# Wavelets in real time digital audio processing: Analysis and sample implementations

Master's thesis

Presented at the
Department of Computer Science IV
Prof. Dr. W. Effelsberg
Advisor Dipl.-Math Claudia Schremmer
University of Mannheim

in May 2000

by
Florian Bömers
from Bremen

in collaboration with
I3 Srl., Rome, Italy

# Contents

# Abbreviations

| | |
|---|---|
| 3D | Three Dimensional |
| AD | Analog to Digital |
| ADC | Analog to Digital Converter |
| AIFF | Audio Interchange File Format |
| CD | Compact Disc |
| CD-ROM | Compact Disc Read Only Memory |
| Codec | Coder/Decoder |
| CPU | Central Processing Unit |
| CWT | Continuous Wavelet Transform |
| DA | Digital to Analog |
| DFT | Discrete Fourier Transform |
| DWT | Discrete Wavelet Transform |
| etc. | and so on (lat.: et cetera) |
| f. | and following pages |
| FFT | Fast Fourier Transform |
| Fig. | Figure |
| FIR | Finite Impulse Response (filter) |
| FT | Fourier Transform |
| FWT | Fast Wavelet Transform |
| GUI | Graphical User Interface |
| Hi-fi | High fidelity |
| HTML | Hypertext Markup Language |
| Hz | Hertz |
| i.e. | that is (lat.: id est) |
| IIR | Infinite Impulse Response (filter) |
| IO | Input/Output |
| IWT | Inverse Wavelet Transform |
| JPEG | Joint Picture Experts Group |
| KB | Kilo Bytes (1024 bytes) |
| KHz | Kilo Hertz |
| MME | Multimedia Extensions |

| | |
|---|---|
| ms | Milliseconds |
| NT | Windows NT (New Technology) |
| p. | Page |
| PC | Personal Computer |
| PCM | Pulse Code Modulation |
| PDF | Portable Document Format |
| pp. | Pages |
| QMF | Quadrature Mirror Filter |
| SNR | Signal-to-noise Ratio |
| STFT | Short Time Fourier Transform |
| UI | User Interface |
| UML | Unified Modeling Language |
| URL | Uniform Resource Locator |
| WT | Wavelet Transform |

# Index of Figures

page

# Index of Tables and Equations

# 1 Introduction

## 1.1 Motivation

Digital processing of audio on personal computers is becoming more and more common. Increasing hardware performance and decreasing price broadens possibilities and quality. Even today's standard PC's are capable of processing CD-quality audio data in real time, making it affordable even for amateurs and small studios to work in the digital domain.

Real time audio processing allows modified audio to be heard while it is processed. Although needing much CPU power, it significantly improves professional digital audio: only when the effect of a changed parameter or setting (e.g. volume of an audio track) can be heard instantly, the desired parameter combination can be found in an acceptable time scale. Real time filters also improve non-destructive audio editing possibilities and can reduce the needed disk space for filtered sections.

This thesis will evaluate the wavelet theory for the use in real time digital audio processing. Wavelets provide a new way of gathering frequency information from musical signals. Contrary to the traditionally employed technique for doing that based on Fourier transforms - the STFT - time information is not lost in a portion of analyzed audio data. This property (along with others, which are discussed later in this thesis) promises that wavelets provide efficient and suitable algorithms for real-time digital audio processing.

For real-world examples, the applications demonstrate usage of the wavelet transform for modification and enhancement of music. Several processing algorithms are evaluated in respect to their suitability.

## 1.2 Limits of this thesis

This is a thesis in the field of computer science – it is focused on the computer-specific aspects of the wavelet theory. Consequently, the mathematical parts are not emphasized.

Most theoretic information is formulated in the text. For further reading on mathematical background of wavelets and filter bank, the reader is referred to [STN96], [MAL98] and [VEK95].

The objective of this thesis is to analyze the wavelet transform for processing digital audio in real time. The early idea of calculating a new wavelet has been dropped due to its immense mathematical complexity. It could fill an entire master's thesis. Instead, focus is put on the implementation of the wavelet transform and on the real-time aspect of audio processing.

## 1.3 Structure of the Thesis

The thesis is divided into 3 parts. The first part in chapters 2 till 4 presents background concepts of the thesis' subject. These cover digital audio, Fourier transform and wavelet transform. The first 2 are written non-technically, as only the understanding of the concepts is the essential aim. More detail would exceed purpose and the page limit. Chapter 4, however, provides more mathematical details, also as some formulas are needed and used in the next chapters.

The second part studies how, and in which form, wavelets can be used for real time digital audio signals. Chapter 5 presents different wavelets and their suitability, while in chapter 6 the computer implementation of the wavelet transform is discussed.

The last part in chapter 7 documents the programs written for the thesis. An overview of the audio framework is outlined, followed by descriptions of the different example applications of the wavelet transform. For each application, the theory is provided, as well.

The thesis ends with a conclusion of the achieved results. Indexes for figures, equations and tables are given at the beginning, the biobliography is at the end. In the appendixes, the classes of the audio framework are described. Furthermore, the contents of the accompanying CD are described. All referenced documents available in electronic format are included on it, along with the example applications.

# 2 Basic Concepts

This chapter provides a general, non-technical overview of digital audio. The presented concepts mainly outline the difference of hearable sounds and how they are represented in a computer. Understanding is required for further reading.

## 2.1 Fundamentals of Digital Audio

### 2.1.1 Analog Domain

Sounds, as the ear can hear them, are small changes in air pressure, which stimulate the eardrum. Any sound, even very complex ones, is an ongoing change of air pressure – lower and higher with different strengths. In analog audio systems, these changes are captured by a microphone and transformed to levels of electrical voltage [ROA96, 20]. Voltage is induced by the changes of air pressure, no change results in no voltage. Air pressure higher or lower than “normal” creates positive or negative voltage, respectively. The more the relative pressure changes, the more electrical current is created. This is a continuous process: the voltage changes continuously its level according to the continuous change of air pressure. At any given instant, a distinguishable voltage level is defined. Therefore, analog signals are called *continuous-time* signals. The level at a given instant is referred to as *amplitude*.

Air pressure has infinite precision as to how much it moves. Voltage is able to map the pressure accordingly (quality depends on the microphone), with infinite precision: it has a continuous range of amplitude levels. When the flow of voltage is fed into a loudspeaker, its membrane moves very similar to the original change of air pressure. This analogy gives the analog domain its name [ROA96, 20].

### 2.1.2 Converting to Digital

A computer cannot handle continuous signals - only sequences of values are possible. The process of converting a continuous signal to a discrete sequences of values is done by *sampling*: in short intervals (e.g. every 1/44100 second), the level of the current is measured. As continuous time is split up into short intervals at which is sampled, sampling does a discretization of time. The number of intervals per time is called

*sample rate* and usually specified in Hz. One sampled value is called a *sample*. Generally, the sampling process of an audio signal with a sufficient sample rate does not introduce errors. More details on the sampling process will be presented in chapter 2.2.

As a computer can only handle finite numbers, the value of a sample needs to be represented by numbers: each measured value is assigned a number; high voltage a high number, low voltage a low number. Common range of numbers is e.g. –32768 to +32767 that can be represented by 2 bytes on a computer[1]. As the analog signal is continuous, there exist many levels in between the numbers, so the level is rounded in order to correspond to a number. This assignment level-to-number is called *quantization*[2]. Quantization introduces loss of information: analog audio has a specific amplitude at a given time, whereas the sampled values only occur in steps of discrete numbers. Otherwise said, one sampled amplitude level corresponds to infinite analog levels.

To summarize: transferring audio data from the analog to the digital domain requires discretization of time and of amplitude – sampling and quantization.

### 2.1.3 Digital Domain

Once the sound has been sampled, its representation is a sequence of discrete amplitude values, which can be stored and processed by the computer. The resulting flow of digital samples is a *discrete signal*.

The first consumer product which used digital audio data was the CD. It uses a sampling frequency of 44100 samples per second, 16bit PCM coded samples in stereo.

Digital signals can be stored and copied without loss of quality. While this is important for producers and convenient for consumers, it presents a problem for other parts of the music industry – performers, music distributors, and vendors assume large loss of turn over because of unlicensed copying of CDs and digitally compressed music files [THO99].

[1] corresponding to 16bit PCM data as used by the audio CD
[2] Quantization also requires division or multiplication of the level values to normalize them.

### 2.1.4 Decibels

Decibel[3] (dB) is a unit to measure the power (level) of a signal relative to a reference power. Usually, the reference power is the threshold of human hearing. Decibel values are logarithmic (as opposed to linear levels), approximating the relative human perception of loudness [KIE97, 21][4]. Decibel values are also used as measurement for a range of levels of power, which is called the dynamic range. Humans have a dynamic range of approximately 125dB: 0dB are hardly heard, 125dB is the limit to pain [ROA96, 40].

### 2.1.5 Filters

In general, the term *filter* means any operation on a signal [ROA96, 185]. In signal processing, however, filters usually denote an algorithm or device that alters frequencies of the signal. For example, an equalizer can be realized with a filter that attenuates and amplifies the frequencies as desired.

Two special filter types are low-pass and high-pass filters. Low-pass filters let all frequencies (in the *pass band*) pass that are below a *cut-off frequency*, whereas the remaining frequency components (in the *stop band*) are removed from the signal. High-pass filters work the vice versa: their pass band is above the cut-off frequency [ROA96, 187]. Added to the cut-off frequency, other parameters characterize a low-pass or high-pass filter. An *ideal filter* exactly separates pass band and stop band. In practice, however, filters are far from ideal[5]: the *transition band* is where the frequency response changes from pass band to stop band (or vice versa for high-pass filters). The steepness is usually indicated in dB/octave. Generally, more steepness requires more effort i.e. computation time for digital filters. The *gain* of a filter is the relative attenuation (or boost) it provides between stop band and pass band.

Filters are visualized as a plot of their frequency response, in the dimensions frequency versus amplitude as can be seen in Fig. 1. It shows a low pass filter. Usually, as it is done in the figure, the sample rate is normalized to 1. The frequency response extends

[3] 1/10 of one *bel*

[4] This is not very precise, but demonstrates the point. The subjective “felt” loudness depends on many other aspects, e.g. the frequency.

[5] Ideal filters have an infinite impulse response (IIR), and cannot be implemented therefore.

to 0.5 as this is the *Nyquist Limit* (see below). This filter's cut off frequency is at about 0.2. At a sampling rate of 44100Hz, this corresponds to 44100Hz*0.2=8820Hz.

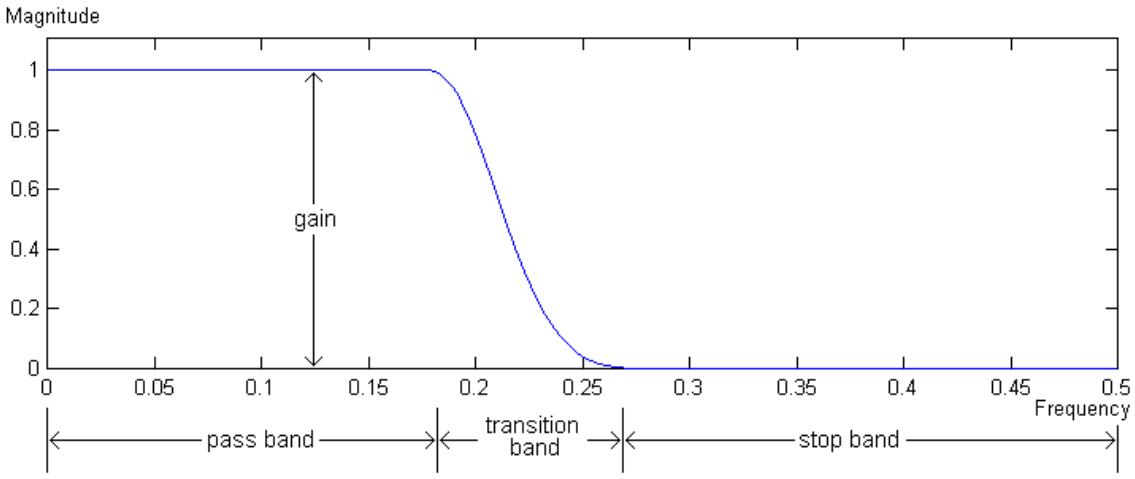



Fig. 1: A low pass filter frequency response

Other special filter types include *band-pass* (letting through a coherent range of frequencies) and *band-reject* filters (or *notch* filter – the inverse of band-pass filter).

## 2.2 AD-DA Conversion

### 2.2.1 The Sampling Theorem

The *Sampling Theorem* (Shannon and Rabe, 1939) is central for digital audio: having a band-limited signal with bandwidth B, it can fully be reconstructed by the sequence of its samples, if and only if the samples are taken with sampling frequency at least 2B. In other words, if a signal is sampled at sample rate f, a signal can be reconstructed perfectly when the signal's bandwidth is at most f/2. This highest frequency for a given sample rate is called the *Nyquist*[6] *limit.*

Frequencies higher than the Nyquist limit cause *aliasing* (*foldover*) effects: these frequencies "fold" into the frequency range below the Nyquist limit. This results in frequency content in the sampled signal, which is not part of the original signal [KIE97, 29].

[6] after the physicist Harry Nyquist (1889-1976) [KIE97, 26]

Aliasing is a serious problem for digital signal processing: in contrast to noise, which covers many frequencies at once, aliasing occurs at certain, folded, frequencies. The ear is more sensible to single frequency components than to a noise floor; aliasing is perceived more than quantization noise.

Considering the Sampling Theorem is not only important for the sampling process: any processing on sampled data must not exceed the Nyquist limit. E.g. when creating sounds in the digital domain, their inherent harmonics can exceed the Nyquist Limit and cause aliasing.

### 2.2.2 AD Conversion

The conversion from analog to digital domain is done by Analog-to-Digital Converters (ADC, spoken “A-D-C”). Input is an analog signal, and the ADC transforms it into an equivalent digital, sampled, discrete representation. Most commonly found are ADC’s which output a sequence of digital encoded samples: voltages are quantized to a linear range of digital values. Other types, like logarithmic scale converters, exist, but are not very common anymore.

The main requirement of the ADC is, that the digital representation reflects the original signal as closely as possible.

To reduce aliasing, a low pass filter needs to be applied to the analog signal before the sampling process: it reduces the signal’s bandwidth so that it contains only frequencies below the Nyquist limit. Although this anti-aliasing filter effectively removes aliasing effects, it introduces new problems: analog filters do not have linear phase – the signal is non-linearly delayed. I.e. the delay is dependent on the frequency of the signal. Especially high frequencies near the cut-off frequency are delayed.

