

C EXTENDED ESSAY

Implementing a parametric EQ plug-in in C++ using the multi-platform VST specification

JONAS EKEROOT

SCHOOL OF MUSIC
Audio Technology

Supervisor: Jan Berg

Implementing a parametric EQ plug-in in C++ using the multi-platform VST specification

Jonas Ekeroot

Division of Sound Recording
School of Music in Piteå
Luleå University of Technology

April 23, 2003

Abstract

As the processing power of desktop computer systems increase by every year, more and more real-time audio signal processing is performed on such systems. What used to be done in external effects units, e.g. adding reverb, can now be accomplished within the computer system using signal processing code modules – *plug-ins*. This thesis describes the development of a peak/notch parametric EQ VST plug-in. First a prototype was made in the graphical audio programming environment Max/MSP on MacOS, and then a C++ implementation was made using the VST Software Development Kit. The C++ source code was compiled on both Windows and MacOS, resulting in versions of the plug-in that can be used in any VST host application on Windows and MacOS respectively. Writing a plug-in relieves the programmer of the burden to deal directly with audio interface details and graphical user interface specifics, since this is taken care of by the host application. It can thus be an interesting way to start developing audio DSP algorithms, since the host application also provides the opportunity to listen to and measure the performance of the implemented plug-in algorithm.

Keywords

Audio, plug-in, C++, parametric EQ, digital filter, FIR, IIR, biquad, Max/MSP, DSP, API, real-time

Contents

1	Introduction	8
1.1	Thesis aim and limitations	8
1.2	Organization of the thesis	8
2	Background	10
2.1	What is a plug-in?	10
2.2	Types of plug-ins	10
2.3	Real-time and non-real-time plug-ins	12
2.4	Digital audio effects	13
2.5	Review of basic DSP theory	13
2.5.1	Signals and systems	13
2.5.2	Quantization	14
2.5.3	Digital filters – FIR and IIR	15
2.5.4	Second-order IIR filter	16
2.5.5	Peak/notch parametric EQ	18
3	Prototype	20
3.1	Max/MSP	20
3.2	Algorithm testing	21
3.3	Pluggo	23
4	Implementation	24
4.1	The VSTSDK C++ framework	24
4.2	The AudioEffect and AudioEffectX classes	25
4.3	The Biquad class	26
5	Results	29
5.1	Aural assessment	30
5.2	MATLAB measurements	30
5.3	Default plug-in GUI	33
5.4	Windows host applications	33
5.4.1	WaveLab	33
5.4.2	AudioMulch	34
5.4.3	Audacity	35
5.5	MacOS host applications	35
5.5.1	Cubase VST	36
5.5.2	Max/MSP	36
5.6	Parameter adjustments	36
5.7	Summary of results	37

6	Discussion	38
6.1	Max/MSP and similar applications	38
6.2	The VSTSDK C++ source code	38
6.3	Optimization	39
6.4	The frequency parameter limits	39
6.5	Cross-platform GUI	40
6.6	Conclusions	40

List of Figures

1	Block diagram representation of a discrete-time system	14
2	Feedforward FIR filter	15
3	Feedback IIR filter	16
4	Second-order IIR system – biquad	17
5	A simple Max/MSP patch	20
6	Patch to calculate biquad coefficients	22
7	Patch to listen to filtered white noise	22
8	A <i>pluggo</i> version of the parametric EQ VST plug-in	23
9	Plug-in API and audio API – a hierarchical view	24
10	Class hierarchy for the Biquad VST plug-in	26
11	The original impulse and the impulse response	31
12	The magnitude and the phase of the frequency response	32
13	Parametric EQ VST plug-in in WaveLab	34
14	Parametric EQ VST plug-in in AudioMulch	34
15	Patch in AudioMulch showing a mono plug-in	35
16	Parametric EQ VST plug-in in Audacity	35
17	Parametric EQ VST plug-in in Cubase	36
18	Parametric EQ VST plug-in in Max/MSP	36

List of Abbreviations

API	Application Programming Interface
AS	AudioSuite
CPU	Central Processing Unit
DFT	Discrete Fourier Transform
DLL	Dynamic Link Library
DSP	Digital Signal Processing
EQ	Equalizer
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
GUI	Graphical User Interface
HTDM	Host Time Division Multiplexing
HW	Hardware
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response
IRCAM	Institut de Recherche et Coordination Acoustique/Musique
I/O	Input/Output
LADSPA	Linux Audio Developer's Simple Plug-in API
MAS	MOTU Audio System
MOTU	Mark of the Unicorn
MSP	Max Signal Processing
RTAS	Real-Time AudioSuite
SGI	Silicon Graphics Incorporated
TDM	Time Division Multiplexing
VSTSDK	Virtual Studio Technology Software Development Kit

1 Introduction

In audio production of today computers play a central role. Computer systems, with appropriate hardware and software applications, are being used for recording, editing, mixing, mastering and streaming on the Internet. The use of various types of audio signal processing like EQ, delay, reverb and dynamic compression that used to be performed by dedicated outboard equipment, is to a large extent carried out within the computer system nowadays. In such a system, the signal processing is not done by a specialized DSP (Digital Signal Processing) chip but by the general-purpose CPU (Central Processing Unit) of the specific system. Typical examples for current desktop computers running commercial operating systems include Motorola or IBM CPUs used by MacOS, and Intel or AMD CPUs used by Windows. The signal processing parts of audio software applications are often placed in separate code modules – called *plug-ins* – that are loaded into the basic application to extend its functionality.

1.1 Thesis aim and limitations

The work being done during the writing of this thesis was focused on plug-in development. Applied basic audio DSP theory in the context of computer audio programming was treated, as was the development process of an audio plug-in in a format that fits in many audio software applications of today on both Windows and MacOS – the *Steinberg VST (Virtual Studio Technology)* format. Parts of the underlying structure of a VST audio plug-in written in C++ was also examined.

The main aim of investigating VST plug-in development was subdivided into two parts – the development of a prototype in a graphical oriented development environment, and the implementation of the final plug-in in C++. The subject was further narrowed down to implement a specific digital audio effects algorithm - a peak/notch parametric EQ plug-in with adjustable frequency, gain and Q values.

Analogue filter prototype design was not part of the work conducted for this thesis, nor was the conversion of such prototypes into digital filters. Instead, descriptions of filters directly in the digital domain were used. Some mathematical equations are stated in the thesis in order to explain the implemented DSP algorithm, but the focus is on presenting conceptual ideas and not on going through rigorous mathematical proofs and derivations.

1.2 Organization of the thesis

Chapter 2 gives a general description of audio plug-ins, and a review of fundamental DSP theory with an emphasis on digital filters. This provides a background for the discussion in the subsequent chapters. Thereafter, the

details of the prototype and the implementation are described in chapter 3 and chapter 4. The results of testing the compiled plug-in in a number of different host applications on both Windows and MacOS are presented in chapter 5. An analysis of the frequency response of the plug-in is performed, and the differences in the default graphical user interfaces (GUIs) are shown. The last chapter contains a discussion of some of the topics treated in earlier chapters, and also gives suggestions for future improvements that could be made to the parametric EQ VST plug-in.

2 Background

This section provides a brief overview of some background concepts that forms the basis for the topics that are treated in subsequent chapters. First, a general definition of an audio plug-in is given, followed by a listing of different plug-in types in common use. Thereafter, some timing related issues as they apply to plug-ins are touched upon. The section ends with a review of fundamental DSP theory with an emphasis on digital filters.

2.1 What is a plug-in?

An audio plug-in is a software component that can not execute within a computer system on its own, but needs a *host application* that makes use of the audio signal processing that the plug-in supplies. In this way the host application provides the basic functionality like audio input and output (I/O) through an audio interface (i.e. a sound card), audio file writing and reading to and from a hard disk, and waveform editing. By placing the file containing the compiled plug-in code in a directory where the host application can find it, the plug-in becomes available from inside the host application. In other words, the plug-in extends the functionality of the host application by supplying specialized audio signal processing. The host does not need to know anything about the DSP algorithm inside the plug-in, and the plug-in does not need to have any knowledge of the audio I/O handling or the GUI of the host. The following quote is from the VST specification [1]:

“From the host application’s point of view, a VST plug-in is a black box with an arbitrary number of inputs, outputs and associated parameters. The host needs no knowledge of the plug-in process to be able to use it.”

Thus, the only thing a host application and a plug-in need to agree about is the way that they communicate digital audio samples and parameter values between themselves. The host continuously supplies the plug-in with chunks of audio samples, *input buffers*, which the plug-in processes using its DSP algorithm and returns back as *output buffers* to the host for playback.

2.2 Types of plug-ins

The way that a plug-in and a host communicate differs among different types of plug-in specifications. There are a number of different formats available on the market, and the topic of this thesis is the plug-in format introduced in 1996 by Steinberg Media Technologies AS, in their Virtual Studio Technology line of products. From the Steinberg web site the *Virtual Studio Technology Plug-In Specification 2.0 Software Development Kit*, or VSTSDK for short, can be freely downloaded [1]. The VSTSDK consists

of a C++ framework, and source code for MacOS, Windows, BeOS and SGI/MOTIF is provided. This means that VST plug-ins are easily developed for platform independence, or at least to have multi-platform support.

