10.58
Mid (475 Hz)		2.10	1.63	1.13	1.62
High (5.7 kHz)		-4.34	-10.79	-8.33	-7.82

Sample 2 – Processed vs. Target

	Amplitude Values (dB)				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	136.52	143.31	142.64	143.35	
Mid (475 Hz)	145.07	145.07	144.83	145.71	
High (5.7 kHz)	100.30	100.85	94.87	98.74	
	Config vs. Target Difference (dB)				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		6.79	6.12	6.83	6.58
Mid (475 Hz)		0.00	-0.23	0.64	0.14
High (5.7 kHz)		0.55	-5.42	-1.56	-2.14

	Amplitude Values (dB)				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	153.71	153.18	153.59	154.68	
Mid (475 Hz)	154.66	153.09	152.91	152.75	
High (5.7 kHz)	109.06	97.64	92.36	95.20	
	Config vs. Target Difference (dB)				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		-0.54	-0.12	0.96	0.10
Mid (475 Hz)		-1.56	-1.74	-1.91	-1.74
High (5.7 kHz)		-11.42	-16.70	-13.86	-13.99

Basic Results

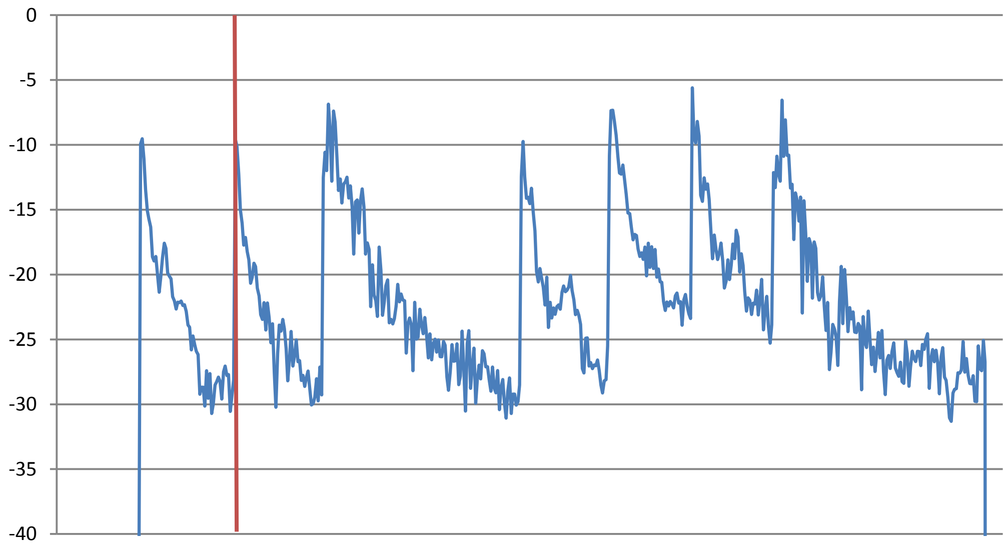
SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Low	-2.94	-3.29	-3.94	**-3.39**
	Mid	-0.72	-1.00	-2.82	**-1.51**
	High	5.17	5.69	4.96	**5.27**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Low	-4.22	-4.56	-3.21	**-4.00**
	Mid	-2.10	-1.86	-0.48	**-1.48**
	High	4.90	5.37	6.77	**5.68**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Low	-5.74	-5.27	-5.81	**-5.60**
	Mid	-3.06	-2.88	-3.34	**-3.09**
	High	4.47	4.64	4.51	**4.54**

	Amplitude Values (dB)				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	153.71	147.44	148.33	148.87	
Mid (475 Hz)	154.66	150.03	150.04	149.41	
High (5.7 kHz)	109.06	102.11	97.00	99.71	
	Config vs. Target Difference (dB)				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		-6.27	-5.38	-4.84	-5.50
Mid (475 Hz)		-4.63	-4.62	-5.24	-4.83
High (5.7 kHz)		-6.95	-12.06	-9.34	-9.45

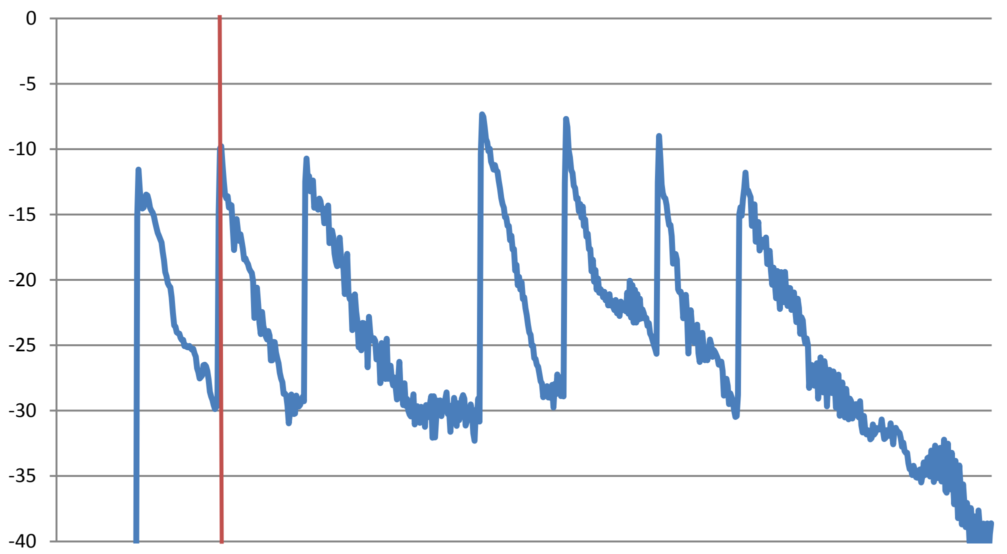
Delta Results

SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Low	2.94	3.29	3.28	**3.17**
	Mid	0.72	1.00	-0.33	**0.47**
	High	-4.33	-5.69	-4.96	**-4.99**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Low	-4.22	-4.56	-3.21	**-4.00**
	Mid	-2.10	-1.40	-0.48	**-1.33**
	High	-3.79	-5.37	-6.77	**-5.31**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Low	5.74	5.27	3.88	**4.96**
	Mid	3.06	2.88	3.34	**3.09**
	High	-4.47	-4.64	-4.51	**-4.54**

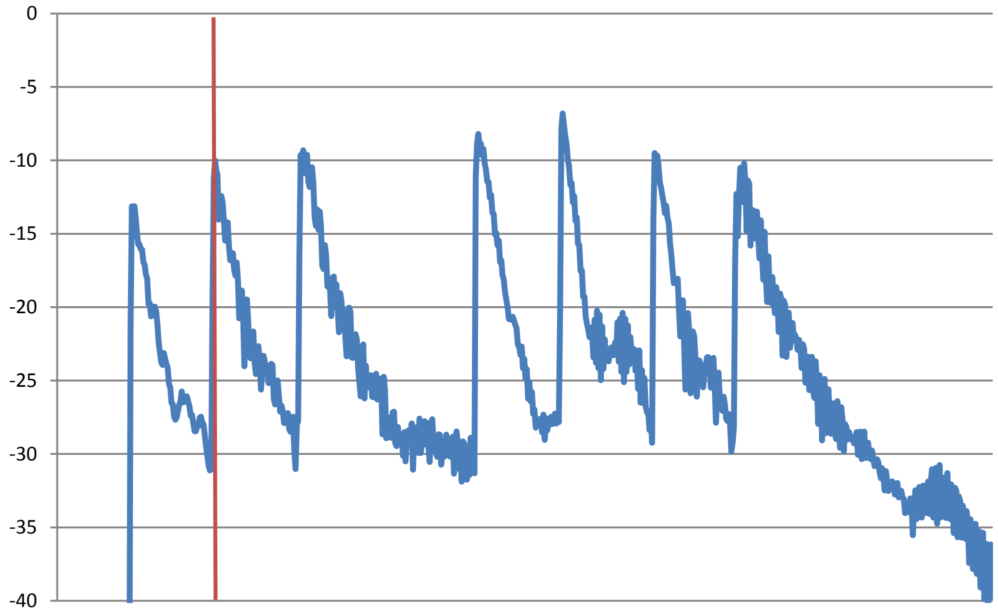
Target – 1 – RMS Plot – Time vs. RMS(dB)



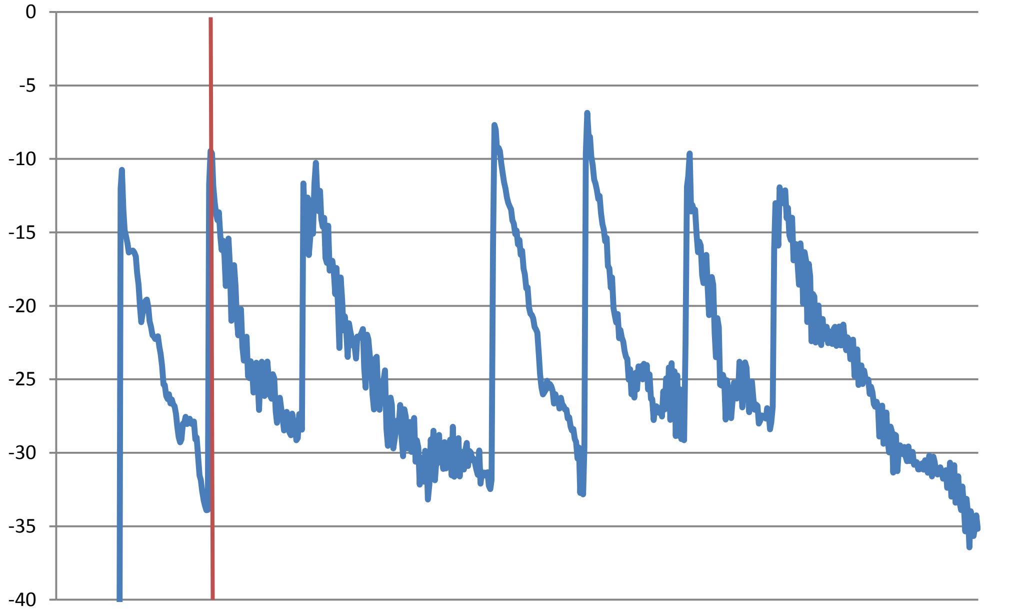
Original – 1 - Config 1 – RMS Plot – Time vs. RMS(dB)



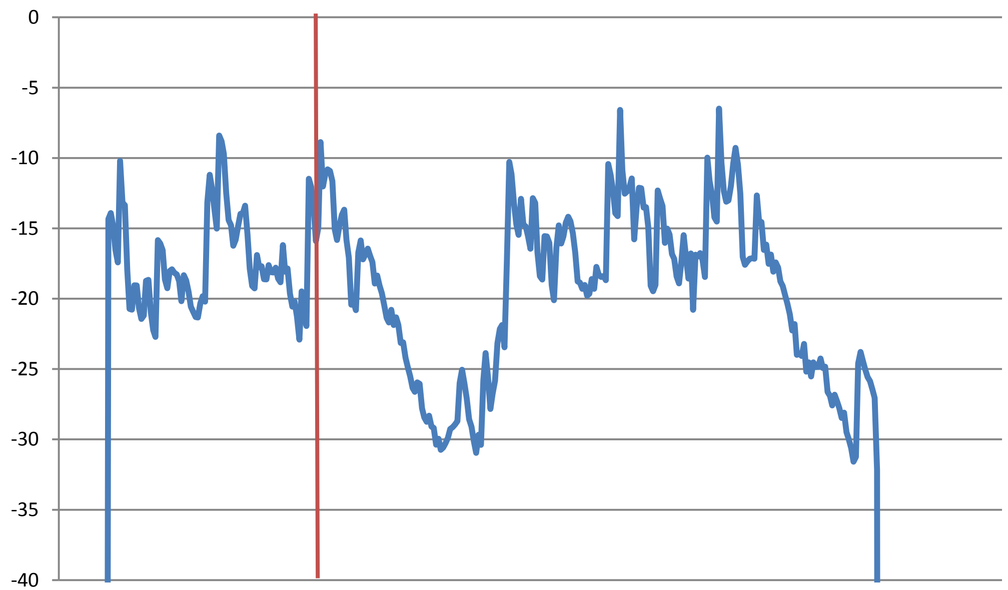
Original – 1 – Config 2 – RMS Plot – Time vs. RMS(dB)



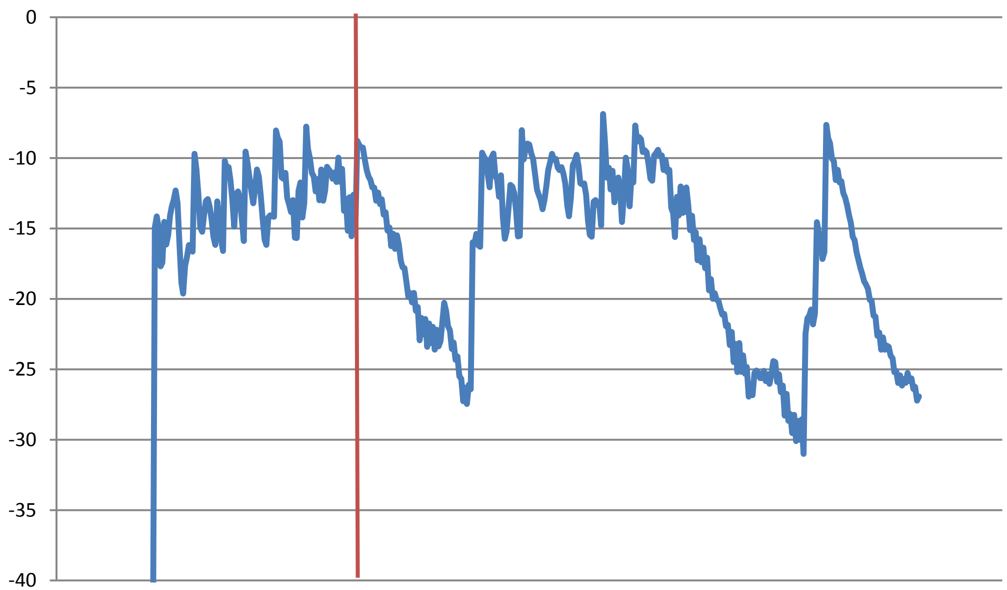
Original - 1 – Config 3 – RMS Plot – Time vs. RMS(dB)



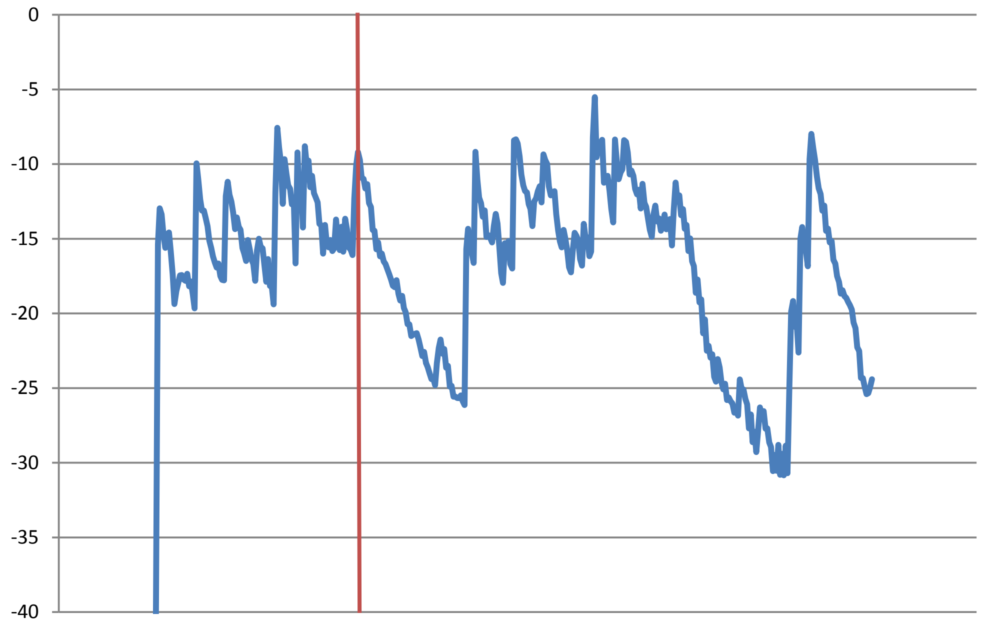
Target – 2 – RMS Plot – Time vs. RMS(dB)



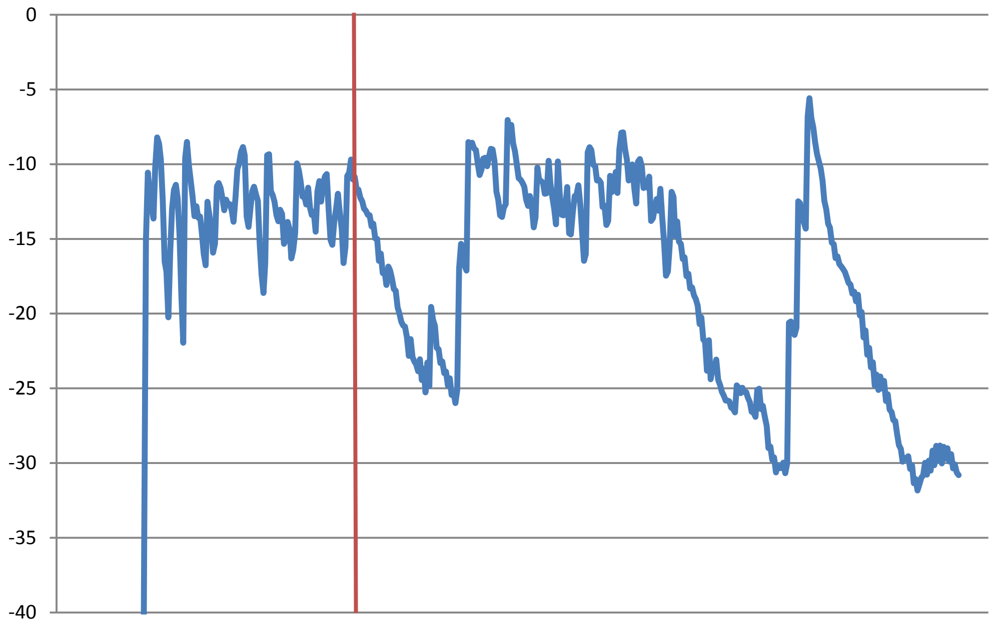
Original – 2 - Config 1 – RMS Plot – Time vs. RMS(dB)



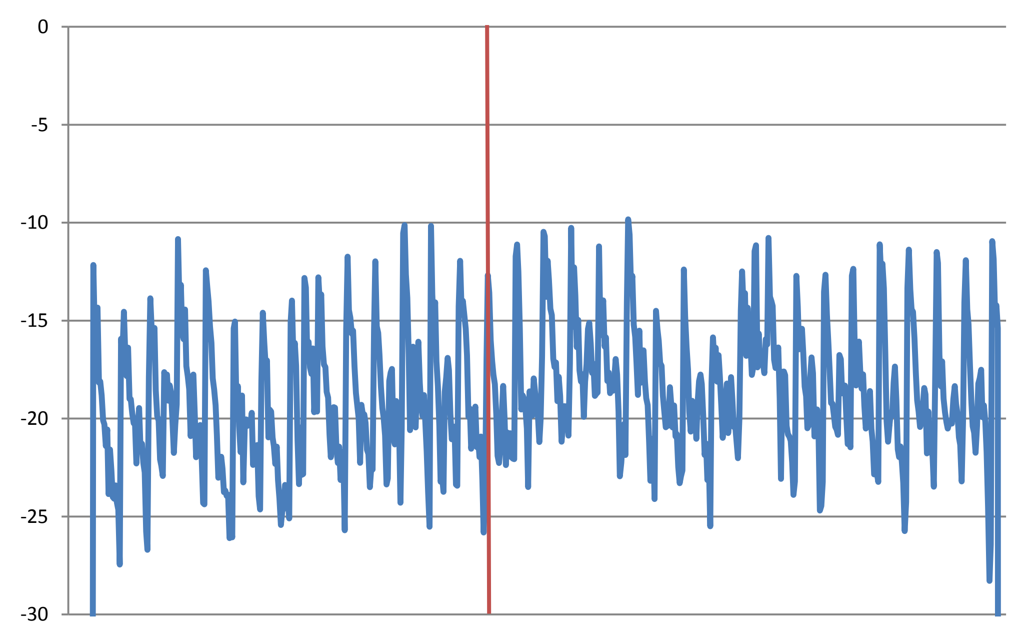
Original – 2 – Config 2 – RMS Plot – Time vs. RMS(dB)



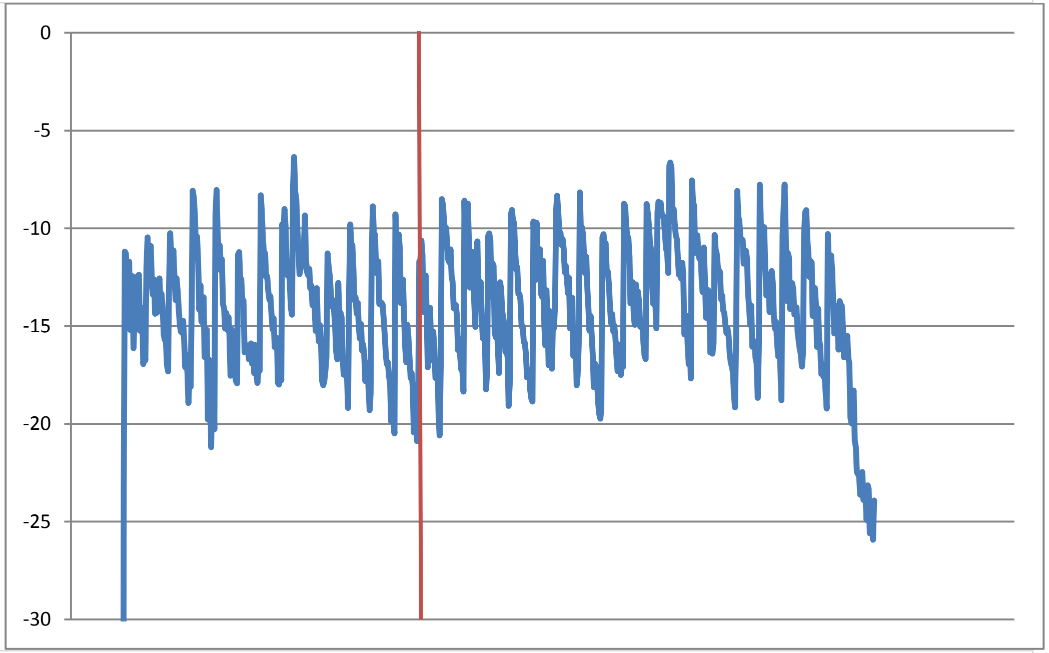
Original - 2 – Config 3 – RMS Plot – Time vs. RMS(dB)



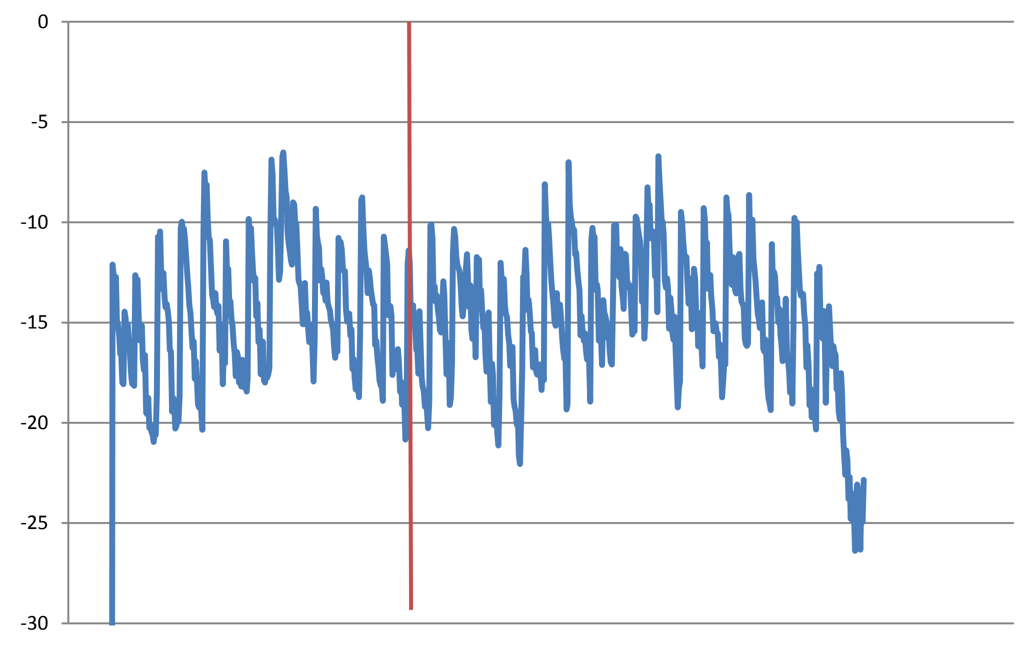
Target – 3 – RMS Plot – Time vs. RMS(dB)



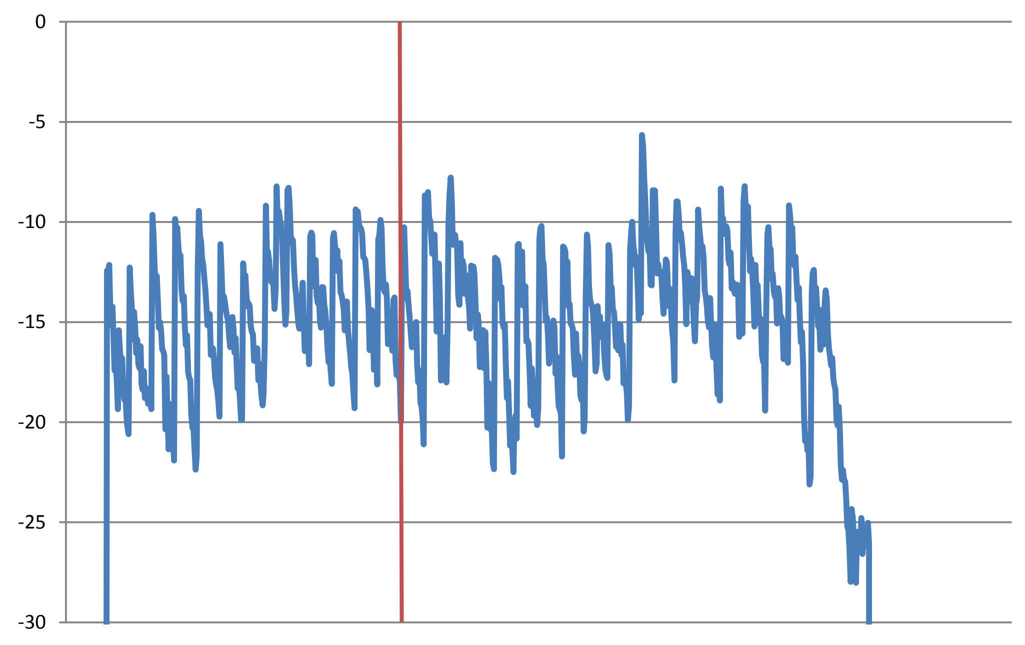
Original – 3 - Config 1 – RMS Plot – Time vs. RMS(dB)



Original – 3 – Config 2 – RMS Plot – Time vs. RMS(dB)



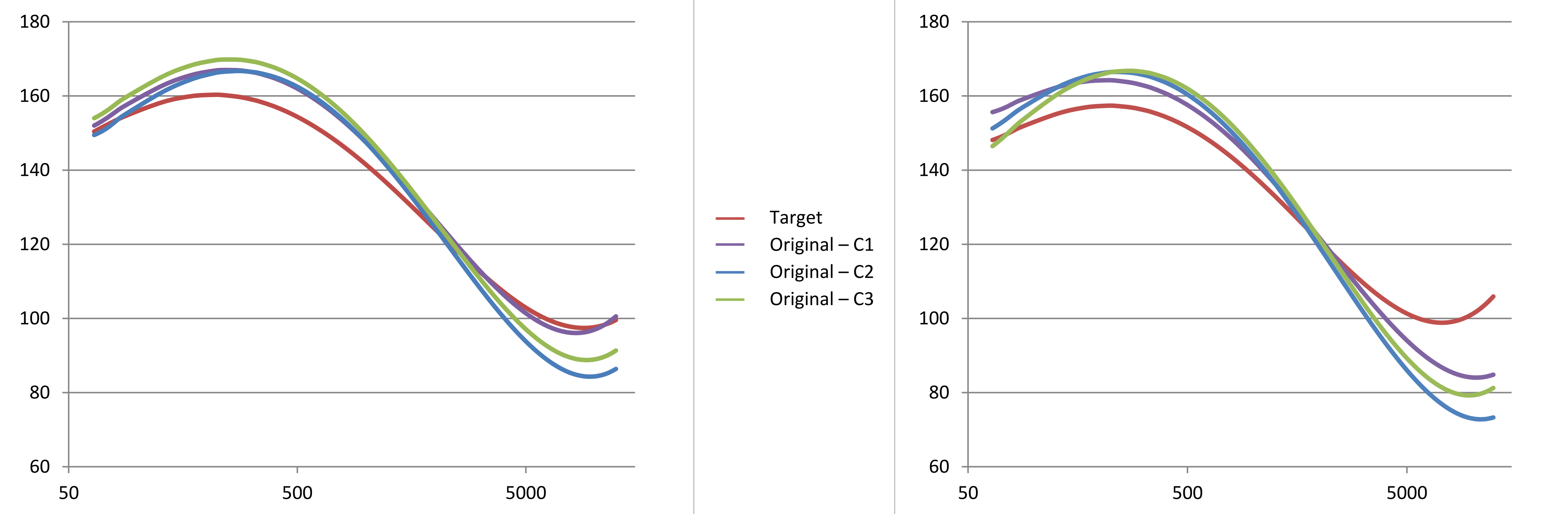
Original - 3 – Config 3 – RMS Plot – Time vs. RMS(dB)