A solution is to move the filter in the digital domain. Good digital filters provide a much more linear phase response than analog ones [ROA96, 42]. Besides that, digital filters are cheaper to manufacture [KIE97, 31]. In order to use a digital anti-aliasing filter, the analog signal is sampled at a higher sample rate than the target sample rate (typically a factor of 4 or 8). The resulting signal is digitally filtered and downsampled. This

technique is called *oversampling*. It has been developed by Motorola. It will be seen that oversampling has another nice property.

As noted in chapter 2.1.2on page 3, quantization introduces errors. These errors are signal dependent. This dependency becomes obvious by looking at sampled silence: as there is no signal, there is no quantization error [ROA96, 34]. Quantization errors create quantization noise. Its level and type depends on the signal, the sample rate, the quality of the ADC and of course, how many bits are used for one sample [ROA96, 36].

### 2.2.3 DA Conversion

The reverse process of sampling is done by the Digital-to-Analog Converter (DAC, spoken “dack”). There are similar problems like for the ADC. To recreate a smooth, continuous signal from the discrete samples, the values between samples are interpolated by a low pass filter following the digital-to-analog conversion [ROA96, 32].

Again, this filter may be moved to the digital domain by oversampling. This has the additional advantage, that the quantization noise is effectively reduced – yielding an improved signal-to-noise ratio. A four-times oversampled signal has 6dB less quantization noise [ROA96, 42]. The resulting immense improvement of quality suggests that it is the more important reason for oversampling.

### 2.2.4 Parameters for optimal Sampling

In order to most accurately capture sounds, the dynamic range and the frequency bandwidth are significant parameters. The number of bits per sample is closely related to the dynamic range of the digital signal. 1bit adds about 6dB dynamic range [OPS85, 447]. The frequency bandwidth is determined by the sample rate.

Audible frequency range is from about 20Hz to 20KHz[7]. Taking this into account, a sampling frequency of 40KHz is the minimum to sample the entire audible frequency

[7] Some people are able to hear higher frequencies, and scientific experiments confirm the physiological and subjective effects of frequencies above 22KHz [ROA96, 31].

spectrum. In order to provide the human's dynamic range of 125dB, about 21 bits per sample are required.

This does not mean that higher values are useless: More bits decrease quantization noise. Digital signals with frequencies near the Nyquist Limit are difficult to handle for ADC's and DAC's, so a higher sample rate gives better quality. And especially for digital processing or synthesis, a "head room" is useful [ROA96, 31]. Therefore it can be said that the more bits are used per sample and the higher the sample rate, the closer the digital signal can represent real sounds.

However, sample rate and number of bits per sample influence directly the amount of digital data produced. This is an important factor for the costs of storage. When audio data is to be sent over a network, the bandwidth of the network connection has to taken into account. Audio processing needs more computational power for higher-quality audio data. Therefore, a "perfect" sample rate and bit resolution does not exist: requirements and possibilities have to be considered.

The audio CD uses 16 bit sample resolution at 44.1KHz. This corresponds to a theoretic dynamic range of 96dB (without oversampling), while having a frequency bandwidth of 22050Hz. When the CD has been developed in the early 1980's, this met the requirements for high-quality audio playback. The possibility of storing 74 minutes of audio (approximately 172KB/s) on one disc was a satisfying limit. Today, 24bit/96KHz systems are becoming popular and available. Their dynamic range (max. 144dB) exceeds human perception, and they provide a good representation of high frequencies (up to 48KHz). There is considerably more "headroom" for digital signal processing or synthesis. Although this format needs 562.5KB/s, today's costs for storage and computational power are lower, and the demand for high-quality digital audio processing is higher than ever.

On the other hand, not in all cases the entire audible range of human hearing needs to be captured. Human speech, for example, only contains frequencies up to 3000Hz and dynamic range is not crucial for the words to remain understandable. For digital telephony (ISDN), a sampling rate of 8000Hz is used by default, with non-linear

quantization providing about 72dB[8]. This dramatically reduces the amount of audio data that is transferred in time: ISDN uses only 62.5KB/s.

## 2.3 Analysis-Resynthesis

In the field of digital signal processing, the terms *analysis* and *resynthesis* stand for conversion from the time domain into another domain and vice versa. Chapter 3 provides an introduction to the Fourier transform (FT) performs analysis to the frequency domain and the corresponding resynthesis. The term **re**synthesis is used, as it transfers back to the original domain. This is distinguished of synthesis, where data, which do not originate in the time domain, are transformed to the time domain[9].

Usually, the analysis algorithm is based on a mathematical *transform*; it consists of a *forward transform* and its counterpart the *inverse transform* for analysis and resynthesis, respectively. An important aspect of a pair of forward transform/inverse transform is its ability to accurately restore the original signal when applied successively. This is called the *perfect reconstruction* property.

For digital signal processing, transforms are powerful tools: modifications in another domain provide new possibilities of altering the signal. A demonstrative example is a way to implement an equalizer (as found on stereo systems) using the FT: after the signal is transformed to the frequency domain, the levels of the frequencies can be accessed directly and thus can be increased or decreased in order to amplify or attenuate certain frequencies. The inverse transform recreates the original signal but with changed frequency components.

Furthermore, analysis transforms provide indispensable possibilities for exploration of and research on signals. There, resynthesis is not needed and the perfect reconstruction property is not important.

---

[8] ISDN samples are uLaw or aLaw encoded with 8 bits per sample. This logarithmic encoding has a subjective dynamic range corresponding to 12bit samples linearly quantized.

[9] This separation of the terms synthesis and resynthesis is not always done in literature. Sometimes, synthesis stands for both.

# 3 The Short Time Fourier Transform

This chapter provides an introduction to the short time Fourier transform. This subject is too large to be covered completely. Only aspects relevant to audio processing are presented, further limited in respect to the following comparison with wavelets.

## 3.1 Overview

In the process of sampling, coded audio data are a sequence of samples, yielding a time-amplitude function. This representation is called the *time domain* [KIE97, 367].

However, music is more than amplitude: one fundamental criterion is pitch. In this context, pitch means the height of a played note, or more general, the frequency of a sound. Every sound, also the most complex one, consists of frequencies. Sounds with extreme frequency content are sine waves on one hand, consisting of exactly one frequency, and white noise on the other hand, containing all frequencies “at once”.

The Fourier transform allows transforming from time domain to *frequency domain* and vice versa (analysis/resynthesis). The frequency domain is in dimensions frequency versus amplitude. After Fourier analysis, the amplitude (or power) of each frequency band can be retrieved[10]. The inverse Fourier transform performs resynthesis from the output of the forward transform. The transform is lossless, i.e. the frequency domain contains the same information as the time domain, only in another representation. Additionally, perfect reconstruction is possible: applying the forward and the inverse transform successively results in exactly the original signal.

[10] The forward FT also produces phase values, which are not regarded here.

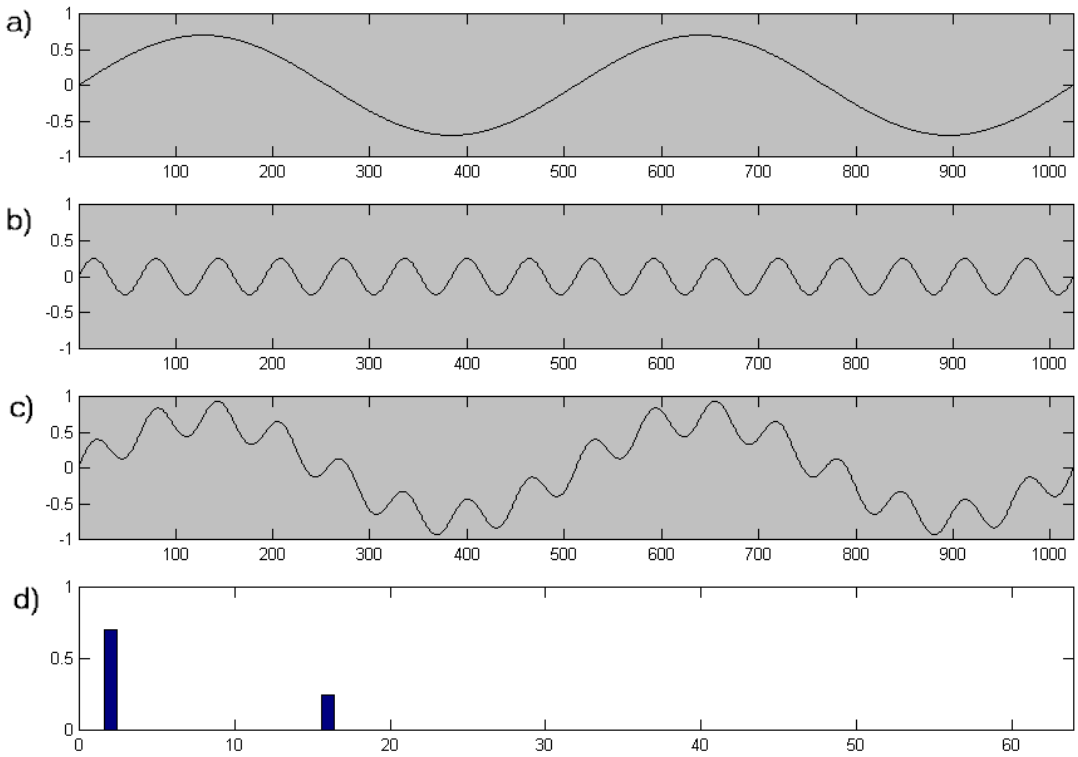



Fig. 2: The Fourier transform

Fig. 2 demonstrates the Fourier transform:

a) a sine wave with low frequency (large period)

b) a second sine wave with higher frequency and less amplitude than a)

c) sum of a) and b).

d) spectrum plot of the output of the Fourier transform of c)

The latter shows the 2 frequency components of c) as well as their magnitude[11] in the frequency domain.

The mathematical foundation of this transform is a theory developed by Jean Baptiste Fourier[12]. He proved that any stationary signal can be represented as an infinite sum of sine waves, each having a specific amplitude and phase [ROA96, 545]. Each sine wave represents one frequency, which can be derived from the period. The amplitude of a specific sine wave represents the amount of that frequency in the signal. For spectral analysis of audio data, the phase of the sine wave is not very important: the ear cannot

[11] "magnitude" is a term for the amplitude of frequency
[12] French mathematician (1768-1830)

hear phase[13] [ROA96, 19]. For resynthesis, all the sine waves, each at their specific amplitude and phase, are added. This results in the original signal.

The original Fourier transform by Jean Baptiste Fourier cannot be applied directly to digital audio data. There are some inherent problems, which will be discussed in the following.

## 3.2 Discrete Analysis and Resynthesis

As digital audio data is discrete, a discrete version of the FT, the discrete FT (DFT) is used, which transforms a discrete signal to a discrete frequency spectrum and vice versa. It maintains the property of exact reconstruction. As opposed to the continuous FT, it may be applied to a limited number of input samples, with the restriction that the analyzed sequence is assumed to be a single period of a periodically repeating waveform [EMB95, 27]. This is due to the periodic nature of the sine waves, fundamental element of Fourier analysis.

For calculating the DFT on computers, various fast algorithms have been developed, which are called *fast Fourier transforms* (FFT). The initial FFT has been discovered by Cooley and Tukey in the 1960s [KIE97, 376]. Modern FFT computation algorithms have a complexity of O(n log(n)) [FRJ00].

## 3.3 Frequency Bands

The output of the DFT can be interpreted as amplitudes of frequency bands[14]. Each band has a fixed bandwidth and a center frequency – the main frequency it analyses. For example, an analysis with 512 frequency bands at a sampling rate of 44100Hz means that the bands are spaced in intervals of approximately 43Hz[15]. The first frequency

[13] Under laboratory conditions, a 180 degree phase shift (polarity inversion) can be heard by some people [ROA96, 19]

[14] Frequency bands are also called *bins* [ROA96, 557].

[15] (22050Hz) / (512 bands). 22050 is the bandwidth of the analyzed signal, it equals the Nyquist frequency

band's center frequency is at 0Hz[16], the second at 43Hz, and so on. 43Hz is the fundamental frequency – all other analyzed center frequencies are multiples of it.

While this property of equal-spaced bands may be useful for other applications of the DFT like in physical analysis, it poses a problem for audio analysis: audio frequencies are heard logarithmically. An interval of one octave always sounds like the same interval, be it 2 low notes or 2 high notes spaced at one octave. Otherwise said, the interval 100Hz to 200Hz sounds like the interval from 5000Hz to 10KHz. As an example, when the DFT is used to extract the note (pitch) of a sound, there is the problem that low frequencies have a low logarithmic resolution compared to the high frequencies. With the 512-band analysis above, there is about one band corresponding for the octave 30Hz to 60Hz, whereas the octave from 3000Hz to 6000Hz is represented by 100 bands. So the pitch of a note at 50Hz cannot be detected, the only information known from the analysis is that it lies in the second band and thus somewhere around 43Hz. For a high note, the pitch can be detected very well – there are 12 (half-) notes per octave, so 100 bands are far more than needed to determine its exact pitch. One can say, the DFT generates too little detail for low frequencies, while generating too much detail for high frequencies in audio analysis.

## 3.4 Windowing

The DFT does not measure exactly the amplitude of the frequencies of one band. Frequencies, which are not multiples of the fundamental frequency, cause *frequency leakage*: in the example above of a 50Hz note, not only the 43Hz-band is affected, but also neighbored bands have little frequency amplitude. Frequency leakage can be so strong, that existing, low amplitude frequencies in a neighbored band are completely hidden by the leakage amplitudes [ROA96, 1102].

Fig. 3 shows such a problem: the 2 mixed sine signals have both a little higher frequency than in Fig. 2. In a) it is visible that the signal's period does not match exactly the length of the analyzed chunk – and thus the frequency components are not multiples

[16] 0Hz actually does not exist. The 0Hz band is called the DC offset [ROA96, 556] and determines an offset to all amplitude values in the time domain [ROA96, 557].

of the fundamental frequency. The frequency domain plot b) shows that the 2 frequencies are not extracted exactly by Fourier analysis: frequency leakage occurs.