The free availability of the VSTSDK has resulted in a large number of plug-ins by third-party developers, and the VST plug-in format is currently one of the major formats supported by host applications. Other common plug-in formats include:

DirectX by Microsoft is really more than a plug-in format. It is a set of application programming interfaces (APIs) for multimedia application development. DirectX plug-ins only work in host applications on Windows. A C++ SDK for writing DirectX applications, including plug-ins, is available from the Microsoft web site¹.

MAS is a plug-in format by Mark of the Unicorn (MOTU). MAS stands for MOTU Audio System. These plug-ins can only be used on MacOS. By signing an agreement with MOTU, third-party developers can gain access to a C++ SDK².

TDM is a plug-in format by Digidesign. TDM stands for Time Division Multiplexing. This type of plug-ins requires the presence of a dedicated DSP chip, and thus differs from the other types listed here, which use the general CPU of the computer for what is called *native* or *host-based* processing. To develop TDM plug-ins, third-party developers have to sign an agreement with Digidesign.

AS/RTAS/HTDM are plug-in formats by Digidesign for native processing. AS stands for AudioSuite and RTAS stands for Real-Time AudioSuite. RTAS allows the resulting sound of the plug-in process to be heard as the calculations proceed, in contrast to the AS plug-ins, where a whole file or audio selection is processed in its entirety, before the results can be auditioned. HTDM stands for Host Time Division Multiplexing and represent a hybrid of the TDM and the RTAS plug-in formats, that allow for host-based processing. To develop AS/RTAS/HTDM plug-ins, third-party developers have to sign an agreement with Digidesign.

LADSPA is a plug-in format for the Linux operating system. LADSPA stands for Linux Audio Developer's Simple Plug-in API, and is released under LGPL (Less-GNU Public License). A C/C++ SDK is available for download³.

¹http://download.microsoft.com/download/whistler/dx/8.1/w982kmexp/en-us/DX81SDK_FULL.exe

²<http://www.motu.com/english/other/developer/index.html>

³http://www.ladspa.org/ladspa_sdk

Audio Units by Apple is a plug-in format that is part of MacOS X Core Audio⁴. An SDK and developer tools are available from Apple⁵. Emagic offers a VST-To-Audio Units porting library to facilitate the porting of existing VST plug-ins to Audio Units plug-ins⁶.

2.3 Real-time and non-real-time plug-ins

Arfib [2] described a *real-time* audio DSP process as one in which the sound could be listened to at the same time as the calculations proceeded. For this to be possible, the mean processing time for one sample of a mono signal must be less than the sampling period of the digital audio signal. Non-real-time plug-ins on the other hand, process the whole of an audio selection before the result can be listened to, and thus have more relaxed timing constraints. Usually a short segment of an audio file can be previewed in order to make adjustments of the plug-in parameters. When the parameters are set, the plug-in applies its process to all of the samples in the file and writes the results to a new audio file, which can then be listened to.

The processing power of current desktop computers quite easily handles real-time plug-ins. Using the VST specification, plug-ins can be developed to provide both real-time and non-real-time, or off-line, usage. The parametric EQ plug-in developed in this thesis is a real-time plug-in.

Another timing related issue is worth mentioning in this context. If a host application reads audio samples from the audio interface input and without further processing of the samples sends them back to the audio interface output, this task takes some time to accomplish and there will be a noticeable delay between the input and the output. This delay is often referred to as *latency*, and was investigated by MacMillan *et al* [3] in the context of desktop operating systems. The latency depends on many parameters such as the CPU speed, the amount of memory, the type of hard disk, the type of sound card, the operating system, the API used to develop the audio software and the type of sound card drivers. Measurements presented in [3] show latency values for some of the best systems of under 5 milliseconds, but values of several hundred milliseconds were also found. So, the fact that a plug-in operates in a real-time mode does not mean that it turns the computer system into an effects processor with a real-time response comparable with a dedicated hardware effects unit. The latency will give a noticeable throughput delay.

⁴<http://www.apple.com/macosx/technologies/audio.html>

⁵<http://developer.apple.com/audio/>

⁶<http://www.emagic.de/support/osx/developer.php?lang=EN>

2.4 Digital audio effects

In this thesis, no general description of algorithms for different digital audio effects is given. This topic has been presented by other authors. Zölzer [4] gave a comprehensive overview of the field, covering filters, delays, modulators, dynamics processing and spatial effects, with software implementations using MATLAB⁷. Bendiksen [5] systematically described digital audio effects according to a classification into amplitude modulation effects, frequency modulation effects, spatial effects, filtering and dynamics compression. He also used MATLAB for software implementation examples. Browning [6] investigated effects like delay, reverb, chorus, flange, distortion and filtering, using pseudo-code examples.

2.5 Review of basic DSP theory

To develop an audio plug-in requires knowledge not only of the programming language and the framework to be used for the implementation, but also of basic DSP theory. Original DSP algorithm development is an advanced topic that involves a deep knowledge of university level mathematics. This was out of the scope of this thesis.

2.5.1 Signals and systems

McClellan *et al* [7] gave an abstract definition of *signals* as patterns of variations that represent or encode information. Such patterns evolve in time to create time waveforms. An example of a *continuous-time signal* (analogue signal) is the varying output voltage from a microphone. This signal can be mathematically represented by a function x of a continuous variable t

$$x(t) \tag{2.1}$$

where t refers to time. By sampling the continuous-time signal at equally spaced time instants, a *discrete-time signal* (digital signal) is obtained

$$x[n] = x(nT_s) \tag{2.2}$$

where n is an integer, and T_s is the sampling period

$$T_s = \frac{1}{f_s} \tag{2.3}$$

where f_s is the sampling frequency. The signal $x[n]$ is a sequence of numbers indexed by the integer n . The numbers in $x[n]$ are the sampled values of $x(t)$ taken once every T_s seconds. In this thesis, the notational convention of enclosing the independent variable of a continuous-time function with

⁷<http://www.mathworks.com>

parentheses (), and enclosing the independent variable of a discrete-time function (sequence) with square brackets [], is used. The square bracket notation for sequences is in analogy with the syntax for arrays of numbers in C++. An array named `x` consisting of four digital audio samples stored as floats would be declared as

```
float x[4];
```

and the individual samples could be accessed as `x[0]`, `x[1]`, `x[2]` and `x[3]`.

In a very general sense, a *system* operates on signals to produce new signals [7]. Using this definition, equation (2.2) could be viewed as a system where the input is a continuous-time signal and the output is a discrete-time signal. The system could be called a continuous-to-discrete (C-to-D) converter [7].

A *discrete-time system* takes an input signal $x[n]$ and produces a corresponding output signal $y[n]$. This can visually be represented by the block diagram in figure 1.

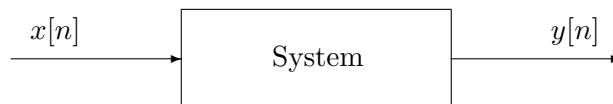


Figure 1: Block diagram representation of a discrete-time system.

2.5.2 Quantization

The actual hardware system for doing C-to-D conversion is an analogue-to-digital (A-to-D) converter. Due to real-world problems such as jitter and the quantization of the sample values to a finite resolution, the A-to-D converter is only an approximation of the perfect sampling of the C-to-D converter.

The quantization resolution of the sample values in $x[n]$ affects the audio quality of the signal. In VST plug-ins, audio samples are handled as 32-bit single precision floating-point numbers [1], according to the IEEE 754 specification, described by Patterson and Hennessy [8]. The sample values are normalized to range from -1.0 to $+1.0$. Thus, a sample value of 1.0 corresponds to 0 dBFS, a value of 0.5 corresponds to -6 dBFS, etc.

The use of 32-bit floating-point sample values in the range from -1.0 to $+1.0$ means that the internal resolution of a VST plug-in is higher than 16-bit integer resolution (audio CD standard). It also gives plenty of headroom to deal with overflow calculations in a controlled manner.

2.5.3 Digital filters – FIR and IIR

According to Roads [9], a committee of signal processing engineers once made the following definition of a digital filter:

“A digital filter is a computational process or algorithm by which a digital signal or sequence of numbers (acting as input) is transformed into a second sequence of numbers termed the output digital signal.”

By this definition every discrete-time system, as in figure 1, is a filter, regardless of what operation the filter performs on the input signal. The term *digital filter* is used in this thesis in a more specific sense, to describe systems that boost or attenuate regions of the frequency spectrum of a signal.

To make such a filter, the functionality of three basic building blocks is required:

- the delay of a sample value by one or several sample periods
- the scaling of a sample value by a gain factor
- the mixing (adding) of two or more sample values

Figure 2 shows a filter that delays a copy of the current sample of the input signal, scales the level of the delayed sample by a gain factor g and mixes the direct and the delayed signals. The filter has a feedforward path.

