FAKULTÄT FÜR !NFORMATIK

Effect Unit for an electric guitar: *Algorithms and Implementation*

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Technische Informatik 938

ausgeführt von

Thomas Semper
Matrikelnummer 0426850

am:
Institut für Nachrichtentechnik und Hochfrequenztechnik

Betreuung:
Betreuer: Univ.Prof. Dr.-Ing. Norbert Görtz
Mitwirkung: Ass.Prof. Dipl.-Ing. Dr.techn. Gerhard Doblinger

Wien, 07. 05. 2010

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Abstract

This diploma thesis deals with the research and implementation of digital audio effects, especially for electric guitars. On the one hand the signal processing aspects are of interest on the other hand a software development platform is created. Thus new audio effects can be added and existing ones can be improved easily. The aim is a software tool to demonstrate and experiment with audio signal processing algorithms. High value is set on a good user interface so that different audio effects can be changed and combined fast. In this thesis three groups of guitar effects have been studied: Delay based effects like the chorus or the flanger effect, spatial based effects like the reverb effect and nonlinear effects like distortion. In a first step all algorithms have been implemented with Matlab [TM10].

Acknowledgements

I thank Univ.Prof. Dr.-Ing. Norbert Görtz for the supervision of this very interesting and practical thesis. His massive support and cooperation were very helpful to write this thesis. Further i would like to thank Ass.Prof. Dipl.-Ing. Dr.techn. Gerhard Doblinger for his assistance ideas of improvement. Lastly I want to thank my parents very much for their support during my whole studies. Without them non of these would have been possible.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

1.1. Motivation

Effect units to alter the sound of electric guitars have been used since the sixties by guitar heroes like Jimmy Hendrix. They help musicians to create many different sounds from one source which will enrich their music. In the past guitar effect units were analog devices which were put between the electric guitar and the guitar amplifier. Today we have powerful digital processors which more and more replace the old analog effect units. For example one digital multi effect unit can replace several analog effects. So the musician just has to handle one device to use many effects like flanger, distortion and reverb at the same time. In the early days they had to plug several single effect units together to get the same result. So at least in the hobby area digital multi effect units have become quite common. In the professional area analog effect units are still popular. This is because certain kinds of effects like distortion based effects just sound better or are at least more familiar with analog devices. For example guitarists swear on valve amplifiers because of their high fidelity and their ability to drive the full frequency range of a loudspeaker. It's perhaps not so much the exact reproduction of the sound but rather harmonics produced by nonlinearities in valve amplifiers. Many algorithms for guitar effects are not very complicated but have to be implemented efficiently to achieve real time capabilities. Thats the reason why they are perfectly suited to demonstrate signal processing basics. It's obviously much easier for students to understand what it means to add a signal to a delayed copy of itself, if they hear the audio effect. By the way this is more or less what all delay based effects do. This master thesis shall provide the basics for an overall project with the aim to create a development environment for signal processing algorithms. The main scope lies on a good user interface and documentation. This allows other students

to expand the multi effects unit project with new effects easily. It's planned to realize the first implementation with Matlab [TM10]. Matlab shall be used to improve, explore and understand the audio effects in the time and in the frequency domain. Another important aspect is the portability on different operating systems. The multi effect unit software has to work on every operating system which is supported by Matlab. In this first version of the guitar effects unit platform, no real time operation is supported. Because it is hardly possible to process the whole effect line which consists of many different effects that have to be processed one after the other in real time on a general purpose computer today. Also the standard Matlab functions which read in audio samples from the sound card do not support real time operation. Instead wave files with guitar samples are used to demonstrate the audio effects. Maybe the effect can be implemented in the C programming language (C) on a digital signal processor (DSP) or in very high speed integrated circuit hardware description language (VHDL) on a field programmable gate array (FPGA) in later projects. These devices usually support very fast audio input and output sampling and have real time capabilities, but this task is well beyond this project.

1.2. Outline of the document

In Chapter 2 delay based effects are dealt with. First in Section 2.1 a general introduction about delay based effects in the time and in the frequency domain is given. Then in Section 2.2 the delay effect is described in detail. In Section 2.3 the chorus effect is covered. Finally in Section 2.4 the flanger effect and its relation to the chorus and the delay effect is explained. Chapter 2 also deals with linear interpolation and the low frequency oscillator (LFO), since these are important to implement delay based effects. In Chapter 3 spatial based effects are dealt with. Again in Section 3.1 a short introduction into many kinds of spatial based effects is given. In fact only one spatial based effect, namely the reverb effect is illustrated in detail. In Section 3.2 the convolution-based approach is explained in detail which is also implemented with Matlab later. A second approach using comb filters to implement a reverb effect is covered in Section 3.3. Furthermore the overlap add method is explained in Subsection 3.2.1 since it plays a main role for the convolution based approach. In Chapter 4 nonlinear effects are examined. Section 4.1 explains the effect of nonlinear operations on input sine waves, to make clear what nonlinear effects are about. Then in Section 4.2 two different methods to achieve a digital distortion effect namely hard clipping and soft clipping are explained. Since all these effects play an important role in electric guitar music their purpose and effect on electric guitar

music is presented too. These abstract chapters are mainly based on existing literature but also include personal experience gathered from designing the different audio effects. Chapter 5 is more practical and describes the implementation of the particular effects. In this chapter all implemented audio effects are illustrated with pieces of Matlab code. Section 5.2 is organized in subsections which describe the single object classes. In Section 5.3 the implementation of the graphic user interface (GUI) with it's most important functions is introduced. Section 5.3.3 gives a simple use case how to use the multi effects unit software. The thesis is concluded with Chapter 6 which gives a short summary of the project and points out how future work could look like.

Chapter 2.

Delay based effects

2.1. Introduction

All delay based effects covered in this thesis have in common that the output signal y[n] is generated by adding the input signal x[n] to a delayed copy x[n-k] of itself. Lets have a look at the difference equation (2.1) for the discrete input signal x[n]:

$$y[n] = x[n] + \beta \cdot x[n-k],\ k > 0 \tag{2.1}$$

The copy of the input signal has a delay of k samples, respectively $\delta = \frac{k}{f_s}$ seconds, where f_s is the sampling frequency. The scale factor β augments the delayed samples of the input signal. The corresponding transfer function is given by

$$H(z) = 1 + \beta \cdot z^{-k},\ k > 0. \tag{2.2}$$

The transfer function is a finite impulse response (FIR) [Dob07b] filter since it has no feedback but only a feed forward path. The magnitude response has comb filter characteristics as illustrated in Figure 2.1 for two different delays k. This kind of filter is called a comb filter since it's magnitude response has many notches similar to a comb. Frequencies that are multiplies of $1/k$ are amplified with the factor β and all other frequencies are attenuated [Zö05]. The highest magnitude equals $1+\beta$ and the lowest magnitude equals $1-\beta$ [Zö05]. In Figure 2.1 magnitude response one originates from a comb filter with delay $k = 9$ samples and magnitude response two originates from a comb filter with delay $k = 3$.

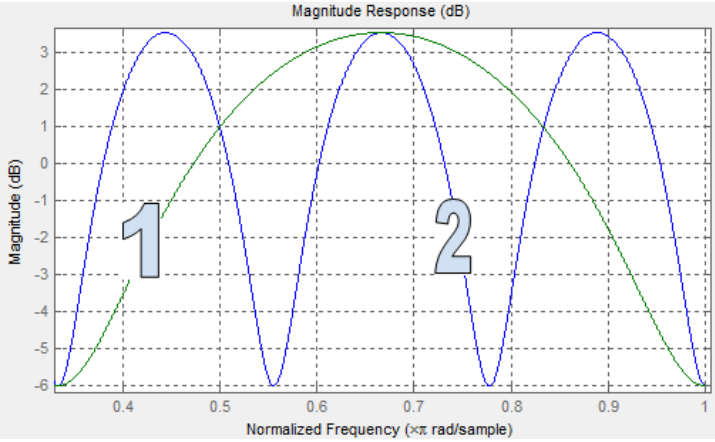


Figure 2.1.: Magnitude Response with two different delays

Because the delay of the first filter is three times bigger than the delay of the second filter, the first one has three times as many spikes as the second one. Our ear can distinguish between the input signal and its delayed copy for large values of k. But for small values the ear can no longer separate the time events.

If the delayed output signal y[n-k] is feedback to the input we get an infinite impulse response (IIR) [Dob07b] comb filter. The corresponding difference equation is given by

$$y[n] = x[n] + \beta \cdot y[n-k],\ k > 0,\ |\beta| < 1. \tag{2.3}$$

In every iteration after the time delay $\delta = \frac{k}{f_s}$ seconds the output is fed back to the input, scaled by feedback factor β. This factor β has to be restricted to values $|\beta| \leq 1$ now. For values $|\beta| > 1$ the output signal would keep growing and for values $|\beta| = 1$ never become zero again, regardless of the input signal. This behavior is typical for an unstable respectively critically stable system. The transfer function of the system in equation (2.3) is given by

$$H(z) = \frac{1}{1 - \beta \cdot z^{-k}},\ k > 0,\ |\beta| < 1. \tag{2.4}$$

The magnitude response of the infinite impulse response (IIR) filter looks similar to the magnitude response of the finite impulse response (FIR) comb filter. But the highest magnitude now equals $\frac{1}{1+\beta}$ and the lowest magnitude equals $\frac{1}{1-\beta}$ [Zö05].

With FIR and infinite impulse response (IIR) comb filters, and of course with combinations of finite impulse response (FIR) and infinite impulse response (IIR) filters, all kinds of delay

based signals can be created.

2.2. Delay

The delay effect can be implemented by a finite impulse response (FIR) or an infinite impulse response (IIR) filter 2.1. Using a mixture with a feedback and a feed forward loop is the more interesting version as it allows higher design flexibility. In Figure 2.2 the block diagram of the delay effect is given.

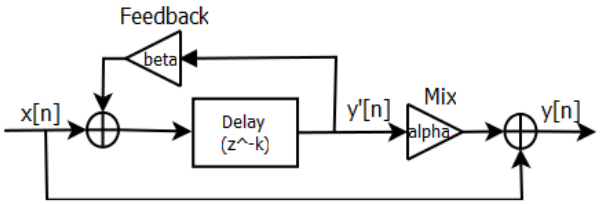


Figure 2.2.: Block diagram of delay effect

There is one feedback loop which scales the fed back samples with the feedback gain factor β. We also have a delay element which delays the input samples by time $\delta = \frac{k}{f_s}$. And we have one feed forward path. The mix gain factor α scales the delayed output signal y'[n], so we get a weighting of the modified output signal y'[n] compared to the unmodified input signal x[n]. If α is zero for example, the output signal y[n] equals the input signal x[n].

2.2.1. Time Domain

In time domain the delay effect can be described with difference equation

$$y[n] = x[n] + \alpha \cdot (x[n-k] + \beta \cdot y'[n-k]),\ k > 0,\ |\beta| < 1. \tag{2.5}$$

This equation can be best described with a temporary signal y'[n-k]. After each time delay δ, y'[n] is multiplied with the feedback gain factor β and added to the unmodified input signal. This temporary signal y'[n-k] is then multiplied with the mix gain factor α and added to the unmodified input signal. The delay time δ, states how long y'[n] is delayed before it is added to the unmodified input signal again. In a typical delay effect for an electric guitar δ can be

in the range from 1 millisecond up to 300 milliseconds. The delay effect can enrich a sound for short delay times up to 25 milliseconds. If the delay time is greater than 50 milliseconds the human ear can distinguish the unmodified input signal and its copy and we will hear an echo [Zö05]. As seen in Section 2.1 the feedback factor β must be smaller than one, otherwise we would get critically stable respectively unstable behavior. The mix factor α controls the amount of mixing the original signal x[n] with the delayed signal y'[n]. If α is bigger one, the delayed signal will sound louder as the unmodified input signal. Of course the samples of the output signal y[n] have to be in the same range as the input signal samples. For audio wave files these samples lie in the range from negative one to positive one. Thus the output samples have to be normalized with their maximum absolute value.

2.2.2. Frequency Domain

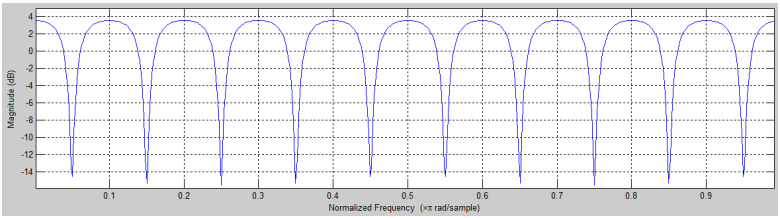
In the frequency domain the delay effect can be described with transfer function

$$H(z) = 1 + \frac{\alpha \cdot z^{-k}}{1 - \beta \cdot z^{-k}},\ k \geq 0,\ |\beta| < 1. \tag{2.6}$$

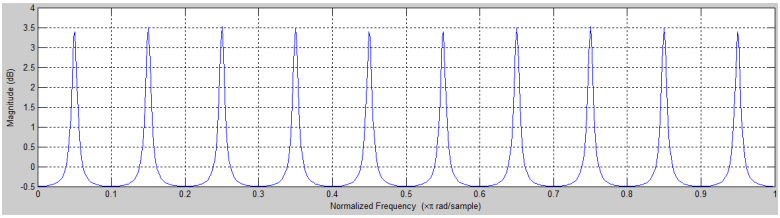
In Figure 2.3(a) and Figure 2.3(b) the magnitude responses of transfer function (2.6) are depicted with two different feedback factors β, but the same delay k and the same mix factor α. It's obvious that the spikes become narrower as β comes closer to one. Also the magnitude changes in the second case, as we get much less attenuation than with the smaller feedback factor. The magnitude response behaves in a similar way for different values of the mix factor α [Zö05]. We have already seen in Section 2.1, what happens for different delay times.