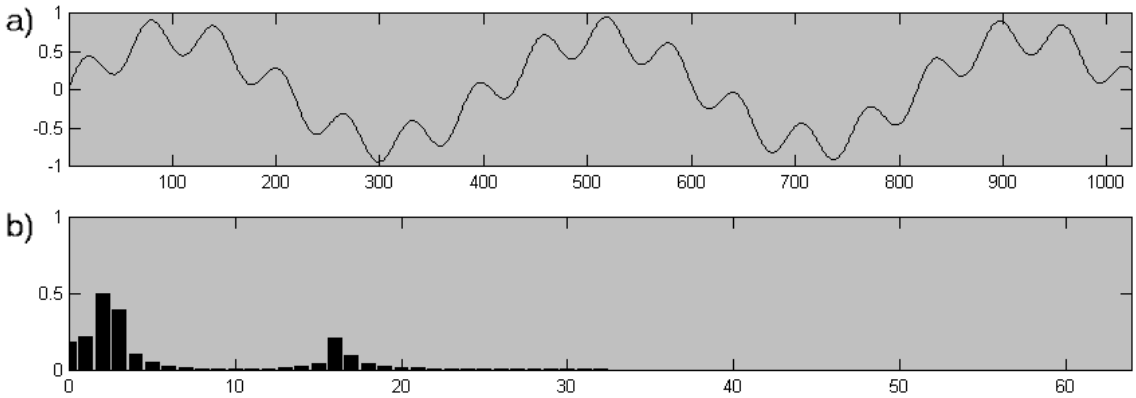



Fig. 3: Frequency leakage

The technique for dealing with the problem of frequency leakage is *windowing*. In the time domain, the signal is enveloped in a window, which reduces the amplitude at the edges [EMB95, 27]. Like this, there is no or little signal at the boundaries, providing a smooth sequence, as you can see in Fig. 4: a) shows a typical window function, the *Hanning window* [PTV94, 554]. In b), the signal from Fig. 4 is enveloped, "windowed" by it. The Fourier analysis c) is not as clean as in Fig. 2, but substantially better than in Fig. 3. For further details on the underlying theory of windowing, the reader is referred to [OPS85, 272f.]

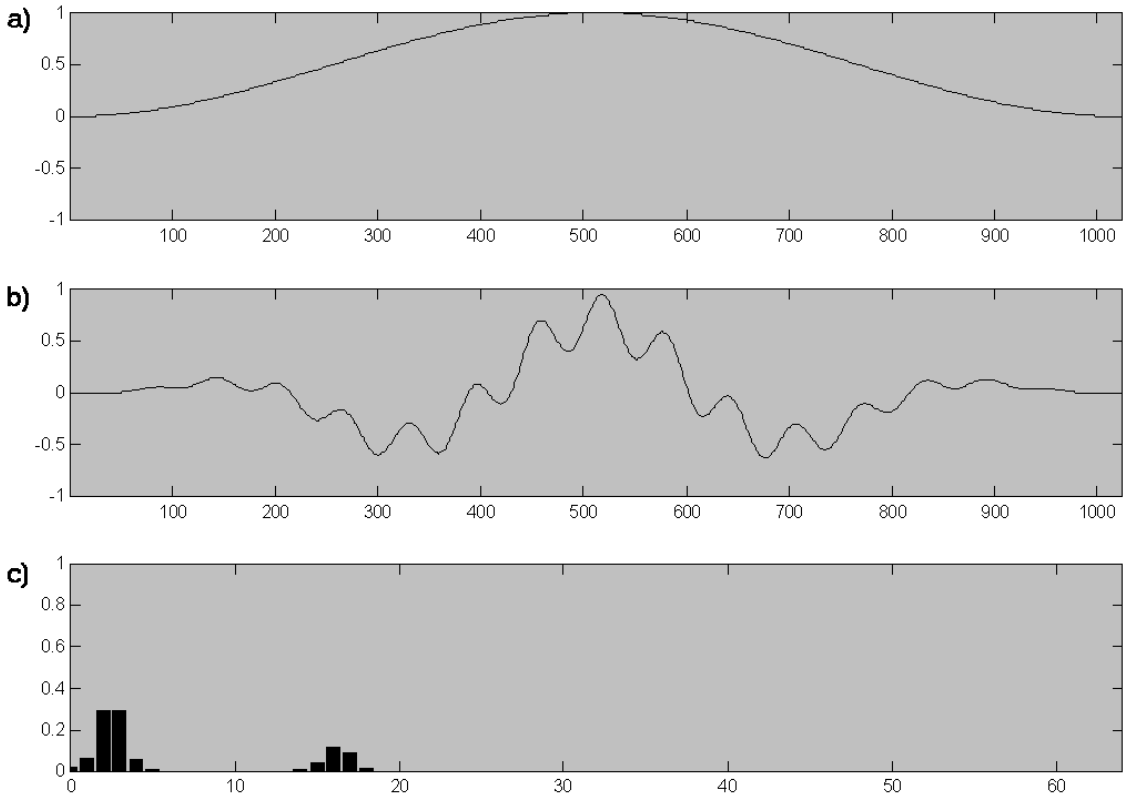



Fig. 4: The window, windowed signal and its frequency plot

A side effect of windowing is that the frequency spectrum is changed slightly. The amplitude of frequencies is lowered: it can be seen that the windowed spectrum has lower amplitudes than the original spectrum [ROA96, 1100]. The side effects depend on the choice of the window function.

There are many different windows with different properties. Other windows like *Blackman* and *Hamming* are commonly used, too. For more information on windows and on their choice, see [PTV94, 554].
Applying the DFT to small windows is called the *short time Fourier transform* (STFT). It was first introduced by D. Gabor in 1946, who established the name *time-frequency domain*, as successive application of the STFT creates a time-varying frequency spectrum [PPR91, 119].

In fact, by limiting the number of analyzed samples when applying the DFT, the data are already windowed by an implicit square window [PTV94, 553]. In this thesis, the term windowing is used for applying a non-square window to the signal.

When resynthesis is needed, it is impossible to exactly reproduce the original signal, as it is affected by the window: the windowed signal will be generated. A common technique to overcome this is to use overlapping chunks of data and mix the overlapped resynthesized parts (*overlap-and-add*). With proper windows, this gives good results, but increases processing demands (computation time): the more data is overlapped, the more samples of the signal are analyzed twice [EMB95, 187]. In real time digital audio, overlapping creates extra latency of the overlapped part.

## 3.5 Time/Frequency Uncertainty

In the frequency domain, time information is lost[17]: the frequency bands (as output by the forward DFT) represent the frequency contents over the entire temporal interval, which has been analyzed. In order to obtain frequency information for sampled, finite-duration, time-varying signals, subsequent chunks of data are analyzed [ROA96, 550]. These chunks are small in size, (typically 32-1024 samples [KIE97, 368]) and thus represent a short time interval, which must be windowed to reduce the limitations of the DFT. The resulting sequence of analyzed chunks is a time-varying spectrum [ROA96, 551].

By using the STFT, a time-frequency domain is obtained, though the time axis has much lower resolution than the time domain. When high time resolution is necessary (i.e. the exact time of an event needs to be known), the analyzed chunks must be very short. However, this results in a coarse frequency spectrum, as with the STFT, the number of frequency bands is proportional to the number of input samples[18]. To keep the example, the event's time will be known precisely, but its frequency content cannot be determined accurately.

Conversely, if high frequency resolution is wished, time resolution is sacrificed, i.e. the exact time of the event cannot be derived [ROA97, 557]. Analyzing time-varying signals using the STFT is thus always a tradeoff of time resolution and frequency resolution. As another example, let a 1 second audio signal with sampling rate 44100Hz

[17] Actually, it is not lost, as it may be recreated by the corresponding resynthesis. More correct is to say, time information is not directly accessible in the frequency domain.
[18] the number of resulting frequency bands equals half the number of analyzed samples [ROA97, 559]

be analyzed using the STFT: for example, when a chunk size of 1024 samples (23ms[19]) is used, the analyzed frequency spectrum has a resolution of 512 frequency bands. So, the bands are spaced in intervals of 43Hz[20]. Thus, an event's time can be determined with accuracy of 23ms, whereas its frequency is known in steps of 43Hz. However, a chunk size of 32 samples results in 0.7ms accuracy in time, but only 1378Hz in frequency.

This coherency of time and frequency is called the *uncertainty principle* due to the similarity to the research results of quantum physicists such as Werner Heisenberg in early 20th century [ROA97, 557]. It is therefore sometimes referred to as *Heisenberg's uncertainty principle*. An exact mathematical derivation can be found in [VEK95, 76].

## 3.6 Spectral Representation

The output of the forward FT, the frequency spectrum, may be visualized in different ways. For the STFT, time information has to be presented, too.

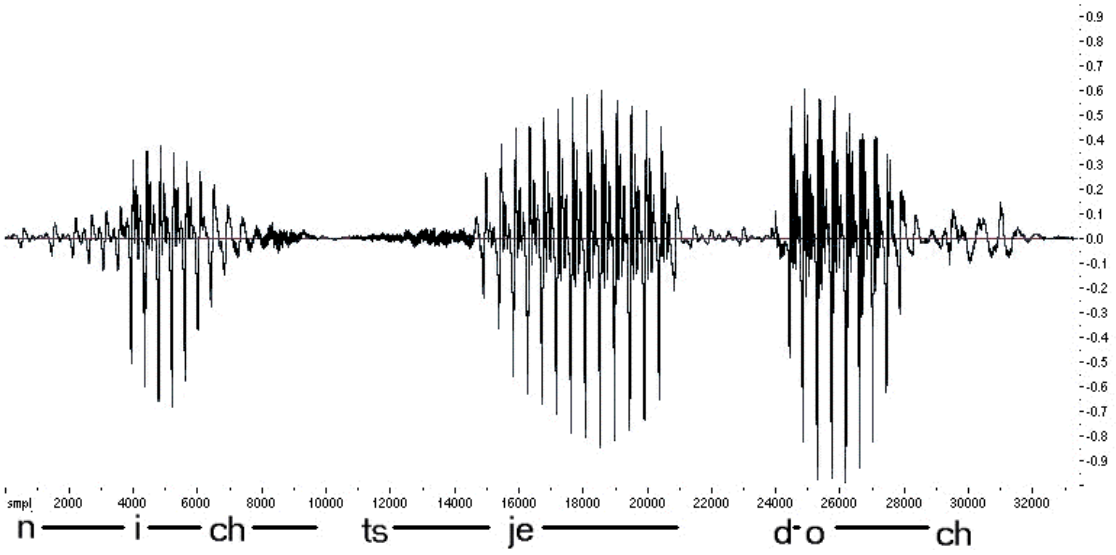



Fig. 5: Original signal in the time domain

[19] (1024samples/s) / (44100Hz) * (1000ms / 1s)
[20] see footnote 15 on page 13

Fig. 5 and Fig. 6 show 2 times the same signal: first as a plot of the time domain and secondly the frequency domain. The time domain plots time vs. amplitude (volume), whereas the frequency domain displays the frequency contents of the same signal: frequency vs. amplitude of the frequency bands. The signal has been analyzed by a 1024 point DFT using a Hanning window. This type of representation is called *discrete* or *line spectrum*. It is in the category of static plots – it displays a "sonic snapshot" or "still image" of a sound [ROA96, 537]. A variation of this uses the power spectrum rather than the amplitude spectrum: as defined in physics, the power spectrum is the square of the amplitude spectrum. Basically, both look similar, but power plots better correspond to human perception [ROA96, 539].

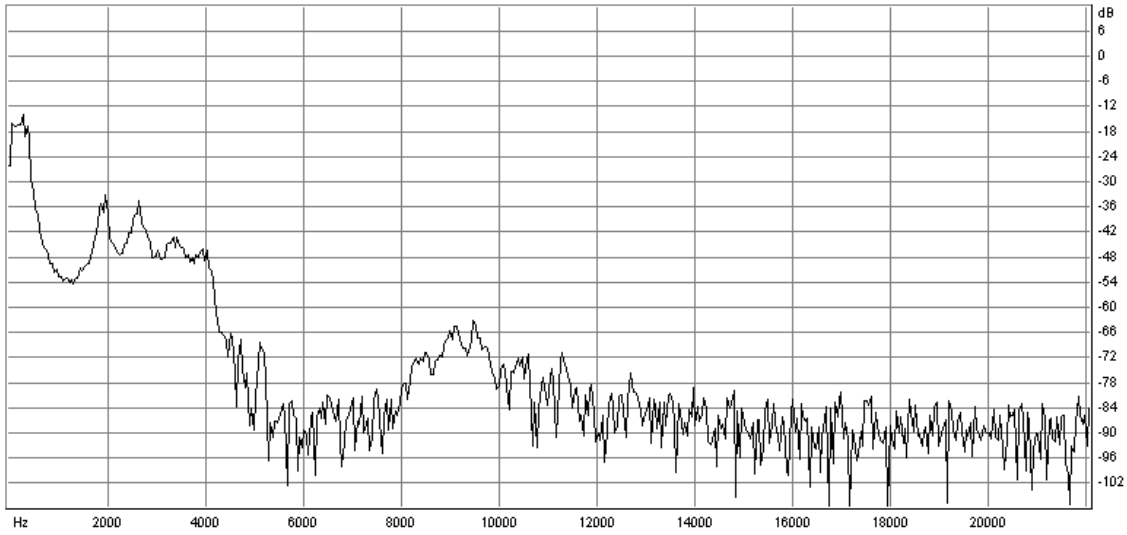



Fig. 6: Line spectrum in the frequency domain

Fig. 7 displays 2 variations of 3-dimensional plots of spectrum versus time, analyzed using the STFT. Essentially, they display a sequence of frequency-amplitude plots. In figure a), the time moves from left to right as it does in time domain representations. Figure b) shows another view, time moving from back to front. A continuous 3d-display of real time data is also referred to as *waterfall display*, since it shows the rising and falling frequency energy in a fluid like depiction [ROA96, 541].

This representation is in the category of time-varying spectrum plots [ROA96, 537]. They allow following the evolution of the sound in time.