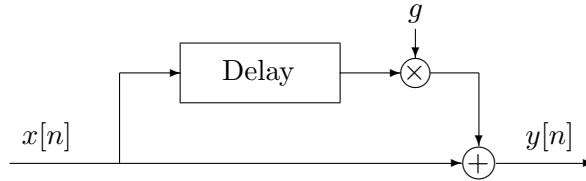


Figure 2: Delay the input and mix (feedforward).

A time-domain (or n -domain) formula for computing $y[n]$ based on the present input sample and past input or output samples, or both, is called a *difference equation* [7]. Given a delay of one sample period, the filter in figure 2 can be described by the difference equation

$$y[n] = x[n] + g x[n - 1] \quad (2.4)$$

where $x[n]$ is the current sample of the input signal and $x[n - 1]$ is the previous (delayed) sample of the input signal. This is called a first-order filter since the maximum delay used in equation (2.4) is one sample period.

A second-order filter has a maximum delay of two sample periods, and so on.

If the input signal to the filter described by equation (2.4) is an *impulse*

$$\delta[n] = \begin{cases} 1 & n = 0 \\ 0 & n \neq 0 \end{cases} \quad (2.5)$$

the output signal, or *impulse response*, will have a finite number of non-zero sample values – the original impulse will be at $y[0]$ followed by the delayed and scaled impulse at $y[1]$. This type of feedforward filter is called a *Finite Impulse Response* or *FIR* filter. Equation (2.4) thus describes a first-order FIR filter.

A filter that delays a copy of the current sample of the output signal, scales the level of the delayed sample by a gain factor g and mixes the direct and the delayed signals, is shown in figure 3.

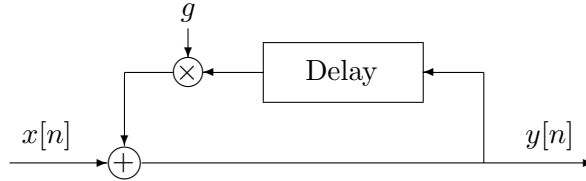


Figure 3: Delay the output and mix (feedback).

The filter has a feedback path and is described by the difference equation

$$y[n] = x[n] + g y[n - 1] \quad (2.6)$$

provided that the delay is one sample period. The impulse response of this filter will theoretically have an infinite number of non-zero sample values due to the feedback path, which always sends back some of the output signal. This type of feedback filter is therefore called an *Infinite Impulse Response* or *IIR* filter, and equation (2.6) thus describes a first-order IIR filter.

By inserting more delays and mixing the delayed and scaled samples with the current input sample, more complex filters can be made.

2.5.4 Second-order IIR filter

By combining the basic filter methods in figure 2 and figure 3 according to the block diagram in figure 4, a structure called a *second-order IIR filter*, Direct form I, is created, as described by Dattorro [10]. The difference

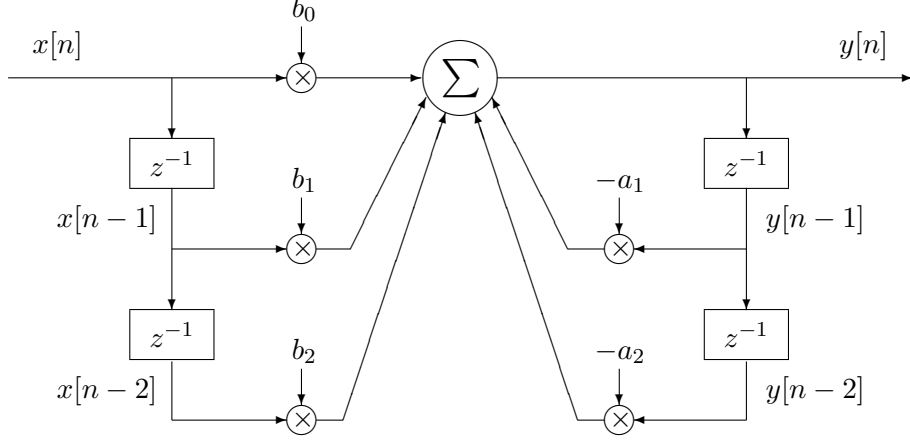


Figure 4: Second-order IIR system, Direct form I. z^{-1} means a delay of one sample period.

equation describing the second-order IIR filter in figure 4 is

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2] \quad (2.7)$$

where b_0 , b_1 , b_2 , a_1 and a_2 are called the *filter coefficients*. By carefully choosing the filter coefficients, many types of filter responses can be realized by equation (2.7). Second-order IIR filters are often used as building blocks to construct more complex filters [9].

Since equation (2.7) describes the operation of the filter in the time-domain, it can be implemented directly in C++ using arrays of samples as input and output signals.

The operation of a system can also be described in the z -domain by using the z -transform. This transform is used primarily as a mathematical analysis tool and not for the implementation of filters, which is usually done in the time-domain [7]. A thorough description of the z -transform goes out of the scope of this thesis. Just briefly though, the transfer function for the system described by the second-order IIR difference equation can be written

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1z^{-1} + b_2z^{-2}}{1 + a_1z^{-1} + a_2z^{-2}} \quad (2.8)$$

where b_0 , b_1 , b_2 , a_1 and a_2 are the same filter coefficients as in equation (2.7) [7]. The delay of one sample period in the time-domain corresponds to a multiplication by z^{-1} in the z -domain. Since the transfer

function $H(z)$ is a ratio of two second-degree (or quadratic) polynomials, a second-order IIR filter is also called a *biquad* [9].

2.5.5 Peak/notch parametric EQ

The plug-in developed during the work for this thesis should let the user adjust the frequency, gain and Q value of a peak/notch parametric EQ. The problem at hand then was to calculate the filter coefficients starting from the frequency, gain and Q values. Formulae by Robert Bristow-Johnson, Wave Mechanics⁸, and also a member of the review board of the Journal of the Audio Engineering Society, were used in the plug-in implementation. The formulae are freely available on the Internet, and were described in Bristow-Johnson [11]. They are based on an analogue filter prototype that has been mapped to a digital filter using the bilinear transform. Details about the derivation of the formulae can be found in Bristow-Johnson [12].

First some intermediate variables (partly using the original notation from [11]) were calculated

$$\begin{aligned} A &= \sqrt{10^{g/20}} = 10^{g/40} \\ \omega &= \frac{2\pi f}{f_s} \\ sn &= \sin \omega \\ cs &= \cos \omega \\ \alpha &= \frac{sn}{2Q} \end{aligned}$$

where g is the gain value in dB, f is the frequency in Hz, f_s is the sampling frequency in Hz and Q is the Q value, determining the bandwidth of the filter.

Bristow-Johnson [12] found that there is little agreement or consistent definition of equalizer bandwidth in the literature, and Harris and Brook-ing [13] wrote:

“There is a near universal source of confusion related to the bandwidth of a boost and of a cut operation in a filter.”

The bandwidth for a peak/notch filter was defined by White [14] as the difference between the frequencies where the filter response deviates from unity by 3 dB. In this case, no bandwidth can be defined for boosts or cuts of less than 3 dB. Another definition was made by Moorer [15]. For a boost or cut of 6 dB or more, he defined the bandwidth as the difference between the frequencies where the response is 3 dB below the peak or above the notch. When the boost or cut is less than 6 dB, the bandwidth is

⁸<http://www.wavemechanics.com>

defined as the difference between the midpoint gain frequencies, where the response is half the boost or cut amount, expressed in dB. The midpoint gain (i.e. $g/2$ dB) definition was adopted by Bristow-Johnson [11, 12] but applied consistently for all gain settings, i.e. also for boosts or cuts of 6 dB or more. For a peak/notch EQ he additionally identified $Q_{EE} = AQ$ to be the classic electrical engineering Q .

In the peak/notch parametric EQ plug-in implemented in this thesis, f , g and Q in the formulae for the intermediate variables above, were the design parameters that the user could adjust using sliders in the plug-in GUI, and the Bristow-Johnson midpoint gain (i.e. $g/2$ dB) definition of the Q value was used. After the intermediate variables had been calculated, the filter coefficients could be calculated as:

$$\begin{aligned} b_0 &= 1 + \alpha A & a_0 &= 1 + \frac{\alpha}{A} \\ b_1 &= -2cs & a_1 &= -2cs \\ b_2 &= 1 - \alpha A & a_2 &= 1 - \frac{\alpha}{A} \end{aligned}$$

The difference equation for the second-order IIR filter in (2.7) has a total of five filter coefficients – b_0 , b_1 , b_2 , a_1 and a_2 . The formulae by Bristow-Johnson gave six filter coefficients for the following difference equation with an additional a_0 coefficient

$$a_0 y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2] \quad (2.9)$$

If a_0 was normalized to be 1, the difference equation could be rewritten

$$\begin{aligned} y[n] &= \frac{b_0}{a_0} x[n] + \frac{b_1}{a_0} x[n-1] + \frac{b_2}{a_0} x[n-2] - \frac{a_1}{a_0} y[n-1] - \frac{a_2}{a_0} y[n-2] \\ &= \frac{1}{a_0} \left(b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2] \right) \end{aligned} \quad (2.10)$$

This was the difference equation that was implemented in the C++ plug-in code.