Sample 1 – Original vs. Target

Sample 1 – Window 1 – Regression Curve – Freq. (Hz) vs Amp. (dB)

Sample 1 – Window 2 – Regression Curve – Freq. (Hz) vs Amp. (dB)



Sample 1 - Window 1

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	158.36	162.86	161.12	165.30	
Mid (475 Hz)	155.10	162.74	163.21	165.51	
High (5.7 kHz)	100.79	99.04	90.48	94.02	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		4.50	2.76	6.93	4.73
Mid (475 Hz)		7.64	8.11	10.41	8.72
High (5.7 kHz)		-1.75	-10.30	-6.77	-6.27

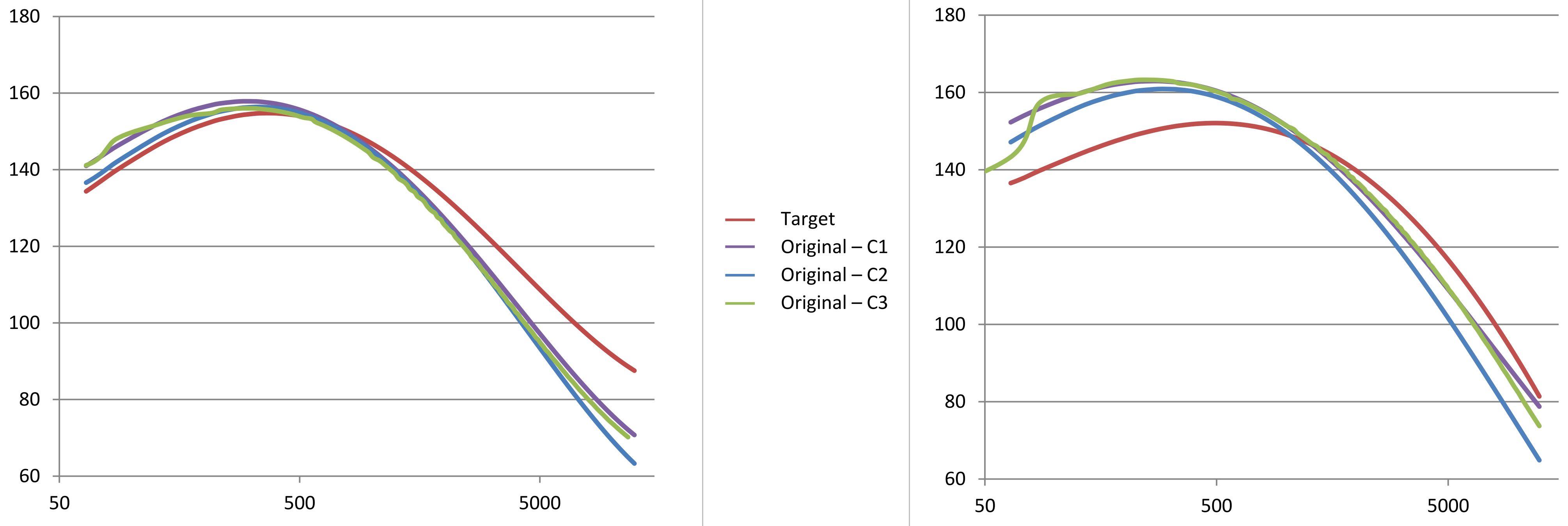
Sample 1 – Window 2

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	155.32	162.44	162.49	160.63	
Mid (475 Hz)	152.31	158.38	161.26	162.75	
High (5.7 kHz)	99.87	90.95	82.18	85.80	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		7.12	7.17	5.31	6.53
Mid (475 Hz)		6.08	8.95	10.45	8.49
High (5.7 kHz)		-8.92	-17.69	-14.07	-13.56

Sample 2 – Original vs. Target

Sample 2 – Window 1 – Regression Curve – Freq. (Hz) vs Amp. (dB)

Sample 2 – Window 2 – Regression Curve – Freq. (Hz) vs Amp. (dB)



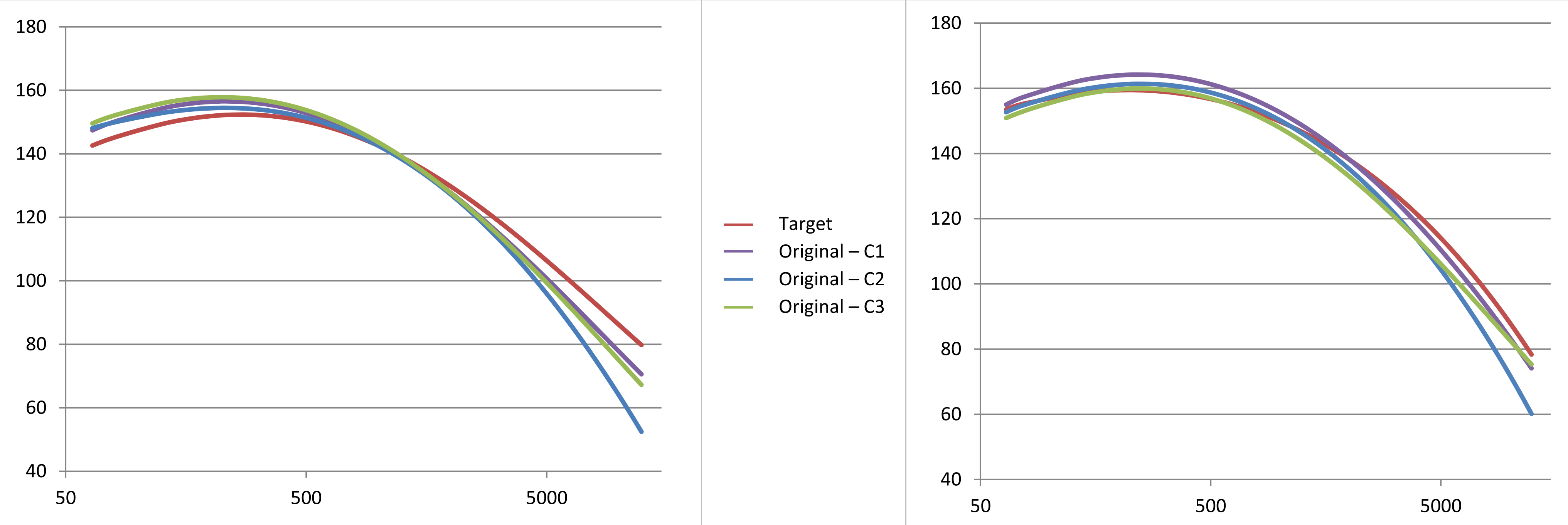
Sample 2 - Window 1

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	146.55	152.06	148.69	148.00	
Mid (475 Hz)	154.26	156.11	155.23	154.97	
High (5.7 kHz)	105.09	92.70	88.42	91.26	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		5.50	2.14	1.44	3.03
Mid (475 Hz)		1.85	0.97	0.71	1.17
High (5.7 kHz)		-12.39	-16.66	-13.82	-14.29

Sample 2 – Window 2

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	144.16	159.86	156.35	162.23	
Mid (475 Hz)	152.07	160.84	159.28	159.97	
High (5.7 kHz)	112.18	104.56	96.33	103.47	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		15.70	12.19	18.06	15.32
Mid (475 Hz)		8.76	7.20	7.90	7.95
High (5.7 kHz)		-7.62	-15.85	-8.71	-10.73

Sample 3 – Original vs. Target


Sample 3 – Window 1 – Regression Curve – Freq. (Hz) vs Amp. (dB)

Sample 3 – Window 2 – Regression Curve – Freq. (Hz) vs Amp. (dB)

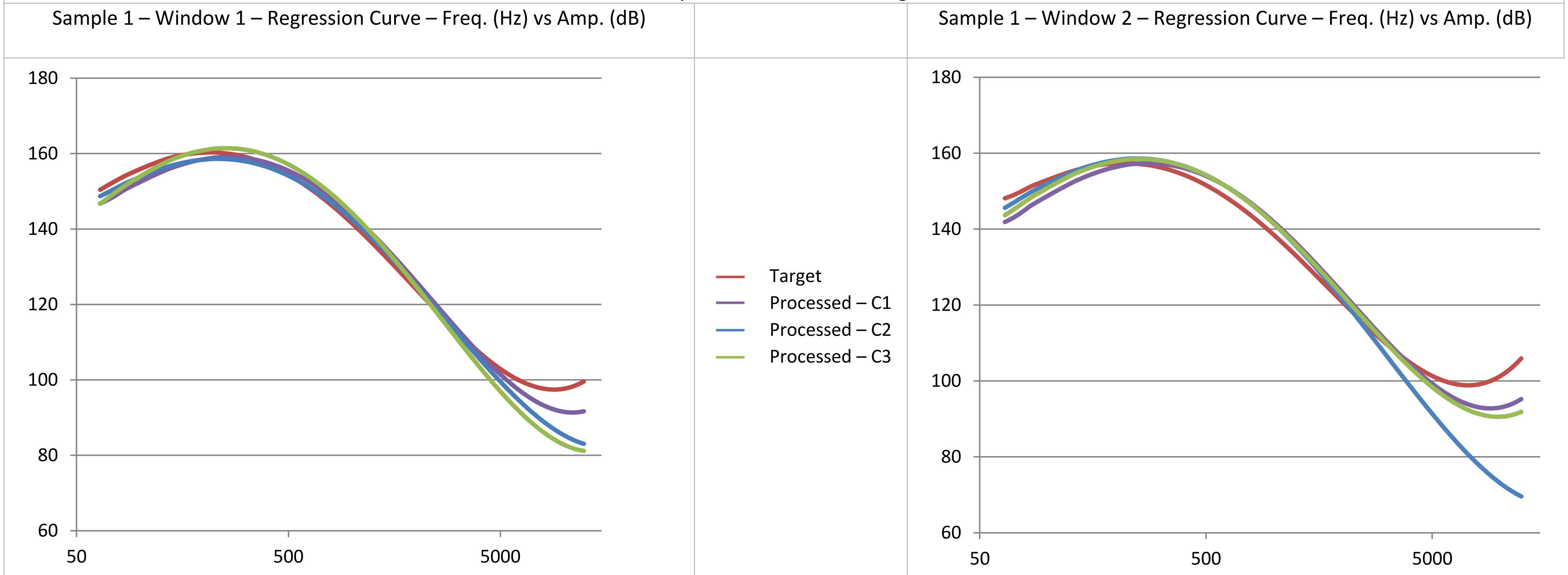
Sample 3 - Window 1

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	149.57	154.49	152.99	156.09	
Mid (475 Hz)	150.54	153.33	151.76	154.27	
High (5.7 kHz)	102.45	96.25	90.20	94.69	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		4.92	3.42	6.53	4.96
Mid (475 Hz)		2.79	1.22	3.74	2.59
High (5.7 kHz)		-6.20	-12.24	-7.76	-8.73

Sample 3 – Window 2

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	158.34	162.01	159.21	157.63	
Mid (475 Hz)	157.08	161.72	159.12	157.37	
High (5.7 kHz)	109.52	105.50	98.62	101.62	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		3.67	0.87	-0.72	1.27
Mid (475 Hz)		4.64	2.04	0.30	2.33
High (5.7 kHz)		-4.02	-10.90	-7.91	-7.61

Sample 1 – Processed vs. Target


Sample 1 – Window 1 – Regression Curve – Freq. (Hz) vs Amp. (dB)

Sample 1 – Window 2 – Regression Curve – Freq. (Hz) vs Amp. (dB)

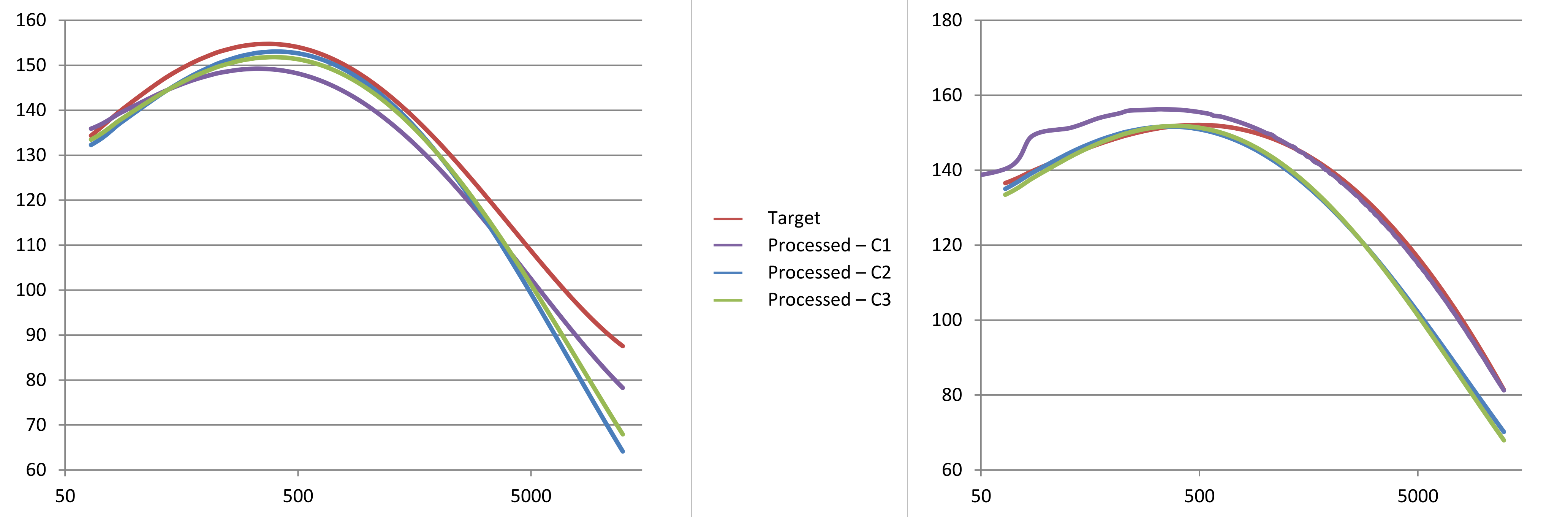
Sample 1 - Window 1

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	158.36	155.40	156.21	157.45	
Mid (475 Hz)	155.10	156.06	154.75	157.78	
High (5.7 kHz)	100.79	98.58	96.16	93.39	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		-2.96	-2.15	-0.91	-2.01
Mid (475 Hz)		0.96	-0.35	2.69	1.10
High (5.7 kHz)		-2.21	-4.63	-7.40	-4.74

Sample 1 – Window 2

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	155.32	152.35	155.12	154.37	
Mid (475 Hz)	152.31	154.58	154.89	154.86	
High (5.7 kHz)	99.87	96.81	87.18	95.74	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		-2.97	-0.20	-0.95	-1.37
Mid (475 Hz)		2.28	2.59	2.55	2.47
High (5.7 kHz)		-3.06	-12.69	-4.13	-6.63

Sample 2 – Processed vs. Target



Sample 2 – Window 1 – Regression Curve – Freq. (Hz) vs Amp. (dB)

Sample 2 – Window 2 – Regression Curve – Freq. (Hz) vs Amp. (dB)

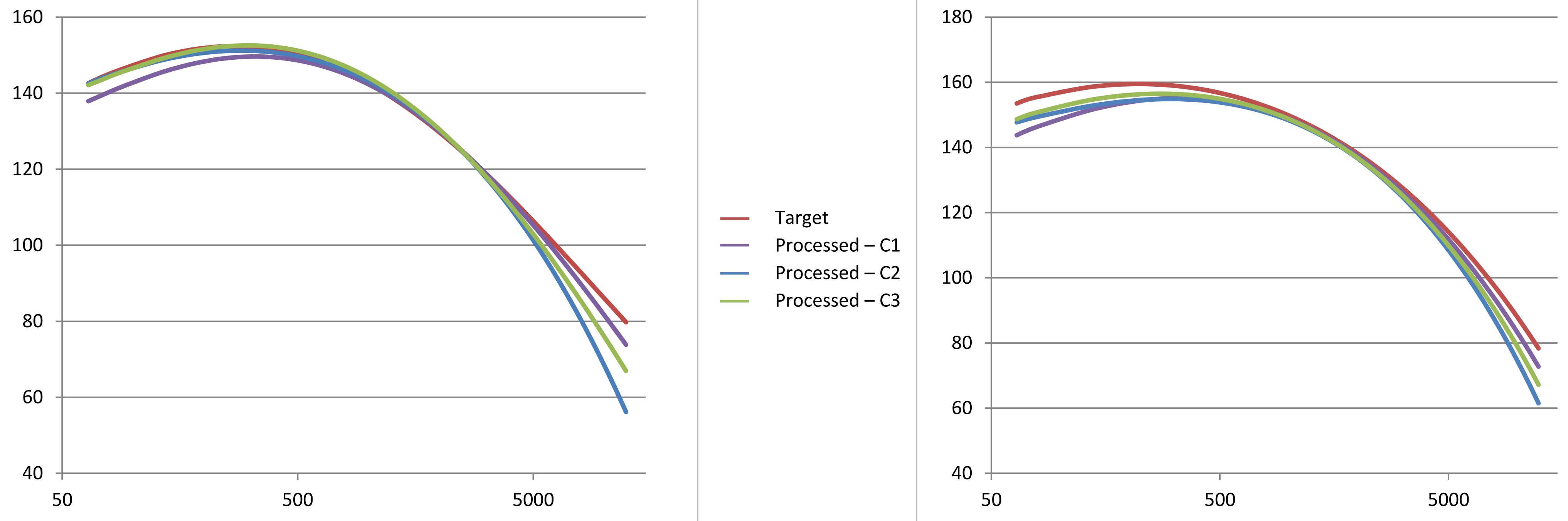
Sample 2 - Window 1

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	146.55	143.89	152.79	139.84	
Mid (475 Hz)	154.26	148.43	156.39	149.87	
High (5.7 kHz)	105.09	98.59	108.63	96.27	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		-2.66	6.24	-6.72	-1.05
Mid (475 Hz)		-5.83	2.13	-4.39	-2.70
High (5.7 kHz)		-6.50	3.55	-8.81	-3.92

Sample 2 – Window 2

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	144.16	153.87	144.80	143.67	
Mid (475 Hz)	152.07	155.35	151.14	151.52	
High (5.7 kHz)	112.18	109.74	97.46	96.35	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		9.70	0.64	-0.50	3.28
Mid (475 Hz)		3.28	-0.93	-0.55	0.60
High (5.7 kHz)		-2.44	-14.72	-15.83	-11.00

Sample 3 – Processed vs. Target



Sample 3 – Window 1 – Regression Curve – Freq. (Hz) vs Amp. (dB)

Sample 3 – Window 2 – Regression Curve – Freq. (Hz) vs Amp. (dB)

Sample 3 - Window 1

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	149.57	145.28	148.51	148.88	
Mid (475 Hz)	150.54	148.88	150.01	151.46	
High (5.7 kHz)	102.45	100.90	95.67	97.99	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		-4.29	-1.06	-0.69	-2.01
Mid (475 Hz)		-1.66	-0.53	0.92	-0.42
High (5.7 kHz)		-1.54	-6.78	-4.46	-4.26

Sample 3 – Window 2

	Amplitude Values				
	Target	Config 1	Config 2	Config 3	
Low (130 Hz)	158.34	151.08	152.47	154.30	
Mid (475 Hz)	157.08	154.56	154.05	155.20	
High (5.7 kHz)	109.52	106.84	102.77	104.64	
	Config vs. Target Difference				
		Config 1	Config 2	Config 3	Mean
Low (130 Hz)		-7.26	-5.88	-4.05	-5.73
Mid (475 Hz)		-2.51	-3.03	-1.88	-2.47
High (5.7 kHz)		-2.68	-6.75	-4.88	-4.77

Basic Results

Window 1

SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Low	-7.45	-4.91	-7.84	**-6.74**
	Mid	-6.68	-8.47	-7.72	**-7.62**
	High	-0.46	5.67	-0.63	**1.53**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Low	-8.17	4.10	-8.16	**-4.07**
	Mid	-7.68	1.16	-5.10	**-3.87**
	High	5.89	20.21	5.01	**10.37**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Low	-9.21	-4.48	-7.22	**-6.97**
	Mid	-4.45	-1.75	-2.82	**-3.01**
	High	4.66	5.46	3.30	**4.47**

Window 2

SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Low	-10.09	-7.37	-6.26	**-7.91**
	Mid	-3.80	-6.36	-7.90	**-6.02**
	High	5.86	5.00	9.94	**6.93**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Low	-6.00	-11.55	-18.56	**-12.04**
	Mid	-5.49	-8.14	-8.45	**-7.36**
	High	5.18	1.13	-7.12	**-0.27**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Low	-10.93	-6.75	-3.33	**-7.00**
	Mid	-7.15	-5.07	-2.18	**-4.80**
	High	1.34	4.14	3.02	**2.84**

Delta Results

Window 1

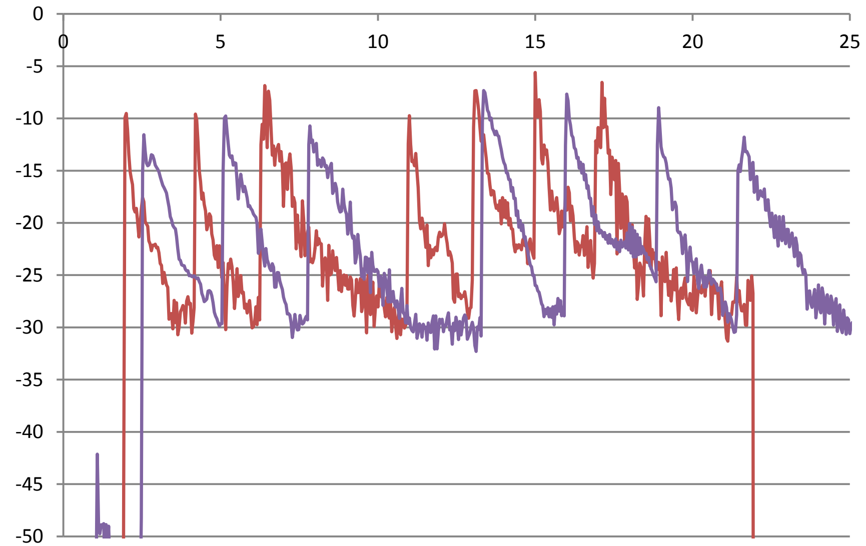
SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Low	-1.54	-0.61	-6.03	**-2.73**
	Mid	-6.68	-7.76	-7.72	**-7.39**
	High	0.46	0.63	0.63	**0.57**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Low	-2.84	4.10	5.27	**2.18**
	Mid	3.99	1.16	3.69	**2.94**
	High	-5.89	-5.01	-5.01	**-5.30**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Low	-0.64	-2.36	-5.84	**-2.95**
	Mid	-1.14	-0.69	-2.82	**-1.55**
	High	-4.66	-3.30	-3.30	**-3.75**

Window 2

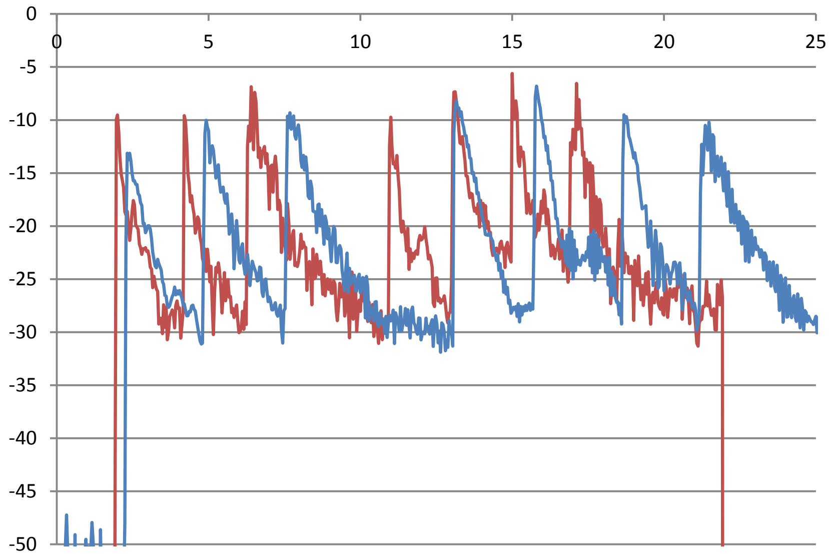
SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Low	-4.15	-6.97	-4.36	**-5.16**
	Mid	-3.80	-6.36	-7.90	**-6.02**
	High	-5.86	-5.00	-9.94	**-6.93**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Low	-6.00	-11.55	-17.57	**-11.71**
	Mid	-5.49	-6.27	-7.34	**-6.37**
	High	-5.18	-1.13	7.12	**0.27**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Low	3.59	5.00	3.33	**3.98**
	Mid	-2.13	0.98	1.58	**0.15**
	High	-1.34	-4.14	-3.02	**-2.84**

Sample 1 – Original vs. Target

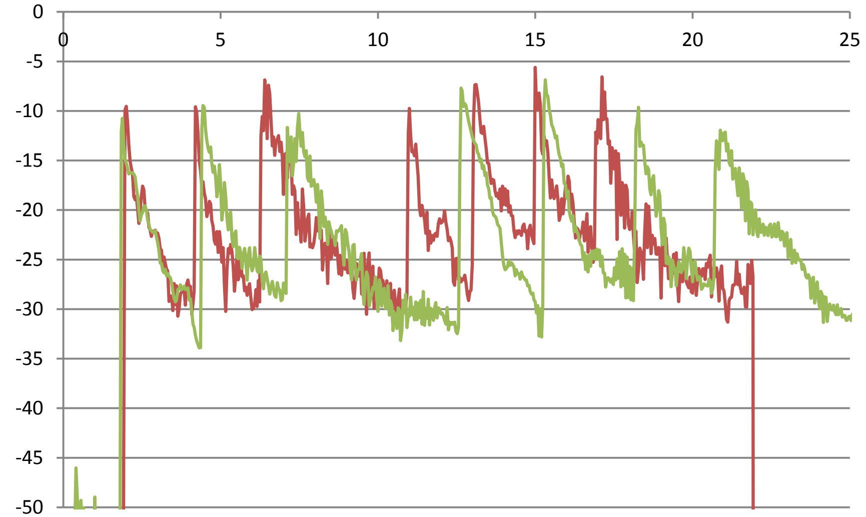
Original 1 – Config 1 – Time(s) vs RMS(dB)



Original 1 – Config 2 – Time(s) vs RMS(dB)



Original 1 – Config 3 – Time(s) vs RMS(dB)



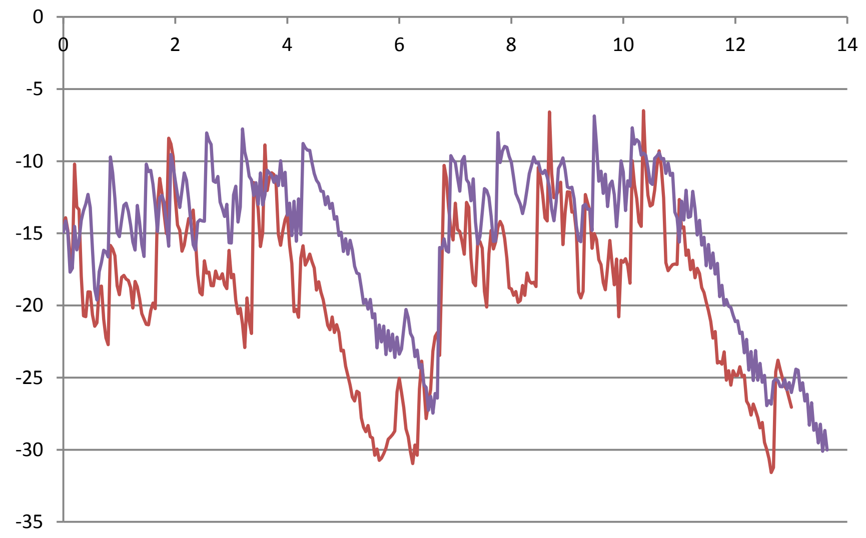
🟥 Target
🟪 Original – C1
🟦 Original – C2
🟩 Original – C3