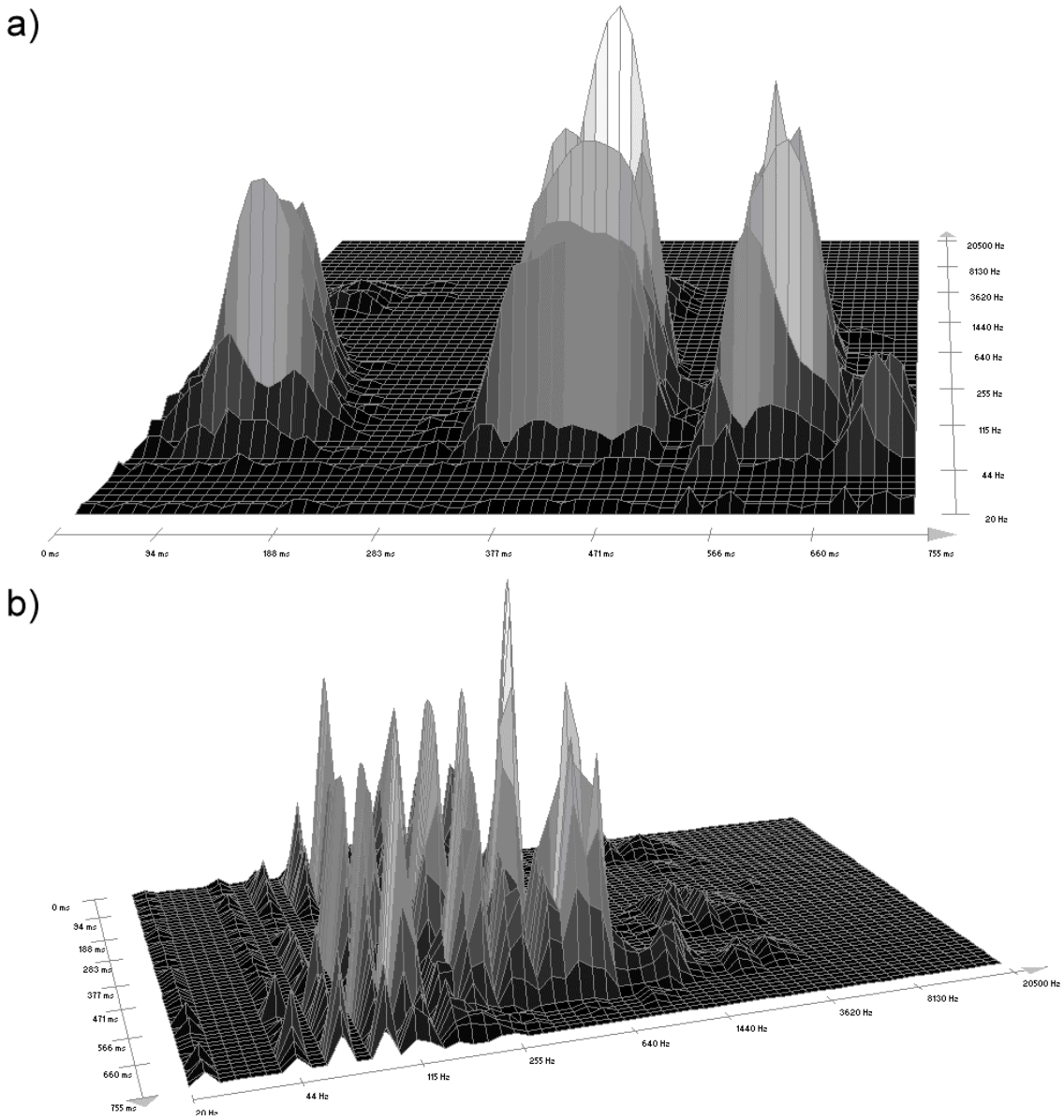



Fig. 7: 3d frequency plots

Fig. 8 shows the so-called *spectrogram*[21] of the same signal as in Fig. 5. Like the 3 dimensional plot, it is in the category of time-varying plots: primarily, it displays time vs. frequency. Additionally, the energy (or amplitude) of each band is represented as the color. In this print, low energy is bright, high energy is dark. Thus, non-existing frequency contents are white. Spectrograms are also called *sonograms*, and as they where first used in speech analysis, they were first referred to as *visible speech* [ROA96, 541].

[21] Created with a Blackman window, 64 frequency bands, on a logarithmic energy scale

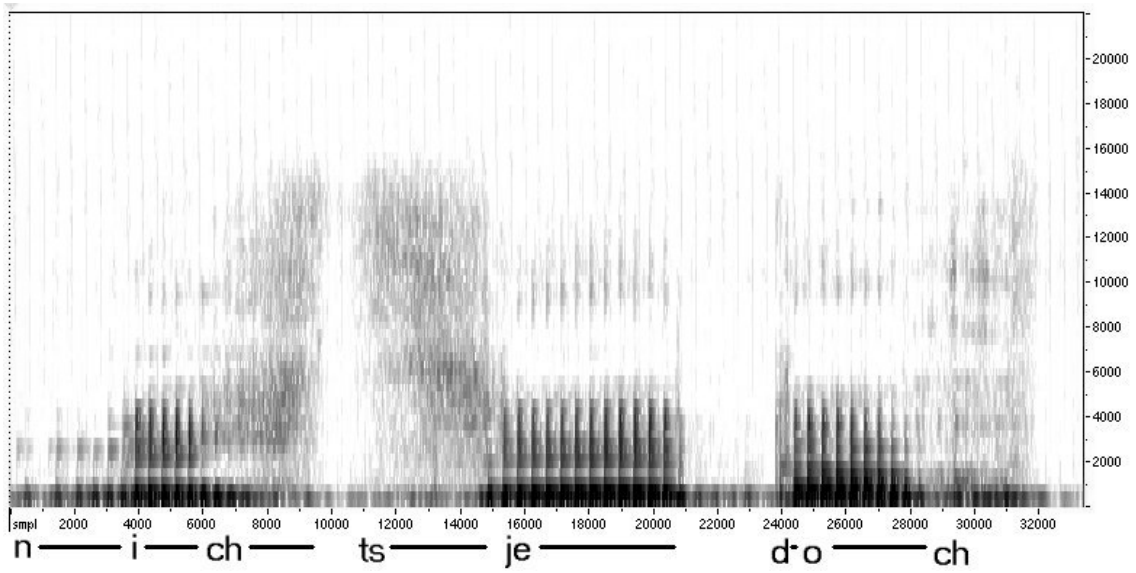



Fig. 8: Spectrogram

The analyzed sound is the author saying the German words “nichts jedoch”. Every letter can be “seen” very well, vocals and consonants are easy to distinguish. Also interesting to note, the last “ch” obviously sounds different than the first “ch”[22]. In the sonogram, many aspects of a sound can be seen, which cannot be seen in the time domain plot.

## 3.7 STFT for processing Musical Signals

The previous discussion showed the main aspects of Fourier analysis of audio data. Here it will be evaluated how suited it is and its drawbacks for real time audio data.

### 3.7.1 Points of Strength

The STFT is a powerful tool to analyze the frequency spectrum of sound. It is possible to obtain very detailed information of the frequency content of a signal.

It is a standard tool for signal processing: much research has been done, and many enhancements have been developed. Fast computation algorithms are available which use low computation time and can be applied in real time applications. Many musical applications today use the Fourier transform for processing, also in professional

[22] The first “ch” is a “shh”-sound produced in front of the mouth, the second one is a gargled noise produced in the throat. It is visible that the first one is composed of a broad coherent range of frequencies, thus resembling white noise, whereas the second one contains distinguishable frequency components.

programs and hardware for high-quality processing. For example, the popular *mp3* file format[23] uses a derivation[24] of the DFT for high quality compression of digital audio data [ISO93, 33].

### 3.7.2 Disadvantages

Despite the general suitability for calculating the frequency spectrum, several drawbacks for audio data, especially in real time, have been identified in the previous chapters:

1. The analysis produces equal-spaced frequency bands. This does not correspond to human perception of frequencies. To get acceptable frequency resolution in low frequencies, high frequency resolution is "over-detailed". Therefore, the FT is quite inefficient for this purpose [ROA96, 592].
2. Digital audio data are not periodic or stationary – in contrary, music, for example, changes continuously. Then, application of the FT produces errors. The more complicated transient phenomena occur in the analyzed signal, the greater the error [ROA96, 593].
3. When the data are windowed to reduce errors, it is even more inefficient, as techniques as overlap-and-add need to be used.
4. For real time application of the STFT, a compromise of time and frequency resolution must be accepted due to the uncertainty principle.

Many alternative spectrum analysis methods have been developed in order to overcome the limitations of the FT approach. However, they all have their respective limitations, and no general method overcomes all limitations without introducing new ones. Rather, they provide better solutions only in specific fields [ROA96, 594]. Especially, most of them do not provide resynthesis, so they are not suitable for processing audio data. Some of these alternative approaches are *autoregression spectrum analysis, linear predictive coding, Walsh functions, cochlegrams*. For further reading, refer to [ROA96, 594f.].

[23] MPEG 1 layer 3
[24] Modified Discrete Cosine Transform

### 3.7.3 Conclusion

The STFT has been employed successfully in real time musical processing systems. Its deep research provides good working solutions for the discussed problems. On the other hand, its inherent problems make it seem more like a compromise than a suitable solution in this field. Much research has been done aiming at new time frequency representations, which overcome the time-frequency uncertainty. The following chapter introduces the wavelet transform that is the outcome of relatively recent research in this field.

# 4 The Wavelet Transform

## 4.1 Introduction

The pre-requisites of the wavelet's history begin in 1910, when Alfred Haar, a German mathematician, developed the now called *Haar function* and associated *Haar matrix*. It is a special kind of matrix: by 2 operations (*translation* – compressing - and *dilation* - shifting) on a "mother vector", the matrix is constructed, all vectors being automatically perpendicular to each other, due to the special "mother" vector. With this scheme it was possible to create orthogonal matrices of any size, all vectors being based on one first vector [STN96, 436]. This is the first known construction of a wavelet, while the term wavelet has not been established at that time.

In the following, much research has been done to overcome and understand the limitations of the FT. One main field of interest was to break up a complicated phenomenon into many simple pieces [JAS94, 4]. In the 30's, these were *Littlewood-Paley techniques*, further developed in the 50's and 60's and leading to applications of the *Calderon-Zygmund theory*. In the 70's, atomic decompositions like in *Hardy space theory* were widely used. G. Weiss and R. Coifman provided much research on these atomic decompositions [GRA95, 4].

In 1980, A. Grossmann and J. Morlet broadly defined wavelets in the context of quantum physics. Little later, J. Strömberg discovered the first orthogonal wavelets. Later in the 80's, Y. Meyer and other independent groups realized discrete calculations of the Littlewood-Paley techniques, followed by the understanding, that this could be effectively a substitute for Fourier techniques. It were Grossmann and Morlet who first suggested the name "wavelets" instead of " Littlewood-Paley theory" [JAS94, 4].

Later development in the 80's and 90's is marked by research of S. Mallat (introducing multiresolution analysis), Y. Meyer (constructing the first non-trivial wavelets) and I. Daubechies (creating compactly supported wavelets of fixed regularity).

## 4.2 Constant Q Filter Bank Analysis

The problem of equal spaced frequency bands with the FT has led to a variety of *constant Q* filter bank analysis transforms. They have been used in audio research since the late 1970s [ROA96, 578]. Examples are the *auditory transform* and the *bounded-Q frequency transform*. Also the wavelet transform can be classified as a constant Q technique.

Q can be seen as the quotient of width of a band to its center frequency (also referred to as Äf / f with f=frequency). So with increasing frequency, the bandwidth becomes greater in constant Q analysis. The analysis bands are thin for low frequencies and wide for high frequencies. The FT transform, though, could be classified as a constant bandwidth transform.

The length of the analysis window is also proportional to the frequency being analyzed: long windows are used to analyze low frequencies, short windows for high frequencies [ROA96, 579]. Like this, the uncertainty principle is not avoided, but it is used effectively. Constant Q analysis trades off time versus frequency resolution “inside” the transform: temporal uncertainty but high frequency resolution in lower octaves (narrow analysis bands) and high temporal resolution with low frequency resolution in higher octaves. As short transients tend to contain high-frequency components, the constant Q scheme allows good time localization of events.

The ear has a similar frequency response as a constant Q response, especially above 500Hz: the human auditory system performs a kind of filter bank analysis with frequency-dependent width of bands. These bands are called *critical bands [ROA96, 579]*.

Constant Q analysis can be performed by applying several low pass (and optionally high pass) filters successively to a signal, or by applying several band pass filters to the same signal. Other approaches exist, e.g. based on FFT algorithms to exploit the high development status of FFT algorithms. While constant Q filter banks typically are less efficient in calculation, they do not need to do as many calculations: e.g. in order to analyze 4 octaves with resolution of half notes (12 half notes per octave), a constant Q

analysis needs 48 bands (each one covering a half note frequency bandwidth), while Fourier analysis needs e.g. 200 bands[25] [ROA96, 581].

## 4.3 Filter Bank Wavelet Transform

In this chapter, the filter bank representation of the wavelet transform is explained. The next chapter will provide background about wavelet functions, followed by the connection back to filter banks.

This approach may appear as starting at the end, since historically the filter bank representation came later than the continuous transform –the link has been made in the late eighties [MAA00, 25], especially by S. Mallat. However, the reader is already familiarized with filters and discrete signals, so this approach integrates better in the logic of the thesis.

### 4.3.1 Digital Filters

Digital filters in the time domain are implemented using a technique called *convolution*. A set of *filter coefficients* (or *taps*) is applied to the samples, by combining previous samples with the coefficients. One output sample is the sum of previous samples multiplied with the filter coefficients. The input sample is multiplied with the first filter coefficient, the previous input sample with the second filter coefficient, and so on. The sum of all products is the resulting, filtered, output sample. As a formula, convolution looks like this (after [VEK95, 49]):

$$y_n = \sum_{i=0}^{N-1} x_{n-i} h_i$$

x: input signal
y: output signal (filtered)
h: filter coefficients
N: number of filter coefficients
n: index of sample

Equation 1: Convolution

Convolution can be written using matrix notation, where signals are written as vectors. It can be written like this:

[25] Approximate minimum amount needed to measure half-note pitches at around 20Hz to 320Hz.

$$\begin{bmatrix} y_n & y_{n+1} & y_{n+2} & \Lambda \end{bmatrix} = \begin{bmatrix} h_1 & h_0 & & & \\ & h_1 & h_0 & & \\ & & h_1 & h_0 & \\ & & & & O \end{bmatrix} \begin{bmatrix} x_{n-1} \\ x_n \\ x_{n+1} \\ M \end{bmatrix}$$

Equation 2: Convolution in matrix representation

Empty spaces in the matrix stand for zeros. Only a filter with length 2 is shown. The convolution matrix is a *right circulant matrix*, and, more specifically, a *Toeplitz matrix* [VEK95, 34][STN96, 36], where each row is a right-shift of the previous row.

In order to retrieve the filter coefficients, a wide variety of design methods exists, which result in filters with different properties. For lowpass and highpass filters, steepness of the transition band and flatness of passband and stopband are important design restrictions and requirements. The reader is referred to [EMB95, 136f.], [STN96, 53f.], [CRO98], [FIS99], and [OPS85] for further details on filter design.

### 4.3.2 Filter Banks

A filter bank is a set of filters, which split up the signal's frequency components in different signals, each with a subset of frequencies. The combined pass bands of the filters cover the entire frequency range, so the filters are complementary. A simple filter bank consists of one low pass filter and one high pass filter, both having a cut off frequency at half the frequency bandwidth. Applying this filter bank to a signal results in 2 new signals, one with the lower half frequencies and one with the upper half frequencies. The FT can be considered as a special filter bank: it splits the signal into many sine waves.

Often, and in the scope of this thesis, only filter banks with the described low pass and high pass filters are used. A block diagram of this filter bank looks like this:

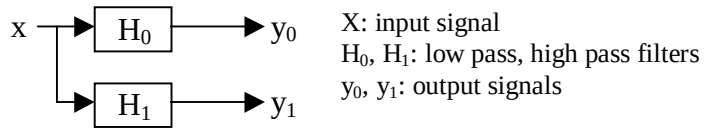



Fig. 9: Simple filter bank

To construct a filter bank with more than 2 frequency bands, $y_0$ could be filtered again by 2 filters, one band pass filter and again one high pass filter which divide the bands up again into 2 bands.