3 Prototype

The method for plug-in development investigated in this thesis comprised two steps – a prototype and an implementation. The prototype was used to gain familiarity with the DSP algorithm and the mathematical calculations necessary to achieve the desired result of realizing a parametric EQ. Within the prototype environment, the DSP algorithm could easily be applied to any audio file or generated test signal, e.g. white noise. Parameter adjustments could be made in real-time and the results of the processing could immediately be listened to. This gave practical experiences with the filter coefficient formulae, that would have taken considerably longer time to get if the C++ implementation had been started on directly.

3.1 Max/MSP

The prototype was made using *Max/MSP*⁹, which is a graphical programming environment for developing real-time MIDI and audio applications on MacOS, described by Puckette in [16]. Max, the basic application, provides the general programming and MIDI functionality, and MSP (Max Signal Processing) adds signal processing extensions to Max that allow the development of audio applications.

The fundamental concept in Max/MSP is the *patch*. A patch is a collection of on-screen boxes connected by virtual patch cords. The boxes represent different signal processing objects. An example of a simple patch is shown in figure 5. This patch shows an oscillator (**cycle~**) generating a

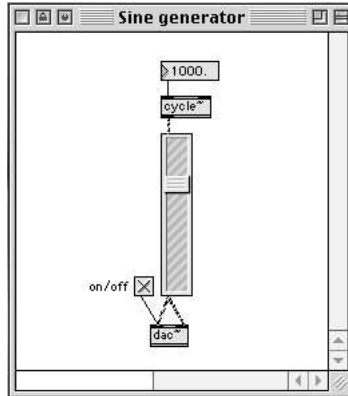


Figure 5: A simple Max/MSP patch.

1 kHz sine tone. The signal level is controlled by the fader object before the signal is sent to the **dac~** object (digital-to-analogue converter, i.e. the audio interface output).

⁹<http://www.cycling74.com/products/maxmsp.html>

Creating patches in Max/MSP is a kind of graphical object-oriented programming. The object boxes represent *unit generators* that are interconnected to form signal processing patches that generate or modify audio signals. Max/MSP is named after Max Mathews, one of the pioneering researchers and experimenters within the field of computer audio at AT&T Bell Laboratories in the 1960s. He originally developed the ideas of the unit generator programming concept [9].

In Max/MSP the signal processing boxes all have names ending in tilde, as in **cycle~** and **dac~**. In figure 5 there are also two other kinds of boxes. The fader is a kind of tilde box used to provide a graphical user interface control, and the number box (with the value of 1000) represents the type of non-tilde boxes that are used to handle non-audio signals, like numbers for calculations, etc.

The graphical object-oriented surface of Max/MSP is built on the traditional programming language C. A user of Max/MSP never interacts directly with the C layer other than if there is a need to develop new specialized unit generator boxes – a process called writing *external objects*.

3.2 Algorithm testing

Max/MSP provides an object called **biquad~**. It implements a second-order IIR difference equation and has control inputs for five filter coefficients in addition to audio signal input and output. For the plug-in prototype, a patch was made that given three control inputs – frequency, gain and Q – delivered five control outputs – the filter coefficients for a biquad. The patch is shown in figure 6. Even though it may look a bit complicated with all the crossed patch cords, a patch like this can be made fairly easily, even by someone not knowledgeable in a traditional programming language like C++.

In order to verify aurally if the filter calculations were working correctly, another patch was made that allowed white noise to be sent through the **biquad~** object. Now, the coefficients could be applied and the filter performance listened to, as the frequency, gain and Q values were adjusted in real-time. The patch is shown in figure 7. Note that the whole patch in figure 6 is included as a sub-patch in figure 7. Max/MSP allows this type of nesting of patches within other patches.

Some additional patches were made to make preliminary measurements using sine tone sweeps to view graphically the magnitude of the frequency response of the filter. Later, in chapter 5, the results of analyzing the C++ version of the plug-in, using an impulse response to find the magnitude and the phase of the frequency response, is presented.

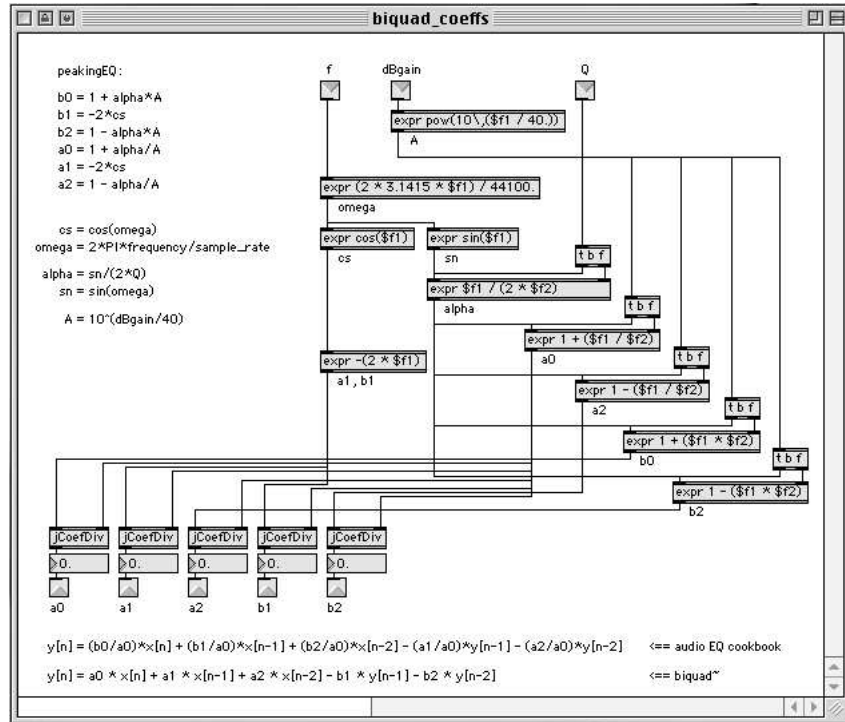


Figure 6: Patch to calculate biquad coefficients.

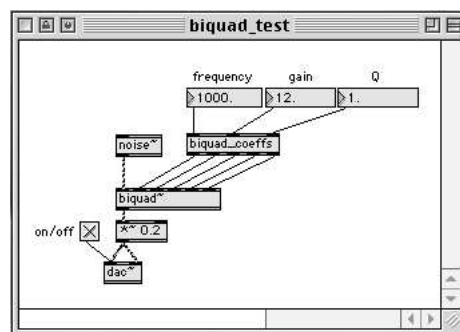


Figure 7: Patch to listen to filtered white noise.

3.3 Pluggo

The easiest way to transform the parametric EQ prototype into a VST plug-in would be to use the plug-in objects – collectively referred to as *pluggo* – provided in Max/MSP. The patch in figure 8 shows the pluggo version of the prototype with the **plugin~** and **plugout~** objects. This essentially is a working VST plug-in. Pluggo can also be used to make RTAS and MAS plug-ins.

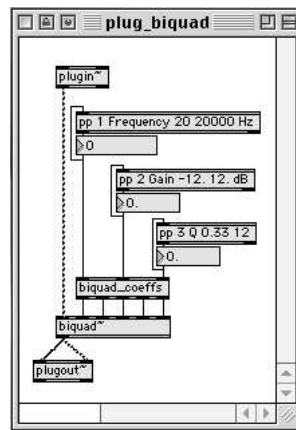


Figure 8: A *pluggo* version of the parametric EQ VST plug-in.

Pluggo plug-ins only work on Macintosh computers. To be able to make versions of the VST plug-in for multiple computer platforms, they have to be written in C++.

4 Implementation

The information contained in this chapter constitutes the innermost specialist layer of the thesis, and as such assumes a prior working knowledge of C++ syntax and terminology. Nevertheless, the presentation in this chapter is such that the interested reader should be able to understand the overall structure of the source code of a VST plug-in, even without expert knowledge in C++.

For the implementation the Steinberg VSTSDK C++ framework was used. The source code was compiled using the Visual C++¹⁰ development environment on Windows, and the CodeWarrior¹¹ development environment on MacOS. For all the supported platforms, the source code of a VST plug-in is identical. The compiled plug-in format differs though. On MacOS a plug-in is a code resource, while on Windows a plug-in is a multi-threaded DLL. For BeOS and SGI/MOTIF a plug-in is a library [1].

4.1 The VSTSDK C++ framework

The VSTSDK consists of an object-oriented C++ framework for building VST plug-ins. All the basic functionality that a plug-in needs is provided by a base class, e.g. the communication of audio sample buffers between the host application and the plug-in, and the handling of user adjustable input parameters. The code in the VSTSDK specifies a plug-in API that abstractly sits on top of the host application. Writing audio plug-ins relieves the programmer of the burden to communicate directly with the audio interface hardware in the computer, since this is taken care of by the host application on a lower level, using one of the audio APIs that the audio interface driver provides. An illustration of this can be seen in figure 9.

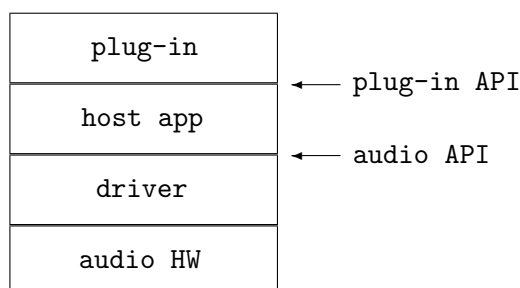


Figure 9: Plug-in API and audio API – a hierarchical view.