Variance Results

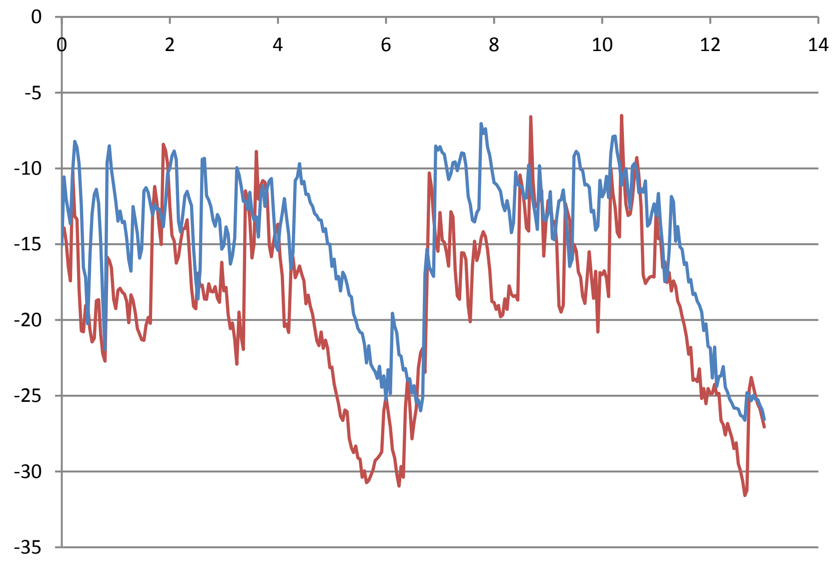
	Std. Deviation	Av. Abs. Deviation	Std. Dev. / Target %	Av. Abs. Dev. / Target %
Target	5.69	4.59	-	-
Config 1	10.49	7.63	84.22	66.02
Config 2	10.24	7.22	79.88	57.07
Config 3	9.86	6.71	73.25	46.15

Sample 2 – Original vs. Target

Original 2 – Config 1 – Time(s) vs RMS(dB)



Original 2 – Config 2 – Time(s) vs RMS(dB)



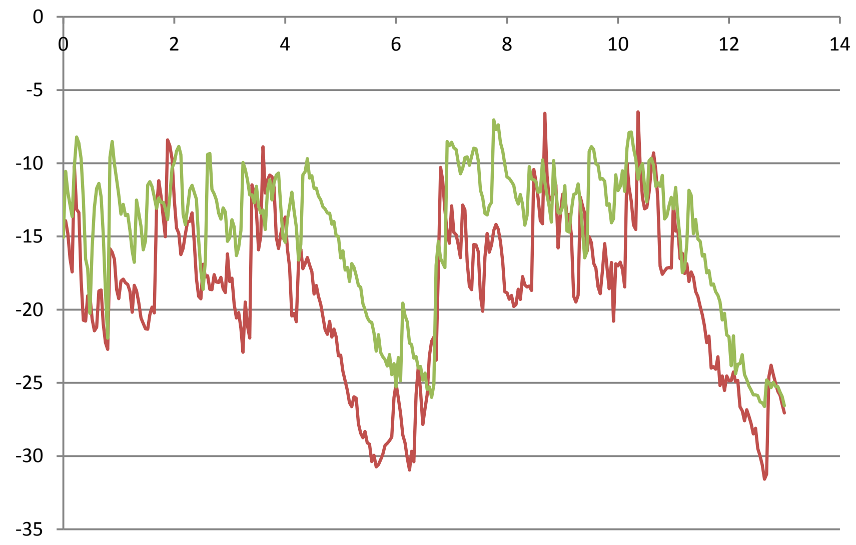
Target

Original – C1

Original – C2

Original – C3

Original 2 – Config 3 – Time(s) vs RMS(dB)

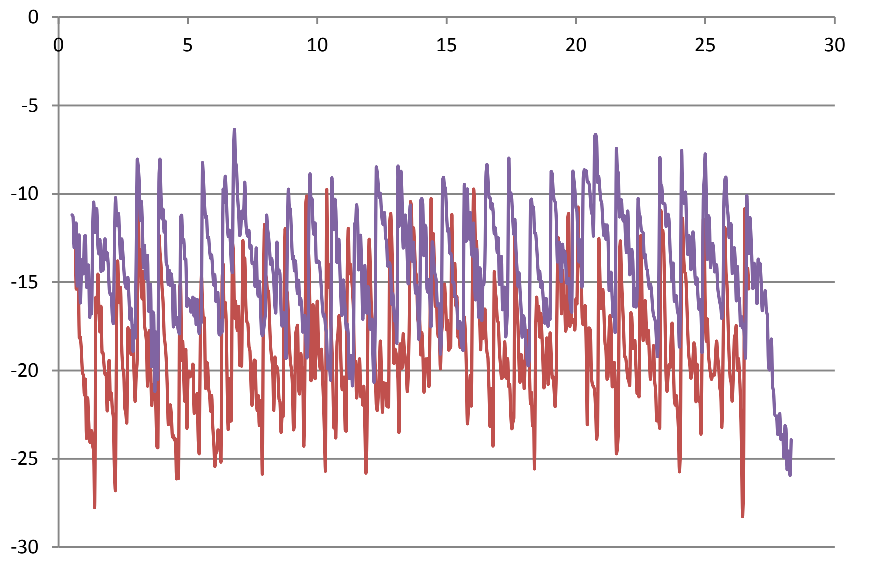


Variance Results

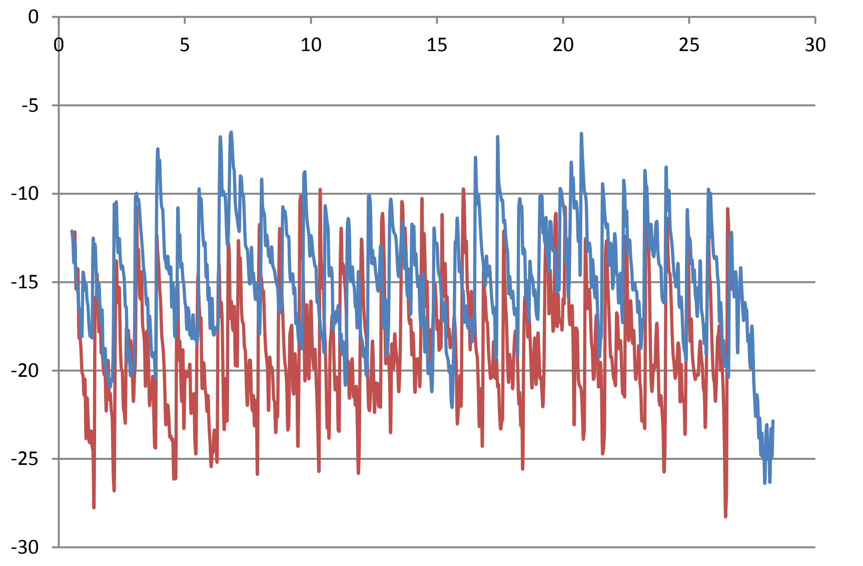
	Std. Deviation	Av. Abs. Deviation	Std. Dev. / Target %	Av. Abs. Dev. / Target %
Target	5.52	4.41	-	-
Config 1	5.71	4.81	3.53	9.05
Config 2	5.45	4.38	-1.23	-0.72
Config 3	6.75	5.79	22.42	31.22

Sample 3 – Original vs. Target

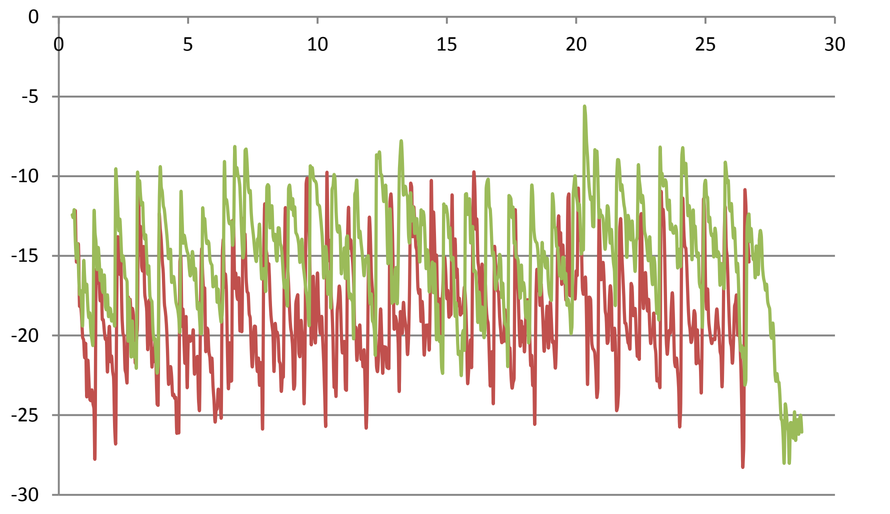
Original 3 – Config 1 – Time(s) vs RMS(dB)



Original 3 – Config 2 – Time(s) vs RMS(dB)



Original 3 – Config 3 – Time(s) vs RMS(dB)



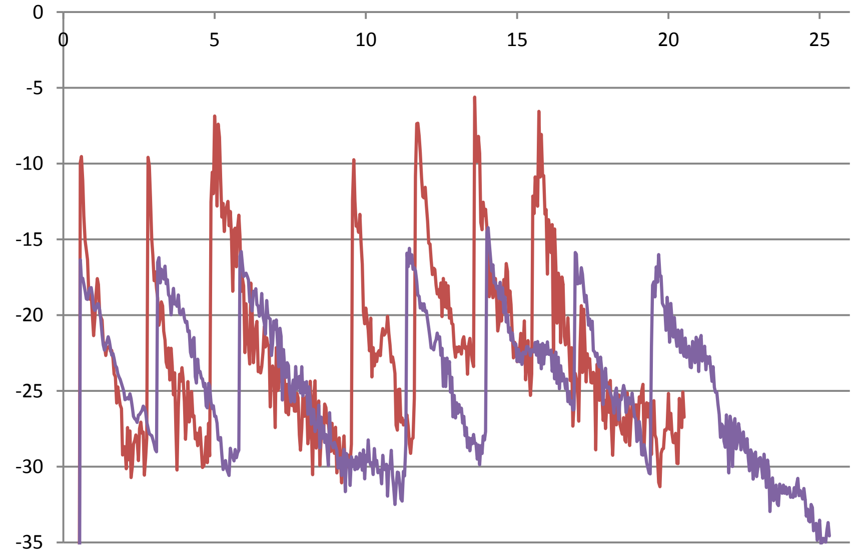
🟥 Target
🟪 Original – C1
🟦 Original – C2
🟩 Original – C3

Variance Results

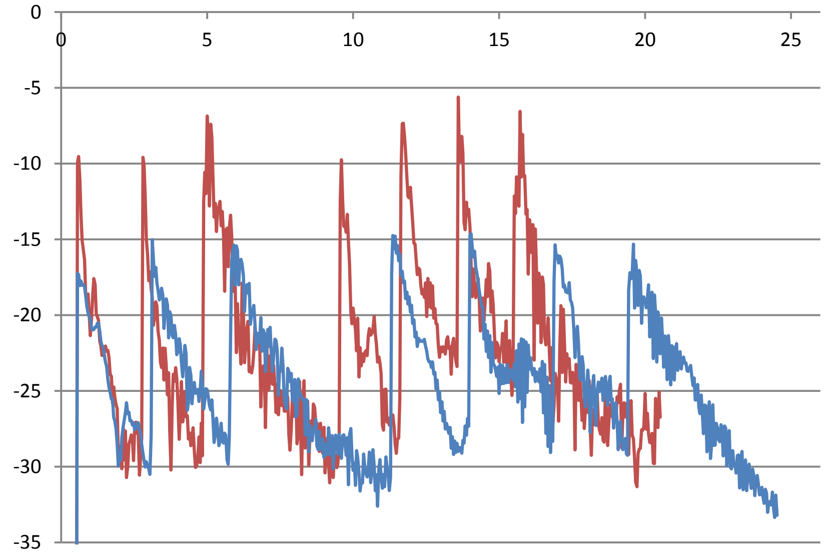
	Std. Deviation	Av. Abs. Deviation	Std. Dev. / Target %	Av. Abs. Dev. / Target %
Target	3.40	2.73	-	-
Config 1	3.27	2.61	-3.77	-4.45
Config 2	3.36	2.67	-1.32	-2.19
Config 3	3.72	2.90	9.51	6.33

Sample 1 – Processed vs. Target

Processed 1 – Config 1 – Time(s) vs RMS(dB)



Processed 1 – Config 2 – Time(s) vs RMS(dB)



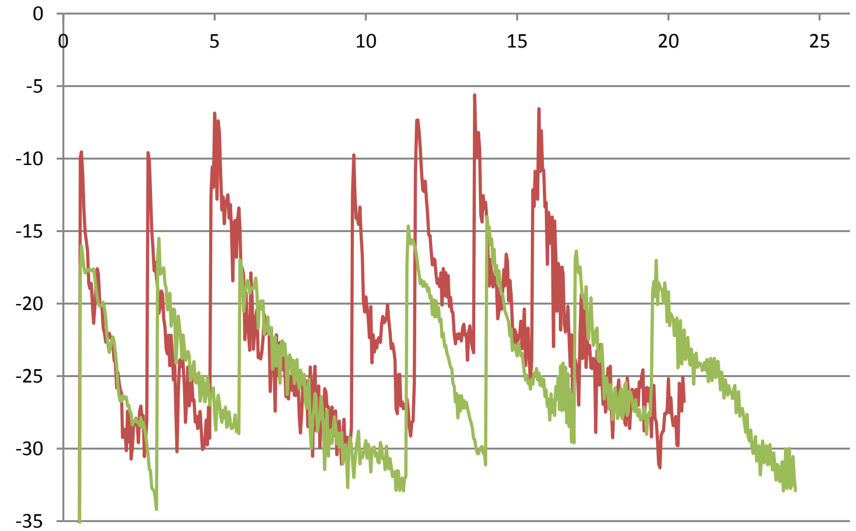
Target

Processed – C1

Processed – C2

Processed – C3

Processed 1 – Config 3 – Time(s) vs RMS(dB)

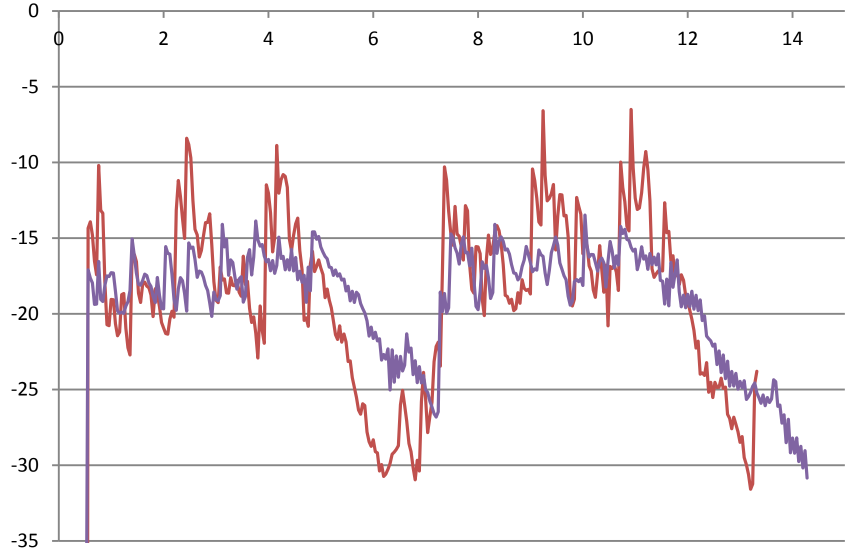


Variance Results

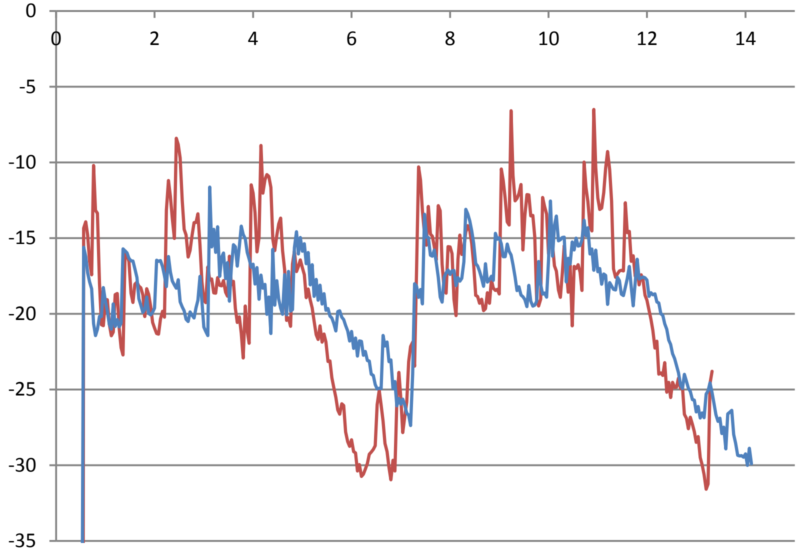
	Std. Deviation	Av. Abs. Deviation	Std. Dev. / Target %	Av. Abs. Dev. / Target %
Target	5.69	4.59	-	-
Config 1	4.71	4.00	-17.34	-12.98
Config 2	4.58	3.82	-19.48	-16.77
Config 3	4.80	3.93	-15.67	-14.44

Sample 2 – Processed vs. Target

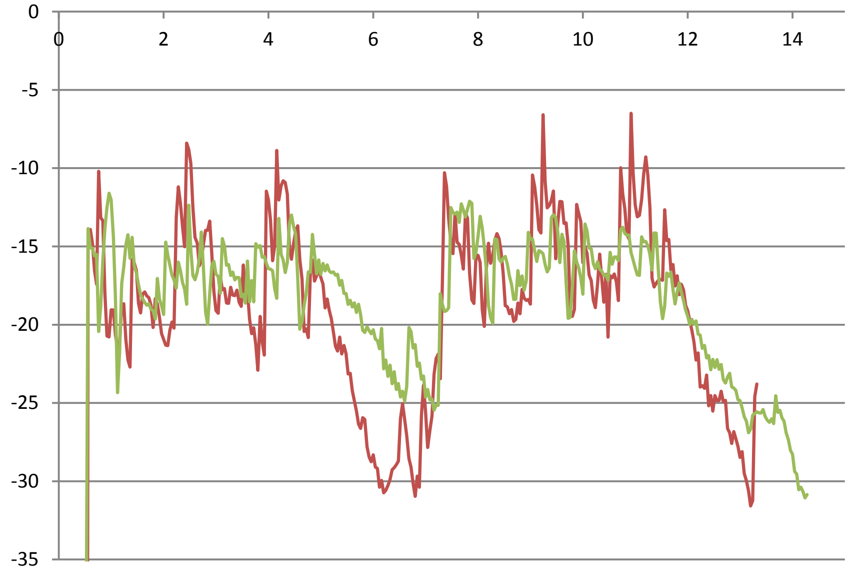
Processed 2 – Config 1 – Time(s) vs RMS(dB)



Processed 2 – Config 2 – Time(s) vs RMS(dB)



Processed 2 – Config 3 – Time(s) vs RMS(dB)



— Target

— Processed – C1

— Processed – C2

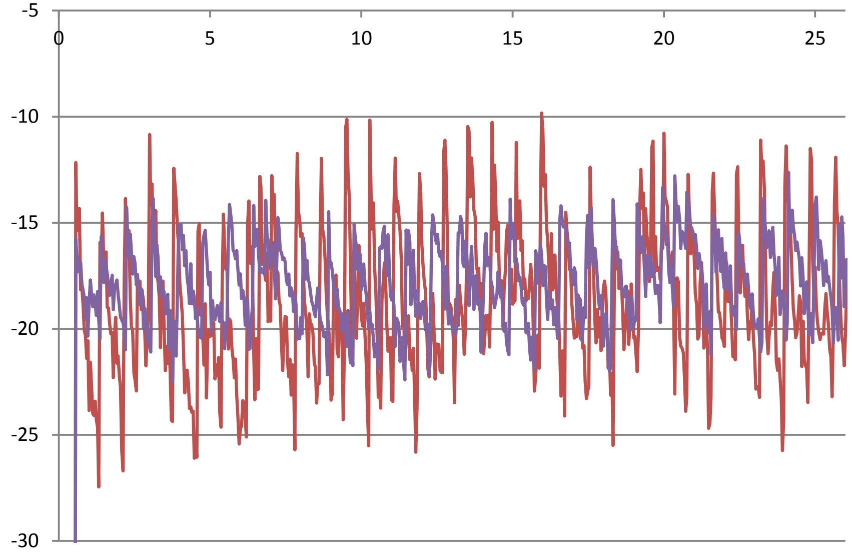
— Processed – C3

Variance Results

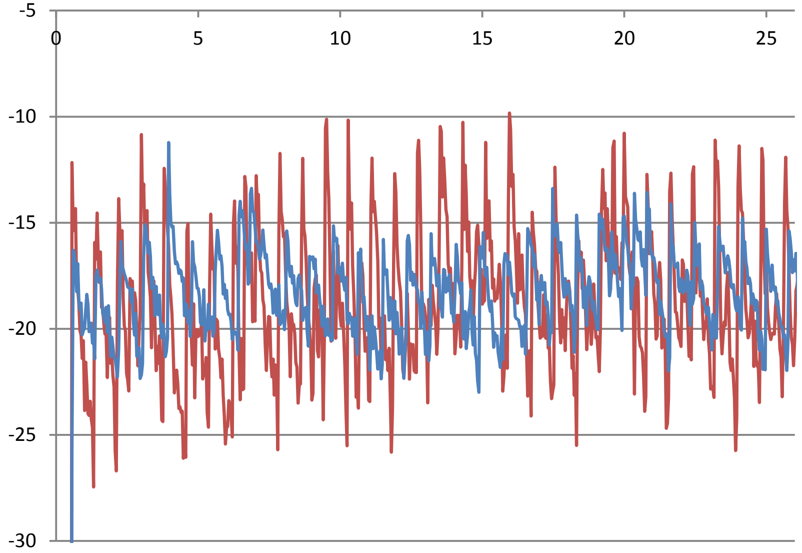
	Std. Deviation	Av. Abs. Deviation	Std. Dev. / Target %	Av. Abs. Dev. / Target %
Target	5.52	4.41	-	-
Config 1	3.55	2.82	-35.65	-36.03
Config 2	3.76	2.90	-31.90	-34.27
Config 3	4.00	3.16	-27.47	-28.28

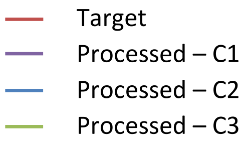
Sample 3 – Processed vs. Target

Processed 3 – Config 1 – Time(s) vs RMS(dB)

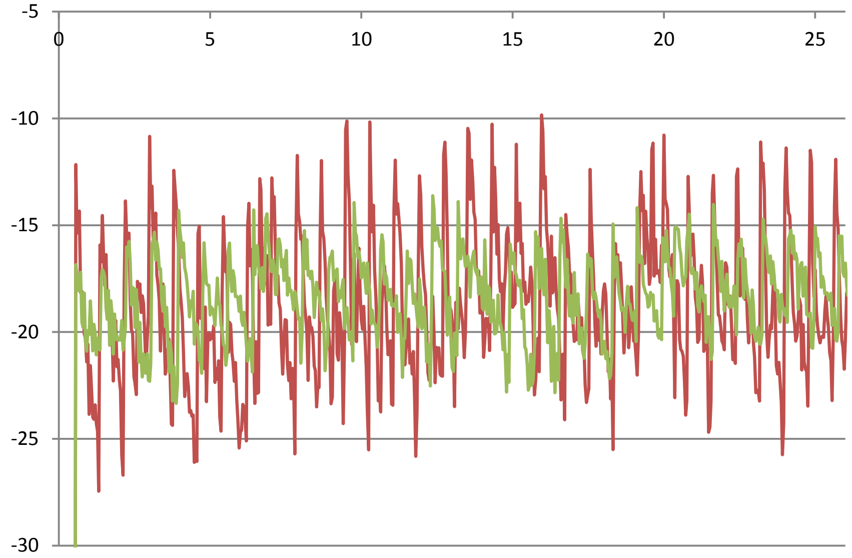


Processed 3 – Config 2 – Time(s) vs RMS(dB)





Processed 3 – Config 3 – Time(s) vs RMS(dB)



Variance Results

	Std. Deviation	Av. Abs. Deviation	Std. Dev. / Target %	Av. Abs. Dev. / Target %
Target	3.40	2.73	-	-
Config 1	2.11	1.67	-38.05	-38.81
Config 2	2.23	1.74	-34.58	-36.32
Config 3	2.24	1.73	-34.09	-36.67

Basic Results					
SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Std Dev	-101.55	-99.36	-88.92	**-96.61**
	Abs Dev	-79.00	-73.84	-60.59	**-71.14**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Std Dev	-39.18	-30.67	-49.89	**-39.91**
	Abs Dev	-45.08	-33.55	-60.59	**-46.41**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Std Dev	-34.27	-33.26	-43.60	**-37.05**
	Abs Dev	-34.37	-34.14	-42.99	**-37.17**

Delta Results					
SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Std Dev	-66.88	-60.40	-57.58	**-61.62**
	Abs Dev	-53.04	-40.31	-31.71	**-41.69**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Std Dev	32.12	30.67	5.05	**22.61**
	Abs Dev	26.97	33.55	-2.95	**19.19**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Std Dev	34.27	33.26	24.59	**30.71**
	Abs Dev	34.37	34.14	30.34	**32.95**

Sample 1 – Original vs. Target

	Standard Deviation			**Absolute Average Deviation**		
	Lo	Mid	Hi	Lo	Mid	Hi
Target	5.16	6.54	7.30	3.94	5.22	5.73
Config 1	9.17	11.50	11.51	7.20	8.65	8.85
Config 2	9.42	12.12	12.20	7.18	8.56	8.84
Config 3	8.93	10.86	10.97	6.74	7.48	7.86
	Config vs. Target Difference (%)					
Config 1	77.93	75.97	57.66	82.51	65.70	54.40
Config 2	82.63	85.31	67.04	82.18	63.88	54.16
Config 3	73.18	66.06	50.24	70.86	43.21	37.09

Sample 2 – Original vs. Target

	Standard Deviation			**Absolute Average Deviation**		
	Lo	Mid	Hi	Lo	Mid	Hi
Target	6.15	5.20	6.23	4.98	4.25	5.04
Config 1	6.75	6.29	6.98	5.63	5.25	5.80
Config 2	6.56	5.97	6.79	5.28	4.82	5.43
Config 3	7.84	7.27	7.98	6.58	6.07	6.65
	Config vs. Target Difference (%)					
Config 1	9.69	20.99	12.09	12.96	23.71	14.97
Config 2	6.63	14.89	9.00	5.93	13.52	7.63
Config 3	27.46	39.83	28.04	32.05	42.95	31.88

Sample 3 – Original vs. Target

	Standard Deviation			**Absolute Average Deviation**		
	Lo	Mid	Hi	Lo	Mid	Hi
Target	4.43	3.52	4.81	3.46	2.78	3.74
Config 1	4.46	3.76	4.96	3.57	3.05	3.97
Config 2	4.90	4.22	5.47	3.97	3.45	4.43
Config 3	5.23	4.50	5.67	4.13	3.60	4.49
	Config vs. Target Difference (%)					
Config 1	0.66	6.87	3.19	3.11	9.38	6.36
Config 2	10.60	19.91	13.82	14.82	23.78	18.54
Config 3	18.21	27.84	18.04	19.44	29.21	20.17

Sample 1 – Processed vs. Target

	Standard Deviation			**Average Absolute Deviation**		
	Lo	Mid	Hi	Lo	Mid	Hi
Target	5.16	6.54	7.30	3.94	5.22	5.73
Config 1	4.93	5.01	2.06	4.02	4.06	1.54
Config 2	5.01	4.84	4.36	4.17	4.09	3.58
Config 3	5.13	5.28	3.83	4.12	4.32	2.92
	Config vs. Target Difference (%)					
Config 1	-4.30	-23.37	-71.80	2.05	-22.33	-73.10
Config 2	-2.80	-25.92	-40.24	5.63	-21.68	-37.56
Config 3	-0.49	-19.31	-47.58	4.39	-17.17	-49.03

Sample 2 – Processed vs. Target

	Standard Deviation			**Average Absolute Deviation**		
	Lo	Mid	Hi	Lo	Mid	Hi
Target	6.15	5.20	6.23	4.98	4.25	5.04
Config 1	3.61	4.41	4.74	2.78	3.43	3.73
Config 2	3.83	4.29	5.69	3.13	3.29	4.56
Config 3	4.76	4.13	5.91	3.61	3.39	4.62
	Config vs. Target Difference (%)					
Config 1	-41.30	-15.25	-23.94	-44.29	-19.33	-26.08
Config 2	-37.83	-17.45	-8.59	-37.13	-22.49	-9.52
Config 3	-22.61	-20.60	-5.15	-27.51	-20.14	-8.33