2.3. Chorus

The chorus effect has no feedback loop and thus can be implemented as a FIR comb filter. In Figure 2.4 the block diagram of the chorus effect is given.



(a) $\alpha = 1.5, \beta = 0.4$



(b) $\alpha = 1.5, \beta = 0.8$

Figure 2.3.: Magnitude Response of delay effect

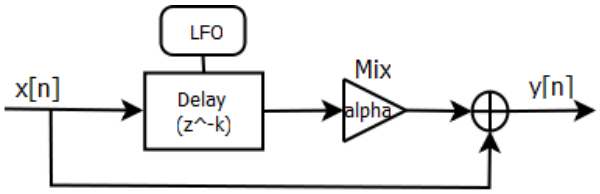


Figure 2.4.: Block diagram of chorus effect

We have a feed forward path which adds the input signal to the delayed signal. And we have a delay element which has variable delay time. The low frequency oscillator (LFO) which is described in Subsection 2.3.2 in detail, defines the current delay k, which changes over time with a certain frequency τ.

2.3.1. Time Domain

The corresponding difference equation of the chorus effect for the current delay k is given in (2.7). Since k changes over time also the different equation respectively the transfer function change too.

$$y[n] = x[n] + \alpha \cdot x[n-k],\ k \geq 0. \tag{2.7}$$

The input signal is delayed with the current delay k in samples, multiplied with the mix gain factor α and added to the unmodified input signal once more. This effect shall create the illusion of several instruments played concurrently to get a brighter and richer sound. Like in a chorus of a group of singers, it makes a single instruments sound like actually several instruments being played. To make the illusion a bit more realistic the total delay is slightly altered over time with a LFO. The total delay $\Delta = \frac{k}{f_s}$ in seconds is the sum of the constant delay δ and the variable delay ε, which is created with the LFO.

$$\Delta = \delta + \varepsilon$$

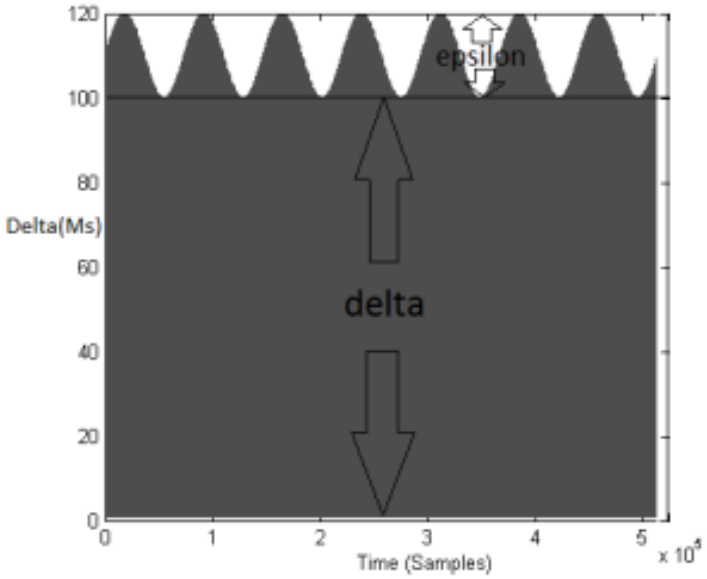
The total delay time typically lies in the range of 10 milliseconds up to 25 milliseconds. The mix factor α controls the amount of mixing the delayed signal with the unmodified input signal.

2.3.2. LFO

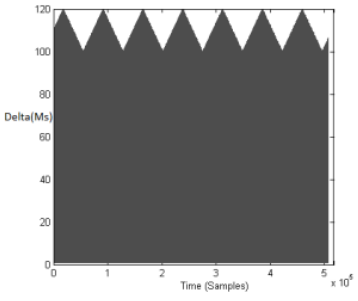
A low frequency oscillator (LFO) creates a periodic signal with frequency τ. In general one can choose the amplitude resp. the maximum value of the curve, the shape of the waveform and the frequency. Common wave forms for guitar effects are a sine curve, a triangle curve or an exponential curve. Common values for the frequency lie between 0.1 Hertz and 5 Hertz. Then the function value of the curve at a certain time equals the variable delay ε in seconds.

$$\varepsilon = LFO[n], 0 \leq n < N$$

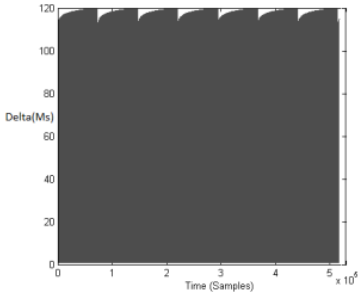
for the cycle duration $N = \frac{f_s}{\tau}$. In Figures 2.5(a), 2.5(b) and 2.5(c) the total delay Δ is given for three different wave forms. The constant delay δ is chosen to be 100 milliseconds and the maximum variable delay ε is 20 milliseconds. So the sine function has an amplitude of 10 milliseconds respectively it's function values are in the range from 0 up to 20milliseconds.



(a) Sinus Wave



(b) Triangle Wave



(c) Logarithmic Wave

Figure 2.5.: Total Delays with different LFO wave forms

2.3.3. Linear Interpolation

To get a delay $\Delta = \frac{k}{f_s}$ we just have to save the last k samples from the input signal in a memory buffer and read out the oldest one. Since Δ changes over time continuously linear interpolation between the discrete samples is needed [Dat97].

In Figure 2.6 a short example why linear interpolation is needed is given for a total delay $\Delta = 5.12$milliseconds resp. k = 1024 samples($f_s = 20000$). The constant delay δ is chosen to be 3 milliseconds and the maximum variable delay ε is 2.12

milliseconds. Thus Δ changes between 3 milliseconds and 5.12 milliseconds over time with frequency $\tau = 1$ Hertz. If the instantaneous delay is 4.696 milliseconds for example we would have to read out the 939.2th sample. But we only have samples of discrete times so we have to interpolate between sample $k = 939$ and sample $k+1 = 940$, which we do as suggested in [Dat97]:

$$v[k] = buffer[k+1] \cdot frac + buffer[k] \cdot (1 - frac)$$

with v[k] the interpolated sample at index k and factor $frac = 0.2$, the fractional part of the sample.

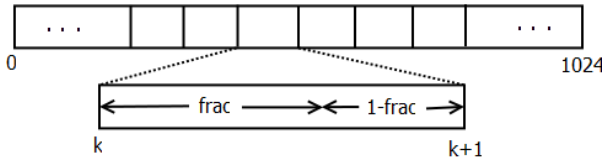


Figure 2.6.: Linear interpolation

2.3.4. Frequency Domain

In the frequency domain the chorus effect can be described by the following transfer function

$$H(z) = 1 + \alpha \cdot z^{-k},\ k \geq 0. \tag{2.8}$$

Since the total delay $\Delta = \frac{k}{f_s}$ is variable the transfer function changes over time. The variable delay together with linear interpolation produces small undulating changes in pitch [Dat97]. In Figure 2.7 six different magnitude responses are given. The delay k is changed from five up to ten samples without linear interpolation. However one can clearly see the notches are at different places, therefore attenuating and amplifying different frequencies. This causes a sweeping pitch change as time passes by.

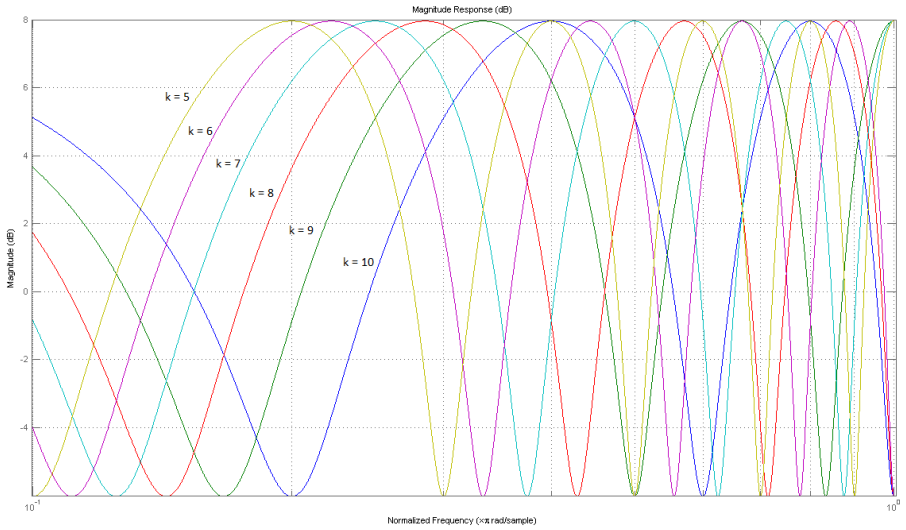


Figure 2.7.: Change of Magnitude Response for different delays Δ

2.4. Flanger

The flanger effect is a mixture of the delay effect from Section 2.2 and the chorus effect from Section 2.3. The block diagram of the flanger effect is given in Figure 2.8

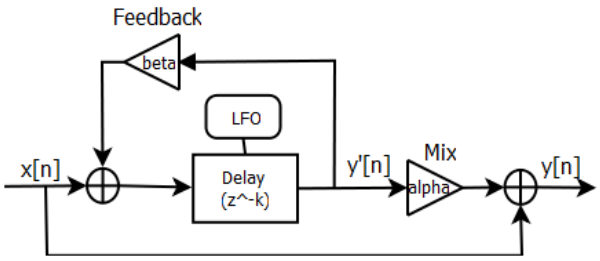


Figure 2.8.: Block diagram of flanger effect

It has both a feedback and a feed forward path. The feedback factor β was already described in Section 2.2, and the mix factor α was already described in Section 2.3. The IIR comb filter causes constructive and destructive interference. This effect can be simple described by looking at a sine wave for example. Imagine a perfect sine wave and add it to a delayed copy of the same sine wave. In one extreme if the phase shift is 0°, the amplitude of the output will be twice as high as that of the original sine wave. But on the other extreme if the

phase shift is 180°, the two waves will cancel each other and the output disappears. The total delay Δ is determined by a low frequency oscillator (LFO) as for the chorus effect. Since the delay changes over time, the interference changes too. This causes a jet plane-like whooshing sound.

2.4.1. Time Domain

The flanger effect has the same difference equation (2.5) as the delay effect 2.2. The only difference is the variable delay ε created with the LFO similar to the chorus effect in the previous Section 2.3. The total delay time Δ is usually short and lies in the range from 1 milliseconds up to 10 milliseconds. Of course the feedback factor β should be smaller than one again to avoid an unstable or critically stable system as described in Section 2.1. A large amount of feedback leads to a "metallic" sound.

2.4.2. Frequency Domain

Since the difference equation is the same as for the delay effect in Section 2.2, also the transfer function equals equation (2.6), except for the variable delay ε. The effects of the variable delay on the frequency spectrum has already been described in Section 2.3.

Chapter 3.

Spatial based effects

3.1. Introduction

Spatial effects shall create the illusion of a certain physical environment. One type of spatial effects is the panorama effect, which works by adjusting the relative amplitude of stereo loudspeakers. The listener gets the impression that the lower loudspeaker is farer away. This leads to a different perception of the own position with respect to the speakers. Another effect is the doppler effect as known from basal physics. It can be implemented by a simple pitch shifter [Zö05]. The position in space is simulated by intensity panning, delay lines and with special filters according to [VZA06]. A method to get interesting reverberation effects is described in Section 3.2. The reverb effect can enrich a dry studio record with an ambiance. There are many implementations using tapped delay lines for this effect. It usually takes very long and many parameter changes are necessary to get a realistic sound. But the easiest way is the convolution of the original signal with a room impulse response.

3.2. Reverb using Convolution

This section describes a simple method to get a reverberation effect from convolving the input signal with an artificial room impulse response. The convolution method leads to a very rich and realistic sound. One possible approach is to record many impulse responses from different places to get different sounding reverb. For example a big cathedral, a storehouse, a cave and so on. But it is very costly to take all this sound samples and it takes much memory to store all of them. This is a big disadvantage since usually a digital signal processor (DSP) does not

have enough memory to use this method. Therefore it makes sense to use an artificial impulse response as described in [Dob07a]. White gaussian noise with a decaying exponential function as envelope as shown in Figure 3.1 is very well suited to simulate a room impulse response.