¹⁰<http://msdn.microsoft.com/visualc/>

¹¹<http://www.metrowerks.com/MW/Develop/Desktop/Macintosh/Default.htm>

The host provides a default GUI for the plug-in if the programmer has not developed a specific GUI, to let the user adjust the input parameters. Thus, the majority of programming effort can be put into the DSP part of the plug-in code.

4.2 The AudioEffect and AudioEffectX classes

In 1996 the VST 1.0 specification was released. The version used for the plug-in in this thesis is the VST 2.0 specification, which is an extension of the 1.0 specification.

In the VST specification, the base class for all plug-ins is called AudioEffect, and is defined in a file named AudioEffect.hpp. Parts of that file look like the following:

```
class AudioEffect                                     // VST 1.0 specification.
{
public:
    AudioEffect();                                   // Constructor.
    virtual ~AudioEffect();                           // Destructor.

    virtual void process();                           // Called by the host for an aux effect.
    virtual void processReplacing();                   // Called by the host for an insert effect.

    virtual void setParameter();                       // Sets a specified parameter value.
    virtual float getParameter();                     // Gets a specified parameter value.

    virtual void getParameterName();                  // e.g. "Frequency" in GUI.
    virtual void getParameterDisplay();               // e.g. "1000.0" in GUI.
    virtual void getParameterLabel();                 // e.g. "Hz" in GUI.

    virtual void setProgramName();                     // Used if the plug-in uses
    virtual void getProgramName();                     // parameter setting presets.

protected:
    float sampleRate;                                // Current sample rate used by the host.
};
```

The code has been edited and shortened (empty parameter lists) to increase clarity, and the comments are by the author. Only the most relevant methods and data members are shown. The VST 2.0 specification extends the base class by defining a sub class AudioEffectX in the file audioeffectx.h:

```
class AudioEffectX : public AudioEffect               // VST 2.0 specification.
{
    // All the code here is left
    // out to increase clarity.
};
```

All the code has been left out since the new methods and data members are not relevant for the parametric EQ plug-in. As the AudioEffectX class inherits from the AudioEffect class, it is compatible to the VST 1.0 specification. The classes are implemented in the files AudioEffect.cpp and audioeffectx.cpp respectively.

VST plug-ins have two methods that contain the actual signal processing code – `process()` and `processReplacing()`. The method `process()` gets called by the host when the plug-in is used in an aux send configuration. If the plug-in is used as an insert effect in the host, then the method `processReplacing()` gets called instead.

4.3 The Biquad class

To make the parametric EQ plug-in, a new sub class – Biquad – was defined and implemented, that inherited from `AudioEffectX`. In this sub class, the methods `process()` and `processReplacing()` got their redefined implementations. The class hierarchy is shown in figure 10.

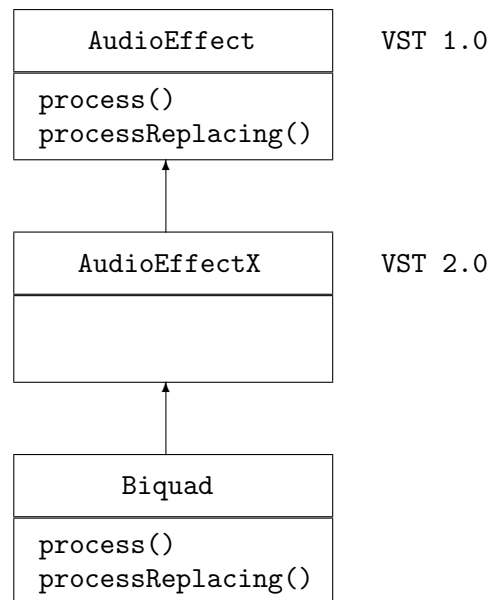


Figure 10: Class hierarchy for the Biquad VST plug-in.

The class Biquad was defined in the following way:

```

class Biquad : public AudioEffectX           // Second-order IIR parametric EQ sub class.
{
public:
    Biquad(audioMasterCallback audioMaster); // Constructor.
    ~Biquad();                             // Destructor.

    virtual void process(float **inputs, float **outputs, long sampleFrames);
    virtual void processReplacing(float **inputs, float **outputs, long sampleFrames);
    virtual void setProgramName(char *name);
    virtual void getProgramName(char *name);
    virtual void setParameter(long index, float value);

```

```

virtual float getParameter(long index);
virtual void getParameterLabel(long index, char *label);
virtual void getParameterDisplay(long index, char *text);
virtual void getParameterName(long index, char *text);

virtual void calcCoeffs(float f, float g, float q); // Compute biquad coefficients.

virtual float calcFreq(float f);           // Convert 0.0...1.0 --> 20 Hz... 20 kHz
virtual float calcGain(float g);           // Convert 0.0...1.0 --> -12 dB...+12 dB
virtual float calcQ(float q);              // Convert 0.0...1.0 --> 0.33... 12.0

protected:
    float fFrequency;                      // 0.0 ... 1.0      Internal
    float fdBGain;                         // 0.0 ... 1.0      parameter
    float fQ;                             // 0.0 ... 1.0      format.

    float jFrequency;                      // 20 Hz ... 20 kHz  GUI
    float jdBGain;                         // -12 dB ... +12 dB parameter
    float jQ;                             // 0.33 ... 12      format.

    float xnm1;                            // x[n-1]
    float xnm2;                            // x[n-2]
    float ynm1;                            // y[n-1]
    float ynm2;                            // y[n-2]

    float a0, a1, a2, b0, b1, b2;          // The biquad coefficients.
};

```

The code shown is not edited but the actual source code used to compile the plug-in. Only some macro definitions (`#define`) and enumerations (`enum`), preceding the class declaration, have been edited out to shorten the code listing.

In the method `calcCoeffs()` the formulae for calculating the filter coefficients were implemented:

```

void Biquad::calcCoeffs(float f, float g, float q)
{
    float A, omega, cs, sn, alpha;          // Intermediate variables.

    A    = pow(10,g/40.0f);
    omega = (2 * M_PI * f) / sampleRate;    // M_PI macro holds value of pi.
    sn    = sin(omega);
    cs    = cos(omega);
    alpha = sn / (2.0*q);

    b0    = 1 + (alpha * A);                 // The filter coefficients.
    b1    = -2 * cs;
    b2    = 1 - (alpha * A);
    a0    = 1 + (alpha / (float)A);
    a1    = -2 * cs;
    a2    = 1 - (alpha / (float)A);
}

```

The actual audio sample processing code was implemented in the method `processReplacing()` in the following way:

```
void Biquad::processReplacing(float **inputs, float **outputs, long sampleFrames)
{
    float *in = inputs[0];    // in points to the first sample in the input buffer.
    float *out = outputs[0];   // out points to the first sample in the output buffer.
    float xn, yn;              // xn/yn holds current input/output sample.

    while(--sampleFrames >= 0) // Go through the buffers sample-by-sample.
    {
        xn = *in++;            // Get xn from the input buffer.

        yn = (b0*xn + b1*xnm1 + b2*xnm2 - a1*ynm1 - a2*ynm2)/a0; // Biquad equation.

        xnm2 = xnm1;          // Shift x[n-1] to x[n-2].
        xnm1 = xn;             // Shift x[n] to x[n-1].
        ynm2 = ynm1;          // Shift y[n-1] to y[n-2].
        ynm1 = yn;             // Shift y[n] to y[n-1].

        *out++ = yn;           // Put yn into the output buffer. (Overwrite)
    }
}
```

Only the method `processReplacing()` is shown here, since the natural way of using an EQ plug-in would be as an insert effect. The VST specification requires that the method `process()` is always provided, while `processReplacing()` is optional, and the specification highly recommends that both methods are always implemented. In the Biquad class, the method `process()` only differed from the method `processReplacing()` in the last line of code,

```
(*out++) += yn;           // Put yn into the output buffer. (Accumulate)
```

where the assignment operator `+=` was substituted for `=`.

The C++ code listings in this section have been kept to a minimum in order to make the general ideas come out clear, without being cluttered by too much detail. By email inquiry to the author, the complete C++ code listings for the peak/notch parametric EQ VST plug-in can be obtained¹².

¹²jonas.ekeroot@mh.luth.se

5 Results

The parametric EQ VST plug-in was tested in a number of different VST host applications on both Windows and MacOS. For the testing on Windows, an IBM T22 ThinkPad laptop computer was used with the following hardware and operating system specifications:

- 900 MHz Intel Pentium III processor
- 256 MB RAM
- Windows XP Professional operating system, version 2002
- built-in audio hardware (Crystal SoundFusion Audio Device by Crystal Semiconductor)

The MacOS testing was performed on an Apple PowerBook G3 laptop computer with the following hardware and operating system specifications:

- 500 MHz PowerPC G3 processor
- 256 MB RAM
- MacOS 9.1 operating system
- built-in audio hardware (Screamer sound IC)

The plug-in appeared without a problem in all of the host applications on both Windows and MacOS. By playing sound files from the hard disk and applying the plug-in, the audio processing of the plug-in was found to be working in all of the applications tested. In this way, aural assessments of how the signal processing of the plug-in affected different audio signals were made.