However, it is possible to further separate the frequency bands by only using the *same* high pass and low pass filters. The bandwidth of $y_0$ and $y_1$ is both half the bandwidth of the original signal – the other half has been removed by the filter. Following the Sampling Theorem, they can be exactly represented by half the number of samples. And exactly this is done with *decimators*[26]. They reduce a signal to have only half the samples, by taking every 2nd sample. This is called *downsampling*, its operator is usually indicated by ↓2. Decimating results in a signal with half the number of samples, but they represent the same time interval as the original signal. Thus, the sample rate is halved, too. The decimated output can then be filtered again with the same filters to again split it up into lower and higher frequency contents.

A filter bank with 4 output bands could then be constructed following this block diagram:

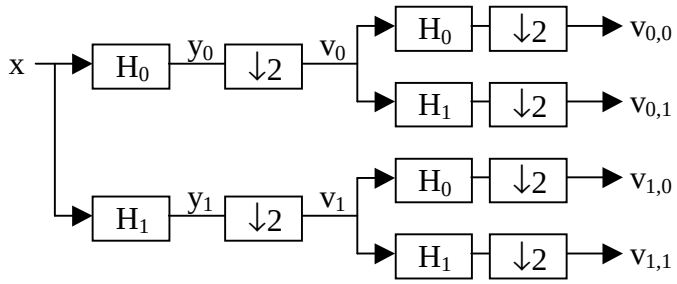



Fig. 10: 2-channel filter bank with 4 output bands

Thus, this type of filter bank works in successive stages. The number of filters per stage is called a *channel*. It is possible to create a 3-channel filter bank with low pass, high pass and band pass (for a range of frequencies between low pass and high pass filter). In general, it is referred to M-channel filter banks. When the frequency bands are of equal distribution, decimators can be used for downsampling, indicated by ↓M. An M-decimator takes every Mth sample and discards the rest. In the following, only 2-channel filter banks are discussed.

[26] The term “decimator” is not historically correct, but widely used. It originates of the Roman practice of killing every 10th soldier of a defeated army, thus meaning “keep 9 out of 10” [VEK95, 66].

### 4.3.3 Perfect Reconstruction

Under certain conditions, a filter bank is reversible, so that the original input can be retrieved from the bands v. Reconstruction is very useful, the filter bank becomes a forward transform/inverse transform pair.

Overall, it depends on the filters whether perfect reconstruction is possible. For reconstruction, *upsampling* (*expanding*) must be done in order to undo the decimation. This is done by inserting a zero after each sample. Additionally, 2 resynthesis filters $F_0$ and $F_1$ are needed to smooth out the zeros, reversing the analysis low pass and high pass filters. The resulting samples are obtained by adding the outputs of the resynthesis filters. This diagram shows a 2-channel filter bank, analysis followed by resynthesis (after [STN96, 103]):

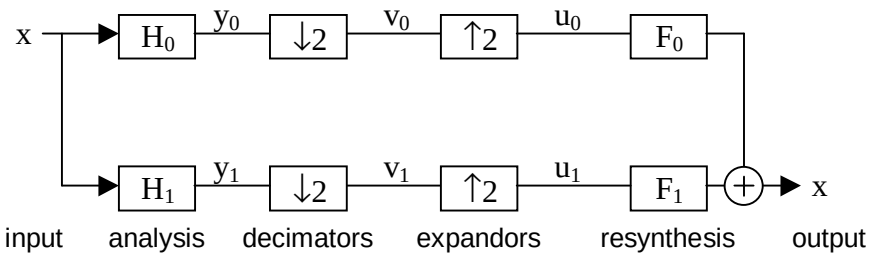



Fig. 11: Analysis/resynthesis filter bank

There are many aspects in order to fulfill the perfect reconstruction property for a filter bank. A selection is presented in the following.

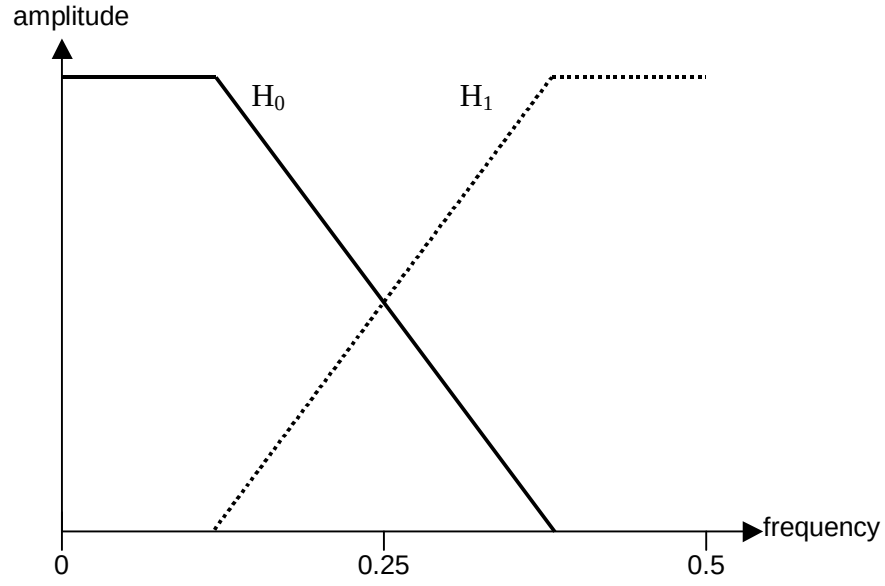



Fig. 12: overlapping lowpass and highpass filter responses (symbolized)

As discrete filters do not have an ideal cut off (i.e. they *do* have a transition band), the low pass and high pass filters' frequency responses overlap: the low pass lets through frequency components of the high pass band, conversely, the high pass filter lets through low frequencies (see Fig. 12). This aspect, causes aliasing when downsampled (as described in chapter 2.2.2 for ADC's, which perform a special kind of downsampling) [STN96, 103]. The solution for perfect reconstruction is to design the reconstruction filters $F_0$ and $F_1$ in such a way that they cancel out the aliasing of the analysis filters [STN96, 104].

As $F_0$ and $F_1$ therefore depend strongly on the analysis filters, it is convenient to calculate them directly from $H_0$ and $H_1$. And indeed there exists a simple formula for calculating them, the *alternating signs pattern*. $F_0$ is derived from $H_1$ by changing the sign of each second filter coefficient starting with the second. $F_1$ is constructed analogously, but sign changing starts with the first filter coefficient.

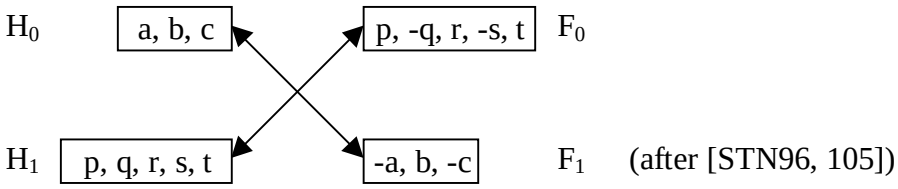



Fig. 13: alternating signs pattern

The diagram shows an example how to construct $F_0$ and $F_1$: a, b, c and p, q, r, s, t are the coefficients of $H_0$ and $H_1$, respectively. Obviously, obtaining the analysis filters from given resynthesis filters works equally well.

The analysis filters also depend on each other. As said before, the frequency responses must be complementary. And to fulfill perfect reconstruction in conjunction with the reconstruction filters, still more conditions have to be met. It would exceed the scope of this elaboration to cover the mathematics behind the *perfect reconstruction condition*. For more detail on this, see [STN96, 107f.].

However, there is a method for deriving $H_1$ from $H_0$ when both filters are to have the same length. It is called the *alternating flip*:

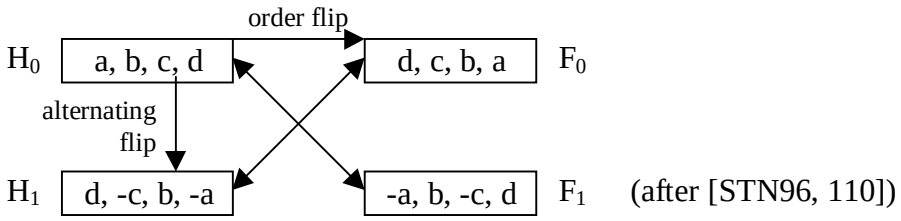



Fig. 14: alternating flip pattern

The coefficients of $H_1$ are the reversed, sign changed coefficients of $H_0$. Still, the perfect reconstruction condition and aliasing cancellation are imposed on the filters. Then, the filters lead to an *orthogonal* filter bank [STN96, 109]. It is called orthogonal, as the convolution matrix is orthogonal, i.e. the transpose is its inverse matrix [BRS89, 155]. There exists another easy way for obtaining $H_1$: it is constructed from $H_0$ using the alternating signs pattern. These filters are called quadrature mirror filter[27] (QMF) banks [STN96, 109]. They lack some useful properties of orthogonal filter banks, so they are not discussed further.

Orthogonality is not required for perfect reconstruction filter banks. The minimum requirement is biorthogonality of forward transform matrix to its inverse counterpart. Then, orthogonality is the special case where the filter bank is biorthogonal to itself. Biorthogonal filter banks do not necessarily have the same length for $H_0$ and $H_1$. In this thesis, the term “biorthogonal” is used for filter banks, which are not orthogonal.

### 4.3.4 Wavelet Filter Bank

Wavelet filter banks are perfect reconstruction filter banks. They appear in trees of filters: in the first level, the frequency spectrum is divided into lower and higher half. After downsampling, these can be split up again, up to a specified level. The *wavelet packet tree* follows the model in Fig. 10: each downsampled filter output is split again in 2 signals. Fig. 15 shows a 3-level decomposition [ALT96].

[27] Also called conjugate mirror filters [COH92, 6]

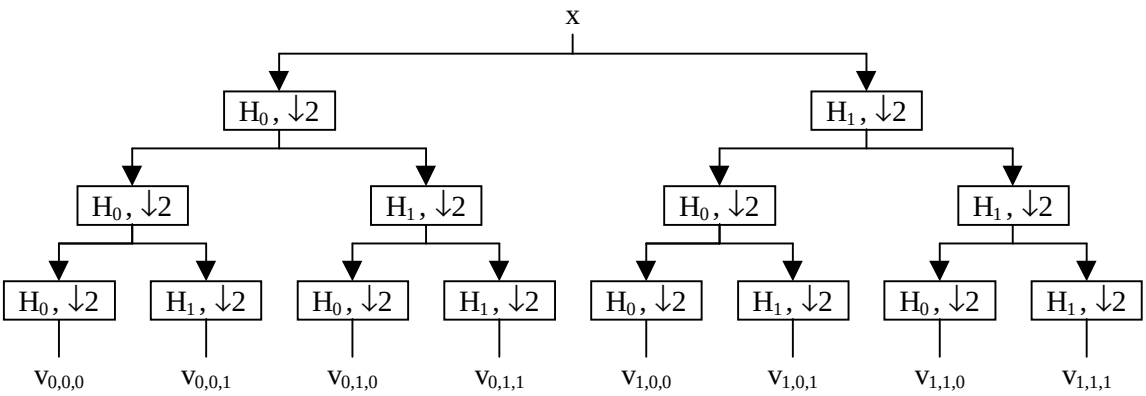



Fig. 15: Wavelet packet tree

The second method is the *wavelet tree*. There, the high pass output is not separated further. It is called the *pyramid algorithm* or *Mallat's algorithm* [STN96, 414].

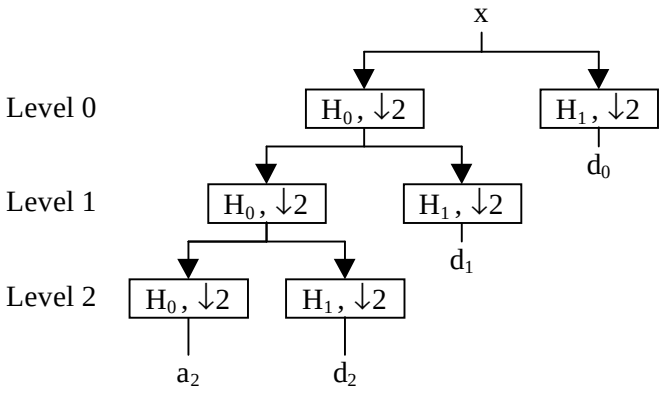



Fig. 16: Wavelet tree

The wavelet tree has some important properties. The output signals that it produces are called *details* d and *approximations* a, they are referred to as the *wavelet coefficients*. The approximations are the decimated output of the low pass filter; conversely the details come from high pass filtering. In each level, the approximations are separated further, and only the approximations of the last level are kept. The numbers of levels determines the number of resulting sets of detail coefficients. $d_0$, the details of level 0, have half the number of coefficients as the number of samples of the original signal, due to decimation. Consequently, $d_1$ has one-fourth the number of coefficients, and so on[28].

When it is looked at the meaning of the coefficients, it will appear obvious that $d_0$ contains the higher half of frequencies of the original signal, $d_1$ contains the range of

[28] Level ordering is reversed for denoting the order of calculation.

frequencies from one fourth to half of the frequencies, etc. As these are octaves, the wavelet tree effectively splits up the signal in octaves. The approximations of the last level contain the remaining lower frequencies. As an example, at a sampling rate of 40KHz (Nyquist limit 20KHz), a 3-level decomposition (as in Fig. 16) gives $d_0$ with frequencies from 10KHz-20KHz, $d_1$ from 5KHz-10KHz, $d_2$ from 2500Hz-5000Hz, $d_3$ from 1250Hz-2500Hz and approximations with range of 0Hz-1250Hz.

This looks like a constant-Q transform. Considering the time resolution of the coefficients, this can be confirmed. Each detail level has half the time resolution of the previous detail level. The $d_0$ coefficients have a very high time resolution, half of the original signal. Further down the tree, localization in time lowers, but, on the other hand, the frequency bandwidth becomes smaller, resulting in better frequency resolution. This is the discrete wavelet transform (DWT).