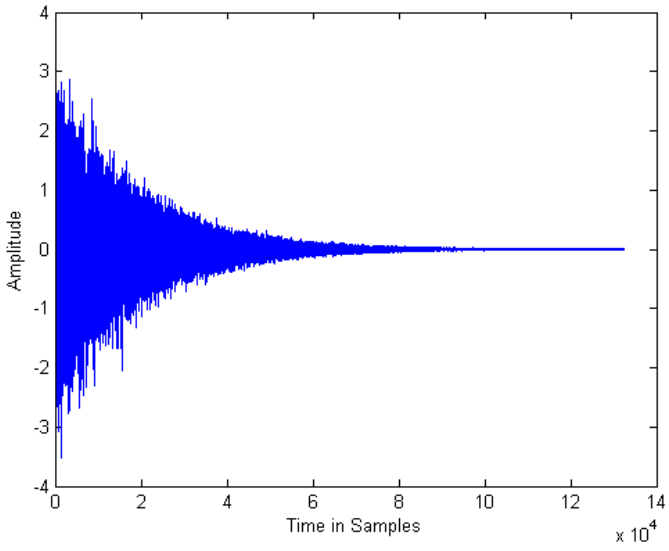


Figure 3.1.: Decaying white gaussian noise

Equation (3.1) describes the reverb effect in the time domain:

$$y[n] = x[n] * h[n] \tag{3.1}$$

The input signal x[n] is convolved with the Gaussian distributed white noise h[n] which has an exponentially decaying envelope. The term “white noise” refers to the fact that the random signal has a constant energy spectrum [Pre04]. The term Gaussian states that the signal is normal distributed with respect to the value. H[n] can be created as follows

$$h[n] = r[n] \cdot e^{-\alpha \cdot n}, n \geq 0. \tag{3.2}$$

In equation (3.2) r[n] is the Gaussian distributed white noise and the exponential function provides the exponential envelope. The factor α can be calculated by

$$\alpha = \frac{3 \cdot ln10}{\tau \cdot f_s}$$

in which the factor τ in seconds controls the length of the impulse response, f_s is the sampling frequency and *ln* is the natural logarithm. The time until the impulse response is zero again can be modified with the factor τ. A bigger τ leads to a longer room impulse response, this creates the illusion of a bigger room. In Chapter 5 an implementation using this approach is discussed.

3.2.1. Overlap-Add method

The overlap add method [Dob07b] is used in combination with the discrete fourier transformation (DFT). The DFT is used on discrete signals which have a finite number of points only and leads to a discrete line spectrum. The main application area of the DFT is the implementation on a DSP or a personal computer. The overlap-add method comes into play then block processing is needed. The input signal x[n] can be composed from a sum of input blocks. In general we have an infinite input signal which is processed block sequentially in non-overlapping blocks of signal samples. So the input signal can be composed in k blocks $x_m[n]$ with fixed size N_x, where k is infinite.

$$x[n] = \sum_{m=0}^{k \cdot N_x} x_m[n]$$

Now one input block $x_m[n]$ with size N_x is convolved with the room impulse response $h[n]$ with size N_h to get the corresponding output block $y_m[n]$.

$$y_m[n] = \sum_{k=0}^{N_h-1} h[k] \cdot x_m[n-k]$$

Because of the linearity of the convolution, the whole output signal can be described as the sum of the single output blocks.

$$y[n] = \sum_{m=0}^{k \cdot N_x} y_m[n]$$

The resulting output signal $y_m[n]$ of one block operation is $N_f = N_x + N_h - 1$ samples long. Of course the block size of the input blocks must equal the block size of the output blocks. Since the result of the convolution with two finite signals leads to a signal which is longer than each of the signals itself we have to overlap add the resulting output blocks $y_m[n]$. We also have to consider that with the DFT circular convolution [Dob07b] comes into play [1]. To achieve a

[1] Remember the shift theorem of the DFT

result which has the same length N_f as with linear convolution, we have to zero pad $x_m[n]$ and $h[n]$. In Figure 3.2 the overlap add method is illustrated.

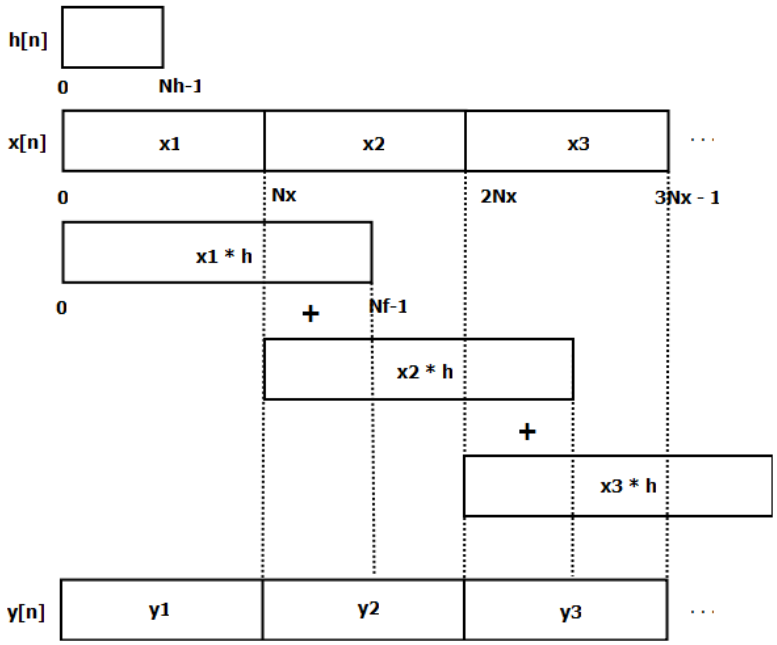


Figure 3.2.: Block processing with overlap add method

3.3. Reverb using comb filters

The oldest method for creating reverberation, which was already used in the seventies, is the use of combinations of comb filters as described in [Zö05]. This approach simulates the dispersion of acoustic waves in a room. In general there is a direct path from the sound source to the listener. But since the sound wave propagates in all directions, it is also reflected from the walls. So the listener will not only hear the sound from the direct path but also the early reflections a little later. Depending on the surface of the walls the reflected sound will be weaker as the direct sound. Of course a sound wave can be reflected several times before it finally reaches the listener. These reflections are called late reflections or diffuse reverberation. As we already know from Chapter 2 the ear can only distinguish between two sounds if the delay time is greater than 50 milliseconds. So echoes can be only perceived in big rooms where the reflections have to travel a far distance from the speaker to the listener. Figure 3.3 provides a graphical explanation.

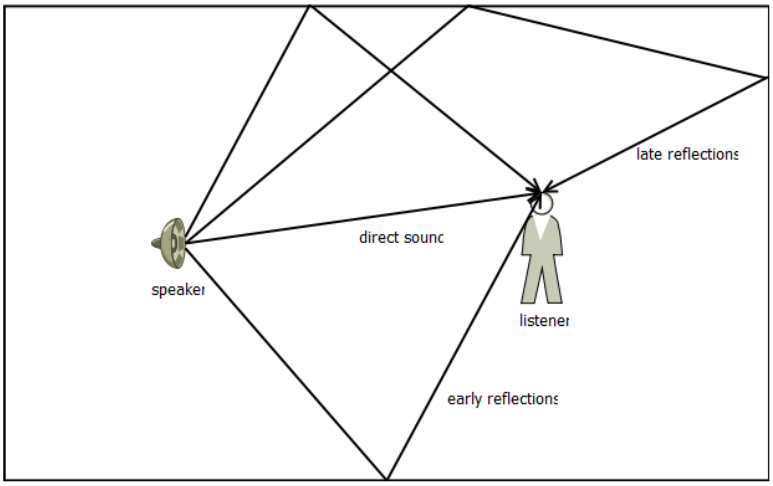


Figure 3.3.: Reverb model for classical approach

The early reflections can be simulated with tapped delay lines, since there is a fixed delay time from the speaker to the listener. But it's difficult to simulate the diffused reverberation, since the delay time changes dynamically according to the shape and the size of the room. The late reverberations decay exponentially until they reach a sound level the listener can't hear anymore. The time until the diffused reverberation is too low to be perceived by the human ear depends also on the surface of the walls.

Chapter 4.

Nonlinear effects

4.1. Introduction

This class of audio effects creates new harmonics or non harmonics in the frequency spectrum. Effects like distortion, overdrive or fuzz belong to this class. First we have to consider what the difference between linear and nonlinear systems is. Linear systems as introduced in [Pre04] must satisfy the superposition and scaling property. If we have a linear system S and two inputs $x_1[n]$ and $x_2[n]$. Then the outputs $y_1[n] = S(x_1[n])$ and $y_2[n] = S(x_2[n])$ results from inputting signals x_1 and x_2 separately to S. Now following relation must hold:

$$c_1 \cdot y_1[n] + c_2 \cdot y_2[n] = S(c_1 \cdot x_1[n] + c_2 \cdot x_2[n])$$

with coefficients c_1 and $c_2 \in \mathbb{R}$. For nonlinear system the superposition property in general doesn't hold [Pre04]. As an example we consider the nonlinear system with the input-output relationship

$$y[n] = u[n]^2$$

with y[n] the output signal and u[n] the input signal. Consider an input signal $u[n] = \cos(\frac{2\cdot\pi\cdot f_1}{f_s} \cdot n) + \cos(\frac{2\cdot\pi\cdot f_3}{f_s} \cdot n)$. Thus a periodic discrete input signal with fundamental frequency $\Theta_0 = \frac{2\cdot\pi\cdot f_1}{f_s} = \frac{2\cdot\pi}{N}$, third harmonic $\Theta_3 = \frac{2\cdot\pi\cdot f_3}{f_s}$ and cycle duration N.

The corresponding output signal y[n] [1] results in:

[1] This can be shown with $\cos(\frac{2\cdot\pi\cdot f}{f_s} \cdot n) = \frac{1}{2} \cdot (e^{j\cdot\frac{2\cdot\pi\cdot f}{f_s}\cdot n} + e^{-j\cdot\frac{2\cdot\pi\cdot f}{f_s}\cdot n})$

$$y[n] = \frac{1}{2} \cdot \cos(2 \cdot \Theta_0 \cdot n) + \cos((\Theta_0 + \Theta_3) \cdot n) + \cos((\Theta_0 - \Theta_3) \cdot n) + \frac{1}{2} \cdot \cos(2 \cdot \Theta_3 \cdot n) + 1 \quad (4.1)$$

The output signal consists of a constant component, a new harmonic with twice the fundamental frequency, a new harmonic with twice the frequency of the third harmonic, and two components that are the sum and difference of the fundamental frequency and the frequency of the third harmonic of the input signal. It follows that with a nonlinear function with power ϑ we can create new harmonics. If the input signal has multiple frequency components as in our example sum and differences of these frequency components occur in the spectrum which sounds very disharmonious. In Figure 4.1 the input signal $u[n] = \cos(\frac{2 \cdot \pi \cdot f_1}{f_s} \cdot n) + \cos(\frac{2 \cdot \pi \cdot f_2}{f_s} \cdot n)$ is illustrated in time domain.

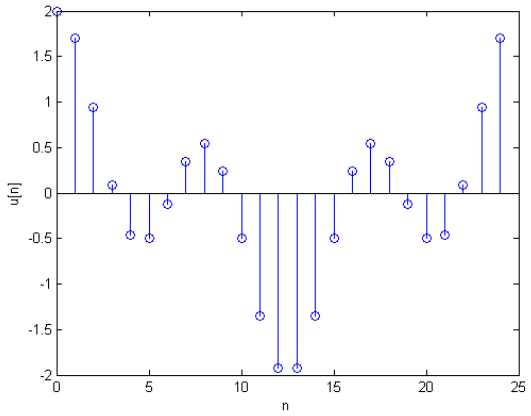
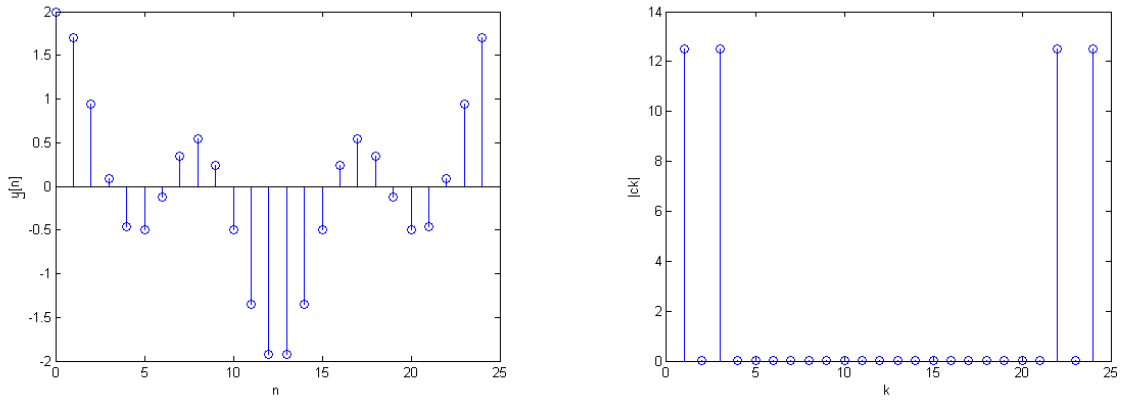


Figure 4.1.: Input signal u[n]

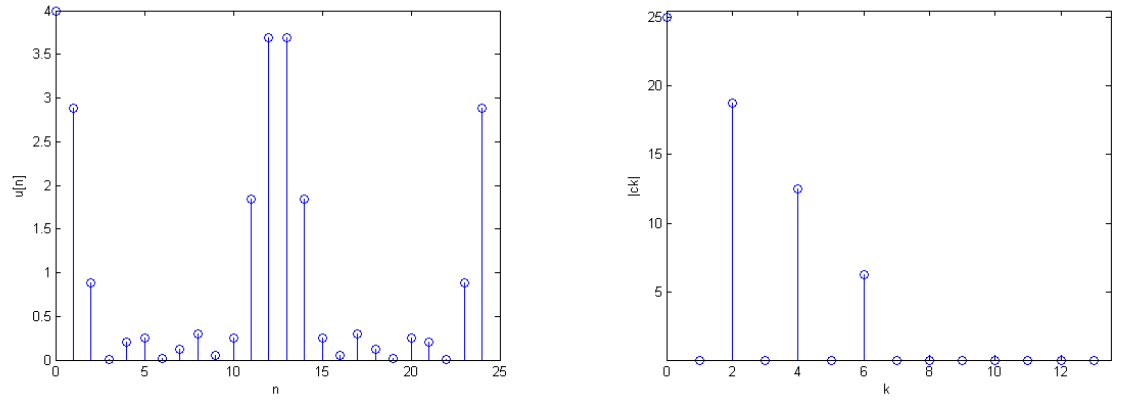
In Figure 4.2(a) and Figure 4.2(b) the output y[n] for a linear system with input u[n] is given in the time and in the frequency domain. In a linear system the shape of a sine wave and through superposition also the shape of a sum of sine waves is not changed but only the amplitude may be changed.