Sample 3 – Processed vs. Target

	Standard Deviation			**Average Absolute Deviation**		
	Lo	Mid	Hi	Lo	Mid	Hi
Target	4.43	3.52	4.81	3.46	2.78	3.74
Config 1	4.43	2.53	3.70	3.67	1.75	2.94
Config 2	4.44	2.25	4.18	3.68	1.75	3.26
Config 3	4.88	2.84	4.09	3.95	1.92	3.19
	Config vs. Target Difference (%)					
Config 1	-0.03	-28.21	-23.01	6.03	-37.16	-21.22
Config 2	0.25	-36.20	-13.10	6.47	-37.09	-12.88
Config 3	10.28	-19.29	-14.83	14.12	-31.00	-14.61

Basic Results

Standard Deviation

SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Low	-82.22	-85.43	-73.67	**-80.44**
	Mid	-99.34	-111.23	-85.38	**-98.65**
	High	-129.47	-107.28	-97.82	**-111.52**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Low	-51.00	-44.45	-50.07	**-48.51**
	Mid	-36.24	-32.34	-60.43	**-43.00**
	High	-36.03	-17.59	-33.20	**-28.94**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Low	-0.69	-10.35	-7.93	**-6.32**
	Mid	-35.08	-56.11	-47.13	**-46.11**
	High	-26.19	-26.93	-32.87	**-28.66**

Average Absolute Deviation

SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Low	-80.46	-76.55	-66.47	**-74.49**
	Mid	-88.02	-85.56	-60.38	**-77.99**
	High	-127.50	-91.72	-86.12	**-101.78**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Low	-57.25	-43.06	-59.55	**-53.29**
	Mid	-43.04	-36.01	-63.09	**-47.38**
	High	-41.05	-17.15	-40.21	**-32.80**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Low	2.93	-8.35	-5.32	**-3.58**
	Mid	-46.54	-60.87	-60.21	**-55.87**
	High	-27.58	-31.42	-34.79	**-31.26**

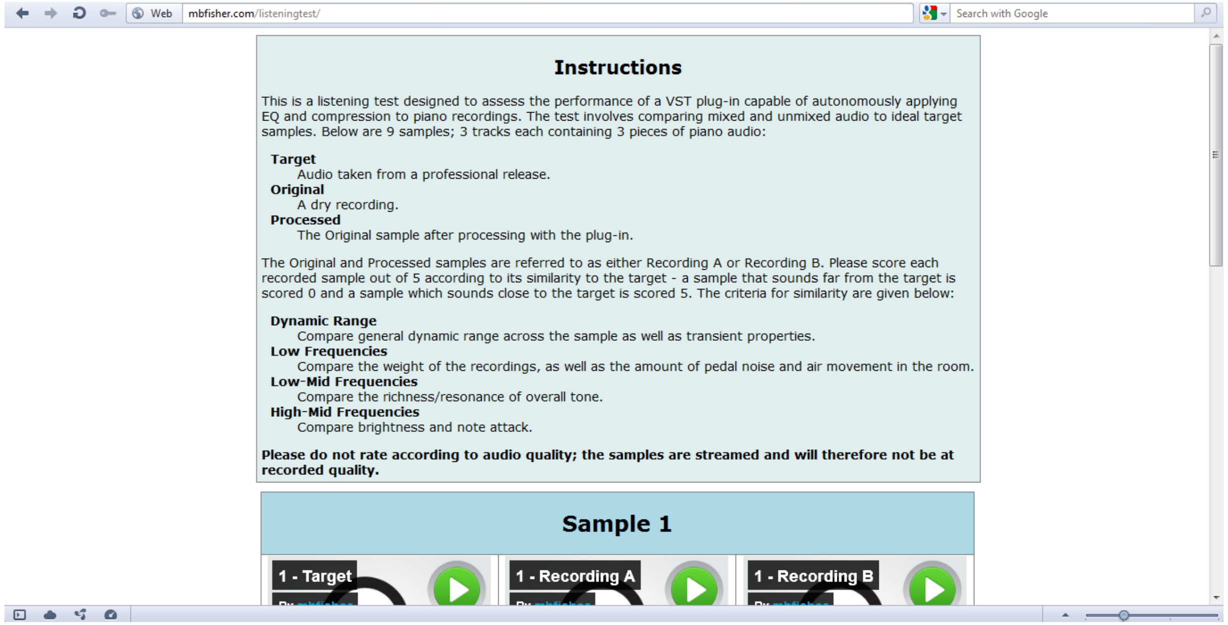
Delta Results

Standard Deviation

SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Low	-73.63	-79.84	-72.68	**-75.38**
	Mid	-52.60	-59.39	-46.75	**-52.92**
	High	14.14	-26.80	-2.66	**-5.11**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Low	31.61	31.20	-4.85	**19.32**
	Mid	-5.74	2.56	-19.23	**-7.47**
	High	11.84	-0.42	-22.89	**-3.82**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Low	-0.63	-10.35	-7.93	**-6.30**
	Mid	21.34	16.29	-8.55	**9.69**
	High	19.82	-0.72	-3.21	**5.29**

Average Absolute Deviation

SAMPLE 1					
		Config 1	Config 2	Config 3	**Mean**
	Low	-80.46	-76.55	-66.47	**-74.49**
	Mid	-43.37	-42.19	-26.04	**-37.20**
	High	18.70	-16.60	11.94	**4.68**
SAMPLE 2					
		Config 1	Config 2	Config 3	**Mean**
	Low	31.33	31.20	-4.54	**19.33**
	Mid	-4.38	8.96	-22.81	**-6.08**
	High	11.12	1.90	-23.54	**-3.51**
SAMPLE 3					
		Config 1	Config 2	Config 3	**Mean**
	Low	2.93	-8.35	-5.32	**-3.58**
	Mid	27.78	13.31	1.79	**14.30**
	High	14.86	-5.66	-5.56	**1.21**



Instructions

This is a listening test designed to assess the performance of a VST plug-in capable of autonomously applying EQ and compression to piano recordings. The test involves comparing mixed and unmixed audio to ideal target samples. Below are 9 samples; 3 tracks each containing 3 pieces of piano audio:

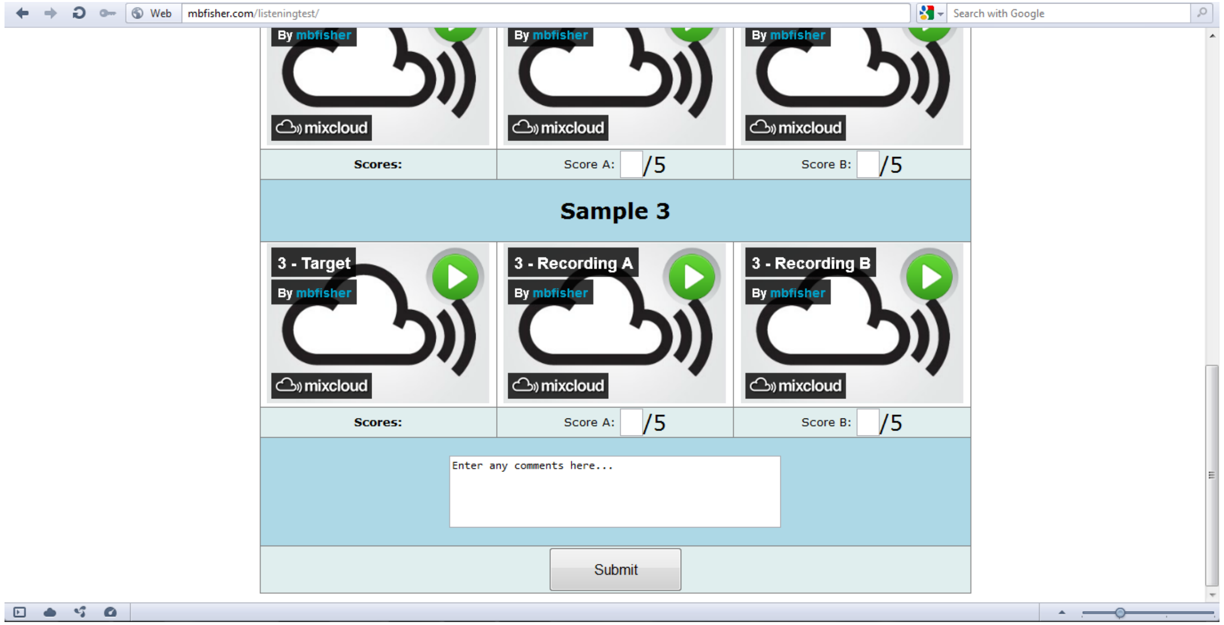
Target
Audio taken from a professional release.
Original
A dry recording.
Processed
The Original sample after processing with the plug-in.

The Original and Processed samples are referred to as either Recording A or Recording B. Please score each recorded sample out of 5 according to its similarity to the target - a sample that sounds far from the target is scored 0 and a sample which sounds close to the target is scored 5. The criteria for similarity are given below:

Dynamic Range
Compare general dynamic range across the sample as well as transient properties.
Low Frequencies
Compare the weight of the recordings, as well as the amount of pedal noise and air movement in the room.
Low-Mid Frequencies
Compare the richness/resonance of overall tone.
High-Mid Frequencies
Compare brightness and note attack.

Please do not rate according to audio quality; the samples are streamed and will therefore not be at recorded quality.

Sample 1



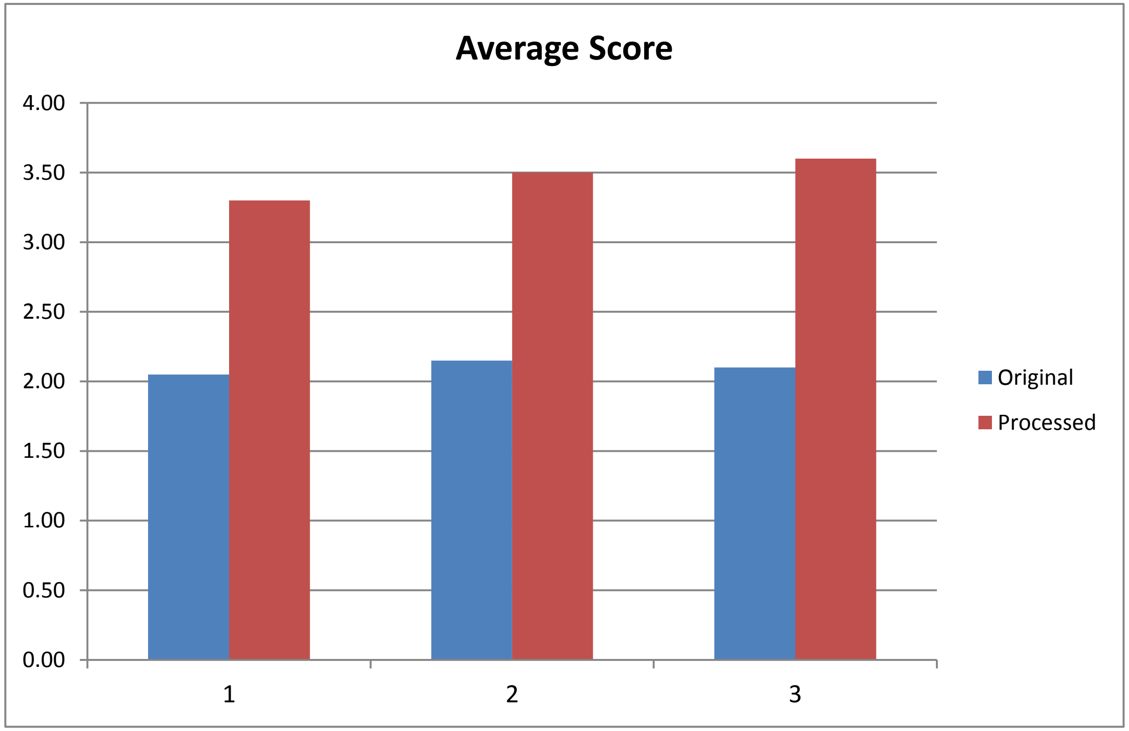
Scores: Score A: /5 Score B: /5

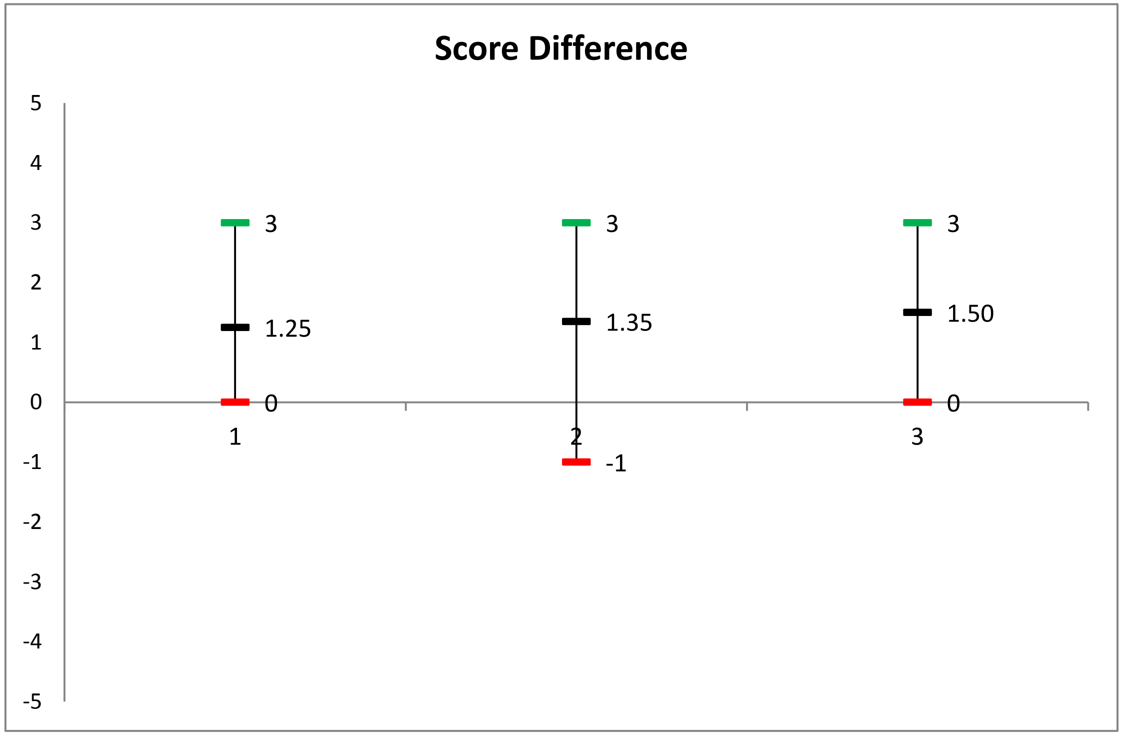
Sample 3

Scores: Score A: /5 Score B: /5

Enter any comments here...

INDEX	O1	P1	O2	P2	O3	P3		P1-O1	P2-O2	P3-O3
0	2	3	1	3	2	4		1	2	2
1	3	3	3	4	3	4		0	1	1
2	2	3	3	4	2	5		1	1	3
3	1	2	1	2	2	3		1	1	1
4	2	3	4	5	4	5		1	1	1
5	3	4	3	4	2	3		1	1	1
6	2	4	0	3	1	3		2	3	2
7	1	4	2	4	1	2		3	2	1
8	2	2	2	4	2	3		0	2	1
9	2	3	3	4	2	4		1	1	2
10	3	4	2	4	2	4		1	2	2
11	3	4	3	4	2	3		1	1	1
12	2	4	1	3	2	5		2	2	3
13	3	5	2	3	3	4		2	1	1
14	2	4	2	3	2	3		2	1	1
15	3	4	3	5	4	5		1	2	1
16	1	2	4	3	2	3		1	-1	1
17	2	3	3	4	2	4		1	1	2
18	1	2	0	1	1	1		1	1	0
19	1	3	1	3	1	4		2	2	3
SUM	41	66	43	70	42	72		25	27	30
MIN	1	2	0	1	1	1		3	3	3
MAX	3	5	4	5	4	5		0	-1	0
AVERAGE	**2.05**	**3.30**	**2.15**	**3.50**	**2.10**	**3.60**		**1.25**	**1.35**	**1.50**


Average Score
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
1
2
3
🟦 Original
🟥 Processed


Score Difference
5
4
3
2
1
0
-1
-2
-3
-4
-5
3
1.25
0
3
1.35
-1
3
1.50
0
1
2
3

Sample	Time Average (µs)				
	Initialisation	**Pre-FFT**	**Effects**	**Post-FFT**	**Total**
Original - 1 - Config 1	5.0	1420.3	893.5	1356.7	3675.4
Original - 1 - Config 2	4.9	1312.4	897.5	1359.0	3573.8
Original - 1 - Config 3	4.9	1313.6	891.1	1290.3	3499.9
Original - 2 - Config 1	5.0	1319.4	903.1	1279.7	3507.3
Original - 2 - Config 2	5.1	1310.2	893.1	1283.1	3491.6
Original - 2 - Config 3	5.0	1327.3	894.5	1347.9	3574.7
Original - 3 - Config 1	5.0	1377.3	970.7	1357.0	3710.0
Original - 3 - Config 2	5.1	1321.9	902.3	1426.7	3656.0
Original - 3 - Config 3	5.1	1318.1	901.2	1289.2	3513.7
Statistic	**Statistic Result (µs)**				
	Initialisation	**Pre-FFT**	**Effects**	**Post-FFT**	**Total**
MINIMUM	4.9	1310.2	891.1	1279.7	3491.6
MAXIMUM	5.1	1420.3	970.7	1426.7	3710.0
RANGE	0.2	110.0	79.6	147.0	218.4
MEAN	5.0	1335.6	905.2	1332.2	3578.0
MEDIAN	5.0	1319.4	897.5	1347.9	3573.8
STANDARD DEVIATION	0.1	35.6	23.5	47.0	78.7

III.3

Analysis

III.3.1 Test Results

The test results can be used to draw three conclusions. The first establishes the operational functionality of the plug-in, informed by the numeric basic results, describing the plug-in's ability to attenuate low and mid-range frequencies, boost high frequencies and reduce dynamic range. The second and more important conclusion, informed by the numeric delta results and listening test results, describes the plug-in's ability to transform audio in a way that imitates a mixing engineer. The third conclusion, informed by the performance test results, describes the computational efficiency of the plug-in. As a general measure of success of the audio tests a percentage "pass rate" was calculated for each test using the basic and delta results; each individual result was compared to the test hypothesis and coloured green or red on the tables above based on its success, and the pass rate expressed as the ratio of green to red results. For the listening test a successful result was a positive difference between an original and processed score. As the testing process has been largely statistical, a measure of significance was employed to be able to declare a test successful with a value of 90%.

Conclusion 1 – Operational Functionality

To establish whether or not the basic functions of the plug-in work as intended the numeric basic results are used. The results are consistently successful with a lowest pass rate of 92%:

Test	Basic Result
Sample Average	100%
Sample Transient	92%
RMS Variance	100%
FFT Band Variance	99%

These tests produce reliable results as to whether the plugin is transforming audio in the way it should as it uses the target only as a reference point to which the original and processed samples are compared; the only factor influencing a result is the processing applied by the plugin.

The results demonstrate that the plug-in attenuates and boosts the relevant frequency bands and reduces dynamic range as it should.

Conclusion 2 – Target Similarity

To establish whether or not the plug-in imitates a professional mixing engineer the numeric delta results and listening test results are used:

Test	Delta Result
Sample Average	58%
Sample Transient	72%
RMS Variance	38%
FFT Band Variance	63%

Sample	Listening Result
Sample 1	90%
Sample 2	95%
Sample 3	95%

The delta results are less successful than the basic results with none achieving the statistical significance of 90% and the RMS Variance test achieving only a 38% pass rate. The combination of good basic results and bad delta results leads to the conclusion that the plug-in is generally over-processing audio; a successful basic result coupled with a failed delta result means that whilst the direction of change is correct the processed sample is further away from the target than the original. For example:

Sample Average Test - Sample 1, Config 1, Low Frequency:

	Amplitude (dB)	Vs. Target (dB)
Target	150.11	
Original	149.63	-0.48
Processed	146.69	-3.42

Whilst 2.94dB of attenuation has been achieved by the plug-in it has moved the processed value further away from the target than the original value.

RMS Variance Test – Sample 2, Config 3, Standard Deviation:

	Standard Deviation (dB)	Vs. Target (%)
Target	5.52	-
Original	6.75	22.42
Processed	4.00	-24.47

Though the dynamic range of the processed sample has been reduced 49.89% compared to the original, it is 5.05% further away from the target.

As described in III.1 the numerical testing process had one main limitation; it was not possible to obtain unmixed versions of the target samples meaning that the difference between target, original and processed samples is not solely produced by the plugin. A successful set of listening test results combined with successful numeric basic results from the tests analysing dynamic characteristics support the theory that poor numeric delta results can be explained by the limitations of the numeric testing process.

Using a positive difference between an original and processed score as a successful result the listening test results show that all three samples achieved a pass rate within the 90% range of statistical significance. The results contain only one point out of 60 showing a negative difference and three out of 60 showing zero difference. The test was carried out by 20 individuals with developed critical listening skills and the identities of the original and processed samples were kept hidden; the evaluation of each individual is trustworthy and unbiased. These results demonstrate that when asked to compare the tonal and dynamic qualities of a set of audio samples a group of trained Music Technologists determined that the plugin successfully transforms piano recordings to be more similar to commercial standard productions.

The results demonstrate that the plug-in produces audio transformed in way that imitates a professional mixing engineer.

Conclusion 3 – Computational Efficiency

With a pass rate of 100% and a maximum total time of 3710.0µs the performance timing test produced extremely successful results compared to a target of 10ms. The plug-in achieves this level of performance with consistency; the total times have a standard deviation of 78.7µs; 2.1% of the mean value.

The performance test results demonstrate that the plug-in operates with significant computational efficiency.

III.3.2 Comparative Analysis

This project aims to produce novel research outcomes in the field of automatic mixing, and so a comparison between the developed plugin and similar products can be used to determine what original concepts are being put forward. The developed plugin is a prototype and therefore features like effect quality, GUI design and overall efficiency are lacking compared to professional commercial plugins, so this comparison takes place at a conceptual level by addressing each of the plugin's headline features.

Autonomous post production

During background research no plugins were found that offer autonomous post production; plugins are available that address the issue of changing parameters over time without the use of time consuming automation, including some that do this in response to signal analysis, but none that offer this functionality with no user input.

Instrument specific design

Plugins such as Waves Bass Rider [Waves Audio Ltd.] offer instrument specific effects; the Bass Rider automatically "rides" the volume of a bass recording in a way that mimics a mixing engineer.

Regression of FFT

All of the plugins found during background research that process individual components of the frequency spectrum or respond to signal analysis use filtering and time dime domain analyses such as RMS measurement to determine the properties of a signal. No plugins were found that use frequency domain analysis to inform effects parameters, and to the best knowledge of the author the developed plugin is unique in its use of polynomial regression to apply temporal smoothing.

Envelope following control of EQ parameters

One plugin was found, the bx_dynEQ, that uses compression techniques to control EQ parameters.

The closest relative of the developed plugin is the dynamic equalizer, and it is comparable to a multiband compressor. Dynamic equalizers use variations in EQ parameters to control the tone of a recording, however no plugins were found that use this technique to also control dynamics. Neither were any plugins found that use FFT analysis to determine the parameter changes; the dynamic equalizers described in II.2 Post Production Systems filter the signal into bands and use RMS measurement to determine amplitude. The multiband compressor filters a signal into bands and compresses each individually with a view to controlling the dynamics of a particular part of the frequency spectrum, and a result of this process can be tonal changes in the recording. Multiband compressors have a large number of parameters to manually manipulate in order to achieve the desired outcome, and though pre-sets can be used to broadly set up the plugin for use with a particular instrument, this is a still a time consuming process.

SECTION IV

CONCLUSIONS

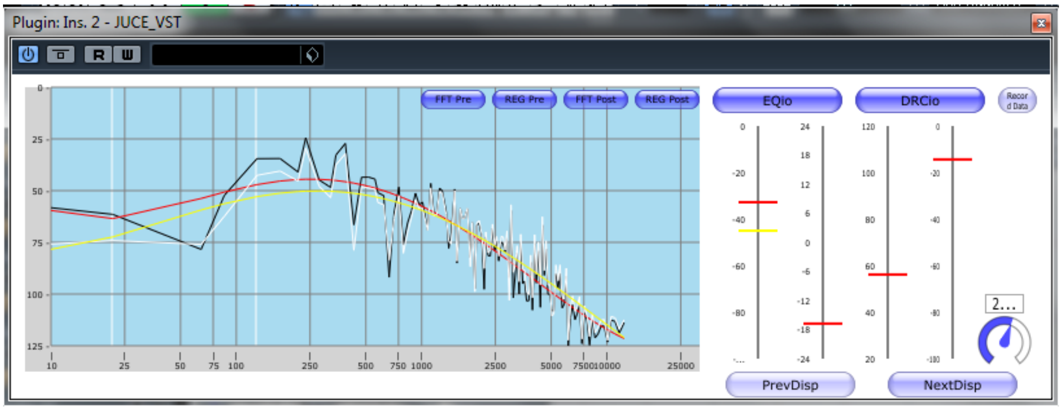


Figure IV.1.1

At the beginning of this programme of research and development a set of goals were identified:

- Develop a plugin capable of autonomously mixing piano recordings;
- Develop a quick and easy to use plugin whose functions are not replicable without significant time and effort;
- Provide an original contribution to the area of automatic mixing research;

The ability of the plugin to autonomously mix piano recordings has been determined by the testing process. Whilst not all testing results were positive they served to prove that the plugin is capable of transforming the tonal and dynamic characteristics of a piano recording to better conform to the standards expected within a modern popular music production. As a prototype the plugin can be declared to be functionally capable.

Testimony regarding the quality of the plugin was gathered:

Colin Elliot, Recording and Mixing Engineer, Yellow Arch Studios, Sheffield:

“This will be a useful plugin once a more efficient GUI is in place. I would use it on a piano track that sat in the back of a mix and it would save a lot of time. For very prominent piano parts I would want the control of individual effects. Needs an AU version!”

Chris Humphries, Music IT Technician, Headington Girl’s School; Music Technology Postgraduate, University of Huddersfield:

“I would definitely use this. I like the FFT curves, makes it really easy to see what’s going on. Would be even more interested in a version that worked on drums.”

The testimonials show a positive response to the plugin. The most useful statement is Mr Elliot's; "I would use it on a piano track that sat in the back of a mix and it would save a lot of time. For very prominent piano parts I would want the control of individual effects," as it provides an indication of the usability of the plugin in a professional context.

The usability of the plugin is excellent as user interaction is minimal due to the autonomous nature of the software; six switches are supplied to enable or disable the filters and dynamic control for each of the three frequency bands, and a input gain knob can used if desired. An example method of partially recreating the plugins functionality would be to use an EQ plugin and automate the gain and frequency parameters of three filters to follow the overall level of a piano recording; this would not recreate the use of FFT analysis to control the parameters and would not be able to respond to level changes as quickly as the plugin. Alternatively a dynamic EQ or multiband compression plugin could be used which could be made to produce similar results, but these plugins usually contain a large number of parameters and need careful configuration. Compared to either of these methods the developed plugin is far simpler and easier to use; it only needs to be inserted into the signal chain and activated.

Comparison with the background research provided in section I.2 demonstrates that several aspects of the plugin represent an original contribution to the area of automatic mixing:

- Though plugins are available which control EQ parameters by responding to an incoming signal, none were found to use FFT analysis in order to do this;
- No plugins were found which vary cut-off frequency using envelope following or vary EQ parameters to control the dynamic range of a recording.
- No plugins were found which are designed for use with a specific instrument and are capable of autonomous processing.

The developed plugin serves as a prototype proof of concept and as such would need further development in order to be commercially viable. The features which require updating have been developed to the minimum standard necessary to enable research into the novel components of the plugin; the suggestions outlined below were not practically employable during the time period available. The most pressing update required is an efficient GUI; the current implementation served only as a feedback facility for use during the development process. Preliminary research suggests that the OpenGL library would be a useful tool for this; it is a widely used graphics library and makes very efficient use of CPU resources (OpenGL.org). The EQ and compression algorithms used are basic and would benefit greatly from a course of research to determine which of the many available methods of implementing each suit the application best. The plugin would also benefit from a custom regression algorithm which could exploit the specifics of the application and boost efficiency. A feature essential before the plugin could be released would be the availability of RTAS and AU versions; as described in Appendix section describing the JUCE framework, JUCE includes the ability to build projects into several plugin formats from one set of source code meaning that with the possession of the relevant SDKs these version could be easily produced.