Then, to measure the plug-in performance objectively, MATLAB was used to analyze an impulse response made with the host application AudioMulch¹³ and the parametric EQ VST plug-in. By recording the impulse response of the plug-in for a specific setting of the parameters (frequency, gain and Q value), and transforming the impulse response into the frequency domain, the magnitude and the phase of the frequency response of the filter were extracted and plotted in graphs.

Finally, the resulting default GUI of the plug-in in a number of different host applications on both Windows and MacOS were compared.

¹³<http://www.audiomulch.com>

5.1 Aural assessment

During the prototype development in Max/MSP, the performance of the **biquad~** object was listened to, using the calculated filter coefficients. The compiled VST plug-in was also listened to in all of the tested host applications. Using both white noise and music as input signals, the filter was aurally verified to be boosting or attenuating the region of the frequency spectrum that was set with the frequency, gain and Q parameters.

5.2 Matlab measurements

A digital filter is completely characterized by its time-domain impulse response, as stated by Pohlmann [17]. Further, the filter can also be described in the frequency-domain by its *frequency response*. The impulse response and the frequency response of a discrete-time system are related by the *Discrete Fourier Transform* (DFT). A computationally efficient implementation of the DFT is the *Fast Fourier Transform* (FFT). McClellan *et al* [7] gave a detailed description of both the DFT and the FFT. The FFT transforms the impulse response into the frequency response, which is a complex valued function. By stepping through the frequency response and for each complex value find the absolute value and the argument (polar notation), the magnitude and the phase of the frequency response of the digital filter is obtained.

For the analysis of the VST plug-in, the following MATLAB script was written to read the impulse response from a WAV file, compute the FFT, and then extract the magnitude and the phase of the frequency response:

```
[H,FS,NBITS] = wavread('ir_L_44_16_1k+12_1.wav'); % Read impulse response from WAV file.

[M,I] = max(H); % Find start of impulse response (IR).

sa_beg = I; % Index of first sample in IR.
sa_end = sa_beg + (FS - 1); % Use FS samples (1 sec) of IR.

Hcut = H(sa_beg:sa_end,1); % Extract IR from H (whole WAV file).

fftHcut = fft(Hcut,FS); % FFT (DFT) of length FS.
posFFT = fftHcut(1:length(fftHcut)/2,1); % Use only positive frequencies.

s1 = subplot(3,1,1);
stem(Hcut(1:128,1),'.') % Plot first 128 samples of IR.

s2 = subplot(3,1,2);
loglog(abs(posFFT)) % Plot the magnitude response.

s3 = subplot(3,1,3);
semilogx(angle(posFFT)) % Plot the phase response.
```

A one channel (mono) WAV file with a sampling frequency of 44.1 kHz was made using MATLAB. The file was two seconds in length (i.e. 88200 samples) and had all sample values set to zero, except for one sample set

to the normalized sample value of 0.5, corresponding to -6 dBFS. This impulse was in the middle of the WAV file, after 1 second of silence. Using AudioMulch, this impulse WAV file was played through the parametric EQ VST plug-in, and the impulse response was recorded into a new WAV file. With the MATLAB script described earlier, the frequency response was calculated by means of the FFT.

The VST plug-in was tested with several different parameter settings. One of these settings is presented in this thesis. Assuming that the sample index n of the original impulse value of 0.5 is $n = 1$, figure 11 shows the first 128 samples of the original impulse and the impulse response of the VST plug-in with the following filter settings: $f = 997.7691$ Hz, $g = +12.0$ dB and $Q = 1.000277$. Since this was an IIR filter, the impulse response theo-

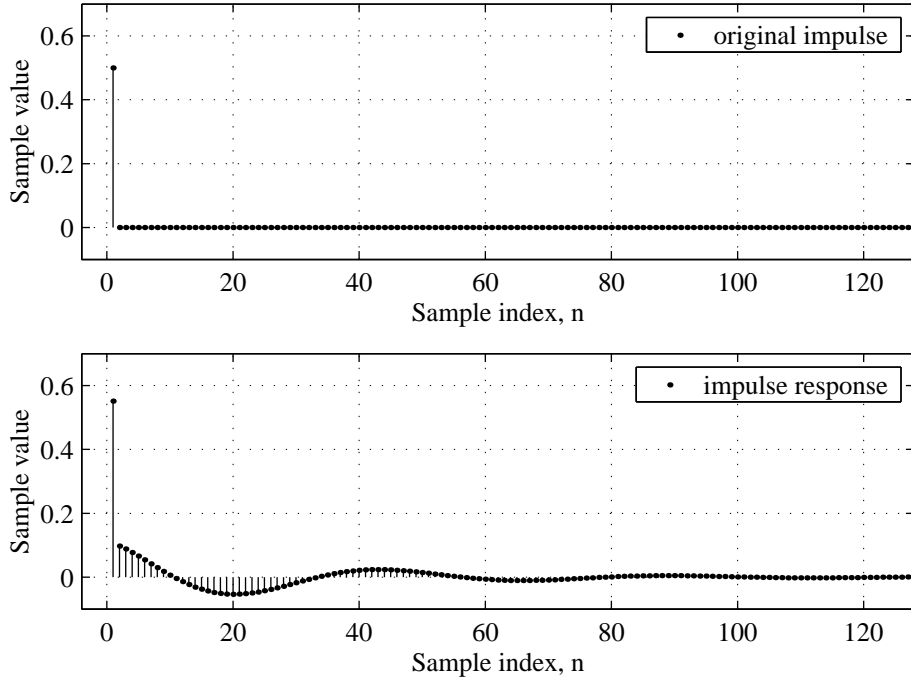


Figure 11: The first 128 samples of the original impulse and the impulse response. ($f = 997.7691$ Hz, $g = +12.0$ dB and $Q = 1.000277$)

retically never decreased to zero and stayed at that value, but continued to oscillate above and below the zero level for ever. For all frequency response calculations made, one second of the impulse response, i.e. 44100 samples at a sampling rate of 44.1 kHz, was used.

Note that the first sample ($n = 1$) in the impulse response in figure 11 has a value of approximately 0.551, which is greater than the original

impulse value of 0.5. For this reason, the original impulse value was chosen to be less than 1.0 so that the increased sample value in the impulse response would stay below the maximally allowed value of 1.0 using normalized floating-point sample values. Failure to observe this requirement gives erroneous frequency response results due to overflow values greater than 1.0 in the impulse response.

The magnitude and the phase of the frequency response of the filter with the impulse response shown in figure 11, are shown in figure 12.

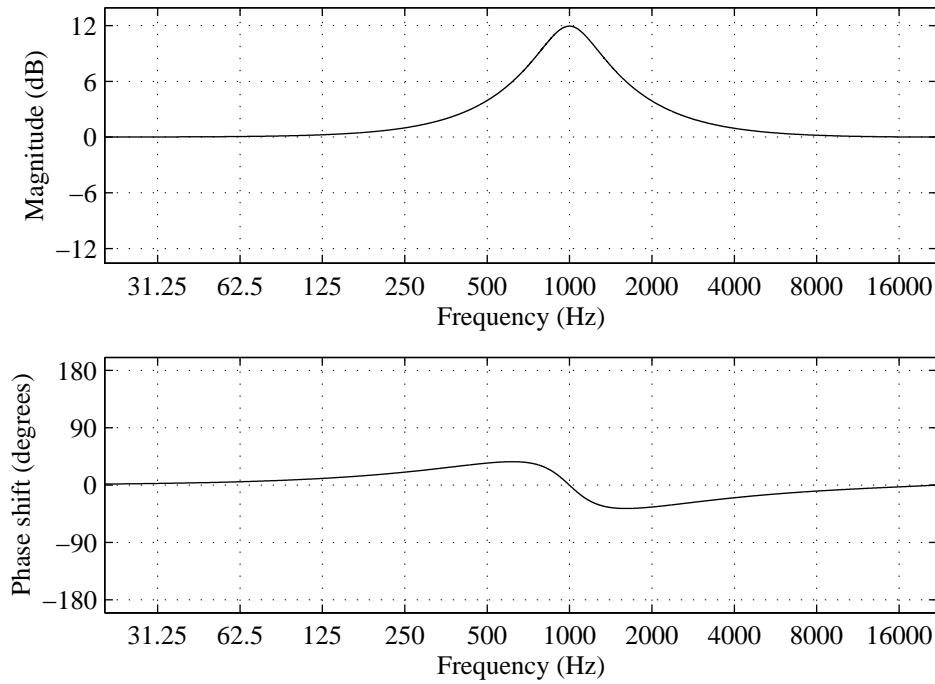


Figure 12: The magnitude and the phase of the frequency response.
($f = 997.7691$ Hz, $g = +12.0$ dB and $Q = 1.000277$)

A bell-shaped curve with a maximum around 1 kHz, with 12 dB gain relative to the 0 dB line, can be seen in the graph of the magnitude response. The MATLAB script sets the number of points in the FFT calculation to the same value as the sampling rate of the WAV file containing the impulse response. This gives a resolution of 1 Hz in the frequency response. The center of the magnitude peak in figure 12 is at $f_{center} = 998$ Hz. The midpoint gain (i.e. $g/2$ dB) frequencies, as discussed in 2.5.5, are at $f_{lower} = 616$ Hz and $f_{upper} = 1613$ Hz. This bandwidth gives a Q value of

$$Q = \frac{f_{center}}{f_{upper} - f_{lower}} = \frac{998}{1613 - 616} \approx 1.001003 \quad (5.1)$$

As can be seen from the lower graph in figure 12, the filter has a *non-linear phase response*. This phase distortion is a known fact for recursive IIR filters [9], and means that the filter smears transients over time due to the frequency dependent delay. FIR filters, in contrast, can generally be designed to have a *linear phase response*, but the disadvantage of an FIR filter is that it is computationally more intensive and has higher memory requirements (longer feedforward delays, i.e. higher order) than an IIR filter with similar magnitude response.