(a) Output signal (b) Spectrum

Figure 4.2.: Linear System

On the opposite the output for the nonlinear system $y[n] = u[n]^2$ is given in Figure 4.3(a) and Figure 4.3(b). Now additional spectral lines at frequencies calculated in equation (4.1) occur. This kind of test for nonlinear distortion is also introduced in [Jon03]. If we feed a sine wave into a nonlinear system we can take the output and measure the total harmonic distortion (THD), which is defined as the square root of the ratio of the sum of powers of all harmonic frequencies to the power of all harmonic frequencies including the fundamental frequency. The higher the THD value, the higher is the nonlinear distortion. Of course for a linear system it is zero.



(a) Output signal (b) Spectrum

Figure 4.3.: Nonlinear System

A guitar tone can be composed in its fourier components[DGKP08]. It consists of the fundamental frequency and all odd or even harmonics.Thus it shows the same characteristics as the sum of sine waves in a nonlinear system. This is remarkable since in basic physics stroking a single string don't generate sine waves.

4.2. Distortion

The distortion effect plays a central role in rock music. It creates a "dirty" distorted sound that is typical for rock bands. A good example for physical distortion is overdriving a valve amplifier. Imagine the range of the amplifier goes up to 100mV [1] and it amplifies an input signal by a factor of two. If the maximum value of an input signal is 10mV, for example, the maximum value of the output signal will be 20mV. But if the maximum value of the input signal is bigger than 50mV the output signal will be clipped. That means that all parts of the input signal bigger 50mV are exactly 100mV in the output signal. This is a nonlinear operation which creates new harmonics in the spectrum. A valve amplifier has a very smooth transition from amplifying to clipping the signal. It distorts all audio frequencies the same because of its transfer characteristics [Jon03]. This smooth transition function creates a very "warm and creamy sound". In Figure 4.4, taken from [Zö05], the curve of a pentode output current I_a over the pentode input voltage U_g is given. One can clearly see the soft transition from the attenuation to the clipping zone. Digital distortion effects try to copy this transfer characteristics. But in hight-quality music production valve amplifiers are still used. Either because of tradition or because of individual preference. A fact is that valve amplifiers deliver the best sound when they work at operating temperature. So it takes a while until they are ready for use. This characteristic of valve amplifiers is described in [Jon03]. Of course a digital distortion effect doesn't have this disadvantage. On the other hand digital distortion sometimes produces a metallic and artificial sounding result that is bad either.

[1] mV stands for millivolts

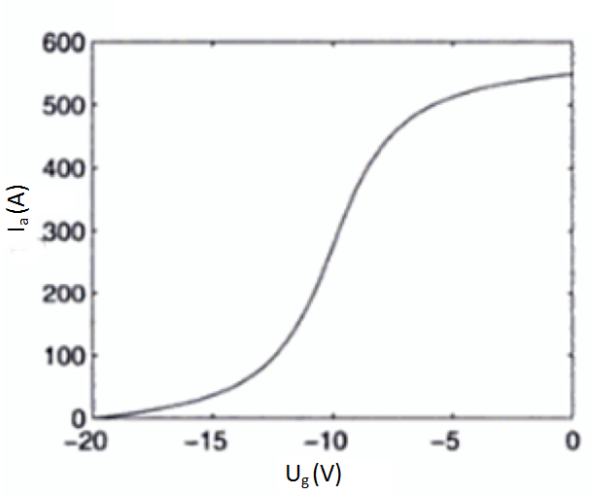


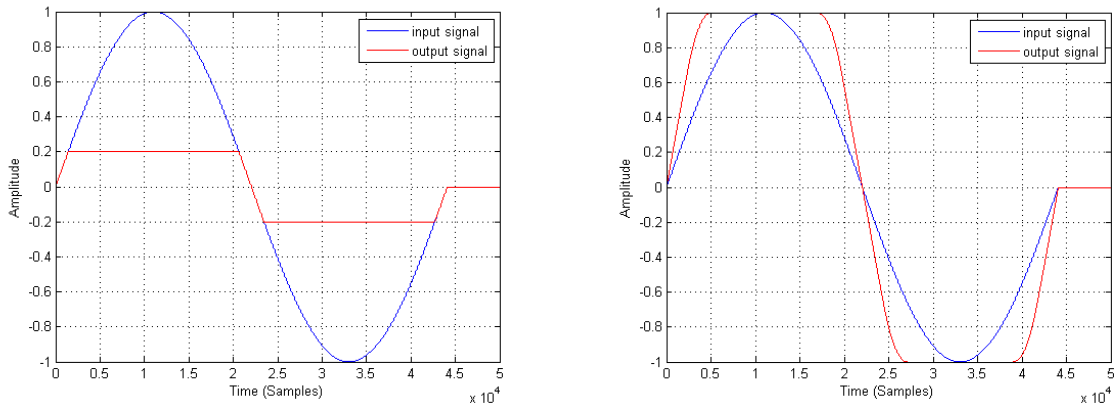
Figure 4.4.: Characteristics of a pentode

4.2.1. Hard Clipping

A digital signal can be clipped using threshold values. The easiest way to clip a signal is hard clipping. A mathematical description is given by

$$y[n] = \begin{cases} -th & x[n] < th \\ x[n] & -th \leq x[n] \leq th \\ th & x[n] > th \end{cases} \tag{4.2}$$

where *th* is called threshold value and x[n] is the normalized input signal, which are both in the range from negative one to positive one. In Figure 4.5(a) and Figure 4.5(b) the intended effect is illustrated in the time domain with a 1Hz sine wave as an input signal and $th = 0.2$.



(a) Hard Clipping

(b) Soft Clipping

Figure 4.5.: Comparison: Hard and Soft Clipping in time domain

As shown in Section 4.1 nonlinear functions create additional harmonics. The smaller the threshold value *th* is, the more of the input signal is clipped. This leads to a sharp distorted sound caused by the sharp transition from clipping the input signal to passing it to the output unchanged.

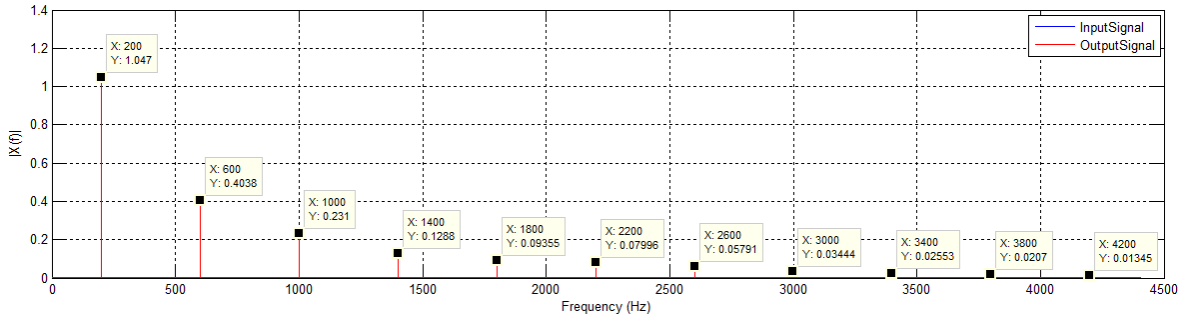
4.2.2. Soft Clipping

In [Zö05] a possible approach for soft clipping with the following input output relation is given:

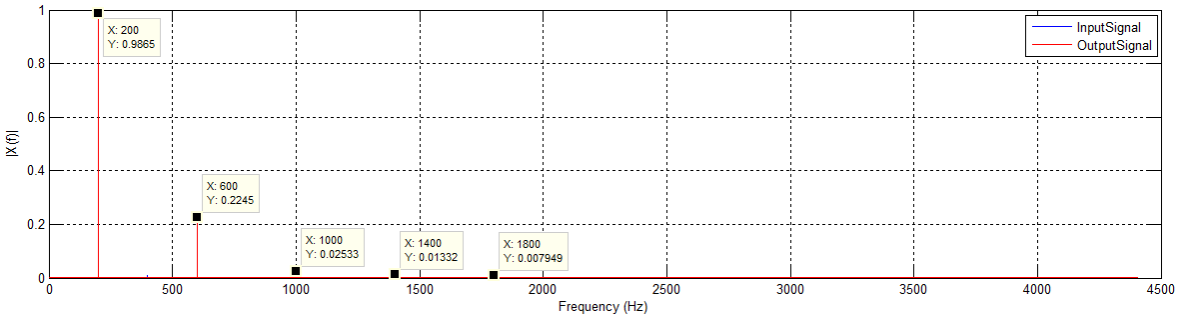
$$y[n] = \begin{cases} -1 & -1 < x[n] < -\frac{2}{3} \\ \frac{3-(2+3\cdot x[n])^2}{3} & -\frac{2}{3} \leq x[n] \leq -\frac{1}{3} \\ 2 \cdot x[n] & -\frac{1}{3} \leq x[n] < \frac{1}{3} \\ \frac{3-(2-3\cdot x[n])^2}{3} & \frac{1}{3} \leq x[n] \leq \frac{2}{3} \\ 1 & \frac{2}{3} < |x[n]| \leq 1 \end{cases} . \tag{4.3}$$

The normalized input signal x[n] is separated into five intervals. In Figure 4.5(b) the effect is illustrated with the sine wave from Figure 4.1 as input signal. This effect creates a "warmer smoother" sound since the transition from clipping the input signal to passing it to the output unchanged is rounded now. This causes a much slower decrease and increase of harmonics

as with hard clipping 4.2.1. In fact with hard clipping harmonics appear with a much higher level [Zö05]. In Figure 4.6(a) and Figure 4.6(b) hard and soft clipping in the frequency domain is illustrated with a 200Hz sine wave as input signal and $th = 0.1$.



(a) Hard Clipping



(b) Soft Clipping

Figure 4.6.: Comparison: Hard and Soft Clipping in frequency domain

Chapter 5.

Implementation

5.1. Introduction

Implementing digital effects with Matlab is rather comfortable as it provides good debugging abilities. In the current Matlab version 7.8.0.347 object-oriented programming is also supported. The only problem is the real-time capability since the object oriented approach causes a huge overhead. It takes Matlab much longer to handle objects instead of simply scripting the Matlab commands one after another. Since it was not the intention to add real-time capability to this first version of the project, not much effort was put in optimizing the code. Instead, the object oriented approach was chosen to provide reusable and readable code. In the following sections the implementation of the effects with Matlab is explained. Also a short introduction to the graphic user interface (GUI) and it's functions is given in Section 5.3.

5.2. Matlab Objects

In this section an overview of the Matlab class files, their implementation and their use in the project is given. All effects are implemented with block processing, since Matlab is optimized for Matrix operations and to leave the possibility to optimize for real-time operation later. This is also the best option for porting to a digital signal processor (DSP), since a DSP is optimized for block processing too.

5.2.1. AnalogIO Class

The *AnalogIO.m* class consists of the Matlab functions to provide read and write access to a hardware sound device. With the following code 5.1 an *AnalogIO* object with an input and an output handle is created and initialized with the block size in samples and the sample rate in hertz.