There are several areas of further development which could be undertaken to improve the plugin in later versions. Concentrating on piano mixing the piano could be expanded to mix other genres of music besides pop; jazz and classical for example. This would require research into common mixing practices within these genres and would only be possible if similarities were observed as they were for popular music. With a coding framework in place to control the way the EQ and compression parameters respond to the incoming signal pre-sets could be designed which provide autonomous setups for mixing other instruments. The strongest candidates for this would be instruments such as kick drums, snare drums and bass guitars; the concept of uniformity within popular music production is most harshly imposed on these instruments and so they would benefit from new methods of controlling dynamics and tone. A step further would be to provide an interface to allow users to design custom setups themselves allowing the effects processing to be used creatively as well as correctively, although this would not be an autonomous solution and would behave very similarly to the dynamic EQ plugins discussed in section I.2. The EQ and compression algorithms could be updated to model analogue hardware; this is a current development trend and a popular feature among consumers of audio plugins.

SECTION V

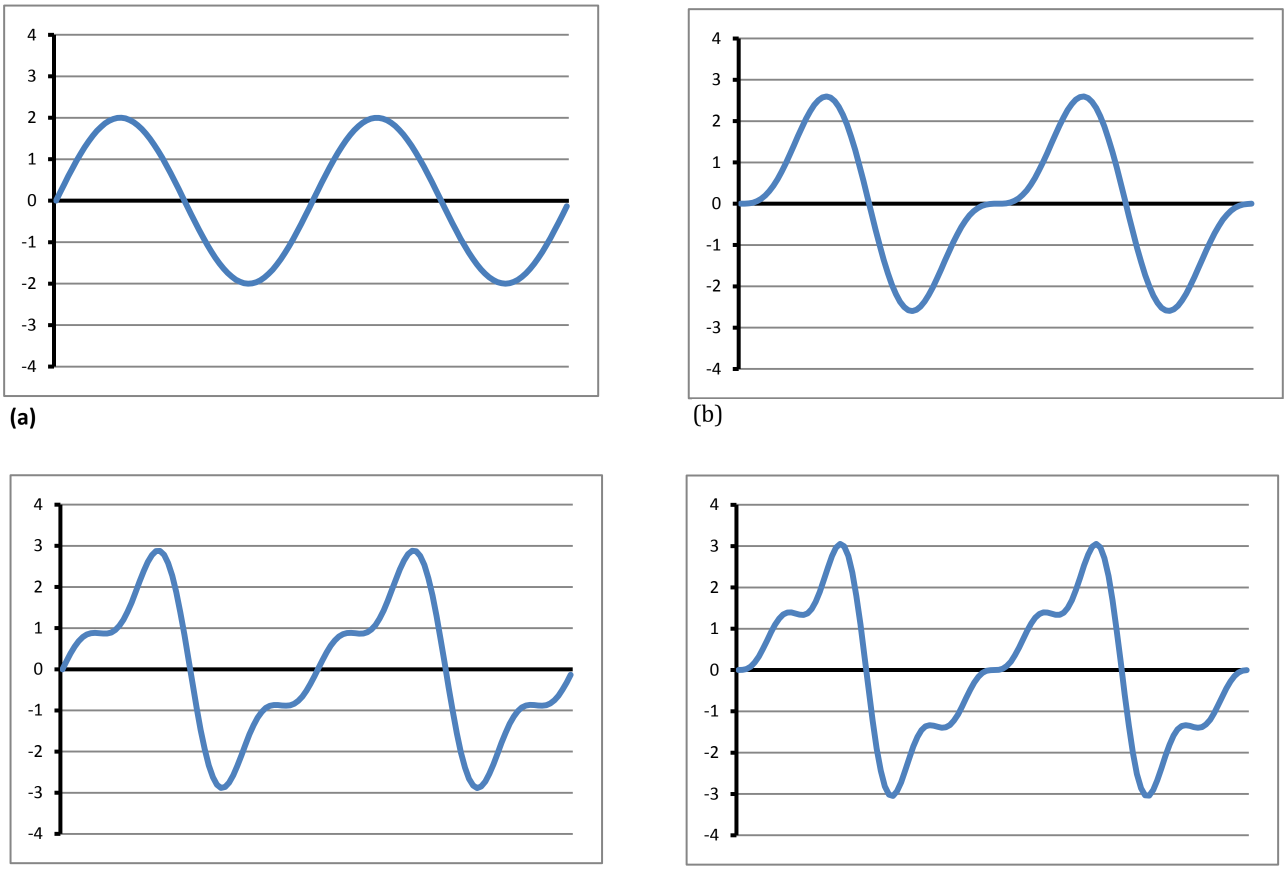
APPENDIX

V.1

Fourier Analysis

V.1.1 Fourier Analysis

Fourier Analysis is a branch of mathematics that is extremely useful in digital signal processing, first published by James Fourier in 1808 (Fourier, 1808). At its core is the idea than any periodic signal can be expressed as the sum of an infinite number of sine and cosine waves, known as the Fourier series, (Croft and Davidson, 2008, p.1139). A common example of this is the use of additive synthesis to create a saw-tooth wave:



(a)

(b)

(c)

(d)

Figure V.1.1

Figure V.1.1 shows graphs of four functions:

a) $\boldsymbol{f(t)} = \mathbf{sin}\,\boldsymbol{t}$

b) $\boldsymbol{f(t)} = \mathbf{sin}\,\boldsymbol{t} - \frac{1}{2}\sin 2t$

c) $\boldsymbol{f(t)} = \mathbf{sin}\,\boldsymbol{t} - \frac{1}{2}\sin 2t + \frac{1}{3}\sin 3t$

d) $\boldsymbol{f(t)} = \mathbf{sin}\,\boldsymbol{t} - \frac{1}{2}\sin 2t + \frac{1}{3}\sin 3t - \frac{1}{4}\sin 4t$

Figure V.1.1(a) shows the function $\mathbf{sin}\,\boldsymbol{t}$. Figure V.1.1(b) shows the function $\mathbf{sin}\,\boldsymbol{t} - \boldsymbol{\frac{1}{2}}\mathbf{sin}\,\boldsymbol{2t}$, the original sine wave with a harmonic added at twice the frequency with half the amplitude. Figures V.1.1(c) and V.1.1(d) show the addition of more harmonics. As more harmonics are added the waveform can be seen to be approaching that of a periodic saw-tooth wave, and if an infinite number of harmonics were added the saw-tooth wave would be represented exactly.

Therefore, the Fourier series of a saw-tooth wave can be expressed as:

$$f(t) = \sum_{n=1}^{\infty} \left(\frac{-2\cos n\pi}{n}\right) \sin nt$$

Using the quantities a_0, a_n and b_n, known as Fourier coefficients, a generalised equation for periodic function *f(t)* with period *T* can be derived to calculate any Fourier series:

$$a_0 = \frac{2}{T}\int_0^T f(t)dt$$

$$a_n = \frac{2}{T}\int_0^T f(t)\cos\frac{2n\pi t}{T}\,dt, \quad n = 1, 2, 3, \ldots$$

$$b_n = \frac{2}{T}\int_0^T f(t)\sin\frac{2n\pi t}{T}\,dt, \quad n = 1, 2, 3, \ldots$$

The Fourier series is given by:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left(a_n\cos\frac{2n\pi t}{T} + b_n\sin\frac{2n\pi t}{T}\right)$$

Evaluating the first few terms of the series makes it clear that a_n are the amplitudes of the sine waves in the series, and b_n are the amplitudes of the cosine waves:

$$f(t) = \frac{a_0}{2} + a_1\cos\frac{2\pi t}{T} + b_1\sin\frac{2\pi t}{T} + a_2\cos\frac{4\pi t}{T} + b_2\sin\frac{4\pi t}{T} + \cdots$$

The Fourier series is a useful tool for analysing theoretical periodic waveforms, but cannot be used to analyse aperiodic waveforms. All sound that is produced in the natural world is comprised of aperiodic waveforms and so a different tool, the Fourier transform, must be used to analyse them.

The Fourier transform expresses an aperiodic function not as the sum of other functions but as an integral (Croft and Davidson, 2008, p. 1149). The Fourier transform of a signal is a function of the continuous frequency variable ω:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}\,dt$$

Where

$$j = \sqrt{-1}$$

The modulus and argument of the complex result of the integral can be plotted as amplitude and phase spectra against ω (Croft and Davidson, 2008, p.1151), providing the magnitudes of the frequency components of the function.

The Fourier transform processes functions of continuous data and assumed infinite length. Digital audio systems, like all computer programs, are discrete systems, meaning that the functions they process must be discrete - comprised of discrete data with finite length. In a digital audio system the continuous function of a waveform in the air is recorded using a microphone, generating a voltage waveform in an electrical wire, which is then connected to a computer and converted into samples. A sample is a measurement of voltage, and hence amplitude, at a specific moment in time. A series of samples, taken with a set interval of time between them, forms a discrete function with finite length. If a small enough sampling interval is used, a discrete function can very accurately simulate a continuous function.

The Discrete Fourier Transform (DFT) allows the theoretical analysis of signals provided by the Fourier transform to be practically realised within a digital system (Smith, 1997, p.150). If x is an input signal and discrete function of N samples, the DFT X is expressed as:

$$X(k) = \sum_{n=0}^{N-1} x(n)W^{kn} \qquad k = 0, 1, \dots, N-1,$$

Where

$$W = e^{-j\frac{2\pi}{N}}$$

This form is used for ease of notation, however if W is evaluated using Euler's Identity:

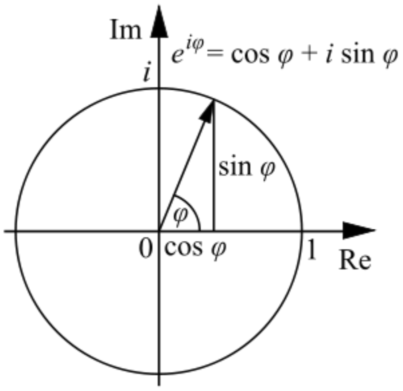


Figure V.1.2

$$e^{-jx} = \cos x - j\sin x$$

$$W = \cos\frac{2\pi}{N} - j\sin\frac{2\pi}{N}$$

It can be seen that, like the continuous Fourier transform, the modulus and argument of the equation's result for each value of k can be plotted to give the amplitude and phase spectra of the input signal. It is important to note that the number of amplitude and phase values produced is the same as the number of samples in the input signal.

The version of the DFT described above can be applied to both real and complex inputs, however as digital audio samples are real numbers, a complex input will be never be used for this application. For the purposes of demonstration within this paper, it is far simpler to utilise the "Real DFT"; an algorithmic version which can only be applied to input signals comprised of real data and hence involves only a few calculations with complex numbers. The Real DFT will be used to begin demonstrating the methods of implementing Fourier analysis as part of a computer program.

The Real DFT algorithm uses basis functions in order to establish a frequency domain representation of a signal (Jackson, 1996, p.189). The basis functions are sine and cosine waves of increasing frequencies to which the input signal is compared. In the frequency domain the DFT produces two signals - the real and imaginary results of the equation above. The real result gives the amplitudes of the cosine basis functions, while the imaginary result gives the amplitudes of the sine basis functions. These two sets of amplitudes are stored by the algorithm using two arrays; *REx* and *IMx*. Weighted with the amplitudes stored in *REx* and *IMx*, the basis functions form the component frequencies of the Fourier series of the signal.

$$REx(k) = \sum_{i=0}^{N-1} \cos 2\pi k \frac{i}{N} \qquad k = 0, 1, 2, \dots, N-1,$$

$$IMx(k) = -\sum_{i=0}^{N-1} \sin 2\pi k \frac{i}{N} \qquad k = 0, 1, 2, \dots, N-1,$$

As mentioned above, the number of results must be the same as the number of samples in the input signal, in this case represented by the variable *N*.

Each of the array values can then be used to calculate the modulus and argument, and hence amplitude and phase, of each of the basis function frequencies present in the input signal, which together form amplitude and phase spectra. For the purposes of the plugin it is only necessary to calculate the amplitude spectrum, which is stored by the algorithm in the array *MAGx*.

$$MAGx(k) = \sqrt{REx(k)^2 + IMx(k)^2}$$

The Real DFT algorithm was prototyped in Visual Basic .NET. The input signal, array *x*, is populated with a sine wave of frequency 10 Hz:

```
Dim N As Integer = 256 'Number of samples in input signal
Dim F As Integer = 10 'Frequency in Hz of input signal sine wave
Dim x(N) As Double 'Array containing input signal
Dim REx(N), IMx(N) As Double 'Arrays containing real and imaginary results
Dim MAGx(N) As Double 'Array containing amplitudes of the basis functions



' Input signal array populated with sine wave
For i = 0 To N - 1
    x(i) = Math.Sin((2 * Math.PI * F) * (i / N))
Next

' Real and imaginary arrays set to 0
For k = 0 To N - 1
    REx(k) = 0
    IMx(k) = 0
Next

' Input signal samples multiplied by basis function samples
For k = 0 To N - 1
    For i = 0 To N - 1
        REx(k) = REx(k) + (x(i) * Math.Cos((2 * Math.PI * k * i / N)))
        IMx(k) = IMx(k) - (x(i) * Math.Sin((2 * Math.PI * k * i / N)))
    Next
Next

' Modulus / amplitude calculated from real and imaginary arrays
For k = 0 To N - 1
    MAGx(k) = Math.Sqrt((Math.Pow(REx(k), 2) + Math.Pow(IMx(k), 2)))
Next
```

The input signal and MAGx array is then plotted using Microsoft Chart Controls .NET:

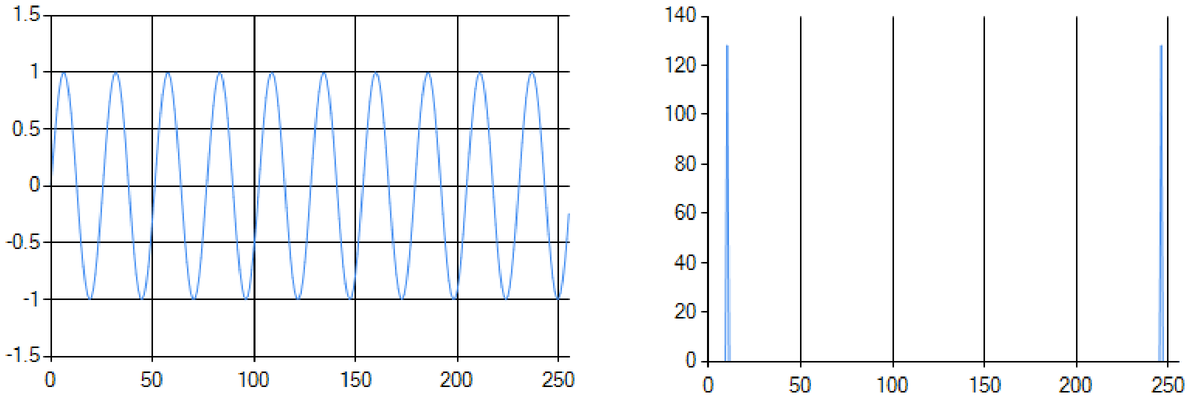


Figure V.1.3

A large amplitude peak can be seen in Figure II.3.3 at 10Hz. However, there is a second peak at 246Hz. This is because amplitude and phase spectra resulting from any Fourier transform are conjugate-symmetric sequences – they are symmetrical about the point N/2. This point is also the Nyquist Frequency – the highest frequency that can be accurately sampled by the system, defined as half the sample rate (Smith, 1997, Chap. 3). This means that whilst the correlation of basis functions needs to be calculated from 0 to N, the arrays *REx*, *IMx* and *MAGx* need only be declared with size N/2. This does not change the fact that the number of results is the same as the number of samples in the input signal, only that half of the results are not required and so the efficiency of the algorithm can be increased by not calculating them.

It is also notable that the amplitude of the peak is 128 units, showing that the *MAGx* values need to be scaled by N/2:

```
       ' Modulus / amplitude calculated from real and imaginary arrays
        For k = 0 To N / 2 - 1
            MagX(k) = Math.Sqrt((Math.Pow(REx(k), 2) + Math.Pow(IMx(k), 2))) / (N/2)
        Next
```

This results in the plot:

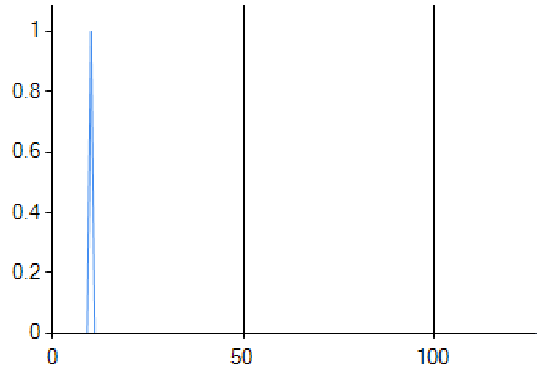


Figure V.1.4

When a series of harmonics are added to the input signal, more amplitudes peaks can be seen on the plot:

```
        For i = 0 To N
            x(i) = Math.Sin((2 * Math.PI * F) * (i / N))
            x(i) += 0.5 * (Math.Sin((2 * Math.PI * (F * 2)) * (i / N)))
            x(i) += 0.25 * (Math.Sin((2 * Math.PI * (F * 4)) * (i / N)))
            x(i) += 0.125 * (Math.Sin((2 * Math.PI * (F * 8)) * (i / N)))
        Next
```

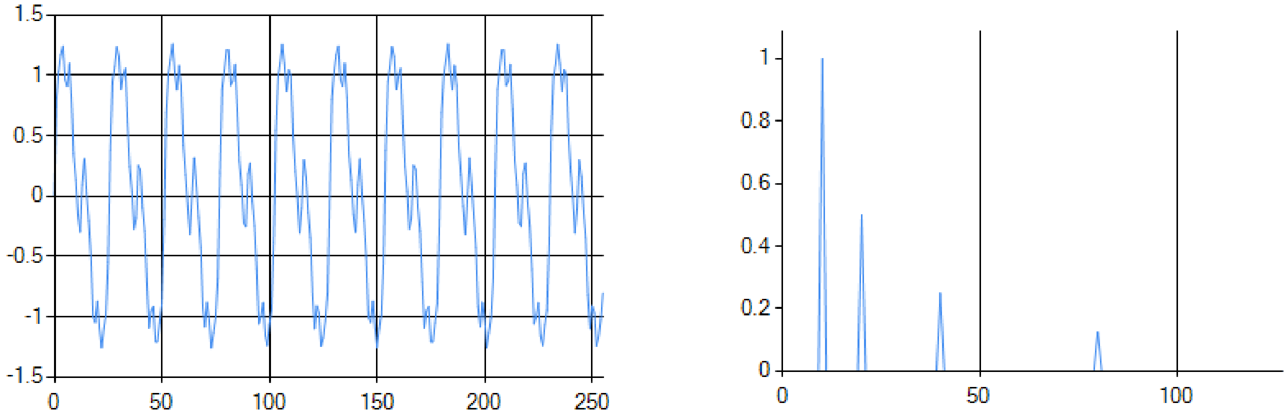


Figure V.1.5

Whilst this algorithm can produce the spectral analysis of signals required, it does so very inefficiently compared to other methods. By definition each DFT sample is a sum of N products and there are N samples to compute, meaning an upper limit of O(N^2) calculations required to compute the spectra. The Fast Fourier Transform (FFT), a variation of the DFT, requires an estimated O($N\log_2 N$) calculations to compute. For example, a DFT of N = 512 requires 262144 calculations to compute, an FFT of the same size requires 4608 calculations, a saving of a factor of 50.

In 1965 J.W. Cooley and John Tukey rediscovered the Fast Fourier Transform algorithm, a re-working of the DFT which computes the transform much faster (Cooley and Tukey, 1965). Many variations of the FFT exist, but all involve some kind of decimation; the separation of an input signal into two signal composed of even and odd indexed samples respectively (Jackson, 1996, p.213):

$$X(k) = \sum_{n=0}^{N-1} x(n)W^{kn}$$

$$= \sum_{n\ even} x(n)W^{kn} + \sum_{n\ odd} x(n)W^{kn}$$

$$= \sum_{m=0}^{\frac{N}{2}-1} x(2m)(W^2)^{km} + W^k \sum_{m=0}^{\frac{N}{2}-1} x(2m+1)(W^2)^{km}$$

Each of these summations can be seen as an *N/2* point DFT of the respective input, and the overall DFT can be written as:

$$X(k) = X_e(k) + W^k X_o(k)$$

W^k is a complex number known as a twiddle factor. A vital contributing factor to the speed of the algorithm is the ability to pre-compute these twiddle factors and store them in memory, so that for the most part only real number calculations need to be carried out. As for the DFT example above, X_e and X_o are periodic in k with period *N/2*, and so the values above *N/2* do not need to be calculated. The decimation is repeated until only 2 2-point DFTs need to be computed. That is, each *N/2* DFT is separated into 2 *N/4* DFTS, each of which is separated into 2 *N/8* DFTs etc., for *r* stages where *N = 2^r^*. The process is demonstrated below with the decimation-in-time FFT on a DFT with *N = 8*.

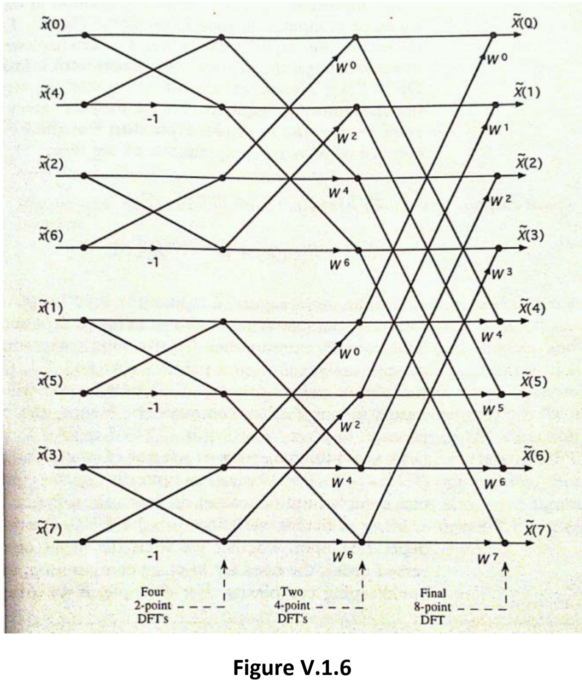


Figure V.1.6

In order for the resulting DFT *X* to be correctly ordered, the input series must be in bit-reversed order:

Input index	Binary	Bit-reversed	Result
0	000	000	0
1	100	001	4
2	010	010	2
3	110	011	6
4	001	100	1
5	101	101	5
6	011	110	3
7	111	111	7

The decimation-in-frequency FFT produces the opposite; a consecutively ordered input and bit-reversed output:

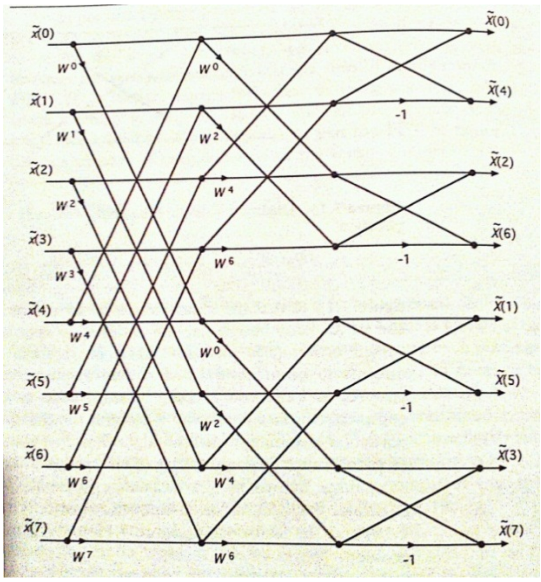


Figure V.1.7

The decimation process restricts the values of *N* which can be implemented because, as mentioned above, it must satisfy the statement $N = 2^r$ so that when *r* is defined from *N* using $r = log_2 N$, *r* is an integer. The value of *N* dictates the resolution of the resulting spectrogram and hence the detail in which the input signal can be analysed. Each FFT amplitude, *X(k)*, is known as a frequency bin. As the FFT algorithm results in the same number of bins as there samples in the input signal, the spectrogram resolution is defined as:

$$Resolution = \frac{Sample\ Rate}{N}$$

The frequency bins are the amplitudes of component periodic waveforms at multiples of this resolution. For example, an FFT where $N = 2^9 = 512$ carried out on a signal sampled at 44.1 kHz has a resolution of approximately 86 Hz, meaning that the lowest frequency component which can be measured is 86 Hz, and from there the values of the frequency bins will correspond to the amplitudes of component frequencies at 172 Hz, 258 Hz, 344 Hz, etc., up to 22.016 kHz at *N/2*.

In order to experiment with the FFT algorithm and related concepts, a Visual Basic prototype was created. FFT was implemented using the Math.Net external library (http://www.mathdotnet.com/), as the plug-in will utilise an external library, but all other functions were designed and implemented from scratch. The following examples are demonstrated using this prototype.

In order to analyse a piano recording, spectral analysis plug-in analyses the audio in batches of *N* samples, most commonly 2048 or 4096, displaying a spectrogram plot for each batch. At 44.1 kHz, a spectrogram of 2048 samples is displayed approximately every 46 ms. The batch is referred to as a window, and the process of windowing prepares a window of samples before the FFT algorithm is applied to it, in order to reduce the effect spectral leakage. Spectral leakage occurs because of the limitations of resolution presented by the discrete digital audio system; any frequency component which does not align exactly with one of the FFT bins will be displayed spread over the two or three bins either side of it:

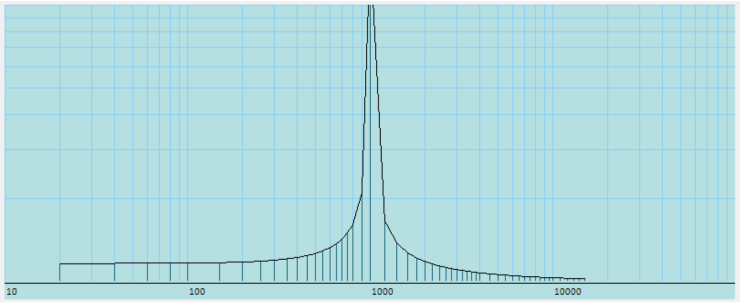


Figure V.1.8

The above plot shows the spectrogram of a 1 kHz sine wave. The amplitude peak of the main beam is very clearly present at 1 kHz, however the base of the peak "spreads out" at lower amplitudes, and the effect is amplified by the logarithmic decibel scale. Side lobes can also been throughout the rest of the spectrum. Spectral leakage gives less accurate spectrograms, hence why windowing is an important process.

Windowing is similar to a crossfade; an amplitude envelope is applied to the window of samples, reducing volume at its beginning and end. There many different windowing envelopes used, and a brief description of the three most common are described below:

Hann window:

$$w(n) = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$$

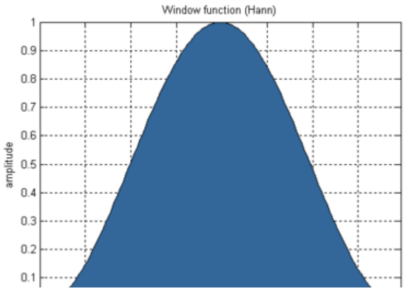


Figure V.1.9

Hamming window:

$$w(n) = 0.54 - 0.46\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$$

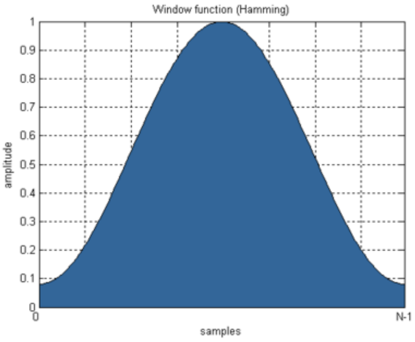


Figure V.1.10

Cosine window:

$$w(n) = \cos\left(\frac{\pi n}{N-1} - \frac{\pi}{2}\right) = \sin\left(\frac{\pi n}{N-1}\right)$$

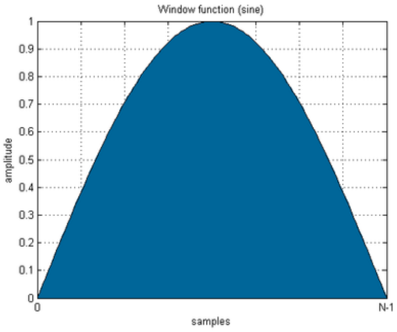


Figure V.1.11

The same 1 kHz sine wave shown above is shown below which each window applied:

Hamming window:

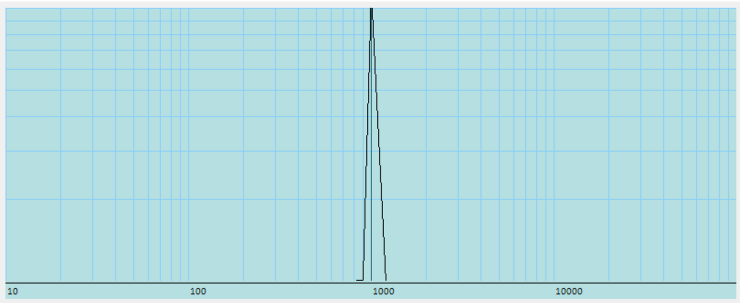


Figure V.1.12a

Hann window:

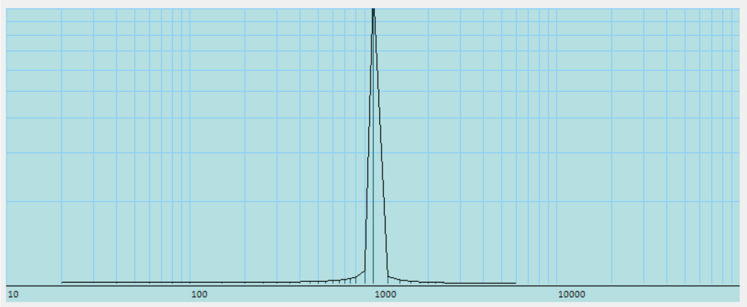


Figure V.1.12b

Cosine window:

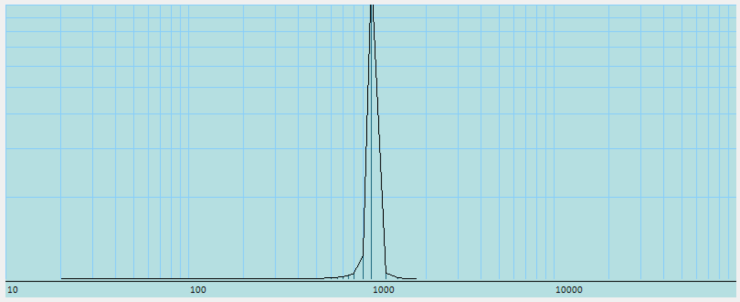


Figure V.1.12c

All three windows reduce the “spreading” effect at the base of the main beam and the amplitude of side lobes throughout the rest of the spectrum.