So, in conclusion, and analyzed with a 1 Hz frequency-domain resolution, the frequency position, the gain value and the bandwidth of the peak corresponded well with the given parameter settings of the VST plug-in.

5.3 Default plug-in GUI

Since the implementation presented in this thesis did not deal at all with the GUI of the plug-in, the graphical representation varied among the applications, as can be seen from figure 13 to figure 18.

5.4 Windows host applications

On Windows the plug-in was tested using three different VST host applications:

- WaveLab 3.04g by Steinberg¹⁴
- AudioMulch 0.9b9p1 – an interactive music studio by Ross Bencina, beta version¹⁵
- Audacity 0.98 – a free, open-source (C++, GNU General Public License), multi-platform (Windows, MacOS and Unix/Linux) digital audio editor, started in 1999 as a development project by Dominic Mazoni at Carnegie Mellon University¹⁶

5.4.1 WaveLab

Figure 13 shows a screenshot of the parametric EQ VST plug-in with the default GUI provided by WaveLab. The plug-in window imitates the front panel of an external hardware effects unit with a display-like view of the parameter values, buttons to switch between the three parameters and a large dial to adjust parameter values. WaveLab also provides additional buttons to bypass or mute the plug-in, and to handle presets, i.e. predefined

¹⁴http://www.steinberg.net/en/ps/products/audio_editing/wavelab/

¹⁵<http://www.audiomulch.com>

¹⁶<http://audacity.sourceforge.net>



Figure 13: Parametric EQ VST plug-in in WaveLab.

parameter settings supplied by the original programmer of the plug-in or set and saved by the plug-in user.

The audio processing of the plug-in is applied in real-time.

5.4.2 AudioMulch

The default GUI provided by AudioMulch is considerably less sophisticated than in WaveLab, as figure 14 shows. A spreadsheet-like layout shows the

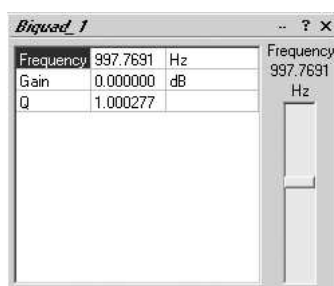


Figure 14: Parametric EQ VST plug-in in AudioMulch.

parameters. By clicking with the mouse on the name, the current parameter to edit is selected, and a vertical slider (fader) then adjusts the parameter value. As in WaveLab, the audio processing is applied in real-time.

The implemented parametric EQ plug-in is a pure mono plug-in, i.e. it has one channel of input and one channel of output. In many host applications, the number of input and output channels of a plug-in is not clearly indicated, which can be a source of confusion depending on how the source code of the plug-in is written. AudioMulch has a patch window, a bit similar to Max/MSP, where the exact number of plug-in inputs and outputs can clearly be seen. Figure 15 shows a patch where the mono output of a test signal generator is connected to the mono input of the parametric EQ VST plug-in. The mono output of the plug-in is then connected to the left channel output of the audio interface in the computer. In this way, AudioMulch gives a good overview of the signal paths of a patch involving a VST plug-in. This was of great help during the plug-in development stage, and also later

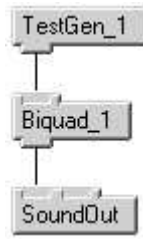


Figure 15: Patch in AudioMulch showing a mono plug-in.

for testing purposes.

5.4.3 Audacity

Audacity provides a simplistic default plug-in GUI, as shown in figure 16.

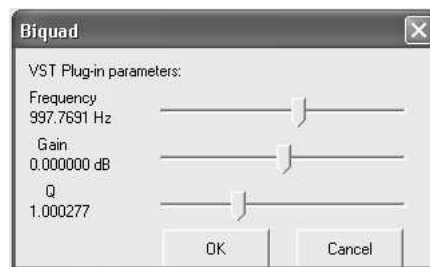


Figure 16: Parametric EQ VST plug-in in Audacity.

The GUI differs in principle from those provided by WaveLab and AudioMulch in that each of the three parameters have their own horizontal slider for value adjustments. As was also the case in AudioMulch, there are no extra buttons provided by the host for bypass, mute or presets.

Audacity has a limitation that means that even though the plug-in is capable of providing real-time processing, all plug-in processing is applied in non-real-time. The filter calculations for all of a selected audio track have to be completed, before the results of the plug-in can be listened to.

5.5 MacOS host applications

The host applications used on MacOS were:

- Cubase VST 5.0 by Steinberg, demo version¹⁷
- Max 4.0.7/MSP 2.0 using the `vst~` object

¹⁷http://service.steinberg.de/webdoc_ps_int.nsf/show/demos_applications_pro_mac_en

5.5.1 Cubase VST

In Cubase VST the parametric EQ plug-in showed up as in figure 17. The



Figure 17: Parametric EQ VST plug-in in Cubase.

GUI is a bit more elaborate than in AudioMulch and Audacity. It uses a separate horizontal slider for each parameter in the same way as in Audacity. An on/off button is provided as well as a popup menu to handle the loading and saving of preset files.

The audio processing of the plug-in is applied in real-time.

5.5.2 Max/MSP

Even though Max/MSP was used as a prototype development environment for the parametric EQ plug-in, it can also act as a totally reconfigurable VST host application using the `vst~` object. The default plug-in GUI provided by this object can be seen in figure 18, which shows that the separate three-slider approach is used in Max/MSP too.

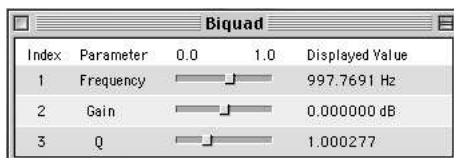


Figure 18: Parametric EQ VST plug-in in Max/MSP.

No extra buttons for bypass, mute or presets are graphically visible. This functionality can be accessed though by connecting additional control objects to the `vst~` object. Note that the internal parameter scale of 0.0 to 1.0 is shown above the sliders.

The audio processing of the plug-in is applied in real-time.

5.6 Parameter adjustments

An important aspect of implementing audio plug-ins is to try to make the parameter adjustments as responsive as possible, so that they do not appear sluggish or create audible artefacts while being changed. The responsiveness

of the parametric EQ VST plug-in implemented in this thesis showed a sluggish tendency in WaveLab and AudioMulch on Windows. In Cubase VST and Max/MSP on MacOS the plug-in showed no sluggish tendency at all, and the responsiveness was completely smooth. Since Audacity on Windows applied the plug-in processing in non-real-time, the responsiveness could not be evaluated in that host application.

The sluggishness of the parametric EQ VST plug-in on Windows was found to be about the same and not worse than the sluggishness experienced when comparing the plug-in with the included EQ-1 plug-in in WaveLab and the included MParaEQ filter in AudioMulch.

5.7 Summary of results

The resulting C++ implementation of the VST plug-in was verified using five different host applications on both Windows and MacOS, and the plug-in was found to be working both visually (the default GUI) and aurally in all of them. Measurements using MATLAB also showed that the plug-in affected the frequency content of the audio in a way that was consistent with the desired parameter settings.

The responsiveness of the parameter adjustments showed a sluggish tendency in WaveLab and AudioMulch on Windows. So, when compiled using different compilers and operating systems, the same C++ source code resulted in a noticeable different sluggishness for the plug-in when used in the tested Windows and MacOS host applications on the computer systems used during the work of this thesis.

6 Discussion

6.1 Max/MSP and similar applications

Max/MSP is currently available only for MacOS, but a Windows version is in development and it was publicly demonstrated in January 2003. This would make it possible to do plug-in prototyping also on Windows in the same way as shown in this thesis. Two software applications that could possibly be used to make prototypes in a similar way are

- Pd (Pure Data) – a real-time music and multimedia environment¹⁸
- jMax – a visual programming environment for interactive real-time music and multimedia¹⁹

Pd is a free, open-source, Max/MSP-like application for Windows, Linux, SGI/IRIX and MacOS X, developed by Miller Puckette – one of the original developers of Max/MSP. The GUI front end of Pd is made using the scripting language Tcl/Tk.