Listing 5.1: *AnalogIO.m* constructor

```
function obj=AnalogIO(sampleRate, blockSize)
    %create soundcard io handles
    obj.input_handle = analoginput('winsound');
    obj.output_handle = analogoutput('winsound');

    %initialize Blocksize and Samplerate
    obj.blockSize=blockSize;
    obj.sampleRate=sampleRate;

    %configure for stereo io
    ch = addchannel(obj.input_handle,[1 2]);
    addchannel(obj.output_handle,[1 2]);

    %set trigger channel
    set(obj.input_handle,'TriggerChannel',ch(1))

    %trigger samples continually
    set (obj.input_handle, 'SamplesPerTrigger', obj.blockSize);
    set(obj.input_handle, 'TriggerType', 'immediate');
    set(obj.output_handle, 'TriggerType', 'immediate');

    %set sample rate
    set (obj.input_handle, 'SampleRate', obj.sampleRate);
    set (obj.output_handle, 'SampleRate', obj.sampleRate);
end
```

The methods *getSamples(obj)* and *putSamples(obj, samples)*, can be used to get a block of samples from the sound device, respectively send a processed block of samples back to the sound device. In fact this class has not been used for the multi effects unit GUI yet, since the Matlab functions do not support real time operation. Maybe this problem can be solved using custom Matlab executable (MEX) files. That allow the execution of C, C++ or the formula translation programming language (Fortran) code with Matlab. Since these programming lan-

guages provide better low level hardware support, a custom driver for the sound device with real time capabilities could be written. But this is not the intention of the overall project since a maximum portability of the multi effects unit shall be ensured. In fact the multi effects unit platform will work on all operation systems that are supported by Matlab.

5.2.2. Delay Class

The delay effect from Section 2.2 can be implemented with the Matlab filter function [1]. This function takes a discrete filter transfer function with nominator and denominator like in equation (2.6). Then it applies the filter to an input vector. For block processing also an internal state is needed. The Matlab help function provides further information on the filter function.

In Listing 5.2 the main routine of *DelayFilter.m* is given. This class has an attribute *transferFunction.num* to save the nominator and an attribute *transferFunction.den* to save the denominator of the filter transfer function. In Listing 5.3 the initialization of the transfer function of the *DelayFilter* object, with descending powers of z is given. Every time the start method is called with a *DelayFilter* object and an input signal with size block size, it returns the filtered output signal.

Listing 5.2: Start method(*DelayFilter.m*)

```
function outputSignal = start(obj,inputSignal)
    %state of last block
    persistent internalState;
    [outputSignal, internalState] = filter(obj.transferFunction.num, obj.
        transferFunction.den, inputSignal, initialCondition);
end
```

Listing 5.3: Initialize transfer function method(*DelayFilter.m*)

```
function transferFunction = setTranferFunction(obj)
    % initialize transfer function
    transferFunction.num = [obj.dry zeros(1,obj.delaySamples-1) (obj.wet -
         obj.dry * obj.feedback)];
    transferFunction.den = [1  zeros(1,obj.delaySamples-1)  -obj.feedback
        ];
end
```

[1] Y = FILTER(B,A,X)

A second order transfer function with a short delay of two samples and the corresponding declaration of the nominator and the denominator vector would look as follows:

$$H[z] = \frac{dry * z^2 + (wet - feedback * dry)}{z^2 - feedback}$$

$$num = [dry \;\; 0 \;\; (wet - feedback * dry)]$$

$$den = [dry \;\; 0 \;\; -feedback]$$

This version of the delay effect has not been used for the multi effects unit GUI because the filter function is very slow for transfer functions of high degrees. Since the degree of the transfer function equals the delay in samples, usually transfer functions with very high degree have to be processed. Hence an implementation in the time domain with a cyclic buffer is preferable. So the *Delay.m* class was used instead. The constructor of the delay class takes following input arguments:

function obj=Delay(sampleRate, blockSize, mix, delayMs, feedback)

Table 5.1.: *Delay.m* constructor arguments

argument	**description**	**unit**
sampleRate	sample rate f_s	Hz
blockSize	block size	samples
mix	mix factor α	%
delayMs	delay Δ	ms
feedback	feedback factor β	%

This class implements the delay using a cyclic buffer in time domain. This approach is also chosen for the chorus 2.3 and the flanger 2.4 effect. There was an attempt to implement a generic cyclic buffer class, and to use it for all effects. But receiving the samples from the memory scope of another object was very slow because Matlab only supports call by value. The size of the cyclic delay buffer is a multiple of the block size. It depends on the delay, the sample rate and of course of the block size.

Listing 5.4: Initialization of cyclic buffer

```
1 %Buffer for delay must be multiple of blockSize, we need one additional
      block for ring buffer
2 obj.totalBuffer = zeros((ceil((obj.delayMs)/1000 * obj.sampleRate / obj.
      blockSize)+1)*obj.blockSize,1);
```

In Listing 5.4 the initialization of the *totalBuffer* is given. First Δ(in milliseconds) is converted to delay in samples with $\Delta = \frac{k}{f_s}$. Next we have to look how many blocks are needed. Finally we need one additional block to implement a ring buffer. This is because we have to be able to refresh the buffer before reading out the delayed samples. If we wouldn't have an additional block we would overwrite the oldest stored samples with the newest input samples all the time.

Listing 5.5: Refresh method for cyclic buffer(*Delay.m*)

```
function obj = refreshTotalBuffer(obj, inputBlock)
    %increment pointer modulo bufferlength
    obj.nextBlockPointer = mod(obj.nextBlockPointer, length(obj.
        totalBuffer)/obj.blockSize)+1;

    %get delay in samples
    delaySamples = floor(obj.delayMs /1000 * obj.sampleRate);

    %get length of buffer
    totalBufLength = length(obj.totalBuffer);

    %fill next buffer slot with lenght blockSize
    for k = 1:obj.blockSize

        %get index with delay
        index = (obj.blockSize*(obj.nextBlockPointer-1)+ k);
        indexDelay = index - delaySamples;

        %correct negativ index
        if (indexDelay <= 0)
            indexDelay  = totalBufLength + indexDelay;
        end

        %before delay: mix input with delayed sample and refresh buffer
        obj.totalBuffer(index) = inputBlock(k) + obj.feedback * obj.
            totalBuffer(indexDelay);
    end
end
```

In Listing 5.5 the *refreshTotalBuffer(obj, inputBlock)* method to save the next block of samples to the buffer is given. The *nextBlockPointer* points to the first index of the *totalBuffer* where the next block of input samples should be written. It has to be updated every time the method is executed. Since it's a cyclic buffer it has to be updated modulo the buffer length. The

constant delay in milliseconds is converted to delay in samples. In the for loop the current input block is processed sample by sample. The newest input sample is taken and added to the delayed sample at index *indexDelay* which is augmented with the feedback factor β. This is how the feedback loop is implemented. Since the buffer length is a multiple of block size, no modulo operation is needed to iterate through the buffer. But if *delaySamples* subtracted from index is smaller than zero we have to wrap around to the end of the buffer and get the fitting samples from the last block of the buffer. This isn't done with a modulo operation because it happens quite often and would slow down the execution speed.

Listing 5.6: Get method for cyclic buffer(*Delay.m*)

```
1
2  %returns next blockSize delayed samples
3  function samples = getTotalBuffer(obj)
4      %get delay in samples
5      delaySamples = floor(obj.delayMs /1000 * obj.sampleRate);
6
7      %get length of buffer
8      totalBufLength = length(obj.totalBuffer);
9
10     %initialize
11     samples = zeros(obj.blockSize,1);
12
13     %fill next buffer slot with lenght blockSize
14     for k = 1:obj.blockSize
15
16         %get index with delay
17         index = (obj.blockSize*(obj.nextBlockPointer-1) + k);
18         indexDelay = index - delaySamples ;
19
20         %correct negativ index
21         if (indexDelay  <= 0)
22             indexDelay  = totalBufLength + indexDelay ;
23         end
24
25         samples(k) = obj.totalBuffer(indexDelay);
26     end
27 end
```

In Listing 5.6 the *getTotalBuffer(obj)* method to receive a block of samples from the buffer is illustrated. The *nextBlockPointer* points to the same index as in the refresh method since it is only updated there. The constant delay is converted to *delaySamples* again. The for loop

iterates through the buffer from index *nextBlockPointer* up to index *nextBlockPointer* plus block size. The constant delay *delaySamples* is subtracted from the index. Negative indexes are corrected, without a modulo operation again. The output sample at index k just equals the buffer sample at index *indexDelay*. Finally a block of processed samples is returned.

Listing 5.7: Start method (Delay)

```
function [obj, outputBlock] = start(obj,inputBlock)
    %save next input block
    obj = refreshTotalBuffer(obj, inputBlock);

    %mix block after delay with original signal
    outputBlock  = inputBlock + obj.mix * getTotalBuffer(obj);
end
```

Now the input output relation can be implemented in the *start(obj,inputBlock)* method given in Listing 5.7. The *start* method is executed for every new block of new input samples. It calls the *refreshTotalBuffer* method with the input samples. As stated out before this method implements the feed back path. Afterwards the output samples are created adding the input samples to the samples from the *getTotalBuffer* method which are augmented with the mix factor α. This is how the feed forward path is implemented.

5.2.3. Chorus Class

The chorus effect from Section 2.3 is implemented in the *Chorus.m* class. Since the block diagram of the chorus effect 2.4 is quite similar to the block diagram of the delay effect 2.2 the chorus.m class is a modification of the delay class 5.2.2. So for a more accurate description of the Matlab code the reader is referred to Section 5.2.2. The constructor of the chorus class 5.2 takes following input arguments:

> function obj=Chorus(sampleRate, blockSize, mix, delayMs, depthMs, rateHz, lfoShape)

Table 5.2.: *Chorus.m* constructor arguments

argument	description	unit
sampleRate	sample rate f_s	Hz
blockSize	block size	samples
mix	mix factor α	%
delayMs	constant delay δ	ms
depthMs	variable delay ε	ms
rateHz	frequency τ of the LFO	Hz
lfoShape	shape of the LFO curve	integer

The main difference between the delay and the chorus class is the variable delay, which has to be implemented. For this purpose a LFO object 5.2.5 belongs to the chorus class attribute's. Again a buffer which is a multiple of block size is used to implement the delay. Since the delay consists of the variable delay ε and the constant delay δ, the buffer must be big enough to capture the total delay Δ. Another difference to the delay effect is that this time there is no feedback loop. That's why we don't have to save the new input block sample by sample to the buffer like in the delay class. Instead we can just save the whole input block in one turn in the *refreshTotalBuffer* method. This is an effort since Matlab is optimized for block processing [TM10].

Listing 5.8: Refresh method for cyclic buffer(*Chorus.m*)

```
function obj = refreshTotalBuffer(obj, inputBlock)
    %increment pointer modulo bufferlength
    obj.nextBlockPointer = mod(obj.nextBlockPointer, length(obj.
        totalBuffer)/obj.blockSize)+1;

    %fill next buffer slot of length blockSize
    obj.totalBuffer(1+obj.blockSize*(obj.nextBlockPointer-1):obj.blockSize
        *obj.nextBlockPointer) = inputBlock;
end
```

In Listing 5.8 the *refreshTotalBuffer(obj, inputBlock)* method is illustrated. The *nextBlock-Pointer* has to be updated modulo the buffer length before the new block of samples can be saved to the buffer.

Listing 5.9: Get methods for cyclic buffer(*Chorus.m*)

```
 %returns next blockSize delayed samples
function [obj samples] = getTotalBuffer(obj)
    %get next blockSize of variable delays
```

```
4       [obj.lfoObject, lfo] = obj.lfoObject.getLookup(obj.blockSize);
5
6       %get one block of Delays
7       totalDelaySamples = (obj.delayMs + obj.depthMs * lfo)/1000 * obj.
            sampleRate;
8
9       %get length of buffer
10      totalBufLength = length(obj.totalBuffer);
11
12      %initialize
13      samples = zeros(obj.blockSize,1);
14
15      %fill next buffer slot with lenght blockSize
16      for k = 1:obj.blockSize
17          %get index with delay
18          index = (obj.blockSize*(obj.nextBlockPointer-1)+ k) -
                totalDelaySamples(k);
19
20          %correct negativ index
21          if (index < 0)
22              index = totalBufLength + index;
23          end
24
25          %linear interpolation
26          fixedIndex = floor(index);
27          frac = index - fixedIndex;
28
29          if(fixedIndex == totalBufLength || fixedIndex == 0)
30              samples(k) = (1-frac) * obj.totalBuffer(totalBufLength) +
                    frac * obj.totalBuffer(1);
31          elseif (fixedIndex > totalBufLength)
32              samples(k) = (1-frac) * obj.totalBuffer(fixedIndex -
                    totalBufLength) + frac * obj.totalBuffer(fixedIndex + 1 -
                    totalBufLength);
33          else
34              samples(k) = (1-frac) * obj.totalBuffer(fixedIndex) +  frac *
                    obj.totalBuffer(fixedIndex + 1);
35              end
36          end
37  end
```

In Listing 5.9 the *getTotalBuffer(obj)* method of the chorus class is given. The *getTotalBuffer* method of the chorus class is a bit more complex than the according method of the delay class. This is because the variable delay described in Section 2.3 has to be implemented with the low frequency oscillator (LFO) from Subsection 2.3.2. The implementation of the *Lfo.m* class is illustrated in Section 5.2.5. First a block of variable delays ε in milliseconds for every discrete sample k is retrieved from the Lfo.m class. Then it is added to the constant delay δ and converted to total delays Δ in samples. These total delays in samples in general have a fractional part depending on the sample rate and the frequency. Hence linear interpolation as described in Subsection 2.3.3 is needed. For this purpose the fractional part *frac* of the total delay at index k is separated from the integer part *fixedIndex*. The resulting delayed sample at index k is a mixture of the sample at *fixedIndex* multiplied with (1-*frac*) and the sample at *fixeIndex* plus one multiplied with *frac*. A negative index has to be corrected first, to replace the modulo operation.