Curve fitting is a technique for modelling the relationship between two variables using a line of best fit. In linear terms, independent variable *x* and dependant variable *y* are related through an equation of the form *y = mx + c*.

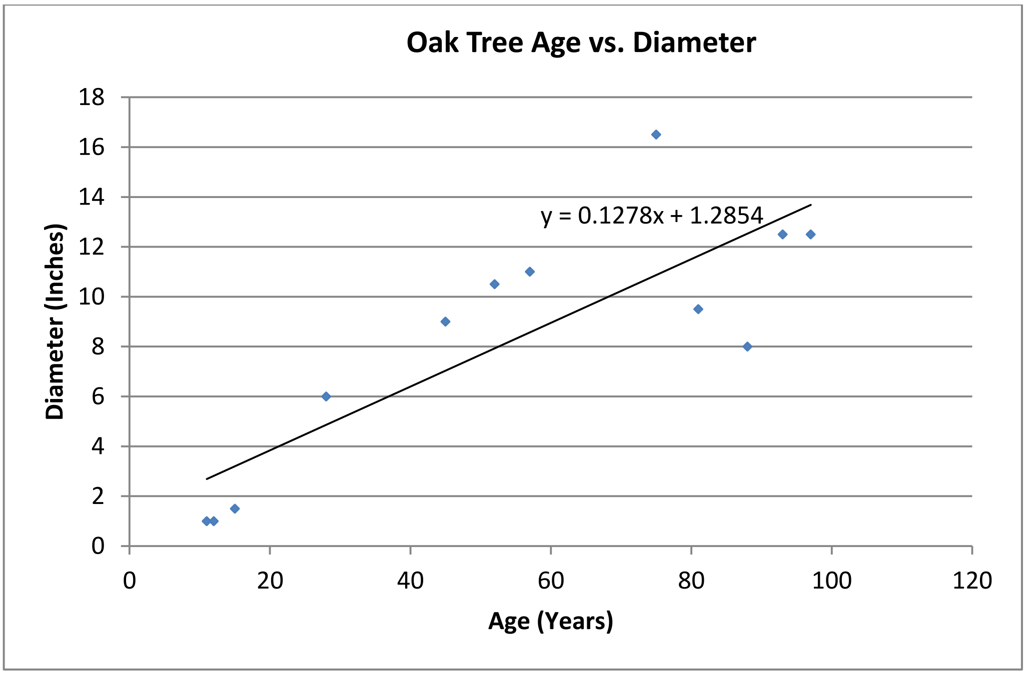


Figure V.2.1

Above is an example of curve fitting using the age of an oak tree and diameter at breast height (Huff, 1964). Using Microsoft Excel's trend line facility, a linear curve has been fitted to the data. The line has a gradient, *m*, of 0.1278 and y-intercept, *c*, of 1.2854, leading to the equation of *y* = 0.1278*x* + 1.2854.

The curve is derived using the process of regression, implemented with the least squares method, which minimizes the sum of the squares of the errors between the actual and predicted data. Assume a set of observed data is denoted x_i, for *i* = 1,…,*N*. An estimate of the data is denoted $\hat{s}_i$ and is comprised of each observed sample weighted by a coefficient, α:

$$\hat{s}_i = \propto x_i$$

The estimate, $\hat{s}_i$, differs from the desired result, s_i, by an error, ε_i:

$$\varepsilon_i = s_i - \hat{s}_i$$

Combining these two equations results in a simple linear estimation:

$$s_i = \hat{s}_i + \varepsilon_i$$

$$\mathrm{s}_i = \alpha x_i + \varepsilon_i$$

A function, *f(α)*, is defined as the sum of the squares of the error components:

$$f(\alpha) = \sum_{i=1}^{N} \varepsilon_i{}^2$$

$$= \sum_{i=1}^{N} (s_i - \alpha x_i)^2$$

When this quadratic function is differentiated with respect to α and the result set to zero,

$$\frac{df(\alpha)}{d\alpha} = \sum_{i=1}^{N} 2(s_i - \alpha x_i)(-x_i) = 0$$

the parameter α is known as the least squares estimate and denoted $\hat{\alpha}$:

$$\hat{\alpha} = \frac{\sum_{i=1}^{N} s_i x_i}{\sum_{i=1}^{N} x_i{}^2}$$

(Giordano and Hsu, 1985)

This is the fundamental principle of least squares curve fitting, and is as much derivation of mathematics as will be entered to in this paper because, as for the Fourier transform, its relevancy to the research does not equal its complexity.

The least squares process can also be used to estimate non-linear relationships. Within the plug-in it is producing a 4th order polynomial of the expected format:

$$y = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

Software algorithms which carry out the process usually return the values of coefficients $a_n,\ldots,a_0$, where n is the order of the polynomial. The purpose of non-linear least squares regression is to allow the use of non-linear curves as lines of best fit. Below is the previously used oak tree example fitted with a second order polynomial, giving a more accurate idea of the relationship trend between age and diameter. The curve, however, is only a model of the relationship as it is obviously incorrect that a tree begins to shrink above 80 years old, but it does demonstrate the non-linear properties of growth; a much faster rate of growth when the tree is young.

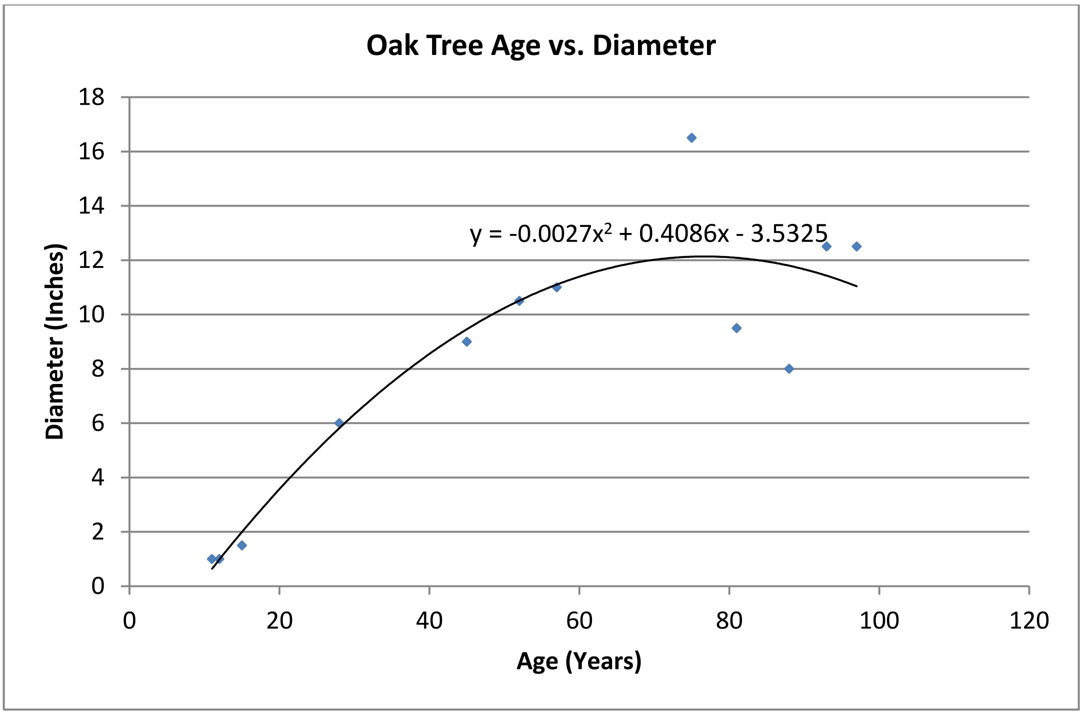


Figure V.2.2

To determine whether polynomial regression is a viable option for smoothing an FFT trace, a piano recording was made and analysed in Cubase 5 using the 1/3 Octave Analyser section of the IXL Spectral Analyser, and data copied by hand into into Microsoft Excel to be regressed. Traces were recorded at four beats of the same bar:

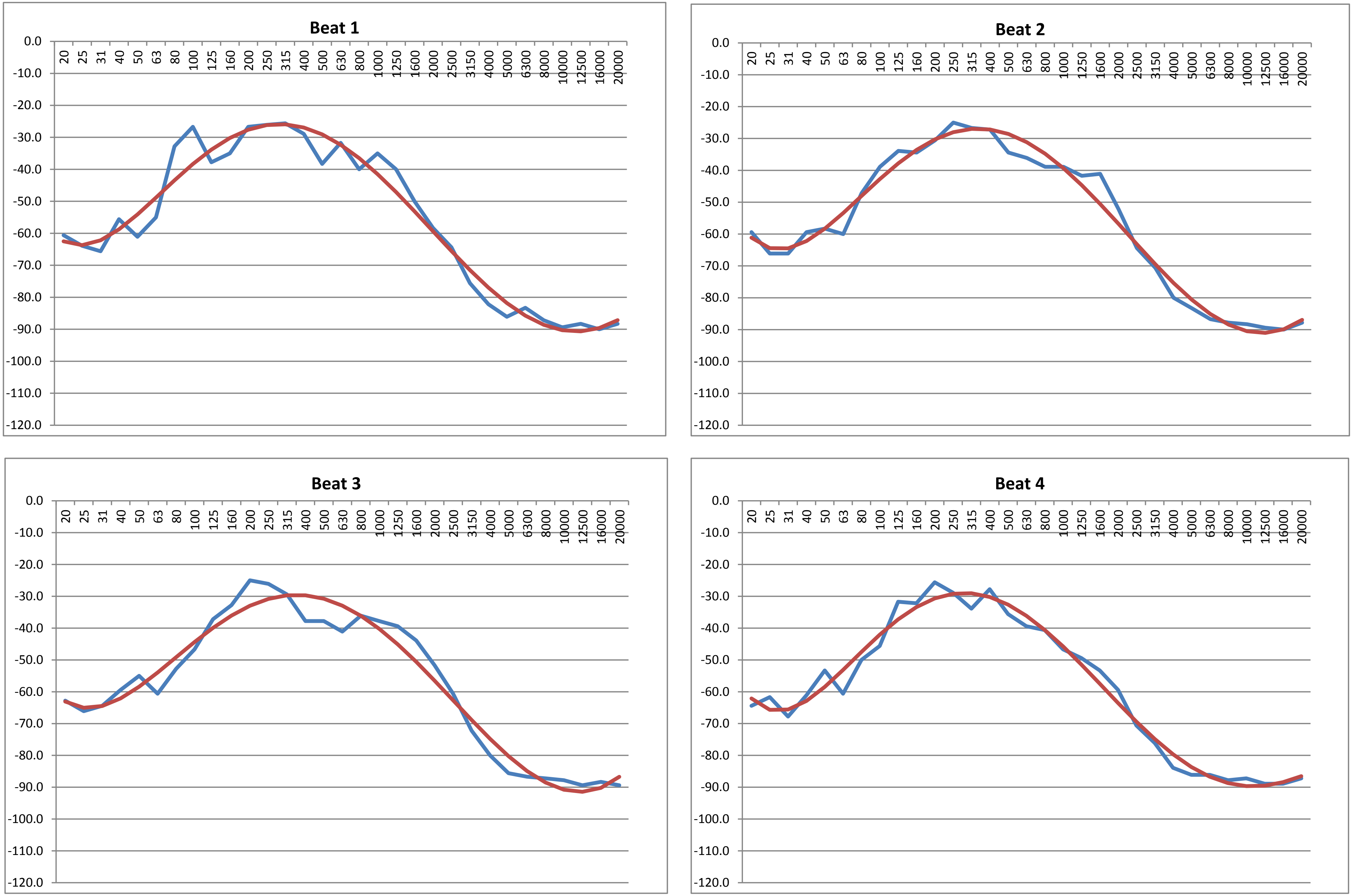

Beat 1
Beat 2
Beat 3
Beat 4
0.0
-10.0
-20.0
-30.0
-40.0
-50.0
-60.0
-70.0
-80.0
-90.0
-100.0
-110.0
-120.0
20
25
31
40
50
63
80
100
125
160
200
250
315
400
500
630
800
1000
1250
1600
2000
2500
3150
4000
5000
6300
8000
10000
12500
16000
20000

Figure V.2.3

The regression very successfully smooths the FFT trace in the frequency domain. The videos* demonstrate that the regression also smooths the FFT trace very successfully in the time domain. The regression also acts as an initial stage of averaging by reducing the variance of the bins. As the parameter calculation algorithm will respond to the regressed FFT trace, this will result in smoother, and hence less audible, changes in parameters. The regression also produces a very clear "shape" in the trace. The shape of the trace can be used as a measure of the tonality of the recording; average amplitudes of relevant spectral bands can be compared to an ideal and altered accordingly.

An example of the usefulness of regression to analyse tonality can be provided by using equalisation to artificially replicate different tonalities and compare pre and post traces. Using the same four beats, the IXL Spectrum Analyser was again used with Microsoft Excel to demonstrate this. The three spectral bands relevant to the plug-in were manipulated using equalisation to simulate a piano recording with correct tonality by applying a high pass filter at 200 Hz, a 9 dB cut with a peaking filter at 500 Hz, and a 9 dB boost with another peaking filter at 4 kHz. Below are four sets of graphs showing pre and post regressed FFT traces, one for each filter and one for all three filters applied at once (see next pages). The plots show very clear differences in the audio pre and post EQ and hence give a useful visual confirmation that the plugin is performing as it should be.

*The videos can be found on the DVD under /Thesis/Videos/Regression

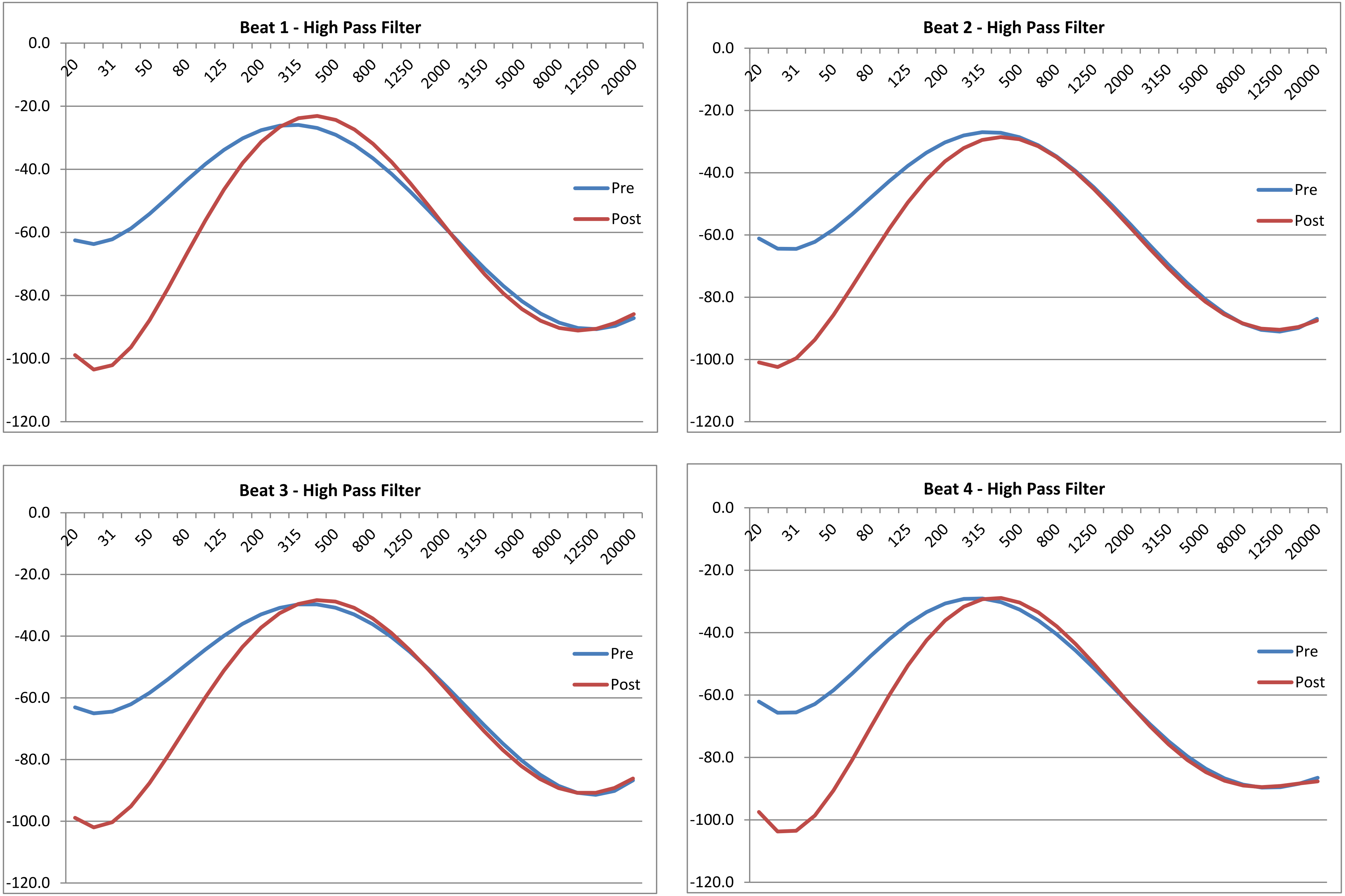

Beat 1 - High Pass Filter
Beat 2 - High Pass Filter
Beat 3 - High Pass Filter
Beat 4 - High Pass Filter
0.0
-20.0
-40.0
-60.0
-80.0
-100.0
-120.0
20
31
50
80
125
200
315
500
800
1250
2000
3150
5000
8000
12500
20000
Pre
Post

Figure V.2.4

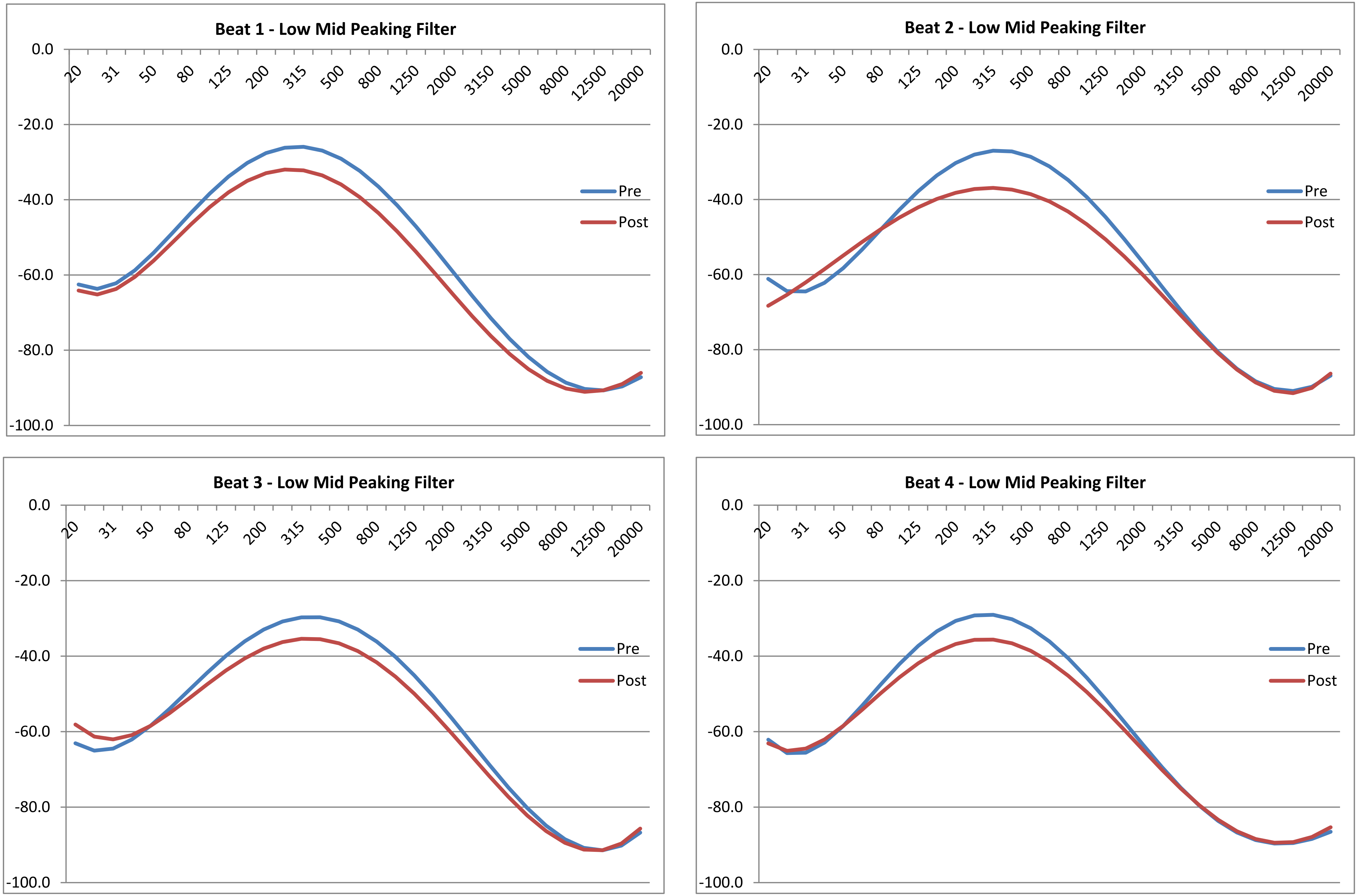

Beat 1 - Low Mid Peaking Filter
Beat 2 - Low Mid Peaking Filter
Beat 3 - Low Mid Peaking Filter
Beat 4 - Low Mid Peaking Filter
0.0
-20.0
-40.0
-60.0
-80.0
-100.0
20
31
50
80
125
200
315
500
800
1250
2000
3150
5000
8000
12500
20000
Pre
Post

Figure V.2.5

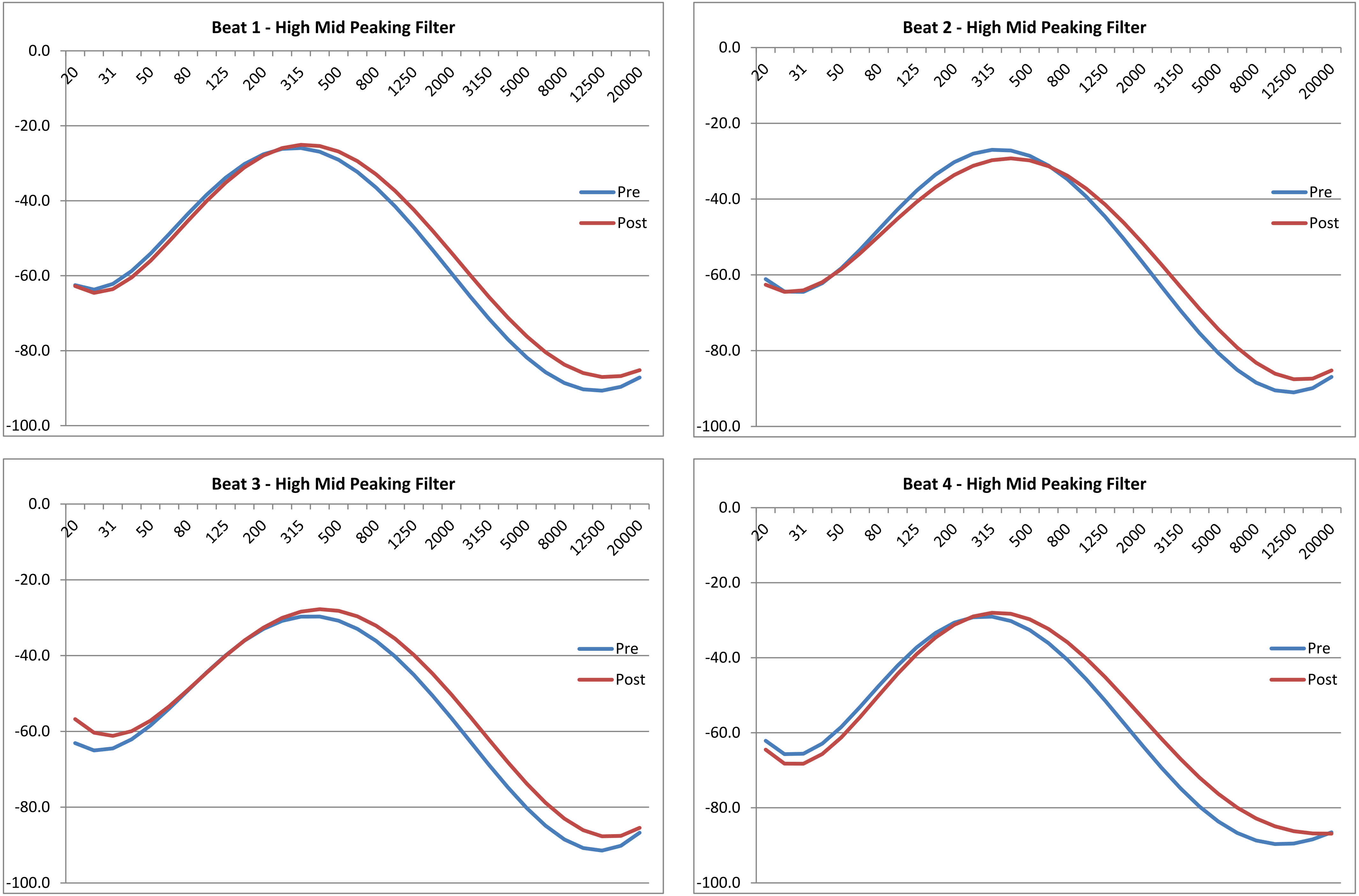

Beat 1 - High Mid Peaking Filter
0.0
-20.0
-40.0
-60.0
-80.0
-100.0
20
31
50
80
125
200
315
500
800
1250
2000
3150
5000
8000
12500
20000
Pre
Post
Beat 2 - High Mid Peaking Filter
0.0
-20.0
-40.0
-60.0
-80.0
-100.0
20
31
50
80
125
200
315
500
800
1250
2000
3150
5000
8000
12500
20000
Pre
Post
Beat 3 - High Mid Peaking Filter
0.0
-20.0
-40.0
-60.0
-80.0
-100.0
20
31
50
80
125
200
315
500
800
1250
2000
3150
5000
8000
12500
20000
Pre
Post
Beat 4 - High Mid Peaking Filter
0.0
-20.0
-40.0
-60.0
-80.0
-100.0
20
31
50
80
125
200
315
500
800
1250
2000
3150
5000
8000
12500
20000
Pre
Post