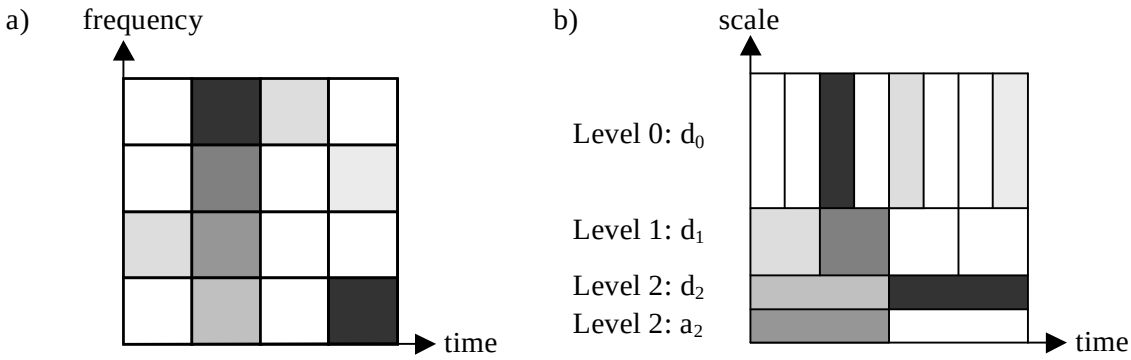



Fig. 17: Time- frequency and time- scale representation

In literature, often the dimension of the levels is not called “frequency”; rather, the dimension is called *scale* (in the following chapter 4.4, it will be seen why). The output of the WT is therefore a time-scale domain. Like a spectrogram, wavelet coefficients can be represented in a *time-scale grid*. Fig. 17 shows a comparison of the representation of the STFT and the WT. Evangelista calls this representation of the WT a *cycle-octave time-frequency grid* [PPR91, 121]. Each box stands for one output coefficient of the forward transform. Due to the different time and frequency resolution, the boxes for each coefficient do not have the same height and width for the WT. As in the sonogram, the color of a box represents the magnitude of the coefficient. It can be well seen that the signal has a short transient in the third box of level 0 coefficients. Using the STFT, this transient cannot be located equally well in time. On the other

hand, the sound, which led to the 2nd coefficient of level 2, can be analyzed considerably more detailed for its frequency content than the STFT. However, the WT cannot locate it precisely in time.

Generally, the number of levels is not limited, except the length of the signal. As each level works on half the number of samples, at some point, there will not be any samples anymore for decomposition.

### 4.3.5 Wavelet Resynthesis

As the filters for a wavelet filter bank need to be suitable for perfect reconstruction, resynthesis is done as described in chapter 4.3.3 on page 29. The detail and approximation coefficients are upsampled and filtered with the resynthesis filters. The sum gives the approximation coefficients for the next level. The process is repeated until level 0 has been resynthesized. The diagram in Fig. 18 shows the procedure.

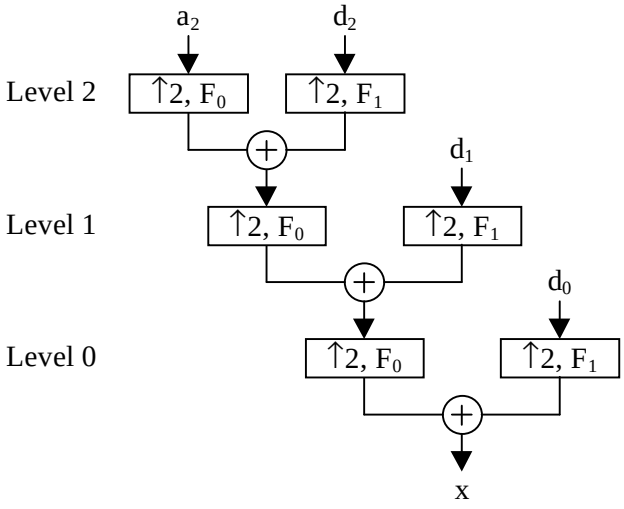



Fig. 18: Wavelet resynthesis

For a complete wavelet packet resynthesis, not all coefficients are needed. Therefore, wavelet packet trees normally do not split up every level in high pass and low pass, but only selected ones.

## 4.4 Wavelet Functions

This chapter is provided for completeness and for the better understanding of the theory behind wavelets. It is not meant to be a mathematically complete coverage. Wavelet

functions, and especially, the formulas, are not used in the work of this thesis, but they may help to understand the idea. It will be shown that the filter bank scheme is sufficient to calculate the wavelet transform.

### 4.4.1 Generalities

*Wavelets* are functions in continuous time that have special properties; usually the letter $\psi$ is used for the wavelet function. The functions need to disappear towards -∞ and ∞. This leads to the term that wavelets are *localized waves* [STN96, xix]. Compact support is not required but useful in many cases.

Another requirement is that the integral is zero[29]. Therefore, the wavelet needs to have at least one change of sign, making its shape look like a small wave – the name "wave-let" is based on this property [SCH97, 111], being the translation of the French "ondelette" [JAS94, 6]. Wavelet functions are the analyzing grains, comparable to the sine waves for the FT. Fig. 19 shows 2 wavelet functions: a) the *Daubechies 2* wavelet and b) the *mexican hat*[30].

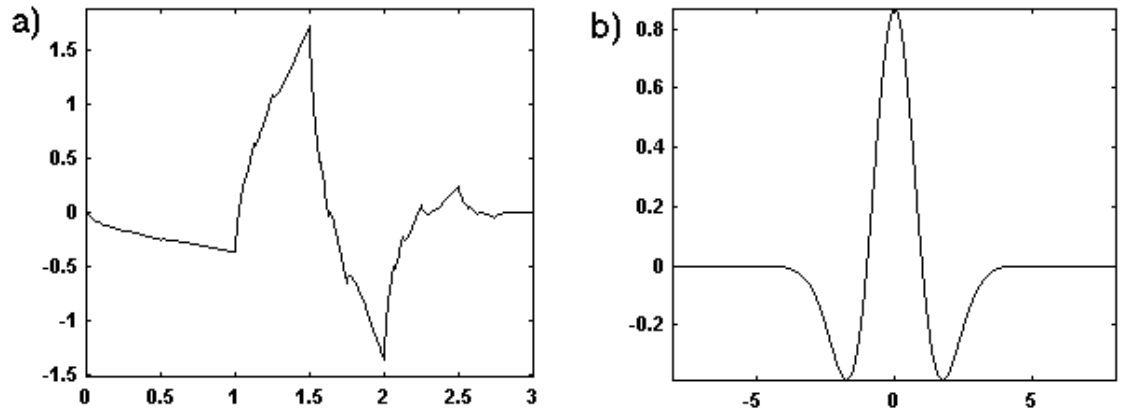



Fig. 19: Typical wavelet functions

### 4.4.2 The Continuous Wavelet Transform

Two important operations on the wavelet function create infinite variations: *shifting* (moving in time, also called *translation*) and *scaling* (compressing in time, also called *dilation*).

[29] Or the sum, for discrete wavelet functions
[30] See ch. 5.2 on p. 46 for a description of common wavelet families

The amount of shifting and scaling is indicated by the indices a (scaling) and s (shifting):

$$\varnothing_{a,s}(t) = \frac{1}{\sqrt{a}} \varnothing\left(\frac{t\text{-}s}{a}\right)$$

Equation 3: Wavelet basis [PPR91, 54]

$\psi_{a,s}$ forms a basis, so that it is possible to represent all admissible functions with a linear combination of the wavelet functions. They are normalized with $1/\sqrt{a}$ to preserve the energy in the wavelet domain [VAL99, (3)]. The special feature of the wavelet basis is that all the elements are derived from a single *mother wavelet* $\psi$ [STN96, 3]. The continuous wavelet transform (CWT) analyzes a given function *f(t)* with the basis functions – the grains.

$$F(a,s) = \int f(t)\psi_{a,s}(t)dt$$

Equation 4: Forward CWT [STN96, 82]

Using the same function as compressed and shifted versions is called *multiresolution*. *F(a,s)* are coefficients with which it is possible to reconstruct the original signal *f(t)*[31]:

$$f(t) = \frac{1}{C}\iint F(a,s)\psi_{a,s}(t)\frac{dads}{a^2} \qquad C = 2\pi\int \frac{|\hat{\psi}(\omega)|^2}{|\omega|}d\omega$$

$\hat{\psi}$ : Fourier transform of $\psi$

Equation 5: Inverse CWT [STN96, 82]

It can be seen that the inverse CWT only exists if the term for the constant C exists, i.e. the integral for C is finite. This can be guaranteed when the integral of $\psi$ is 0 [STN96, 82].

The CWT as given in the equations is *over-complete* [STN96, 82]. The coefficients *F(a,s)* are redundant. It is sufficient to scale the wavelet in powers of 2, as is done in the following discrete version.

The over-complete output of the CWT is commonly displayed as a *scalogram*, like in Fig. 20.

[31] For non-orthogonal wavelets, the inverse CWT uses the dual wavelet for resynthesis.

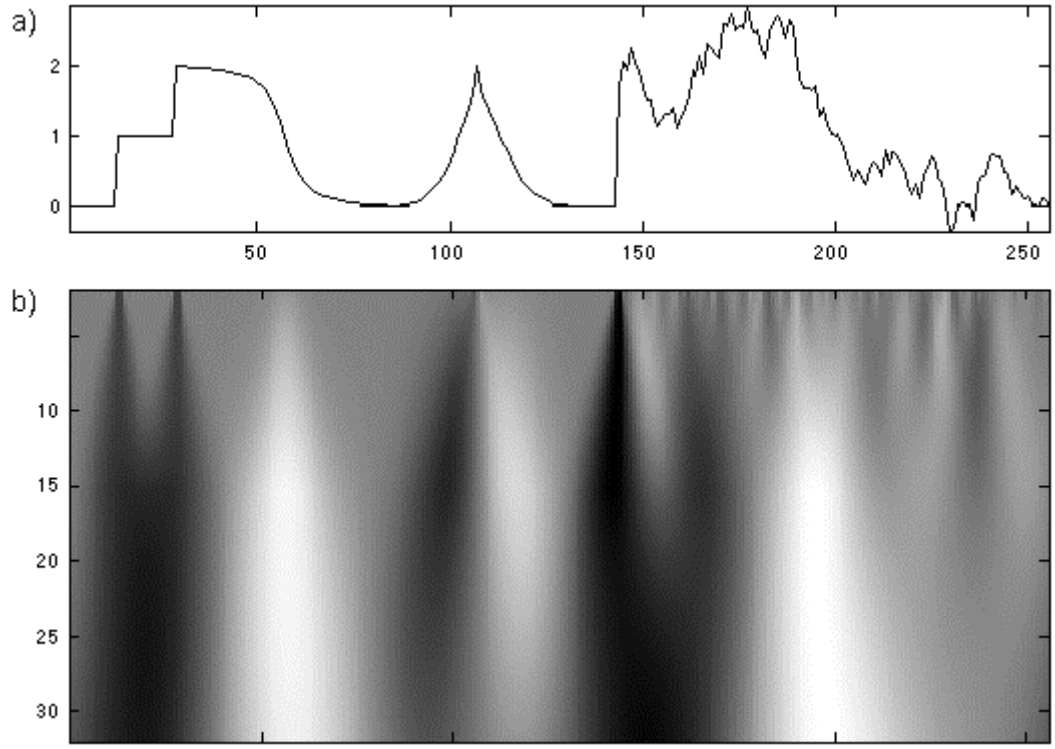



Fig. 20: Time-domain function and its scalogram of the over-complete WT[32]

A scalogram is similar to the spectrogram, it displays the wavelet coefficients in the planes time vs. scale (or level). As for the spectrogram, a scalogram is usually based on the energy of the wavelet coefficients [CHA99, transformees/Transforms.html]. In the picture, slightly more than 30 levels are analyzed. Singularities of the time-domain function appear as cones – they have frequency contents in the entire spectrum. Finer levels have good time resolution, so the peak of the cone (at finest level) indicates the exact position of the singularity.

### 4.4.3 The Discrete Wavelet Transform

The discrete wavelet transform (DWT) is calculated analogously to the CWT. Here is presented the dyadic DWT, which is scaled in powers of 2, resulting in the following discrete transform [STN96, 432]:

$$\psi_{j,k} = 2^{j/2}\psi(2^j t - k) \qquad b_{j,k} = \int f(t)\psi_{j,k}(t)dt$$

Equation 6: Forward DWT

[32] The picture has been taken from [CHA99]

The $b_{j,k}$ coefficients are the wavelet coefficients, analogous to the *F(a,s)* coefficients. The discrete inverse transform is straightly adding the translated, dilated wavelets, weighted by the coefficients[33]:

$$f(t) = \sum_{j,k} b_{j,k} \psi_{j,k}(t)$$

Equation 7: inverse DWT [VAL99, (13)]

## 4.5 Connection of Filter Banks and Wavelet Functions

Normally in practice, and in particular in this thesis, wavelet functions are not used for calculation of the DWT. And mostly, they are not even the starting point of development of a wavelet. Rather, they are derived from the low pass and high pass filters of the corresponding perfect reconstruction filter bank. For this, an auxiliary function, the scaling function $\phi$ is introduced. It can be calculated using the *dilation equation*:

$$\phi(t) = 2\sum_{k=0}^{N} h_0(k)\phi(2t-k)$$

$h_0$: low pass filter coefficients
N: number of filter coefficients
$\phi$: scaling function

Equation 8: Dilation equation [STN96, 22]

As the dilation equation is recursive to itself, there is not always a solution for $\phi$. The scaling function is a function in continuous time, but is not likely to be continuous; rather it may not be smooth and even contain jumps.

Finally, the wavelet function $\psi$ can be calculated from the scaling function with the *wavelet equation*:

$$\psi(t) = 2\sum_{k=0}^{N} h_1(k)\phi(2t-k)$$

H1: high pass filter coefficients
N: number of high pass filter coefficients
$\psi$: wavelet function

Equation 9: Wavelet equation [STN96, 24]

It can be seen that once the scaling function is known, the mother wavelet can be calculated directly – without recursion.

[33] Also here, foot note 31 applies.

The filter bank calculation scheme has the same output as the DWT – the detail coefficients are exactly the $b_{j,k}$ coefficients calculated by the DWT. The scaling index $j$ becomes the level, whereas the translation index $k$ corresponds to the time plane in the filter bank. Therefore, it is possible to calculate the DWT entirely without wavelet functions, as it is done with the tree-structured filter bank. Each level of the filter bank corresponds to one scale of the wavelet. Consequently, when the filters are known, the DWT can be computed exclusively with the filter bank. As the filter bank uses decimation for the scaling, it is referred to as *decimated DWT*.

The wavelet depends on the high pass filter; therefore it is logical that it creates the detail coefficients for a decomposition using the CWT or DWT. On the other hand, the scaling function corresponds to the low pass filter, so by applying it, the remaining approximation coefficients can be retrieved (analysis) or resynthesized.

To sum up, Fig. 21 shows a) scaling function, b) wavelet function, c), d), e) and f) the filter coefficients and g) the frequency responses of the analysis filters for the *Daubechies 2* wavelet. The Daubechies wavelet family has reached an enormous importance and was first developed by Ingrid Daubechies. It is of order 2, corresponding to 4 filter coefficients[34].