Listing 5.10: Start method(*Chorus.m*)

```
function [obj,outputBlock] = start(obj,inputBlock)
    %save next input block to buffer
    obj = refreshTotalBuffer(obj,inputBlock);

    [obj, samples] = getTotalBuffer(obj);

    %process next output block
    outputBlock  = inputBlock + samples;
end
```

The input output relation is implemented in the *start(obj,inputBlock)* method given in Listing 5.10. It equals the start method from the delay class 5.7. But this time the *getTotalBuffer* implements no feedback path. Another difference is that the *getTotalBuffer* method refreshes the LFO. That's why a temporary variable samples is needed, to retrieve both the refreshed chorus object and the output samples.

5.2.4. Flanger Class

As stated in Section 2.4 the flanger effect is a mixture of the delay and the chorus effect. Hence the *Flanger.m* class uses source code from Subsection 5.2.2 and Subsection 5.2.3. As discussed above in Section 2.4 the flanger effect has a feed back and a feed forward path. It

also has a variable delay ε like the chorus effect. The constructor of the flanger class 5.3 takes following input arguments:

function obj=Flanger(sampleRate, blockSize, mix, delayMs, depthMs, feedback, rateHz, lfoShape)

Table 5.3.: *Flanger.m* constructor arguments

argument	**description**	**unit**
sampleRate	sample rate f_s	Hz
blockSize	block size	samples
mix	mix factor α	%
delayMs	constant delay δ	ms
depthMs	variable delay ε	ms
feedback	feedback factor β	%
rateHz	frequency τ of the LFO	Hz
lfoShape	shape of the LFO curve	integer

The *refreshTotalBuffer(obj, inputBlock)* and the *start(obj,inputBlock)* methods are more or less the same as the according methods in Listing 5.5 and Listing 5.7 of the delay class in Section 5.2.2. The *getTotalBuffer(obj)* method equals the according method in Listing 5.9 of the chorus class in Section 5.2.3.

5.2.5. LFO Class

The *Lfo.m* class implements the LFO from Subsection 2.3.2. It is used for the chorus and the flanger effect from the previous subsections. The constructor of the *Lfo.m* class 5.4 takes following input arguments:

function obj=Lfo(sampleRate, rateOfChange, waveShape, amplitude)

Table 5.4.: *Lfo.m* constructor arguments

argument	**description**	**unit**
sampleRate	sample rate f_s	Hz
rateofChange	frequency of the periodic wave f_s	Hz
waveShape	shape of the periodic wave f_s	integer
amplitude	amplitude of the periodic wave f_s	integer

The lookup tables for the math functions are created on object creation. The length of the tables depends on the required sample rate and the desired frequency of the periodic wave. Of course only one period of the math function has to be sampled. If the wave frequency is changed later the *refreshLookup(obj)* method has to be called again, to sample a new lookup table.

Listing 5.11: Lookup method(*Lfo.m*)

```
function obj = refreshLookup(obj)
    %constant?
    if (obj.rateOfChange == 0)
        obj.lookup = obj.amplitude;
    else
    switch  obj.waveShape
        %lookup for sin wave
        case 1
            %calculate sin values
            lut = sin(2*pi*obj.rateOfChange/obj.sampleRate*(0:(obj.
                sampleRate/obj.rateOfChange)-1));
            %do scaling
            obj.lookup = (lut + 1)*obj.amplitude;

    %lookup for triangle wave
        case 2
            lut = 0:(1/4)*obj.sampleRate/obj.rateOfChange-1;
            lut = [lut (1/4)*obj.sampleRate/obj.rateOfChange:-1:(-1/4)*obj
                .sampleRate/obj.rateOfChange+1];
            lut = [lut (-1/4)*obj.sampleRate/obj.rateOfChange:obj.
                amplitude-obj.rateOfChange/obj.sampleRate];

            %do scaling
            obj.lookup = (lut / max(lut) * obj.amplitude) + obj.amplitude;
            %lookup for log wave
        case 3
            %calculate exp values
            lut = exp(2*pi*obj.rateOfChange/obj.sampleRate*(1:(obj.
                sampleRate/obj.rateOfChange)));

            %do scaling
            lut = (lut - min(lut));
            obj.lookup =  lut * 2*obj.amplitude / max(lut);
        case 4
            %calculate log values
```

```
32              lut = log(2*pi*obj.rateOfChange/obj.sampleRate*(1:(obj.
                    sampleRate/obj.rateOfChange)));
33
34              %do scaling
35              lut = (lut - min(lut));
36              obj.lookup =  lut * 2 * obj.amplitude / max(lut);
37
38          %constant
39          otherwise
40              obj.lookup = obj.amplitude;
41          end
42      end
43  end
```

The *refreshLookup(obj)* method can create lookup tables for several different wave forms. Depending on the *waveShape* attribute, a lookup for a sine wave, a triangle wave an exponential wave or a logarithmic wave form can be created. For the triangle wave, three linear functions which rise resp. fall with 45° are used to build the course of the function. All other lookups can be created with standard Matlab functions [1]. Since the logarithmic and the exponential wave are no periodic functions they just rise for the duration of one sine period before they are set to zero again. After sampling all functions have to be scaled to the *amplitude* attribute. This attribute gives the maximum deflection of the wave in one direction.

5.2.6. Reverb Class

There are several versions of the reverb class. *ReverbV1.m* implements the simple convolution approach without early and late reflections. The following code samples are taken from the *ReverbV3.m* class, which uses both, on the one hand the convolution method described in Section 3.2 on the other hand the delay based approach described in Section 3.3.

function obj=ReverbV3(sampleRate, blockSize, reverbTimeMs, earlyPreDelayMs, latePreDelayMs, damping)

[1] See the Matlab Help for further information. SIN(X), LOG(X), EXP(X)

Table 5.5.: *ReverbV3.m* constructor arguments

argument	**description**	**unit**
sampleRate	sample rate f_s	Hz
blockSize	block size	samples
reverbTimeMs	reverb time τ determines length of artificial pulse response	ms
earlyPreDelayMs	time before first reflection reaches listener	ms
latePreDelayMs	time before diffuse reflections start	ms
damping	attenuation factor for early and late reflections	%

Listing 5.12: Method to get artificial room impulse response

```
%returns a artificial pulse response
function obj = setPulseResponse(obj)
    %get length in samples
    responseLength = ceil(obj.reverbTimeMs/1000*obj.sampleRate);
    preDelayLength = ceil(obj.latePreDelayMs/1000*obj.sampleRate);

    alpha = 3*log(30)/(responseLength);

    %decay is zero till late reflections
    decay = [zeros(1,preDelayLength) exp(-alpha*(preDelayLength:
        responseLength-1))]';

    %Nf = L+Nh-1;
    fftSize = obj.blockSize + responseLength - 1; %in samples

    obj.pulseResponse = fft(ReverbV3.getGauss(responseLength) .* decay,
        fftSize);
end

%returns blockSize gaussian distributed random numbers with zero mean
function  gauss = getGauss(length)
    randNums=rand(length,2);  %generate 2 * blockSize random numbers [0,1]
    gauss = cos(2*pi*randNums(:,1)) .* sqrt(-2 * log(randNums(:,2))); %box
        -muller transformation

end
```

In the *getPulseResponse(reverbTime, sampleRate)* method first the *responseLength* and the *preDelayLength* in samples are calculated. Then the impulse response is composed from a zero part, to simulate the pre delay, and an exponential decaying part. The exponential decay with length in samples is created with the Matlab exponential function [1] . The white Gaussian noise is created with the Matlab random function [2]. This function creates uniform distributed random numbers in the range from zero to one. The Box Muller Transformation [Dut05] is needed to convert the uniform distributed numbers to Gaussian distributed numbers. The FFT size N_f is determined by the *blocksize* N_x and the *responseLength* N_h. Finally the pulse response in the frequency domain can be created multiplying the decay with the white noise and applying the discrete fourier transformation (DFT) to the result.

Listing 5.13: Convolves input with white noise(*ReverbV3.m*)

```
%creates diffuse reverberation resp. late reflections
function [obj, diffuseBlock] = getDiffuseReverberation(obj,inputBlock)
    %calculate pulse length
    pulseSize = ceil(obj.reverbTimeMs/1000*obj.sampleRate);

    %Nf = L+Nh-1;
    fftSize = obj.blockSize + pulseSize - 1; %in samples

    %initialize
    if (length(obj.diffuseBlockOld) == 1)
        obj.diffuseBlockOld = zeros(fftSize,1);
    end

    diffuseBlockNew = ifft(fft(inputBlock,fftSize).*obj.pulseResponse);

    %do overlap add
    diffuseBlock = diffuseBlockNew(1:obj.blockSize) + obj.diffuseBlockOld
        (1:obj.blockSize);

    obj.diffuseBlockOld(1:fftSize-obj.blockSize) = diffuseBlockNew(obj.
        blockSize+1:fftSize) + obj.diffuseBlockOld(obj.blockSize+1:fftSize
        );
end
```

The *getDiffuseReverberation(obj,inputBlock)* method convolves the input block with the artificial pulse response to get diffuse reverberation. Here the overlap add method described in

[1] EXP(X)
[2] RAND(N)

Subsection 3.2.1 is implemented. First the input block has to be transformed to frequency domain with the DFT. Then the input block can be multiplied with the *pulseResponse* in the frequency domain [1]. The result then is transformed to time domain with the inverse discrete fourier transformation (IDFT) again. A temporary state variable *diffuseBlockOld* is needed to save the last result of the convolution. It is initialized with zeros at the first call of the function. At subsequent calls of the *getDiffuseReverberation(obj,inputBlock)* method the overlapping $N_h - 1$ samples of the *diffuseBlockOld* variable are added to the result of the current convolution.

Listing 5.14: Start method(ReverbV3)

```
function [obj,outputBlock] = start(obj,inputBlock)
    %save next input block for early reflections
    obj = refreshTotalBuffer(obj, inputBlock);

    [obj, diffuse] = getDiffuseReverberation(obj,inputBlock);

    %direct sound + early reflections + diffuse reverberation
    outputBlock = inputBlock + getTotalBuffer(obj) * obj.damping + diffuse
         * obj.damping/2;
end
```

In Listing 5.14 the start method of the Reverb3.m class is given. The early reflections are created with a simple circular buffer as described in Subsection 5.2.2. The diffuse reflections are created with the *getDiffuseReverberation(obj,inputBlock)* method, as described above. So finally the output block can be created using the unmodified input block plus the delayed input block from the *getTotalBuffer(obj)* plus the diffuse reverberation. The *damping* variable is a scale factor simulation the attenuation of the reflected waves at the walls.

5.2.7. Distortion Class

The *Distortion.m* class implements the distortion effect from Section 4.2. One can chose between soft clipping 4.2.2 and hard clipping 4.2.1.

function obj=Distortion(sampleRate, blockSize, distortion, level)

[1] Remember the convolution theorem of the DFT

Table 5.6.: *Distortion.m* constructor arguments

argument	**description**	**unit**
sampleRate	sample rate f_s	Hz
blockSize	block size	samples
distortion	chose between hard and soft clipping	integer
level	amount of distortion	%

Listing 5.15: Start method(*Distortion.m*)

```
%length of input signal equals blocksize!
function outputBlock = start(obj,inputBlock)

    %initialize output vector
    outputBlock = zeros(obj.blockSize,1);

    switch  obj.distortion
        %hard clipping
        case 1
            outputBlock = inputBlock;

            %set threshold according percentage
            threshold = 1-obj.level;

            %cut of  input signal
            topIndex = abs(inputBlock) > threshold;

            %distinguish between positiv and negativ range!
            negIndex = inputBlock < 0;
            posIndex = inputBlock > 0;

            %replace top values
            outputBlock(topIndex & negIndex) = -threshold ; %input = -1
            outputBlock(topIndex & posIndex) = threshold ; %input = 1

        %symmetrical soft clipping
        case 2
            %set threshold according percentage
            threshold = 1-obj.level;

            %get index for 3 threshold steps
            botIndex = abs(inputBlock) < 1/3; %-1/3<input<1/3
```

```
33              topIndex = abs(inputBlock) > 2/3; %input<-2/3 and input>2/3
34              midIndex = ~logical(botIndex + topIndex);%-2/3<=input<=-1/3 &
                    1/3<=input<=2/3
35
36              %distinguish between positiv and negativ range!
37              negIndex = inputBlock < 0;
38              posIndex = inputBlock > 0;
39
40              %replace bottom values
41              outputBlock(botIndex) = 2 * inputBlock(botIndex); %input * 2
42
43              %replace mid values
44              outputBlock(midIndex & negIndex) = -1 * (3 - (2 + 3 *
                    inputBlock(midIndex & negIndex)).^2)/3; %(3 - (2 - 3 *
                    input).^2)/3
45              outputBlock(midIndex & posIndex) = (3 - (2 - 3 * inputBlock(
                    midIndex & posIndex)).^2)/3; %(3 - (2 - 3 * input).^2)/3
46
47              %replace top values
48              outputBlock(topIndex & negIndex) = -1; %input = -1
49              outputBlock(topIndex & posIndex) = 1; %input = 1
50          %constant
51          otherwise
52          end
53      end
54  end
```

The *Distortion.m* class more or less implements only one important method. First the *start(obj, inputBlock)* method selects the kind of distortion depending on the *distortion* attribute. If hard clipping as described in Section 4.2.1 is selected all absolute input values bigger than *threshold* are set to *threshold*. The *threshold* variable is set according to the *level* attribute. Matlab is able to use relational operators on a whole block of input values. It returns an index vector with boolean values zero and one. First it has to be checked if the absolute values of the input block are smaller than the threshold. Then the positive values have to be separated from the negative ones. Finally these two information can be logical combined to generate the output block. If soft clipping as described in Section 4.2.2 is selected it's a bit more complicated. Because we have to find the indexes for three intervals plus the indexes for positive and negative values now. The first interval A contains the indexes of all absolute values of the input block smaller than one-third. The third interval C contains the indexes of all absolute values to the input block that are bigger than two-third. Finally the indexes of the second interval B are a logical

combination of the first and the second interval namely B equals not A and C. All input values with indexes that lie in interval A are just multiplied with two. All input values with indexes that lie in interval C are set to one. All values with indexes that lie in interval B are manipulated to make a smooth transition from the first interval A to the third interval C. Of course positive and negative values have to be distinguished again.