Figure V.2.6

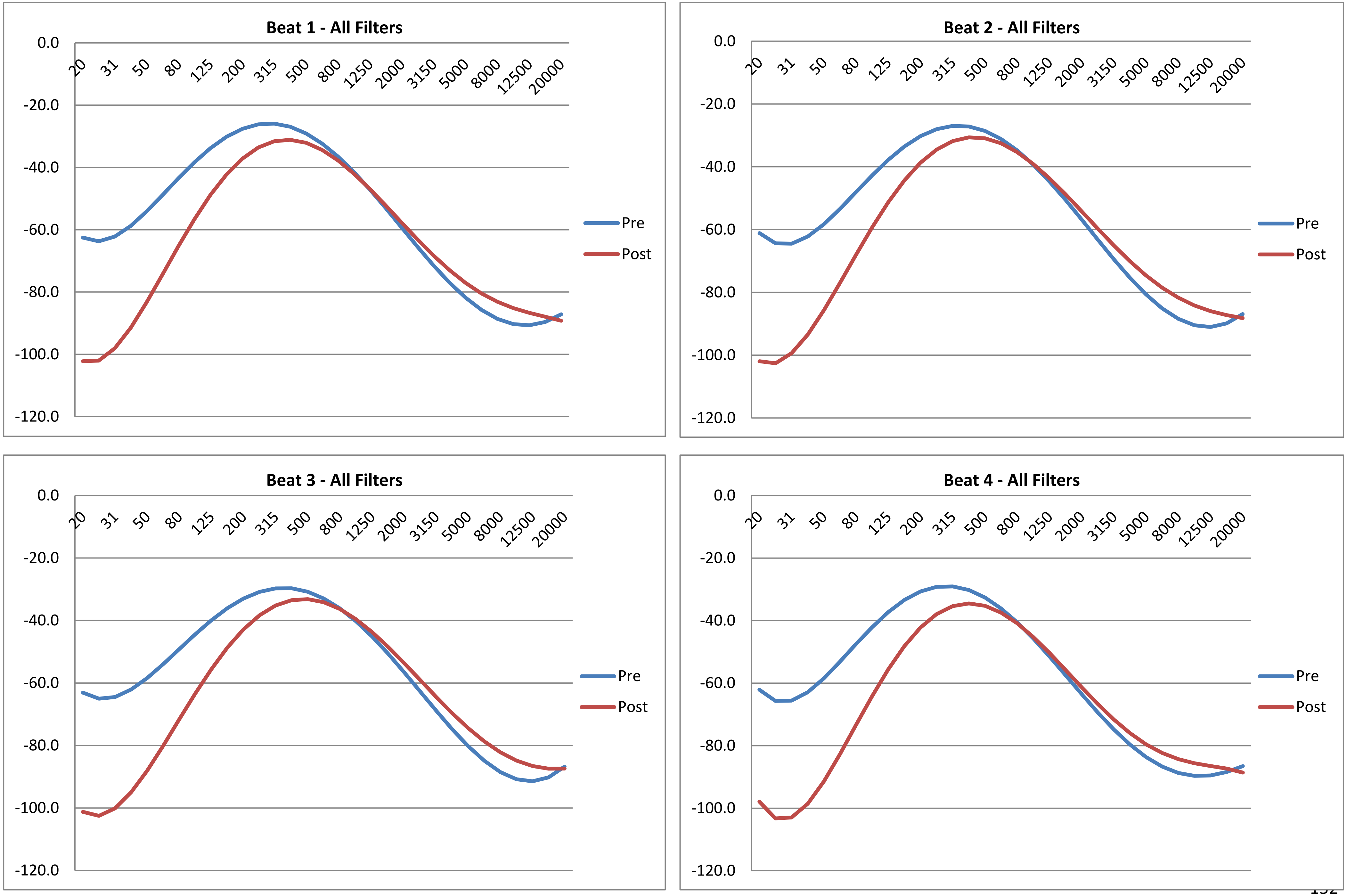

Beat 1 - All Filters
Beat 2 - All Filters
Beat 3 - All Filters
Beat 4 - All Filters
0.0
-20.0
-40.0
-60.0
-80.0
-100.0
-120.0
20
31
50
80
125
200
315
500
800
1250
2000
3150
5000
8000
12500
20000
Pre
Post

Figure V.2.7

V.3

Source Code

V.3.1 Juce_vstAudioProcessor

PluginProcessor.h

```
#ifndef __wtypes_h__
#include <wtypes.h>
#endif

#ifndef __WINDEF_
#include <windef.h>
#endif

#ifndef __PLUGINPROCESSOR_H_641D2B40__
#define __PLUGINPROCESSOR_H_641D2B40__

// JUCE Includes
#include "../JuceLibraryCode/JuceHeader.h"
#include "../JuceLibraryCode/JucePluginCharacteristics.h"

// Object Inclues
#include <FFTALG/FFTALG.h>
#include <Polyfit/Polyfit.h>
#include <FBand/FBand.h>
#include <Bristow-Johnson/EQ.h>
#include <LogScales/LogScales.h>
#include <WriteToFile\WriteToFile.h>

// StringStream and File Output Includes
#include <Windows.h>
#include <tchar.h>
#include <string>
#include <stdio.h>
#include <sstream>

using namespace std;

//==============================================================================

class Juce_vstAudioProcessor  : public AudioProcessor

{
public:

       /*----------------------------------- JUCE Declarations ---------------------------------*/

    //==============================================================================
    Juce_vstAudioProcessor();
    ~Juce_vstAudioProcessor();

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock);
    void releaseResources();

    void processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages);

    //==============================================================================
    AudioProcessorEditor* createEditor();
    bool hasEditor() const;

    //==============================================================================
    const String getName() const;

    int getNumParameters();

    float getParameter (int index);
    void setParameter (int index, float newValue);
```

```
const String getParameterName (int index);
const String getParameterText (int index);

const String getInputChannelName (int channelIndex) const;
const String getOutputChannelName (int channelIndex) const;
bool isInputChannelStereoPair (int index) const;
bool isOutputChannelStereoPair (int index) const;

bool acceptsMidi() const;
bool producesMidi() const;

//==============================================================================
int getNumPrograms();
int getCurrentProgram();
void setCurrentProgram (int index);
const String getProgramName (int index);
void changeProgramName (int index, const String& newName);

//==============================================================================
void getStateInformation (MemoryBlock& destData);
void setStateInformation (const void* data, int sizeInBytes);

   /*--------------------------------- Constant Declarations -------------------------------*/

   static const int SR = 44100; // Sample Rate
   static const int N = 2048; // Logical FFT Size
   static const int FFTRES = 110; // FFT Output Resolution
   static const int LOGBASE = 250; // FFT Output Logarithm Base

   /*----------------------------------- FFT Declarations ----------------------------------*/

   // FFT Objects
   FFTALG FFTpre;
   FFTALG FFTpost;

   // Circular Indexes
   int fftpre_index, fftpost_index;

   // FFT Output Pointers
   float *FFTpre_out, *FFTpost_out;

   /*------------------------------- Regression Declarations -------------------------------*/

   // Regression Objects
   Polyfit REGpre;
   Polyfit REGpost;

   // Output Pointers
   float *REGpre_out, *REGpost_out;

   // X Co-ordinates Pointers
   float *fftX;

   /*-------------------------------- Parameter Declarations -------------------------------*/

   // Parameter Pointer - 3 Frequency Bands
   float F[3]; // Cutoff Frequency
   float G[3]; // Gain
   float Q[3]; // Q
   float T[3]; // Threshold
   float S[3]; // Slope
   float A[3]; // Attack
   float R[3]; // Release
   bool EQio[3]; // EQ Enable
   bool DRCio[3]; // DRC Enable
```

```
// Input Gain
float in_gain;

// Index Values for Parameters
enum Param
{
        _F=0,
        _G,
        _Q,
        _T,
        _S,
        _A,
        _R,
        _EQio,
        _DRCio,
        _St, // FBand Analysis Start
        _En, // FBand Analysis End

        _rms = 37,
        _rec, // Record Data
        _reg_x_print, // Print Data Headers
        _in_gain // Input Gain
};

// Parameter Pointer for Editor
float *ediParams;

// FBand Objects
FBand LS, MP, HP;

// Band Identifiers
enum Band
{
        _LS=0,
        _MP,
        _HP
};

// FBand Initialisation Test
bool Init;

// RMS Containers
float rms, rms_db;
float rms_post, rms_post_db;

/*---------------------------- Performance Timer Declarations ---------------------------*/

// StringStream Object
stringstream ss;

// Clock Count Containers - _s = Start, _e = End
__int64 init_s, init_e, pre_s, pre_e, eff_s, eff_e, post_s, post_e;

// Clock Frequency Container
__int64 tim_res;

// Time Calculation Function
int timeSpan(__int64 s, __int64 e);
float span;

// Time Taken Containers
int init, pre, eff, post, tot;

// Buffer Counter
int numBuffers;
```

```
       /*----------------------------- Miscellaneous Declarations ----------------------------*/

       // Buffer Length
       int numSamples;

       // Stereo Sample Pointers
       float *Lx, *Rx;

       // EQ Series Connection Pointer
       float *out;

       // Function called by Editor to receive parameters for GUI display
       float* getParams(int i);

       // Record Data Test
       bool rec;

       // Print Data Headers Test
       bool reg_x_print;

private:
    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (Juce_vstAudioProcessor);
};

#endif  // __PLUGINPROCESSOR_H_641D2B40__
```

PluginProcessor.cpp

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

// WriteToFile Declaration and Initialisation - need to be global to be accesible in processBlock
WriteToFile reg_coeffs("D:\\Libraries\\MSc\\Project\\Testing\\Data Outputs\\reg_coeffs.csv");
WriteToFile dynamics("D:\\Libraries\\MSc\\Project\\Testing\\Data Outputs\\dynamics.csv");
WriteToFile parameters("D:\\Libraries\\MSc\\Project\\Testing\\Data Outputs\\parameters.csv");
WriteToFile performance("D:\\Libraries\\MSc\\Project\\Testing\\Data Outputs\\performance.csv");

Juce_vstAudioProcessor::Juce_vstAudioProcessor()
{
       /*--------------------------------- FFT Initialisation ------------------------------------*/

       // FFT object
       FFTpre.Init(N, FFTRES, float(LOGBASE));
       FFTpost.Init(N, FFTRES, float(LOGBASE));

       // Circular Indexes
       fftpre_index = 0;
       fftpost_index = 0;

       // FFT Output Pointers
       FFTpre_out = new float[FFTRES];
       FFTpost_out = new float[FFTRES];

       // Set FFT outputs to 0
       for (int i=0; i<FFTRES; i++)
       {
              FFTpre_out[i] = 0.0;
              FFTpost_out[i] = 0.0;
       }

       /*------------------------------- Regression Initialisation --------------------------------
*/
```

```
// Regression Objects
REGpre.Init(4);
REGpost.Init(4);

// Output Pointers
REGpre_out = new float[FFTRES];
REGpost_out = new float[FFTRES];

/*------------------------------- Plotting Initialisation ---------------------------------*/

// Plotting Object - X Co-ordinates Pointer for Regression
Plot fft;
fft.Init(SR, N, FFTRES, LOGBASE, 30, 10, 760, 264);
fftX = fft.X;

/*------------------------------ Parameter Initialisation ---------------------------------*/

F[_LS] = 20.0;
G[_LS] = -24.0;
Q[_LS] = 1.0;
T[_LS] = -45.0;
S[_LS] = 50.0;
A[_LS] = 20.0;
R[_LS] = 200.0;
EQio[_LS] = false;
DRCio[_LS] = false;

F[_MP] = 500.0;
G[_MP] = -6.0;
Q[_MP] = 0.5;
T[_MP] = -40.0;
S[_MP] = 50.0;
A[_MP] = 20.0;
R[_MP] = 200.0;
EQio[_MP] = false;
DRCio[_MP] = false;

F[_HP] = 5000;
G[_HP] = 8.0;
Q[_HP] = 0.5;
T[_HP] = -40.0;
S[_HP] = 40.0;
A[_HP] = 10.0;
R[_HP] = 80.0;
EQio[_HP] = false;
DRCio[_HP] = false;

// Input Gain
in_gain = 1.0;

// FBand Initialisation Test
Init = false;

// Parameter Pointer for Editor
ediParams = new float[12];
for (int i=0; i<12; i++)
       ediParams[i] = 0.0;

/*------------------------- Performance Testing Initialisation --------------------------*/

// Clock Frequency
QueryPerformanceFrequency((LARGE_INTEGER *)&tim_res);

// Clock Containers
init_s = 0;
init_e = 0;
```

```
       pre_s = 0;
       pre_e = 0;
       eff_s = 0;
       eff_e = 0;
       post_s = 0;
       post_e = 0;

       // Total Time Taken
       tot = 0;

       // Buffer Counter
       numBuffers = 0;


       /*----------------------------- Miscellaneous Initialisation ------------------------------*/

       // Record Data Test
       rec = false;

       // Print Data Headers Test
       reg_x_print = false;

}

Juce_vstAudioProcessor::~Juce_vstAudioProcessor()
{
}

//===============================================================================

float Juce_vstAudioProcessor::getParameter (int index)
{
       // Called by Editor to retrieve individual parameter value stored in Processor

       // 3 frequency bands, 12 parameters available per band
       // Index values : 0-11, 12-23, 24-36

       if (index <= 36)
       {
              // Sent index value used to calculate:
              // i : frequency band (0-2)
              // index : parameter index (0-11)

              int i = int(index / 12);
              index -= i * 12;

              // new index value used to retrieve paramter from i'th pointer position

              switch (index)
              {
                     case _F:                    return F[i];                             break;
                     case _G:                    return G[i];                             break;
                     case _Q:                    return Q[i];                             break;
                     case _T:                    return T[i];                             break;
                     case _S:                    return S[i];                             break;
                     case _A:                    return A[i];                             break;
                     case _R:                    return R[i];                             break;
                     case _EQio:                 return EQio[i];
       break;
                     case _DRCio:         return DRCio[i];                         break;

                     case _St:
                            switch (i)
                            {
                                   case 0:       return LS.fft_frequency; break;
                                   case 1: return MP.fft_frequency; break;
                                   case 2: return HP.fft_frequency; break;
                            }

              break;
                     case _En:
                            switch (i)
                            {
                                   case 0:       return LS.fft_width; break;
                                   case 1: return MP.fft_width; break;
                                   case 2: return HP.fft_width; break;
                            }

              break;
                     default:                    break;
              }
       }
       else
       {
              // Other Parameters

              switch (index)
              {
                     case _rms: return rms; break;
              }
       }
}

float* Juce_vstAudioProcessor::getParams(int i)
{
       // Called by Editor to retrieve parameters for GUI display
       // Receieves band identifier and populates ediParams pointer accordingly

       // FBand Parameters

       switch (i)
       {
       case _LS:
              ediParams[0] = 20*log10(LS.Comp.level);
              ediParams[1] = LS.F;
              ediParams[2] = LS.G;
              ediParams[3] = LS.Q;
              ediParams[4] = LS.fft_frequency;
              ediParams[5] = LS.fft_width;
              break;
       case _MP:
              ediParams[0] = 20*log10(MP.Comp.level);
              ediParams[1] = MP.F;
              ediParams[2] = MP.G;
              ediParams[3] = MP.Q;
              ediParams[4] = MP.fft_frequency;
              ediParams[5] = MP.fft_width;
              break;
       case _HP:
              ediParams[0] = 20*log10(HP.Comp.level);
              ediParams[1] = HP.F;
              ediParams[2] = HP.G;
              ediParams[3] = HP.Q;
              ediParams[4] = HP.fft_frequency;
              ediParams[5] = HP.fft_width;
              break;
       }

       // Other Parameters
       ediParams[6] = init;
       ediParams[7] = pre;
       ediParams[8] = eff;
       ediParams[9] = post;
       ediParams[10] = tot;
       ediParams[11] = rms_db;

       return ediParams;
}


void Juce_vstAudioProcessor::setParameter (int index, float newValue)
{
       // Called by Editor to send new parameter values from GUI to Processor
       // See getParameter()

       if (index <= 36)
       {
              int i = int(index / 12);
              index -= i * 12;

              switch (index)
              {
                     case _F:                   F[i] = newValue;
              break;
                     case _G:                   G[i] = newValue;
              break;
                     case _Q:                   Q[i] = newValue;
              break;
                     case _T:                   T[i] = newValue;
              break;
                     case _S:                   S[i] = newValue;
              break;
                     case _A:                   A[i] = newValue;
              break;
                     case _R:                   R[i] = newValue;
              break;
                     case _EQio:                EQio[i] = (newValue == 1) ? true : false; break;
                     case _DRCio:        DRCio[i] = (newValue == 1) ? true : false;      break;

                     case _St:
                            switch (i)
                            {
                                   case 0:       LS.fft_frequency = int(newValue); break;
                                   case 1: MP.fft_frequency = int(newValue); break;
                                   case 2: HP.fft_frequency = int(newValue); break;
                            }

              break;
                     case _En:
                            switch (i)
                            {
                                   case 0:       LS.fft_width = int(newValue); break;
                                   case 1: MP.fft_width = int(newValue); break;
                                   case 2: HP.fft_width = int(newValue); break;
                            }

              break;

                     default:                   break;
              }

              // Send new parameters to Compression / EQ Objects if necessary

              if (index != _EQio && index != _DRCio)
              {
                     switch (i)
                     {
                     case _LS:
```

```
                if (EQio[i] && DRCio[i])
                    LS.Comp.setParameters(T[i], S[i], A[i], R[i]);
                else if (EQio[i] && !DRCio[i])
                    LS.EQ.setCoefficients(F[i], G[i], Q[i]);
                break;
            case _MP:
                if (EQio[i] && DRCio[i])
                    MP.Comp.setParameters(T[i], S[i], A[i], R[i]);
                else if (EQio[i] && !DRCio[i])
                    MP.EQ.setCoefficients(F[i], G[i], Q[i]);
                break;
            case _HP:
                if (EQio[i] && DRCio[i])
                    HP.Comp.setParameters(T[i], S[i], A[i], R[i]);
                else if (EQio[i] && !DRCio[i])
                    HP.EQ.setCoefficients(F[i], G[i], Q[i]);
                break;
            }
        }
    }
    else
    {
        switch (index)
        {
            case _rec: rec = (newValue == 1) ? true : false;break;
            case _reg_x_print: reg_x_print = true;                        break;
            case _in_gain: in_gain = pow(10.0,newValue/20.0);      break;
        }
    }
}


void Juce_vstAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{

}

void Juce_vstAudioProcessor::releaseResources()
{

}

void Juce_vstAudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages)
{

    // Performance Timer - Initialisation Start
    QueryPerformanceCounter((LARGE_INTEGER *)&init_s);

    // Get buffer length
    numSamples = buffer.getNumSamples();

    // FBand Initalisation if necessary

    if (!Init)
    {
        // FBand Object - (buffer size, sample rate, compression type, analysis start pointer 
index,
        // analysis width in pointer indeces)
        LS.Init(numSamples, SR, EQ::LowShelf, 0, 1, 3);
        // FBand Parameters - (frequency base, frequency ratio, gain base, gain ratio)
        LS.setParams(20.0, 5.0, -24.0, 1.0);
        // EQ Parameters
        LS.EQ.setCoefficients(F[_LS], G[_LS], Q[_LS]);
        // Comression Parameters
        LS.Comp.setParameters(T[_LS], S[_LS], A[_LS], R[_LS]);
        MP.Init(numSamples, SR, EQ::Parametric, 0, 10, 9);
        MP.setParams(500.0, 0.0, 0.0, -1.0);
        MP.EQ.setCoefficients(F[_MP], G[_MP], Q[_MP]);
        MP.Comp.setParameters(T[_MP], S[_MP], A[_MP], R[_MP]);

        HP.Init(numSamples, SR, EQ::Parametric, 0, 75, 25);
        HP.setParams(5000.0, 0.0, 6.0, -1.0);
        HP.EQ.setCoefficients(F[_HP], G[_HP], Q[_HP]);
        HP.Comp.setParameters(T[_HP], S[_HP], A[_HP], R[_HP]);

        Init = true;
}

/*--------------------------------- Initialisation Section --------------------------------*/

// Get samples from buffer

Lx = buffer.getSampleData(0);
Rx = buffer.getSampleData(1);

// Calculate RMS

float total = 0.0;

for (int i=0; i<numSamples; i++)
{
        Lx[i] *= in_gain;
        Rx[i] *= in_gain;
        total += (Lx[i] * Lx[i]) + (Rx[i] * Rx[i]);
}

total /= numSamples;

rms = sqrt(total);

rms_db = 20 * log10(rms);

// Reset circular indeces if needed (failsafe)

if (fftpre_index == N)
        fftpre_index = 0;

if (fftpost_index == N)
        fftpost_index = 0;

// Performance Timer - Initialisation End
QueryPerformanceCounter((LARGE_INTEGER *)&init_e);
// Performance Timer - Pre-Effect Analysis Start
QueryPerformanceCounter((LARGE_INTEGER *)&pre_s);

/*------------------------------ Pre-Effect Analysis Section ------------------------------*/

// Loop through samples

for (int i=0; i<numSamples; i++)
{
        // If circular index has reached logical FFT size, do FFT and Regression. If not, add
        // current sample to FFT input pointer

        if (fftpre_index >= N)
        {
                // Window samples for FFT
                FFTpre.Window(wintype::hann);

                // Do FFT and return to output pointer
                FFTpre_out = FFTpre.Do();
                // Do regression and return to output pointer
                REGpre_out = REGpre.Solve(fftX, FFTpre_out, FFTRES);

                // reset circular index and exit sample loop
                fftpre_index = 0;
                break;
            }
            else
            {
                // Add current sample to FFT input pointer
                FFTpre.Lc[fftpre_index] = Lx[i];
                FFTpre.Rc[fftpre_index] = Rx[i];

                // Increment circular index
                fftpre_index++;
            }
        }

        // Performance Timer - Pre-Effect Analysis Start
        QueryPerformanceCounter((LARGE_INTEGER *)&pre_e);
        // Performance Timer - Effects Processing Start
        QueryPerformanceCounter((LARGE_INTEGER *)&eff_s);


        /*------------------------------ Effects Processing Section ------------------------------*/

        // Loop through frequency bands

        for (int i=0; i<3; i++)
        {
            // Check if effects enabled

            if (EQio[i] || DRCio[i])
            {
                switch (i)
                {
                // Split channel input, interleaved output
                case _LS:
                    out = LS.Do(Lx, Rx, REGpre_out, EQio[i], DRCio[i]);
                    break;
                case _MP:
                    out = MP.Do(Lx, Rx, REGpre_out, EQio[i], DRCio[i]);
                    break;
                case _HP:
                    out = HP.Do(Lx, Rx, REGpre_out, EQio[i], DRCio[i]);
                    break;
                }

                // Transfer interleaved effected samples back into split channel pointers -
series
                // connection
                int j = 0;
                for (int k=0; k<numSamples*2; k+=2)
                {
                    Lx[j] = out[k];
                    Rx[j] = out[k+1];
                    j++;
                }
            }
        }

        // Performance Timer - Effects Processing End
        QueryPerformanceCounter((LARGE_INTEGER *)&eff_e);
        // Performance Timer - Post-Effect Analysis Start
        QueryPerformanceCounter((LARGE_INTEGER *)&post_s);
```

```
   /*------------------------------ Post-Effect Analysis Section -----------------------------*/

   // Same as Pre-Effect

   for (int i=0; i<numSamples; i++)
   {
          if (fftpost_index >= N)
          {
                 FFTpost.Window(wintype::hann);
                 FFTpost_out = FFTpost.Do();

                 REGpost_out = REGpost.Solve(fftX, FFTpost_out, FFTRES);

                 fftpost_index = 0;

                 break;
          }
          else
          {
                 FFTpost.Lc[fftpost_index] = Lx[i];
                 FFTpost.Rc[fftpost_index] = Rx[i];


                 fftpost_index++;
          }
   }

   // Performance Timer - Post-Effect Analysis End
   QueryPerformanceCounter((LARGE_INTEGER *)&post_e);

   /*-------------------------------------- Output Section -----------------------------------*/

   // RMS Calculation

   total = 0.0;

   for (int i=0; i<numSamples; i++)
   {
          total += (Lx[i] * Lx[i]) + (Rx[i] * Rx[i]);
   }

   total /= numSamples;

   rms_post = sqrt(total);

   rms_post_db = 20 * log10(rms_post);

   // Copy effected samples into buffer
   buffer.copyFrom(0, 0, Lx, numSamples);
   buffer.copyFrom(1, 0, Rx, numSamples);

// If more outputs than inputs, clear unused input buffers in case of garbage

for (int i = getNumInputChannels(); i < getNumOutputChannels(); ++i)
{
    buffer.clear (i, 0, buffer.getNumSamples());
}

   /*------------------------------- Performance Timer Section -------------------------------*/

   // Increment buffer counter
   numBuffers++;

   // Calculation of number of buffers in period of output frequency
```

```
float buffer_s = float(int(float(numSamples) / float(SR) * 1000)) / 1000.0;
int _5Hz = int((1.0/5.0)/buffer_s);
int _25Hz = int((1.0/25.0)/buffer_s);
int _1Hz = int(1.0/buffer_s);

// Check recording enabled

if (rec)
{
        // Print X values for regression curves at the start of each batch of data

        if (reg_x_print)
        {
                // Empty stringstream
                ss.str("");
                ss << "x,";

                float fhz;

                // Loop through FFT bins

                for (int i=0; i<FFTRES; i++)
                {
                        // Converting the array indexes to Hz
                        fhz = FFTpre.log.index[i] * (float(SR) / float(N));

                        // Add current value to stringstream with ',' for CSV formatting
                        ss << fhz << ",";
                }

                // WriteToFile Object writes stringstream to file
                reg_coeffs.Write(ss.str());

                reg_x_print = false;
        }

        // Modulus of numBuffers value with period of desired output frequency in buffers
        controls
        // output frequency

        // Regression Plot @ 10Hz

        if (numBuffers % _5Hz == 0)
        {
                ss.str("");
                ss << numBuffers << ",";
                float yval;

                for (int i=0; i<FFTRES; i++)
                {
                        // Inverting the dB amplitudes to allow them to be plotted on
                        // logarithmic axes - Excel can't manage negative values
                        yval = 200 - abs(REGpost.curve[i]);
                        ss << yval << ",";
                }

                reg_coeffs.Write(ss.str());
        }

        // RMS + FFT Band @ 25Hz;

        if (numBuffers % _25Hz == 0)
        {
                // Calculating individual frequency band output levels

                float ls = 0, mp = 0, hp = 0;
                for (int i=LS.fft_frequency; i<LS.fft_frequency+LS.fft_width; i++)
                        ls += FFTpost.out[i];

                ls /= LS.fft_width;
                ls = (0.8 * ls) + (0.2 * rms);

                for (int i=MP.fft_frequency; i<MP.fft_frequency+MP.fft_width; i++)
                        mp += FFTpost.out[i];

                mp /= MP.fft_width;
                mp = (0.8 * mp) + (0.2 * rms);

                for (int i=HP.fft_frequency; i<HP.fft_frequency+HP.fft_width; i++)
                        hp += FFTpost.out[i];

                hp /= HP.fft_width;
                hp = (0.8 * hp) + (0.2 * rms);

                ss.str("");
                ss << numBuffers;
                ss << ",";

                ss << rms_post_db;
                ss << ",";
                ss << ls;
                ss << ",";
                ss << mp;
                ss << ",";
                ss << hp;

                dynamics.Write(ss.str());
        }

        // Parameters @ 25Hz;

        if (numBuffers % _25Hz == 0)
        {
                ss.str("");
                ss << numBuffers;
                ss << ",";

                ss << LS.F;;
                ss << ",";
                ss << LS.G;
                ss << ",";
                ss << LS.Q;
                ss << ",";
                ss << MP.F;;
                ss << ",";
                ss << MP.G;
                ss << ",";
                ss << MP.Q;
                ss << ",";
                ss << HP.F;;
                ss << ",";
                ss << HP.G;
                ss << ",";
                ss << HP.Q;
                ss << ",";

                parameters.Write(ss.str());
        }
}

// Performance Timers @ 1Hz + RESET
```

```
       if (numBuffers % _1Hz == 0)
       {
              // Calculate time span using clock values
              init = timeSpan(init_s, init_e);
              pre = timeSpan(pre_s, pre_e);
              eff = timeSpan(eff_s, eff_e);
              post = timeSpan(post_s, post_e);

              tot = init + pre + eff + post;

              if (rec)
              {
                     ss.str("");
                     ss << init << "," << pre << "," << eff << "," << post << "," << tot;

                     performance.Write(ss.str());
              }
       }

}

int Juce_vstAudioProcessor::timeSpan(__int64 s, __int64 e)
{
       span = e - s;
       span /= tim_res;
       span *= 1e6;
       s = 0;
       e = 0;
       return int(span);
}

//====================================== JUCE Functions =====================================//

bool Juce_vstAudioProcessor::hasEditor() const
{
    return true; // (change this to false if you choose to not supply an editor)
}

AudioProcessorEditor* Juce_vstAudioProcessor::createEditor()
{
    return new Juce_vstAudioProcessorEditor (this);
}