[34] More information on Daubechies wavelets will be given in ch. 5.2.2 on p. 46.

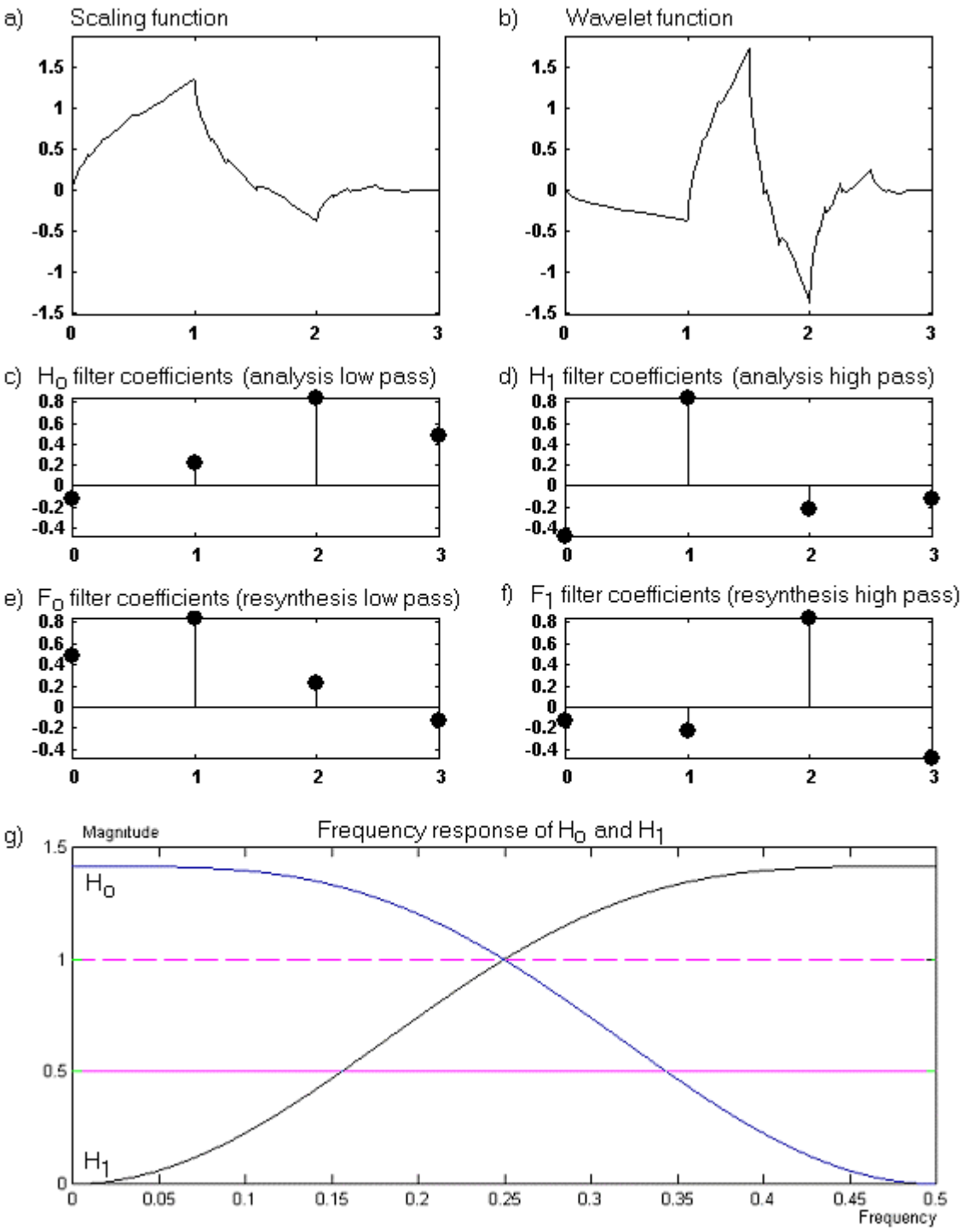



Fig. 21: Daubechies 2 wavelet

## 4.6 Properties of the Wavelet Transform

The WT is *linear*. This means that the transform of the sum of 2 signals equals the sum of their transforms [PPR91, 59]. Furthermore, the transform *conserves energy*, i.e. the energy of the signal equals the energy of the coefficients [PPR91, 59]. The local property of the grains allows *localization* of events of the original signal.

Wavelets approximate the signal. Thus the shape of the wavelet determines the accuracy. One important aspect for the accuracy is the number of *vanishing moments* of the wavelet function. A wavelet with p vanishing moments can approximate a polynomial of order p-1 [STN96, 227]. Thus, the more vanishing moments, the more "concentrated" the wavelet coefficients describe the signal [CHE96, chapter 2].

Furthermore wavelets can be classified according to [MMO96, 6-62] and [JAS94, 20]:

- Whether the **scaling function exists**. This is true when the low pass filter is known and Equation 8 has a solution.
- Whether **filters exist** (and thus the filter bank calculation scheme). Some wavelets are specified as a function for the wavelet function, and not all of them can be expressed as filters.
- **Orthogonality**: whether the filter bank is orthogonal or biorthogonal.
- **Symmetry**: Symmetric wavelet and scaling functions lead to respective symmetric filters. Compactly supported orthogonal wavelets cannot be symmetric both for low pass filter and high pass filter, as the alternating flip construction (Fig. 14) demonstrates. However, orthogonal wavelets can be *antisymmetric*. Symmetric and Antisymmetric filters are linear phase [STN96, 419].
- **Compact support**: Compactly supported scaling function and wavelet function lead to finite filters (FIR). Non-compactly supported wavelets should have a fast decay so that FIR filters can be approximated reasonably well.
- **Smoothness**: For many signal processing applications, smoothness is important, as changed wavelet coefficients shall result in a smooth output signal. Smoothness is connected to the regularity of the wavelet function.

## 4.7 Wavelet Applications

Wavelets are used successfully for signal processing. Most prominent field is compression of digital signals. Quite established are wavelet algorithms in digital image processing. The ability of the WT to extract the main features (most important for the eye) results in high compression without loosing much quality. The compression quality showed to be superior to the usual JPEG compression, which is based on a FT. The FBI adopted wavelet compression for their archive of digital fingerprint images [STN96, 364]. Also for video compression, wavelets are used successfully [VEK95, 431].

Noise reduction works well for similar reasons: low coefficients are likely to contain uncorrelated wide-spectrum noise. By setting coefficients below a certain *threshold* to 0, the image can be denoised. This method works well for audio signals, too, and will be explained in more detail in chapter 7.6.

Other fields of signal processing, where the WT is efficient, include detecting of singularities or breaks, determining long-term evolution of the signal, and pattern-recognition [MMO96, ch.4]

For sound processing, experiments have been done as described in [PPR91] and [CHE96]. Also compression of sound has been successfully developed with good results [STN96, 385].

Furthermore, wavelets can be used for linear algebra: [PTV94, 603f.] shows an application for solving linear systems efficiently by using the wavelet transform, [MMO96, 4-48] demonstrates its application for fast multiplication of large matrices.

## 4.8 The WT for processing real-time Musical Signals

It has been anticipated that the WT provides some features especially useful for processing musical signals.

Its multiresolution decomposition offers high temporal localization for high frequencies while offering high frequency resolution for low frequencies. A high frequency event (e.g. a cymbal crash) will be analyzed by many “fast”, short, and high frequency

wavelets. Low notes will be analyzed by “slow”, long, low frequency wavelets. Non-stationary transients can be located and analyzed well. Generally, this fails with Fourier analysis.

The logarithmic decomposition of the frequency bands resembles human perception of frequencies. The WT offers logarithmically equal frequency bands (octaves) while the FT has logarithmically low resolution for low frequencies. The WT adopts all advantages of a constant-Q transform.

For real time processing, the WT does not need a special window to be applied, as it decomposes the time by itself. So, the advantage of locality includes this advantage.

Many aspects depend on the analyzing wavelet[35]. Investigation is needed, which wavelet is suitable for the specific application. Additionally, for computation of the wavelet decomposition, it has to be decided how many levels (scales) are calculated. The WT needs considerable more parameterization then the FT.

The WT can be calculated efficiently with the pyramid filter bank algorithm. Although it is of complexity O(n), computation is in general more time-consuming than computing the FFT [ROA96, 589]. However, it is fast enough for real-time analysis and resynthesis of audio data.

The WT creates octave-wide frequency bands. So a fine analysis is not possible. Using more bands per level (scale) could be a solution, which has not been researched very much in respect to efficient computer based calculation. For many applications, the bandwidth of the octaves is fine enough, but for e.g. pitch detecting algorithms, a finer frequency resolution is needed.

In comparison with the FT, it can be said that the WT provides properties, which are well adapted for analyzing and processing real time audio data.

[35] or on the corresponding filters

# 5 Choosing a Wavelet for processing Musical Signals

Audio signals come in many different flavors. Classical music has different characteristics than speech, and both are again different to pop music. This thesis focuses on musical audio signals, without distinction of the characteristics. Speech signals could be regarded as a subset, so most assumptions for musical signals remain valid for speech signals. The idea is to develop algorithms that work well for many kinds of audio signals in real time.

## 5.1 Requirements

### 5.1.1 Quality

The quality of wavelet decomposition especially depends on the ability of approximating the signal with wavelets. When the applied wavelet does not resemble the shape of the analyzed signal, the wavelet coefficients will not extract the main "features" of the signal – resulting in many non-zero wavelet coefficients to approximate the signal. Thus, the better the analysis, the fewer significant wavelet coefficients result – they can be described as "concentrated" coefficients [CHE96, chapter 2, paragraph V].

Musical signals are always some kind of smooth wave, significantly smoother than pictures [STN96, 437]. Pictures may have sharp edges, fine lines and high contrast. Short filters corresponding to non-smooth wavelets like Daubechies 2 (see Fig. 21 on page 40) have proven to approximate well pictures. Musical signals, however, lead to the requirement of a sufficiently smooth wavelet, or in other words, a high regularity is preferred.

The size of the transition band of low pass and high pass filters is an important factor, too. Larger transition bands (i.e. low steepness), cause high overlapping of low pass and high pass bands. So the output bands of the filter bank are not separated well, and aliasing effects are enforced when the coefficients are changed [DEW97, 1899]. Especially in applications where the wavelet coefficients are related directly to frequency (i.e. in pitch shifting), highly separated low pass and high pass frequency

response is important. Recursive wavelet filters have been designed which greatly decrease the transition band, however they need a special implementation and could not be researched further for this thesis [MAL98, 253].

Furthermore, linear phase response is crucial for high quality audio filters. When the filters do not have at least an approximate linear phase, certain frequencies are delayed in the wavelet domain. The inverse transform undoes this phase distortion. However, when the wavelet coefficients are changed, unwanted modifications may occur to the frequencies, which are “out of phase”. Linear phase response can be achieved by using symmetric filters [STN96, 10].

Last, but not least, different wavelets have different temporal localization. Wavelets with short compact support can localize an event’s time better than others. So, for exact temporal analysis, a short wavelet is required, the faster decay the better. This conflicts with the ability of separation of the frequency bands and smoothness – there, longer filters provide better results [UYW99, 6].

### 5.1.2 Real-time Aspects

Especially in a real time environment, wavelet transforms lead to the requirement of a sufficiently fast algorithm so that the processor is able to compute the forward and inverse wavelet transform faster than the resulting chunk is played. In the example of chunks of 23ms duration, any processing of the chunk may not take more than 23ms – otherwise the flow of chunks will have breaks. The faster the processing has been completed, the better, as the remaining processor time can be used for additional processing on the audio signal, operating system tasks, etc. Additionally, some headroom is required, so that the real-time environment operates stable at any time, also when high peaks of processor usage occur. This headroom needs to be especially large for operating systems with preemptive multitasking, as the system may interrupt the chunk processing at any time for other tasks [EFF98, 5-13].

As the length of the filters directly affects computation time of analysis and resynthesis, shorter filters are preferred. However, in general, more vanishing moments and smaller transition bands lead to longer filters [STN96, 216]. As this is preferred for audio filters, a reasonable compromise of filter length has to be found.

Computers normally process integer numbers faster than floating point numbers [KIE97, 36]. It would be an idea to use one of the integer-based wavelet transforms, e.g. following the procedure described in [CDS96] or [UYW99]. However, integer sample values are not very well suited for high quality sound processing, resulting in round off errors, unsmooth waves, causing alias effects. High-quality audio processing systems can be assumed to work with signals in a floating point format, so using an integer transform would be of little benefit, while reducing overall quality. All modern PCs are equipped with fast floating point processors, so the performance impact is not very important. Also, the increasing popularity of other programs using floating-point calculations extensively (i.e. 3D games) plays a role for processor manufacturers to develop high performance floating point processing units.

## 5.2 Common Wavelets and their Properties

Some selected wavelets and their properties are presented in this section. In general, constructing a wavelet is not a very difficult task. However, highly sophisticated mathematics is involved when wavelets with special “good” properties are wished. All wavelets presented here were designed with such specialties, so their construction has not been trivial at all (maybe except Haar).

### 5.2.1 Haar Wavelet

The Haar wavelet is a special one. It has only 2 filter coefficients, so a long transition band is guaranteed. The wavelet function is a square wave; smooth audio signals cannot be approximated well. It is the only wavelet that is at once symmetric and orthogonal [STN96, 152]. Regarding computation speed, it is perfect for real-time processing. However, the quality is not sufficient: any modification of wavelet coefficients results in strong aliasing.

### 5.2.2 Daubechies Wavelets

The compactly supported and orthogonal wavelets created by Ingrid Daubechies in the late 1980’s gained much attention. They were one of the first to make discrete wavelet analysis practicable [MMO96, 1-31].

She constructed them by designing orthogonal filters with maximum flatness of the frequency response at 0 and one half the sampling rate (*maxflat filters*) [STN96, 164]. So the restriction for design was the highest number of vanishing moments for a given support width. For a given number of vanishing moments *p*, the filters have 2*p* coefficients. The minimum support constraint leads to maximum temporal resolution. The resulting filters and wavelets are called Daubechies *p* or just D*p*. For the special case of *p*=1, the resulting wavelet is Haar [MMO96, 1-31].