5.2.8. Equalizer Class

The *Equalizer.m* class implements a three band equalizer. It uses the Matlab Butterworth digital and analog filter design function [1] and the filter function already mentioned in Subsection 5.2.2. A three band equalizer divides the frequency spectrum in three bands with fixed size. We have a bass band for the low frequencies. A mid band for the mid frequencies. And a treble band for the high frequencies. Figure 5.1 illustrates the magnitude response for all three bands.

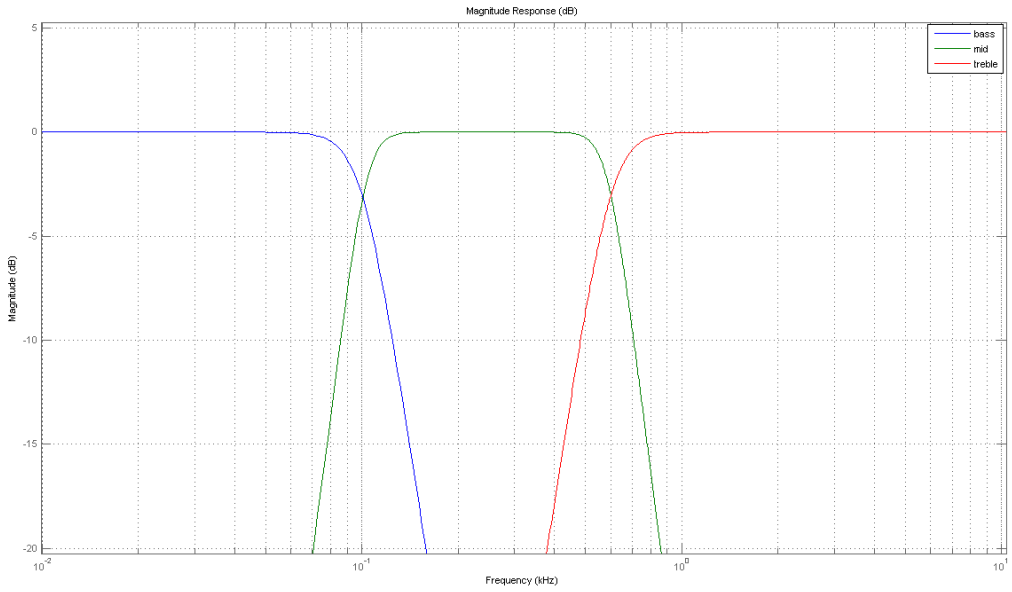


Figure 5.1.: Frequency spectrum of 3 band eualizer

The constructor in Listing 5.16 required the sample rate in hertz and the gain in decibel for all three bands. The gain in decibel has to be converted to a gain factor α so it can be used to scale the output later.

[1] [B,A] = BUTTER(N,Wn,'high')

Listing 5.16: Constructor (*Equalizer.m*)

```
function obj=Equalizer(sampleRate,bassGain,midGain,trebleGain)

    %initialize properties
    obj.sampleRate = sampleRate;

    %convert from dB to factor
    if(bassGain == 0)
        obj.bassGain = 1;
    elseif(bassGain > 0)
        obj.bassGain = bassGain/3;
    else
        obj.bassGain = (-1) * 3/bassGain;
    end

    %convert from dB to factor
    if(midGain == 0)
        obj.midGain = 1;
    elseif(midGain > 0)
        obj.midGain = midGain/3;
    else
        obj.midGain = (-1) * 3/midGain;
    end

    %convert from dB to factor
    if(trebleGain == 0)
        obj.trebleGain = 1;
    elseif(trebleGain > 0)
        obj.trebleGain =trebleGain/3;
    else
        obj.trebleGain = (-1) * 3/trebleGain;
    end

    %calculate filters
    obj = refreshFilter(obj);

end
```

The *refreshFilter(obj)* method in Listing 5.17 calculates the filter coefficients for the three bands using Butterworth filters. The bass band uses a lowpass filter with stop frequency at eighty hertz. The mid band goes from eighty one up to six hundred hertz and uses a bandpass

filter. The treble band uses a highpass filter with a stop frequency at six hundred and one hertz. It is very important to use a even order for the bandpass filter since it has to be multiplied by two. So *butter(6/2,[81 600]/(obj.sampleRate/2),'bandpass')* creates a butterworth bandpass filter, normalized with the Nyquist frequency(*sampleRate*/2), pass-band from eighty one up to six hundred hertz of order six.

Listing 5.17: Refresh filter coefficients(*Equalizer.m*)

```
function obj = refreshFilter(obj)
    %bass band goes from 0 up to 80Hz
    %create butterworth filter with order 6
    [obj.bassCoeffs.num,obj.bassCoeffs.den] = butter(6, 80/(obj.sampleRate
        /2),'low');

    %mid band goes from 81 up to 600Hz
    %create butterworth filter with EVEN order 6
    [obj.midCoeffs.num,obj.midCoeffs.den] = butter(6/2,[81 600]/(obj.
        sampleRate/2),'bandpass');

    %treble band > 600Hz
    %create butterworth filter with order 6
    [obj.trebleCoeffs.num,obj.trebleCoeffs.den] =butter(6,601/(obj.
        sampleRate/2),'high');
end
```

The *start(obj, inputBlock)* method in Listing 5.18 gets a block of input samples. Like in Subsection 5.2.2 the Matlab filter function is used to apply the filter to the input block. Again a state variable is needed to safe the inner state of the filter for the block processing. Finally the output block can be assembled multiplying all three bands with the specific gain factor α and adding them to each other. This accords to a parallel connection.

Listing 5.18: Start method(*Equalizer.m*)

```
%returns next blocksize samples and filter state
function [obj, outputBlock] = start(obj, inputBlock)
    %apply filter to input
    [bassOutput, obj.bassState]= filter(obj.bassCoeffs.num, obj.bassCoeffs
        .den, inputBlock, obj.bassState);

    %apply filter to input
    [midOutput, obj.midState]= filter(obj.midCoeffs.num, obj.midCoeffs.den
        , inputBlock, obj.midState);

```

```
9       %apply filter to input
10      [trebleOutput, obj.trebleState]= filter(obj.trebleCoeffs.num, obj.
            trebleCoeffs.den, inputBlock, obj.trebleState);
11
12      %parallel combination
13      outputBlock = bassOutput * obj.bassGain + midOutput  * obj.midGain +
            trebleOutput  * obj.trebleGain;
14  end
```

5.3. GUI

The graphic user interface (GUI) was created with the Matlab open GUI layout editor [1]. Since all effects are written with Matlab code they can be easily integrated. Another advantage is the easy integration of new effects, as planned for later versions. This section provides a documentation of the GUI, especially how the user interface for the existing effects was created and how to use it. Guide provides a mixture of drag and drop and programming operational controls manually. In Figure 5.2 possible operational controls and the multi effects unit GUI are given. The toolbar provides push buttons, sliders, checkboxes, tables and so on. Figures can be applied to the GUI using the axes control.

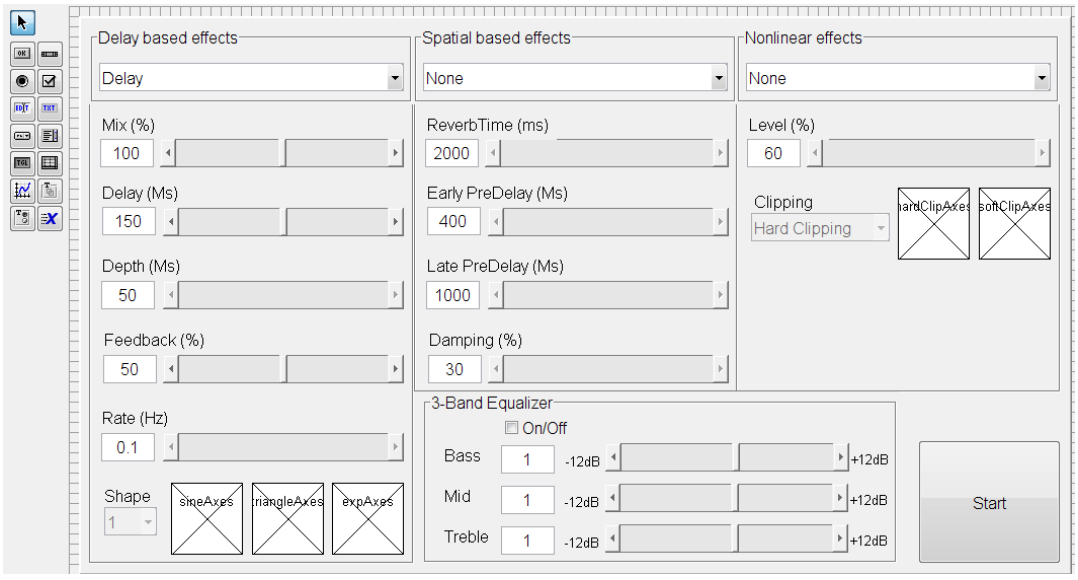


Figure 5.2.: Guide toolbar and multi effects unit GUI

[1] GUIDE

The effects are arranged in effect groups. On the left all delay based effects with their particular parameters are given. In the middle all spatial based effects are positioned. And on the right all distortion based effects are placed. The three band equalizer from Subsection 5.2.8 is positioned on the bottom of the GUI. In fact the effects are applied to the input signal in exactly this order. First one delay based effect can be applied. Then one spatial based effect can be applied. Then one distortion based effect can be applied. Finally the equalizer can be used to attenuate or amplify the specific wave bands of the output signal. To start processing the *Start* button has to be pressed. Of course it makes a difference if the effects are executed in a another order because of nonlinearity. But the current GUI only provides execution of the effects in the order given above. The specific parameters of the effects have been already explained in the previous sections. The GUI provides the facility to easily change and experiment with different parameter configurations.

5.3.1. Menu items

The menu item *File* has only one entry *Open*. It is used to load a input wave file using the Matlab standard dialog box for opening files [1], given in Figure 5.3, and the wave read function [2]. After opening a file it is displayed in a separate window 5.4. Furthermore the input wave file is played using the Matlab sound function [3]. The sample rate and the mono stereo selection are changed according to the input signal. If a stereo signal is loaded, the left and the right channel are displayed separately. This view can be changed from the time to the frequency domain in the *Settings* menu with menu item *Time/Frequency*. Of course the several effects have to be applied to both channels separately for a stereo signal. A stereo signal can be treated as a mono signal with the *Mono/Stereo* entry in the *Settings* menu. If *Mono* is selected for a stereo signal the sum of the left and the right channel is taken and divided by two to get the average of both channels. If *Stereo* is selected for a mono signal, the signal is just duplicated to the left and to the right channel. Of course the processing time reduces for a mono signal. In the *Settings* menu the sample rate and the block size for block processing can be changed too. In general a larger block size leads to a smaller processing time. If a sampling rate different from the input signal is selected, undersampling resp. oversampling of the output signal can be achieved. In Figure 5.4 the input signal and the output signal are displayed for a stereo input signal on which several effects were applied. The *Presets* menu has two entries *Save* and

[1] [FILENAME, PATHNAME, FILTERINDEX] = UIGETFILE(FILTERSPEC, TITLE)
[2] Y=WAVREAD(FILE)
[3] SOUND(Y,FS)

Load. Here the current configuration of the selected effects can be saved as comma separated values (CSV) file with the Matlab standard dialog box for saving files [4] and the CSV write function [5]. Thus good sounding effects can be loaded again later easily with the Matlab CSV read function [5].