//==============================================================================
void Juce_vstAudioProcessor::getStateInformation (MemoryBlock& destData)
{
    // You should use this method to store your parameters in the memory block.
    // You could do that either as raw data, or use the XML or ValueTree classes
    // as intermediaries to make it easy to save and load complex data.
}

void Juce_vstAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    // You should use this method to restore your parameters from this memory block,
    // whose contents will have been created by the getStateInformation() call.
}

//==============================================================================
// This creates new instances of the plugin..
AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new Juce_vstAudioProcessor();
}

const String Juce_vstAudioProcessor::getName() const
```

```
{
    return JucePlugin_Name;
}

int Juce_vstAudioProcessor::getNumParameters()
{
    return 0;
}

const String Juce_vstAudioProcessor::getParameterName (int index)
{
    return String::empty;
}

const String Juce_vstAudioProcessor::getParameterText (int index)
{
    return String::empty;
}

const String Juce_vstAudioProcessor::getInputChannelName (int channelIndex) const
{
    return String (channelIndex + 1);
}

const String Juce_vstAudioProcessor::getOutputChannelName (int channelIndex) const
{
    return String (channelIndex + 1);
}

bool Juce_vstAudioProcessor::isInputChannelStereoPair (int index) const
{
    return true;
}

bool Juce_vstAudioProcessor::isOutputChannelStereoPair (int index) const
{
    return true;
}

bool Juce_vstAudioProcessor::acceptsMidi() const
{
#if JucePlugin_WantsMidiInput
    return true;
#else
    return false;
#endif
}

bool Juce_vstAudioProcessor::producesMidi() const
{
#if JucePlugin_ProducesMidiOutput
    return true;
#else
    return false;
#endif
}

int Juce_vstAudioProcessor::getNumPrograms()
{
    return 0;
}

int Juce_vstAudioProcessor::getCurrentProgram()
{
    return 0;
}
```

```
void Juce_vstAudioProcessor::setCurrentProgram (int index)
{
}

const String Juce_vstAudioProcessor::getProgramName (int index)
{
    return String::empty;
}

void Juce_vstAudioProcessor::changeProgramName (int index, const String& newName)
{
}
```

V.3.2 FFTALG

FFTALG.h

```
// ALGLIB Include
#include <fasttransforms.h>
// Object Include
#include <LogScales/LogScales.h>

#ifndef __mbf_fftalg__
#define __mbf_fftalg__

using namespace alglib;

class FFTALG
{
public:

       // Initialisation Function
       void Init(int _N, int _L, float log_base);
       // FFT Function
       float* Do();
       // Convert to dB Function
       float dB(float in);
       // Magnitude Function
       float c2r(float c, float r);
       // Windowing Function
       void Window(int type);
       // Sinc Function
       float sinc(float x);

       // FFT Logical Size, Output Size
       int N, L;

       // ALGLIB Real Input
       real_1d_array in;
       //ALGLIB Complex Output
       complex_1d_array X;

       // Sample and Output Pointers
       float *Lc, *Rc;
       float *out;

       // LogScales Object
       LogScales log;

       // PI
       double pi;

       // Windowing Coefficient
       float a;
};

// Identifiers for window type

class wintype
{
public:
       enum
       {
              none = 0,
              hann,
              hamming,
              tukey,
              cosine,
              lanczos
```

```
        };
};


#endif;
```

FFTALG.cpp

```
#include <FFTALG/FFTALG.h>
#include <math.h>

void FFTALG::Init(int _N, int _L, float LOGBASE)
{
        // Copy FFT logical and output sizes to local variables
        N = _N;
        L = _L;

        // Set size of ALGLIB inputs and sample pointers
        in.setlength(N);
        X.setlength(N);

        Lc = new float[N];
        Rc = new float[N];

        // Define PI to avoid expensive pi() calls
        pi = 3.14159265358979323846264338327950;

        // Windowing Coefficient
        a = 0.5;

        // Generate logarithmic index
        log.Do(N, L, LOGBASE);
}

float* FFTALG::Do()
{
        // Populate ALGLIB inputs from input pointers

        for (int i=0; i<N; i++)
        {
                in[i] = (Lc[i] + Rc[i]) / 2;
        }

        // Do ALGLIG FFT
        fftr1d(in, X);

        // Empty output pointer
        out = new float[L];

        // Populate output pointer with converted values on logarithmic scale
        for(int i=0; i<L; i++)
        {
                out[i] = dB(c2r(float(X[log.index[i]].x), float(X[log.index[i]].y)));
        }

        return out;
}

float FFTALG::dB(float in)
{
        // Convert FFT bin amplitude to Decibels
        return float(20*log10(2 * in / float(N)));
}

float FFTALG::c2r(float c, float r)
{
        // Caluclate magnitude of complex number
        return float(sqrt((c*c) + (r*r)));
}

/*---------------------------------------- Windowing ----------------------------------------*/

void FFTALG::Window(int type)
{
        // Sample coefficient to apply window
        float multi = 0;

        if (type != wintype::none)
        {
                // Loop through samples calculating relevant coefficient

                for (int i=0; i<N; i++)
                {
                        switch (type)
                        {
                        case wintype::hann:
                                multi = float(0.5 * (1 - cos((2 * pi * i) / (N - 1))));
                                break;

                        case wintype::hamming:
                                multi = float(0.54 - (0.46 * cos((2 * pi * i) / (N - 1))));
                                break;

                        case wintype::tukey:
                                if ( (0 <= i) & (i <= (a*N)/2) )
                                        multi = float(0.5* (1 + cos(pi * ((2*i)/(a*N)) - 1)));
                                if ( (N*(1-(a/2)) <= i) & (i <= N) )
                                        multi = float(0.5* (1 + cos(pi * (((2*i)/(a*N)) - (2/a) + 1))));
                                break;

                        case wintype::cosine:
                                multi = float(sin((pi * i) / (N - 1)));
                                break;

                        case wintype::lanczos:
                                multi = float(sinc((float(2 * i)/float(N - 1)) - 1));
                                break;
                        }

                        // Apply coefficient to current sample
                        Lc[i] *= multi;
                        Rc[i] *= multi;
                }
        }
}

float FFTALG::sinc(float x)
{
        // Sinc function
        return float(sin(pi * x) / (pi * x));
}
```

LogScales.h

```
#ifndef __mbf_log__
#define __mbf_log__

class LogScales
{

public:
        // Main Function
        void Do(int N, int L, float LOGBASE);

        // Ratio of logical size to output size
        int inc;
        // Index values pointer
        int *index;
        // Intermediate variable
        float x;
};

#endif
```

LogScales.h

```
#include "LogScales.h"
#include <math.h>

void LogScales::Do(int N, int L, float LOGBASE)
{
        // Usable FFT size - conjugate symmetry
        int N21 = int(float(N)/2 - 1);

        // Increment required to loop through N21 samples in L steps
        inc = N21 / L;

        // Define pointer
        index = new int[L];

        int j = 0;

        // Populate pointer

        for(int i=0; i<N21; i+=inc)
        {
                // logb(x) = loga(x) / loga(b)
                x = (1 - (float(i) / float(N21))) * (LOGBASE - 1) + 1;
                index[j] = int((1 - (log10(x) / log10(LOGBASE))) * (float(N21)));
                j++;
                if(j >= L)
                {
                        break;
                }
        }
}
```

V.3.4 Polyfit

Polyfit.h

```
// ALGLIB Inclues
#include <ap.h>
#include <interpolation.h>

#ifndef __mbf_polyfit__
#define __mbf_polyfit__

using namespace alglib;

class Polyfit
{
public:
       // Initialisation Function
       void Init(int degree);
       // Regression Function
       float* Solve(float *_x, float *_y, int L);

       // ALGLIB Cartesian Inputs
       real_1d_array x;
       real_1d_array y;

       // ALGLIB Polynomial Degree
       ae_int_t m;
       // ALGLIB Error Reporting
    ae_int_t info;
       // ALGLIB Goodness of Fit Report
    polynomialfitreport rep;
       // ALGLIB Barycentric Polynomial
    barycentricinterpolant p;
       // ALGLIB Polynomial Coefficients
       real_1d_array polyco;

       // Output Pointer
       float *curve;
};


#endif
```

Polyfit.cpp

```
#include <Polyfit\Polyfit.h>
#include <math.h>

void Polyfit::Init(int degree)
{
       m = degree + 1;
}

float* Polyfit::Solve(float *_x, float *_y, int L)
{
       // Set ALGLIB input sizes
       x.setlength(L);
       y.setlength(L);

       // Transfer Cartesian co-ordinates into ALGLIB inputs
       for (int i=0; i<L; i++)
       {
              x[i] = _x[i];
              y[i] = _y[i];
       }
```

```
	// Do Regression
	polynomialfit(x, y, m, info, p, rep);

	// Convert barycentric polynomial to coefficients
	polynomialbar2pow(p, polyco);

	// Define output pointer
	curve = new float[L];

	// Populate output pointer with Y values of regression plot
	for (int i=0; i<L; i++)
	{
		curve[i] = 0.0;

		for (int j=0; j<m; j++)
		{
			curve[i] += float(polyco[j] * pow(x[i], j));
		}
	}

	return curve;
}
```

V.3.5 EQ

EQ.h

```
#ifndef __mbf_eq__
#define __mbf_eq__

class EQ
{

public:
        // Initialisation Function
        void Init(int filterType, int _SR);
        // Coefficient Calculation Function
        void setCoefficients(float F, float G, float Q);
        // EQ Function
        float Do(float in, int channel);

        // Filter Type Identifiers
        enum
        {
                LowShelf=0,
                Parametric,
                HighShelf
        };

        // Stereo Channel Identifiers
        enum
        {
                L=0,
                R
        };

        // Local Filter Type Identifier
        int filterType;
        // Sample Rate
        int SR;
        // Sample Delay
        int n;
        // Delayed Sample Pointers
        float **x, **y;

        // Intermediate Variables
        float A, w0, cosw0, sinw0, alpha, sq;
        // EQ Coefficients
        float b0, b1, b2, a0, a1, a2;

};

#endif
```

EQ.cpp

```
#include <Bristow-Johnson/EQ.h>
#include <math.h>

double pi = 3.14159265358979323846264338327950;

void EQ::Init(int _filterType, int _SR)
{
        // Copy filter type and sample rate to local variables
        filterType = _filterType;
        SR = _SR;

        // Maximum sample delay
        n = 2;
```

```
       // Define delayed sample pointers - 3 stereo samples - and set to 0

       x = new float*[3];
       y = new float*[3];

       for (int i=0; i<3; i++)
       {
              x[i] = new float[2];
              y[i] = new float[2];

              for (int j=0; j<2; j++)
              {
                     x[i][j] = 0.0;
                     y[i][j] = 0.0;
              }
       }
}

void EQ::setCoefficients(float F, float G, float Q)
{
       // Calculate intermediate variables

       A = pow(10, G/40);

       w0 = float(2 * pi * F / SR);

       cosw0 = cos(w0);
       sinw0 = sin(w0);

       if(filterType == Parametric)
              alpha = sinw0 / (2 * Q);
       else
              alpha = sinw0/2 * sqrt( (A + 1/A)*(1/Q - 1) + 2 );

       sq =  2*sqrt(A)*alpha;

       // Calculate EQ coefficients for chosen filter type

       switch (filterType)
       {
       case LowShelf:

              b0 =    A*( (A+1) - (A-1)*cosw0 + sq );
        b1 =  2*A*( (A-1) - (A+1)*cosw0      );
        b2 =    A*( (A+1) - (A-1)*cosw0 - sq );
        a0 =        (A+1) + (A-1)*cosw0 + sq  ;
        a1 =   -2*( (A-1) + (A+1)*cosw0      );
        a2 =        (A+1) + (A-1)*cosw0 - sq  ;

              break;

       case HighShelf:

               b0 =    A*( (A+1) + (A-1)*cosw0 + sq );
         b1 = -2*A*( (A-1) + (A+1)*cosw0      );
         b2 =    A*( (A+1) + (A-1)*cosw0 - sq );
         a0 =        (A+1) - (A-1)*cosw0 + sq  ;
         a1 =    2*( (A-1) - (A+1)*cosw0      );
         a2 =        (A+1) - (A-1)*cosw0 - sq  ;

               break;

       case Parametric:

               b0 =   1 + alpha*A;
         b1 =  -2 * cosw0  ;
         b2 =   1 - alpha*A;
         a0 =   1 + alpha/A;
         a1 =  -2 * cosw0  ;
         a2 =   1 - alpha/A;

               break;
       }
}

float EQ::Do(float in, int channel)
{
       // Copy latest input sample to delayed samles pointer
       x[n][channel] = in;

       // Calculate latest output sample
       y[n][channel] = (b0/a0)*x[n][channel] + (b1/a0)*x[n-1][channel] + (b2/a0)*x[n-2][channel]
                                                                       - (a1/a0)*y[n-1][channel] -
(a2/a0)*y[n-2][channel];

       // Shift samples through delayed samples pointer
       for (int i=1; i<=n; i++)
       {
              x[i-1][channel] = x[i][channel];
              y[i-1][channel] = y[i][channel];
       }

       return y[n][channel];
}
```

V.3.6 Compression

Compression.h

```
#include <math.h>

class Compression
{
public:
        // Initalisation Function
        void Init(int _type, int _size, int _SR);
        // Parameters Function
        void setParameters(float thold, float slope, float att, float rel);
        // Audio Input Function
        void setAudio(float *_L, float *_R, float _log_fft_band);
        // Compression Function
        float Do(int i);

        // Compression type identifiers
        enum
        {
                Compress=0,
                Compand
        };

        // Type, pointer size and sample rate
        int type, size, SR;

        // Sample pointers
        float *L, *R;

        // Lookahead time and RMS calculation size
        int look, width;
        // RMS running total and container
        float summ, rms;

        // Local logarithmic FFT band level
        float log_fft_band;
        //Linear FFT Band level
        float fft_band;

        // Blended RMS and FFT level
        float level;

        // Linear envelope level
        float env;
        // Logarithmic envelope level
        float log_env;
        // Envelope ballistics coefficent
        float theta;

        // Compression parameters
        float log_thold, slope, att, rel;
        // Gain reduction
        float GR;

};
```

Compression.cpp

```
#include <Compression/Compression.h>

void Compression::Init(int _type, int _size, int _SR)
{
       // Copy compression type, pointer size and sample rate to local variables
       type = _type;
       size = _size;
       SR = _SR;

       // Convert lookahead variables from ms to samples
       look = int(3 * (SR * 1e-3));
       width = int(1 * (SR * 1e-3));

       // Set envelope to 0
       env = 0.0;
}

void Compression::setAudio(float *_L, float *_R, float _log_fft_band)
{
       // Copy stereo input and current FFT band level to local variables
       L = _L;
       R = _R;
       log_fft_band = _log_fft_band;
}

void Compression::setParameters(float _thold, float _slope, float _att, float _rel)
{
       // Copy and convert parameters to local variables
       log_thold = _thold;
       slope = _slope * float(1e-2);
       att = (_att == 0.0) ? (0.0f) : exp(-1.0f / (SR * (_att/10.0f) * 1e-3f));
       rel = (_rel == 0.0) ? (0.0f) : exp(-1.0f / (SR * (_rel/10.0f) * 1e-3f));
}
float Compression::Do(int i)
{
       // Calculate RMS
       summ = 0.0;

       for (int j=i+look; j<i+look+width; j++)
           summ += (0.5f*L[j] + 0.5f*R[j]) * (0.5f*L[j] + 0.5f*R[j]);

       rms = sqrt(summ / float(width));

       // Convert FFT band level to linear
       fft_band = pow(10, log_fft_band / 20);

       // Calculate blended level
       level = (0.8f * fft_band) + (0.2f * rms);

       if (type == Compress)
       {
           // Calculate ballistics coefficient
           theta = level > env ? att : rel;

           // Calculate envelope level
           env = (1.0f - theta) * level + theta * env;

           // Convert envelope to dB
           log_env = 20*log10(env);

           // Check envelope value is real number
           if (env != env)
           {
               GR = 0.0;
              }

              // Calculate gain reduction
              GR = 0.0;
              if (log_env > log_thold)
               GR -= (log_env - log_thold) * slope;
        }

        if (type == Compand)
        {
                theta = level < env ? att : rel;

                env = (1.0f - theta) * level + theta * env;
                log_env = 20*log10(env);
                if (env != env)
                {
                        GR = 0.0;
                }
                GR = 0.0;
                if (log_env < log_thold)
                        GR += (log_thold - log_env) * slope;
        }

        return GR;
}
```

FBand.h

```
// Object Includes
#include <Bristow-Johnson/EQ.h>
#include <Compression/Compression.h>
#include <math.h>

class FBand
{
public:
	// Initialisation Function
	void Init(int _size, int SR, int _filterType, int _compType, float _fft_frequency, float
	_fft_width);
	// Parameters Function
	void setParams(float _f_base, float _f_ratio, float _g_base, float _g_ratio);
	// Effects Function
	float* Do(float *L, float *R, float *fft, bool EQio, bool DRCio);

	// EQ and Compression Objects
	EQ EQ;
	Compression Comp;

	// Pointer size, sample rate, compression type and filter type
	int size, SR, compType, filterType;

	// Analysis band start point and size
	int fft_frequency;
	int fft_width;

	// FFT band level
	float log_fft_band;

	// Gain reduction
	float GR;
	// FBand parameters
	float f_base, f_ratio, g_base, g_ratio;
	// EQ parmeters
	float F, G, Q;

	// EQ cascade and output pointers
	float *Lcas, *Rcas, *Lout, *Rout;

	// Output pointer
	float *out;


};
```

FBand.cpp

```
#include <FBand/FBand.h>

void FBand::Init(int _size, int SR, int _filterType, int _compType, float _fft_frequency, float
_fft_width)
{
	// Copy pointer size, filter type and compression type to local variables
	size = _size;
	filterType= _filterType;
	compType = _compType;

	// Initialise EQ and Compression
```

```
        EQ.Init(filterType, SR);
        Q = (filterType == EQ::LowShelf) ? 1.0f : 0.5f;
        Comp.Init(compType, size, SR);

        // Copy analysis band start and size to local variables
        fft_frequency = int(_fft_frequency);
        fft_width = int(_fft_width);
        log_fft_band = 0.0;

        // Define output pointers
        Lcas = new float[size];
        Rcas = new float[size];
        Lout = new float[size];
        Rout = new float[size];
        out = new float[size*2];
}

void FBand::setParams(float _f_base, float _f_ratio, float _g_base, float _g_ratio)
{
        // Set FBand parameters, inverting when necessary
        f_base = _f_base;
        f_ratio = (compType == 0.0) ? _f_ratio * -1 : _f_ratio;
        g_base = _g_base;
        g_ratio = (compType == 0.0) ? _g_ratio * -1 : _g_ratio;
}

float* FBand::Do(float *L, float *R, float *fft, bool EQio, bool DRCio)
{
        // Calculate FFT band level
        log_fft_band = 0.0;
        for (int i=fft_frequency; i<fft_frequency + fft_width; i++)
                log_fft_band += fft[i];
        log_fft_band /= fft_width;

        // Copy stereo audio to Compression object
        Comp.setAudio(L, R, log_fft_band);

        // Loop through audio
        for (int i=0; i < size; i++)
        {
                // Gain reduction for current sample

                GR = Comp.Do(i);

                // Update every 3 samples - EQ has 3 sample delay

                if (i % 3 == 0 && DRCio)
                {
                        // Calculate cutoff and gain
                        F = f_base + GR * f_ratio;
                        G = g_base + GR * g_ratio;

                        // Calculate EQ coefficients
                        EQ.setCoefficients(F, G, Q);
                }

                // Apply EQ if enabled, cascading for low shelf

                if (EQio && filterType == EQ::LowShelf)
                {
                        Lcas[i] = EQ.Do(L[i], EQ::L);
                        Rcas[i] = EQ.Do(R[i], EQ::R);

                        Lout[i] = EQ.Do(Lcas[i], EQ::L);
                        Rout[i] = EQ.Do(Rcas[i], EQ::R);
                }
                else if (EQio)
                {
                        Lout[i] = EQ.Do(L[i], EQ::L);
                        Rout[i] = EQ.Do(R[i], EQ::R);
                }
                else
                {
                        Lout[i] = L[i];
                        Rout[i] = R[i];
                }
        }

        // Populate interleaved output pointer
        int j=0;
        for (int i=0; i<size*2; i+=2)
        {
                out[i] = Lout[j];
                out[i+1] = Rout[j];
                j++;
        }

        return out;
}
```

V.3.8 WriteToFile

WriteToFile.h

```
#ifndef __mbf_wtf__
#define __mbf_wtf__

#include <iostream>
#include <fstream>
#include <string>

class WriteToFile
{

public:
       WriteToFile(char *filename)
              :filename(filename)
       {}

       void Write(std::string input);

       char *filename;

};

#endif
```

WriteToFile.cpp

```
#include "WriteToFile.h"

void WriteToFile::Write(std::string input)
{
  std::ofstream file;
  file.open(filename, std::ios::app);
  file << input << "\n";
  file.close();
}
```

SECTION VI.

REFERENCES AND BIBLIOGRAPHY

References

ALGLIB, 2012, ALGLIB, Available at: http://www.alglib.net/ [Accessed 07/04/2012]

Bristow-Johnson, R. (Unknown) *Cookbook formulae for audio EQ biquad filter coefficients*, Available at: http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt [Accessed 14/01/2012]

Cooley, J.W. and Tukey, J. W. (1965) *An Algorithm for the Machine Calculation of Complex Fourier Series*, Mathematics of Compuation, 19, (90), pp.297-301, Available at: http://www.jstor.org/stable/2003354 [Accessed 14/01/2012]

Croft, A. and Davidson R. (2008) *Mathematics for Engineers*, Pearson Education, Essex, p. 1139

Croft, A. and Davidson R. (2008) *Mathematics for Engineers*, Pearson Education, Essex, p. 1149

Croft, A. and Davidson R. (2008) *Mathematics for Engineers*, Pearson Education, Essex, p. 1151

FFTW User Manual (Unknown), *Complex One-Dimensional DFTs*, Available at: http://www.fftw.org/doc/Complex-One_002dDimensional-DFTs.html#Complex-One_002dDimensional-DFTs [Accessed 15/01/2012]

Fisher, M. (2010) *Optimal recording and mixing technqiues for a stereo piano recording in the context of a modern popular music production*, University of Huddersfield

Fourier, J (1808) *Mémoire sur la propagation de la chaleur dans les corps solides* p.215-221, Presented 21st December 1807 at l'Institut national - Nouveau Bulletin des sciences par la Société philomatique de Paris, Available at: http://mathdoc.emath.fr/cgi-bin/oetoc?id=OE_FOURIER__2 [Accessed 14/01/2012]

Giordano, A. A. and Hsu, F. M. (1985) *Least Square Estimation with Applications to Digital Signal Processing*, John Wiley & Sons Inc., USA, p. 8

Huff, D. (1954) Least Squares Fitting, in Huff, D. *How to Lie with Statistics*, [Online] Available at: http://www.physics.csbsju.edu/stats/least_squares.html [Accessed 15/01/2012]

Jackson, L. B. (1996) *Digital Filters and Signal Processing,* Kluwer Academic Publishers, London, p. 189

Jackson, L. B. (1996) *Digital Filters and Signal Processing,* Kluwer Academic Publishers, London, p. 213

Mellor, D (1995) *EQ: How & When To Use It*, Sound on Sound, March 1995, Available at: http://www.soundonsound.com/sos/1995_articles/mar95/eq.html [Accessed 14/01/2012]

OpenGL.org, Available at: http://www.opengl.org/ [Accessed 15/01/2012]

Orion-Smith, J. (2007) "The Simplest Lowpass Filter", in Orion-Smith, J. *Introduction to Digital Filters with Audio Applications*, [Online], Available at: https://ccrma.stanford.edu/~jos/filters/Simplest_Lowpass_Filter_I.html [Accessed 14/01/2012]

Preve, F (2008) September 24th 2008, Tutorial: How to use compression, Available at: http://www.beatportal.com/feed/item/tutorial-how-to-use-compression/ [Accessed 14/01/2012]

Price, S (2008) *Side-chain Compression in Reason*, Sound on Sound, September 2008, Available at: http://www.soundonsound.com/sos/sep08/articles/reasontech_0908.htm [Accessed 14/01/2012]

Robjohns, H (1999) *How & When To Use Mix Compression*, Sound on Sound, June 1999, Available at: http://www.soundonsound.com/sos/jun99/articles/mixcomp.htm [Accessed 14/01/2012]

Savitch, W. (2010) *Absolute C++*, 4th Ed., Pearson Education Inc., US, p.169

Senior, M. (2008) *Piano Recording: All You Need To Know To Capture A Great Acoustic Piano Sound*, Sound on Sound, January 2008, Available at: http://www.soundonsound.com/sos/jan08/articles/pianorecording_0108.htm [Accessed 15/01/2012]

Senior, M (2009) *Techniques for Vocal De-Essing*, Sound on Sound, May 2009, Available at: http://www.soundonsound.com/sos/may09/articles/deessing.htm [Accessed 14/01/2012]
Senior, M (2011) *Listen and Learn: Analysing Commercial Mixes*, Sound on Sound, April 2011, Available at: http://www.soundonsound.com/sos/apr11/articles/listen-and-learn.htm [Accessed 14/01/2012]

Smith, S. W. (1997) *The Engineers Guide to Digital Signal Processing*, California Technical Publications, LA., p. 150

Smith, S. W. (1997) “Chapter 3: ADC and DAC”, From Smith, S.W., *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publications, Available at: http://www.dspguide.com/ch3/2.htm [Accesses 14/01/2012]

T.C. Electronic Dynamic EQ Overview, Available at: http://www.tcelectronic.com/dynamiceq.asp [Accessed 14/01/2012]

Voxengo GlissEQ Overview, Available at: http://www.voxengo.com/product/glisseq/ [Accessed 14/01/2012]

Walden, J (2007) *Multiband Compression in Cubase*, Sound on Sound, April 2007, Available at: http://www.soundonsound.com/sos/apr07/articles/cubasetech_0407.htm [Accessed 14/01/2012]

Walker, M (2007) *Mixing on Headphones*, Sound on Sound, January 2007, Available at: http://www.soundonsound.com/sos/jan07/articles/mixingheadphones.htm [Accessed 14/01/2012]

Waves Audio Ltd. (2012) “Bass Rider”, Available at: http://www.waves.com/content.aspx?id=11856 [Accessed 09/05/2012]

Weisberg, S. (1985) “Simple Linear Regression”, in Weisberg, S. *Applied Linear Regression*, John Wiley & Sons Inc, USA

Weisstein, Eric W. (1999) "Homogeneous Coordinates." From MathWorld--A Wolfram Web Resource, Available at: http://mathworld.wolfram.com/HomogeneousCoordinates.html [Accessed 15/01/2012]

White, P (2009) *Brainwork BX_dynEQ*, Sound on Sound, July 2009, Available at: http://www.soundonsound.com/sos/jul09/articles/bxdynamiceq.htm [Accessed 14/01/2012]

Zölzer, U. and Dutilleux, P. (2002) *DAFX: Digital Audio Effects*, John Wiley & Sons, Chichester, p.95

Anon. (2011) 3rd Party JUCE Applications, [Online], Available at: http://www.rawmaterialsoftware.com/wiki/index.php/3rd-party_JUCE_Applications, [Accessed on 15/09/11]

Anon. (2011) Raw Material Software – The JUCE Library, [Online], Available at: http://www.rawmaterialsoftware.com/juce.php, [Accessed on 15/09/11]

Anon. (2011) Steinberg VST Plugin Zone, [Online], Available at: http://www.steinberg.net/en/products/partner_products/pluginzone.html, [Accessed on 15/09/11]

Bibliography

Collings, S. N. et al. (1977) “Simple Linear Regression”, in Collings, S. N et al. *Fundamentals of Statistical Inference - Unit 14 – Linear Models: Introduction and Least Squares Estimations*, The Open University Press, Milton Keynes

Croft, A. and Davidson R. (2008) “An introduction to Fourier series and the Fourier transform”, in Croft, A and Davidson, R, *Mathematics for Engineers*, 3rd ed., Pearson Education, Essex

Giordano, A. A. and Hsu, F. M. (1985) “Introduction”, in Giordano, A. A. and Hsu, F. M. *Least Square Estimation with Applications to Digital Signal Processing*, John Wiley & Sons Inc., USA

Jackson, L. B. (1996)”Discrete Fourier Transform”, in Jackson, L. B. *Digital Filters and Signal Processing,* 3rd ed., Kluwer Academic Publishers, London

Smith, S. W. (1997) “The Discrete Fourier Transform”, in Smith, S. W. *The Engineers Guide to Digital Signal Processing*, California Technical Publications, LA.

Zölzer, U. and Dutilleux, P. (2002) “Nonlinear Processing” in Zölzer, U. (ed.), *DAFX: Digital Audio Effects*, John Wiley & Sons, Chichester

Anon. (2005) *The Frequencies of Music*, Available at: http://www.psbspeakers.com/articles/The-Frequencies-of-Music [Accessed 14/01/2012]