The historical roots of Max/MSP can be found at IRCAM²⁰ in Paris. Today IRCAM provides the application jMax as a freely downloadable Max/MSP-like application that runs on Windows, Linux, SGI/IRIX and MacOS X. jMax is distributed under the GNU General Public License. It uses Java for the GUI front end.

Among commercial products Reaktor²¹ from Native Instruments makes use of graphical patch programming in the spirit of Max/MSP. Reaktor is available for Windows and MacOS, and the whole development environment can actually be used as a VST plug-in, a DirectX plug-in or an Audio Units plug-in.

6.2 The VSTSDK C++ source code

The C++ source code in the VSTSDK provides a well structured framework to be used as a starting point for plug-in development. Anyone who knows digital audio and C++ should not find it very hard to understand. This fact, and the possibility that from the same source code files create plug-ins for multiple computer platforms, should make it interesting for students who want to learn about audio DSP in software. The majority of programming effort can be put into the audio DSP part of the code, since all hardware interaction, e.g. audio input and output, is taken care of by the host application. For companies focusing on plug-in development, the same source code base can be used to make plug-ins for both MacOS and

¹⁸<http://www-crca.ucsd.edu/~msp/software.html>

¹⁹<http://www.ircam.fr/jmax>

²⁰<http://www.ircam.fr/index-e.html>

²¹http://www.native-instruments.com/index.php?reaktor_us

Windows, eliminating the need to maintain separate source code bases for the different platforms. The VST format is a popular plug-in format, well established on the market among third-party developers.

For more complex plug-ins, issues like the handling of threads on different platforms, might pose problems and make the source code platform dependent in a way that is not illustrated by the comparatively simple parametric EQ plug-in developed in this thesis.

6.3 Optimization

The C++ code written for the parametric EQ plug-in was not optimized in any way for speed or size. It was simply a straightforward implementation of a second-order IIR filter and the necessary formulae to calculate the filter coefficients. As such, the code is more an illustration of the basic principles of plug-in programming, rather than a highly optimized top performing EQ plug-in.

Future work in this area could try to minimize the CPU load of the plug-in. This is an important area in general, since all the plug-ins in a host application share the same finite amount of available CPU power, and less CPU load per plug-in means that more plug-ins can be used at the same time.

6.4 The frequency parameter limits

The parametric EQ VST plug-in allows the frequency parameter to be set in the range 20 - 20000 Hz. This is regardless of the sampling rate used by the host application, and thus also used by the plug-in. Equation (2.10) on page 19 involves the division by the filter coefficient a_0 . Because of this, the value of a_0 must not be zero. For $a_0 = 1 + \alpha/A$ to become zero, the fraction α/A must be -1 . $A = 10^{g/40}$ is always positive. Q , used in the calculation of $\alpha = sn/2Q$, is per definition also positive, so for α to take on negative values, $sn = \sin 2\pi f/f_s$ has to be negative. This occurs for $f_s/2 < f < f_s$. The sampling theorem states:

“A continuous-time signal $x(t)$ with frequencies no higher than f_{max} can be reconstructed exactly from its samples $x[n] = x(nT_s)$, if the samples are taken at a rate $f_s = 1/T_s$ that is greater than $2f_{max}$.” [7]

In other words, since a digital signal only contains frequency components up to half the sampling rate ($f_s/2$), the plug-in frequency parameter should not be allowed to exceed this limit either. The parametric EQ VST plug-in implementation uses a fixed range of 20 - 20000 Hz for the frequency parameter, which means that α can take on negative values for a sampling

frequency below 40 kHz. So for a sampling frequency of 44.1 kHz or 48 kHz, a_0 will never be zero in the plug-in implementation.

An example of a problematic case would be a sampling frequency of 22050 Hz with $f = 16537.5$ Hz, $g = 0$ dB and $Q = 0.5$. The frequency parameter should in this case really not be allowed to be set any higher than $f = 11025$ Hz as a maximum, but $f = 16537.5$ Hz is possible in the current plug-in implementation. This results in $A = 1$ and $\alpha = -1$ and thus $a_0 = 0$. To summarize, a future version of the parametric EQ VST plug-in should not have a fixed upper limit for the frequency parameter, but use a value of $f_{upper\ limit} < f_s/2$ in order to cope with lower sampling rates.

6.5 Cross-platform GUI

As shown by the figures in chapter 5, the default GUI appearance of the parametric EQ plug-in differs considerably among host applications and operating systems. By adding code to handle a custom GUI, the plug-in can be made to have the same look regardless of the host application used.

GUI programming generally can be rather complicated, and tends to be very platform specific. Included in the VSTSDK are the *VSTGUI Libraries* to be used to create and handle a GUI with faders, knobs, dials, etc. This provides a way of implementing a GUI without having to deal with platform specific details. Creating a custom GUI was out of the scope for this thesis, but might be an interesting path for future examination.

6.6 Conclusions

This thesis focused on the development of a VST audio plug-in. By starting with a prototype made in a graphical programming environment, hands-on experience with the necessary DSP theory and calculations was obtained. This experience was then applied in the implementation of the plug-in in C++. Because of the preparatory algorithmic work that was done in the prototype, the C++ implementation was made without any major problems. In addition to this development method in two stages, the thesis also demonstrated a way of verifying the audio processing of the final plug-in, by analyzing impulse response measurements using MATLAB.

References

- [1] Steinberg VST 2.0 Software Development Kit (SDK).
URL: <http://www.steinberg.net/en/ps/support/3rdparty/>
- [2] Arfib, D. (1998) Different Ways to Write Digital Audio Effects Programs. In *Proceedings of the COST-G6 Workshop on Digital Audio Effects Processing (DAFx'98)*, 19-21 Nov 1998, Barcelona, Spain, pp. 188-191.
URL: <http://www.iaa.upf.es/dafx98/papers/ARF36.PS>
- [3] MacMillan, K., Droettboom, M. and Fujinaga, I. (2001) Audio Latency Measurements of Desktop Operating Systems. In *Proceedings of the International Computer Music Conference (ICMC)*, 17-22 Sep 2001, Havana, Cuba, pp. 259-262.
URL: <http://gigue.peabody.jhu.edu/~mdboom/latency-icmc2001.pdf>
- [4] Zölzer, U. (2002) *DAFX – Digital Audio Effects*. John Wiley & Sons, Chichester, West Sussex.
- [5] Bendiksen, R. (1997) *Digitale lydeffekter*. Diplomoppgave i akustikk, Norges teknisk-naturvitenskapelige universitet, Institutt for teleteknikk, Trondheim.
URL: <http://www.notam02.no/~rbendiks/Diplom.html>
- [6] Browning, P. (1997) *Audio Digital Signal Processing In Real Time*. Master's thesis, West Virginia University, Morgantown.
URL: <http://www.tcicomp.com/paul/dsp/docword97.zip>
- [7] McClellan, J., Schafer, R. and Yoder, M. (1998) *DSP First: A Multimedia Approach*. Prentice-Hall, Upper Saddle River, New Jersey. Reprinted with corrections June, 1999.
- [8] Patterson, D. and Hennessey, J. (1998) *Computer Organization and Design – The Hardware/Software Interface*, 2nd ed. Morgan Kaufmann Publishers, San Francisco.
- [9] Roads, C. (1996) *The Computer Music Tutorial*. The MIT Press, Cambridge, Massachusetts.
- [10] Dattorro, J. (1988) The Implementation of Recursive Digital Filters for High Fidelity Audio. *J. Audio Eng. Soc.* Vol.**36**, No.11, pp. 851-878.
- [11] Bristow-Johnson, R. *Cookbook formulae for audio EQ biquad filter coefficients*.
URL: <http://www.harmony-central.com/Computer/Programming/Audio-EQ-Cookbook.txt>

- [12] Bristow-Johnson, R. (1994) The Equivalence of Various Methods of Computing Biquad Coefficients for Audio Parametric Equalizers. Presented at *AES 97th Convention, San Francisco*. Preprint 3906.
URL: http://www.harmony-central.com/Effects/Articles/EQ_Coefficients/EQ-Coefficients.pdf
- [13] Harris, F. and Brooking, E. (1993) A Versatile Parametric Filter Using an Imbedded All-Pass Sub-Filter to Independently Adjust Bandwidth, Center Frequency and Boost or Cut. Presented at *AES 95th Convention, New York*. Preprint 3757.
- [14] White, S. (1986) Design of a Digital Biquadratic Peaking or Notch Filter for Digital Audio Equalization. *J. Audio Eng. Soc.* Vol.**34**, No.6, pp. 479-483.
- [15] Moorer, J. (1983) The Manifold Joys of Conformal Mapping: Applications to Digital Filtering in the Studio. *J. Audio Eng. Soc.* Vol.**31**, No.11, pp. 826-841.
- [16] Puckette, M. (1991) Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal* **15**(3), pp. 68-77.
URL: <http://www-crca.ucsd.edu/~msp/Publications/cmj91-max.ps>
- [17] Pohlmann, K. (2000) *Principles of Digital Audio*, 4th ed. McGraw-Hill, New York.