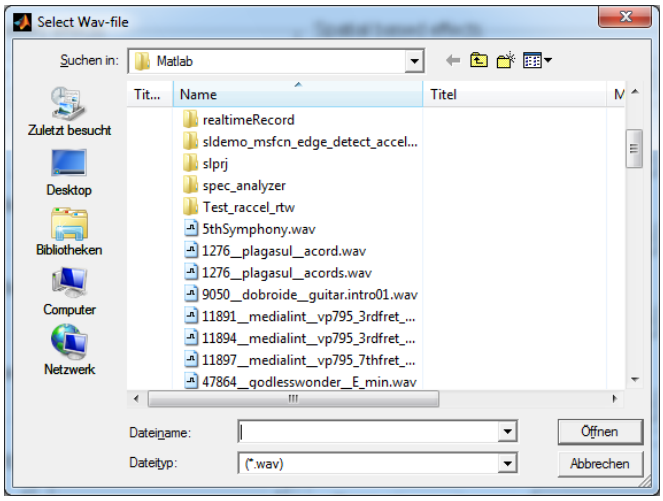


Figure 5.3.: Open dialog box

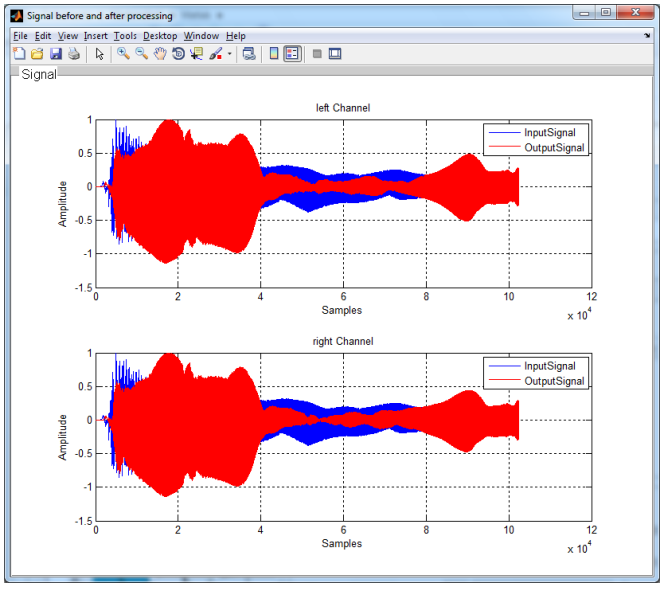


Figure 5.4.: Signal before and after processing window

[4] [FILENAME, PATHNAME, FILTERINDEX] = UIPUTFILE(FILTERSPEC, TITLE)
[5] CSVWRITE(FILENAME,M)
[5] M = CSVREAD('FILENAME')

5.3.2. GUI functions overview

In this section a short description of the most important GUI functions is given. This includes the callback functions but also the create function and opening functions for the user controls. Error handling is done for most functions using try and catch statements:

```
TRY
    statement, ..., statement,
CATCH ME
    statement, ..., statement
END
```

Like in other object oriented programming languages like the Java programming language (Java) or the C++ programming language (C++), errors respectively exceptions that occur between the try and catch statement can be handled between the catch and the end statement. Matlab does not distinguish between different error types but simply catches all types of errors. The specific error message *ME* can be used to give the user feedback what caused the error.

multiEffectsUnit_OpeningFcn(hObject, eventdata, handles, varargin) This function is executed after the operational controls are created and initialized. The sample rate,*Mono/Stereo* selection and *Time/Frequency* selection initialization is made here. Further the handles for the shape and the clipping parameters are set here.

startProcessing_Callback(hObject, eventdata, handles) This callback is executed when pressing the *Start* button. Here the selected effect with the appropriate parameters is applied to the input signal one after the other. After that the refreshed output signal is displayed using the refreshIOPlot(handles) function. Also a wave file 'effectsUnitOutput.wav' is created in the working directory with the Matlab wave write function [1]

sampleRateSelection_Callback(hObject, eventdata, handles) This function belongs to the sample rate selection menu item described in Subsection 5.3.1.

blockSizeSelection_Callback(hObject, eventdata, handles) This function is belongs to the block size selection menu item described in Subsection 5.3.1. If the input signal is already loaded it is made a multiple of the new block size with zero padding.

[1] WAVWRITE(Y,FS,NBITS,WAVEFILE)

monoStereoSelection_Callback(hObject, eventdata, handles) This function belongs to the *Mono/Stereo* selection menu item described in Subsection 5.3.1.

timeFrequencySelection_Callback(hObject, eventdata, handles) This function belongs to the *Time/Frequency* selection menu item described in Subsection 5.3.1.

saveCsv_Callback(hObject, eventdata, handles) This function belongs to the *Save* CSV menu item described in Subsection 5.3.1. First the selection for all effects and values for all parameters are read out. Then they are put into a settings matrix and saved as CSV file.

loadCsv_Callback(hObject, eventdata, handles) This function belongs to the *Load* CSV menu item described in Subsection 5.3.1. First the settings matrix is read from a CSV file. Then the selected effects and parameters are set according to the settings matrix.

openFile_Callback(hObject, eventdata, handles) This function belongs to the *Open* menu item described in Subsection 5.3.1. After the input file is loaded it is made a multiple of block size using zero padding.

blockSizeSelection_CreateFcn(hObject, eventdata, handles) This function is called on creation of the *blockSize* menu entry. Here the possible default values for the block size are initialized.

sampleRateSelection_CreateFcn(hObject, eventdata, handles) This function is called on creation of the *sampleRate* menu entry. Here the possible default values for the sample rate are initialized.

shape_Callback(hObject, eventdata, handles) This callback function uses the Matlab image show function [2] to display the right image for the shape selection parameter. An active selection has a blue background color. An inactive selection has a white background color.

popupDelay_Callback(hObject, eventdata, handles) This callback function changes the active parameters and sets the default values according to the delay based effects selection.

popupSpatial_Callback(hObject, eventdata, handles) This callback function changes the active parameters and sets the default values according to the spatial based effects selection.

[2] IMSHOW(I,[LOW HIGH])

popupNonlinear_Callback(hObject, eventdata, handles) This callback function changes the active parameters and sets the default values according to the nonlinear effects selection.

clipping_Callback(hObject, eventdata, handles) This callback function uses the Matlab image show function to display the right image for the clipping selection parameter. An active selection has a blue background color. An inactive selection has a white background color.

refreshIOPlot(handles) This function opens the *Signal before and after processing* window 5.4, if an input signal has been loaded, using the Matlab figure function [3]. Depending whether the mono or stereo parameter is selected and whether the time or frequency parameter is selected in the *Settings* menu, a different plot is created. Following Matlab code is used to plot a mono signal in the time domain.

```
plot(handles.inputSignal, 'b');
xlabel('Samples');
ylabel('Amplitude');
```

And following Matlab code is used to plot a mono signal in the frequency domain.

```
NFFT = 2^nextpow2(inputSignalN);
%fft input signal
U = fft(handles.inputSignal,NFFT)/inputSignalN;
%get frequency axes
f = sampleRate/2*linspace(0,1,NFFT/2+1);
plot(f,2*abs(U(1:NFFT/2+1)), 'b');
xlabel('Frequency (Hz)');
ylabel('|X(f)|');
grid on;
```

5.3.3. Simple Use Case

Figure 5.5 illustrates a simple use case for the multi effects unit GUI. The list's numbers give a short explanation to the according numbers in Figure 5.5. For a detailed

[3] FIGURE(H)

explanation of the GUI menu items go back to Subsection 5.3.1.

1. In a first step an audio wave file has to be loaded. The loaded audio file is displayed in a separate window and played once.
2. With this popup window one delay based effect can be chosen. Next the according parameters can be changed to alter the delay based effects behavior.
3. With this popup window spatial based effect can be chosen. Next the according parameters can be changed to alter the spatial based effects behavior.
4. With this popup window one nonlinear effect can be chosen. Next the according parameters can be changed to alter the nonlinear effects behavior.
5. Using this button all chosen effects are applied to the input signal one after the other. The output file is displayed in a separate window, played once and saved as 'effectsUnitOutput.wav' in the working directory.

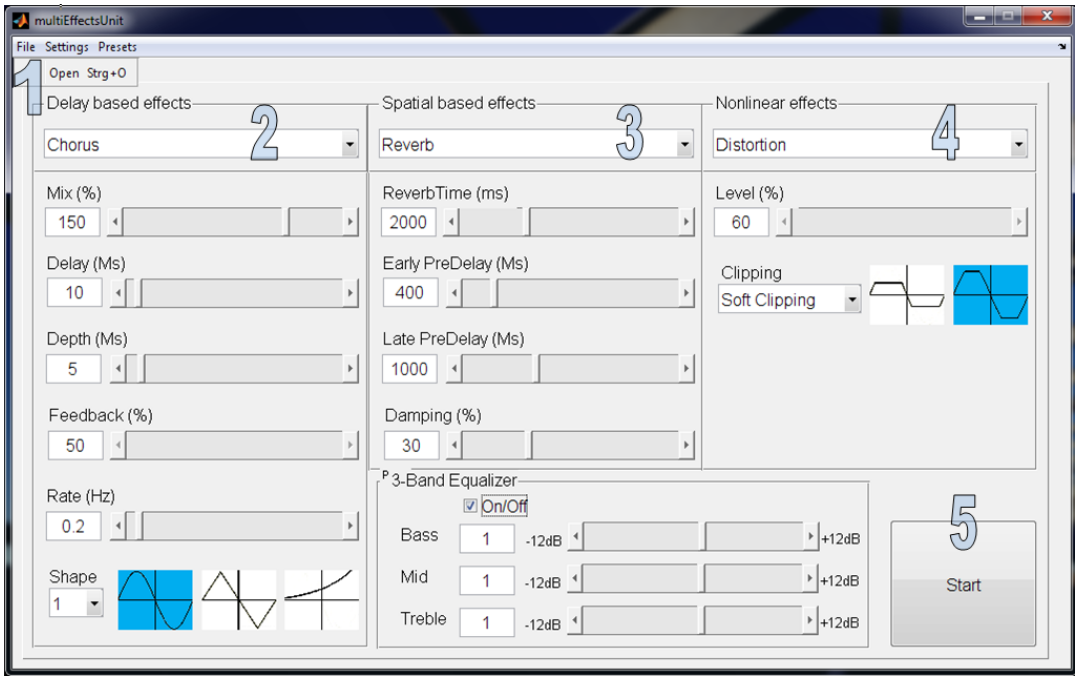


Figure 5.5.: Simple use case

Chapter 6.

Conclusion and Outlook

This work provides an introduction to several well known audio effects. They are grouped in delay based, spatial based and nonlinear effects according to their implementation details and the way humans perceive them. All effects are described both in the time and in the frequency domain. To make understanding easier the delay based effects are represented using block diagrams. Also the similarities of all delay based effects are figured out. Since low frequency oscillator (LFO) and linear interpolation play an important role for the implementation of delay based effects, they are mentioned too. For the reverb effect, which belongs to the spatial based effects, two different approaches are introduced. The first approach uses convolution and works in the frequency domain. The second approach uses comb filters and works in the time domain. In Chapter 4 higher harmonics are created using nonlinear operations on a sum of sine wave. This gives a better understanding how nonlinear effects work and why they create such rich sounds. Later the distortion effect with hard and soft clipping is explained as an important representative. Matlab provides an easy and fast way to implement the covered effects. But since sound design has much to do with human perception, it takes very long to create a well sounding result. For example it usually makes a big difference in which order nonlinear effects are applied to a sound sample. Future work will focus on implementing new effects and improving the existing effects. Adding real time capabilities will be an important topic for future students.

Appendix A.

Acronyms

ADC analog to digital converter

C the C programming language

C++ the C++ programming language

CSV comma separated values

DSP digital signal processor

DFT discrete fourier transformation

FFT fast fourier transformation

FIR finite impulse response

Fortran the formula translation programming language

FPGA field programmable gate array

GUI graphic user interface

IDFT inverse discrete fourier transformation

IIR infinite impulse response

Java the Java programming language

LFO low frequency oscillator

MEX Matlab executable

THD total harmonic distortion

VHDL very high speed integrated circuit hardware description language

Appendix B.

Bibliography

[Dat97] DATTORRO, J.: Effect Design Part 2: Delay-Line Modulation and Chorus. In: *Journal of the Audio Engineering Society* (1997)

[DGKP08] DRMOTA, M. ; GITTENBERGER, B. ; KARIGL, G. ; PANHOLZER, A.: *Mathematik für Informatik*. Heldermann Verlag, 2008

[Dob07a] DOBLINGER, Dr. G.: *Digitale Signalprozessoren Vertiefung*. Institut für Nachrichten und Hochfrequenztechnik, TU Wien, 2007

[Dob07b] DOBLINGER, Dr. G.: *Zeitdiskrete Signale und Systeme*. J. Schlembach Verlag, 2007

[Dut05] DUTTER, R.: *Statistik und Wahrscheinlichkeitsrechnung*. Institut für Statistik und Wahrscheinlichkeitstheorie, 2005

[Jon03] JONES, Morgan: *Valve Amplifiers*. Third. Elsevier, 2003

[Pre04] PRECHTL, A.: *Signale und Systeme 1*. Institut für Grundlagen unt Theorie der Elektrotechnik, TU Wien, 2004

[TM10] THE MATHWORKS, Inc.: *The MathWorks Deutschland - MATLAB and Simulink for Technical Computing*. Internet, 1994-2010. – http://www.mathworks.de/

[VZA06] VERFAILLE, Vincent ; ZÖLZER, Udo ; ARFIB, Daniel: Adaptive Digital Audio Effects (A-DAFx): A New Class of Sound Transformations. In: *IEEE Transactions on audio, speech and language processing , VOL. 14, NO. 5* (2006)

[Zö05] ZÖLZER, Udo: *DAFX - Digital Audio Effects*. J. Wiley & Sons, Ltd, 2005