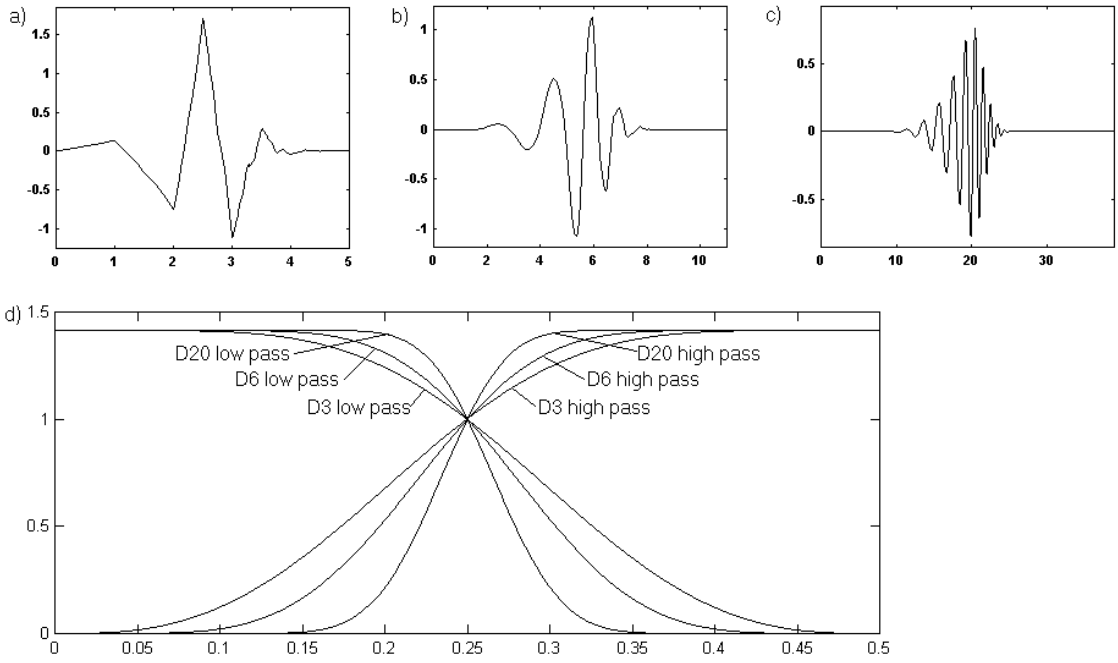



Fig. 22: Daubechies wavelet family

Most Daubechies wavelets are not symmetric – in contrary, some are very asymmetric. For small *p*>1, they are not smooth but still continuous. With increasing *p*, the wavelet function becomes smoother [STN96, 163]. For example, the D2 wavelet has singularities at the points $p/2^n$ (p and n integer) where it is left-differentiable but not right-differentiable [PTV94, 598]. Due to the flatness, the filters do not separate the frequency bands very well [DEW97, 1899]. The steepness of the filter's frequency response is proportional to the square root of 2*p* [STN96, 172].

Fig. 22 shows a) D3 wavelet, b) D6 wavelet, c) D20 wavelet. In d), the respective filter responses are plotted. It can easily be seen that the higher the order *p*, the steeper the transition curve.

### 5.2.3 Other Orthogonal Wavelets

Daubechies constructed a series of other orthogonal wavelets: “symmlets” have similar good features like the Daubechies family (compact support, *p* vanishing moments) but they were designed with the requirement to optimize symmetry and linear phase [MAL98, 252]. Still, as it is impossible for orthogonal wavelets, they are not perfectly symmetric.

Another family of wavelets (also constructed by I. Daubechies) are the so-called Coiflets. She constructed them on request of R. Coifman[36], who needed wavelets similar to the Daubechies family, but with an additional constraint on the scaling function: not only the wavelet function, but also the scaling function has to have *p* vanishing moments [MMO96, 6-66]. This has the advantage that the approximation coefficients can be approximated by the signal samples themselves. However, the support, and therefore the length of the filters, is longer (length of filter $6p$ instead of $2p$[37]), so this additional property costs efficiency.

A special wavelet family is the one of *Meyer* wavelets. The wavelet and scaling function are constructed in the frequency domain with an auxiliary function. Their support is infinite, but still the functions have a fast decay [MMO96, 6-69]. They are infinitely differentiable; furthermore they are symmetric and orthogonal, but have no vanishing moments. FIR Filters cannot be constructed, so a filter bank implementation is not possible.

### 5.2.4 “Crude” Wavelets

In [MMO96, 6-73], wavelets which lack many interesting properties are called “crude”: the *Morlet wavelet* and the *mexican hat*[38] both have an explicit expression for $\psi$, but a scaling function cannot be constructed. They have neither compact support, nor vanishing moments, and are not orthogonal. Due to these limitations, filters cannot be

[36] Often in literature, the name leads to the wrong conclusion that these wavelets were constructed by Coifman himself.
[37] In [MAL98, 253], a number of $3p$ coefficients is implicitly specified, though the filter of coiflet of order $p$=5 has 30 coefficients in Matlab.
[38] The mexican hat can be seen in Fig. 19 on p.35.

calculated, and only the forward CWT is possible. They are useful for mathematical demonstrations, as the wavelet function exists as a formula.

### 5.2.5 Biorthogonal Wavelets

There exist a number of well-studied biorthogonal wavelets. The major advantage of biorthogonal wavelets is the possibility to create symmetric transforms: both wavelet and scaling function are symmetric. This requires an odd length of both analysis filters [STN96, 111]. Biorthogonal wavelet functions and scaling functions are different for analysis and resynthesis, so for a filter bank transform, 2 analysis filters and 2 different resynthesis filters need to be used. Common practice for biorthogonal transforms is to indicate the analysis wavelet and scaling function with $\tilde{\psi}$ and $\tilde{\phi}$, respectively.

It is apparent that the filters may have different properties for analysis and resynthesis. Consequently, useful properties for analysis are designed into the analysis filters (e.g. vanishing moments) while the resynthesis filters may be designed in respect to useful properties for reconstruction (e.g. regularity) [MMO96, 6-68].

*Battle and Lemarié* introduced biorthogonal wavelets based on polynomial splines. For splines of degree *m*, the resulting wavelet function has *m*+1 vanishing moments [MAL98, 248]. Unlike Daubechies wavelets, they are not compactly supported; finite filters can only be approximated by cutting of at the edges. However, the wavelet function has exponential decay, so reasonably truncating the filters does not introduce much error [COH93, 4]. Polynomial spline wavelet functions can be specified explicitly in the frequency domain, and since they are polynomial splines, they are *m*-1 times continuously differentiable, resulting in quite smooth wavelets. For odd *m*, these wavelets are symmetric. An orthogonalization scheme allows making the Battle-Lemarié family of filters orthogonal [MMO96, 6-71]. In short, spline wavelets provide maximum regularity with symmetry and minimum support [STN96, 258].

Other biorthogonal wavelets are Binlets, also based on splines (proposed in [STN96, 217]). They are symmetric, have short support and the coefficients are binary: all

coefficients of a filter are integers divided by the same power of 2. This allows efficient implementation on computers – division by a power of 2 is "natural" for computers[39].

## 5.3 Decision

The parameterization possibilities of the wavelet transform provide a high degree of flexibility on its properties and performance. By fixing the wavelet and its parameters for the transform used in the example applications, the flexibility of the wavelet transform would be lost. It would degrade its potential; therefore no definitive choice shall be made. For example, when high separation of the frequency bands is needed, long filters with high demands on processing power are needed. By providing the length of the filters as a parameter to the user, the quality can be adjusted with respect to the performance of the computer. However, some decisions can be taken – mostly by exclusion.

The demand for linear phase leads to symmetric biorthogonal wavelets. A high degree of regularity and frequency band separation is preferred. On the other hand, temporal resolution is not a major concern – steep filters are more important. Biorthogonal spline wavelets provide all these properties. Studies of wavelet transforms for audio or audio-like signals agree on this ([CHE96, ch. 2], [DEW97, 1899], [STN96, 258]). Consequently, symmetric spline-based wavelets or Battle-Lemarié wavelets will be used for the example applications. Other wavelets are included for comparison purposes.

[39] This is due to the internal binary representation of integers. Bit shift commands effectively divide by a power of 2.

# 6 Computer-based Algorithm of the Wavelet Transform

This chapter outlines the implementation of the fast wavelet transform (FWT). The functions are used in the example applications in chapter 7.

## 6.1 Algorithm

The most important at first: the filter bank calculation scheme of the wavelet transform is the FWT ! So it is sufficient to implement the filter bank in order to have a fast calculation. As its complexity is O(n), it is among the fastest algorithms.

### 6.1.1 General Algorithm

The filter bank is applied recursively: each level (or scale) requires exactly the same algorithm. Furthermore, the high pass and low pass filters are the same for each level. So there is one function, which applies the filter bank to one level. An outer function successively calls this function for each level.

Each analysis level takes a set of input coefficients or samples and produces one set of detail coefficients (details) and one set of approximation coefficients (approximations). The number of each set of coefficients equals half the input coefficients or samples. The details are saved as output. The approximations are the new input for the next level. In this implementation, the filter bank is iterated until only 2 details and approximations are left – it is a nearly complete analysis. At the end of the decomposition, the overall output of the analysis transform is a set of detail coefficients for each level, plus the 2 approximations of the last level. As an example, when 1024 samples are analyzed, the forward transform produces 9 details: the $0^{th}$ level has 512 coefficients, the $1^{st}$ 256, and so on.

The inverse transform works analogously. It is started from the last level. To the 2 details and 2 approximations the inverse filter bank is applied. The result is 4 approximations, which are applied again to the inverse filter bank, along with the details of the before-last level. This scheme is repeated until the $0^{th}$ set of coefficients is resynthesized.

### 6.1.2 Matrix Representation

In the following, the transforms are represented as matrices. As convention, upper case letters are matrices, lower case letters are vectors. In general, the (vertical) vector x is the input signal, $a_i$ are the approximations of level i, $d_i$ the details of level i. Filters are denoted as the (horizontal) vector of the filter, where an index 0 stands for low pass and index 1 stands for high pass. Furthermore, h stands for an analysis filter, f for a resynthesis filter. Square brackets are used to indicate a specific element of a vector, numbering beginning with 0. So, $h_1[0]$ is the first analysis high pass filter coefficient. Filters have N elements, the last element is N-1. The index s is used for the level. The original signal corresponds to the approximations of level "-1".

A subscript T stands for the transposed matrix or vector. Empty elements in a matrix denote zeros.

### 6.1.3 Forward Transform

Following Fig. 16 on page 32, a one-level decomposition applies the low and high pass filters to the input and decimates each output. As a step-by-step algorithm, it would look like this:

Convolution matrix for filter h:

$$H = \begin{bmatrix} h[N-1] & h[N-2] & h[N-3] & \Lambda \\ & h[N-1] & h[N-2] & \Lambda \\ & & h[N-1] & \Lambda \\ & & & O \end{bmatrix}$$

Decimation matrix U:

$$U = \begin{bmatrix} 1 & 0 & & \\ & & 1 & 0 \\ & & & O \end{bmatrix}$$

Approximations level 0:

$$a_0 = U\left(H_0\, x\right)$$

Details level 0:

$$d_0 = U\left(H_1\, x\right)$$

Approximations level *s*+1:

$$a_{s+1} = U\left(H_0\, a_s\right)$$

Details level *s*+1:

$$d_{s+1} = U\left(H_1\, a_s\right)$$

Equation 10: Generic forward transform

The convolution is not exactly like presented in Equation 1: the input signal is shifted "up" by N-2. Effectively, this creates a delay of N-2 samples in the wavelet domain. As this delay is frequency-independent (in contrast to phase distortion), there is no impact on quality. The inverse transform undoes the delay.

Fortunately, analysis can be integrated into just one matrix for a complete one-level decomposition. The first step to this is done by using the associativity of matrix multiplication: H is multiplied with U first. Effectively, this deletes every second row. An example with a filter of length 3:

$$H' = U\,H = \begin{bmatrix} h[2] & h[1] & h[0] & & & \\ & & h[2] & h[1] & h[0] & \\ & & & & h[2] & \Lambda \\ & & & & & O \end{bmatrix}$$

Equation 11: Decimated convolution matrix

Like this, decimation is done before convolution; it reduces the number of operations by the half. The second step is to create one convolution matrix $H_d$ by appending the high pass convolution matrix below the low pass matrix. There will be only one output vector, which is composed of the approximations and appended, the details. This is called the direct filter bank matrix [STN96, 124.].

| Direct form of the filter bank matrix: | Analysis level *s*: |
|---|---|
| $H_d = \begin{bmatrix} H'_0 \\ H'_1 \end{bmatrix}$ | $\begin{bmatrix} a_{s+1} \\ d_{s+1} \end{bmatrix}^T = H_d\, a_s$ |

Equation 12: Forward transform with direct filter bank matrix

For an efficient implementation of the direct matrix, the rows of $H'_0$ are interleaved with the rows of $H'_1$. After the first row of $H'_0$ comes the first row of $H'_1$, then the second row of $H'_0$, then second of $H'_1$, and so on. It is called the block filter bank matrix $H_b$ (it is in *block Toeplitz form* or a *polyphase matrix in the time domain* [STN96, 114]). The output vector is composed of the interleaved coefficients of $a_0$ and $d_0$. The reason for this change of order is that in the computer algorithm, one loop is sufficient for calculating one approximation coefficient and one detail coefficient at once. Additionally, the input data is processed quasi-linearly, resulting in high locality of memory access. This optimizes performance as it favors the use of the fast cache memory of the processor.

### 6.1.4 Inverse Transform

The inverse transform is based on Fig. 18 on page 34. Expansion is also integrated into the inverse direct filter bank matrix (here by deleting every other column). The derivation is analogous to derivation of $H_d$. The resulting transform looks like this:

Expanded convolution matrix:

$$F' = \begin{bmatrix} f[0] & & & \\ f[1] & & & \\ f[2] & f[0] & & \\ f[3] & f[1] & & \\ f[4] & f[2] & f[0] & \\ M & M & & O \end{bmatrix}$$

Direct form:

$$F_d = \begin{bmatrix} F_0' & F_1' \end{bmatrix}$$

Recomposition of level *s*:

$$a_{s-1} = F_d \begin{bmatrix} a_s \\ d_s \end{bmatrix}$$

Equation 13: Inverse transform with direct filter bank matrix

The implementation optimizes this again by calculating high pass and low pass in one iteration of the convolution loop. For this, the columns of $F_d$ are interleaved, yielding the inverse polyphase matrix of the time domain.

It is interesting to note that $F_d$ is the inverse matrix of $H_d$. For orthogonal transforms, the inverse matrix is the transpose. And due to the construction of the inverse filters with order flip, i.e. f[i]=h[N-1-i], it can easily be seen that $F_d$ is the transpose of $H_d$. For biorthogonal transforms (which are not orthogonal), $F_d$ is biorthogonal to $H_d$ and therefore the inverse of $H_d$.

## 6.2 Implementation

Once the matrices are set up, the implementation is straightforward. For a one-level decomposition and a one-level resynthesis there is each a function, which implements the respective block Toeplitz matrix. In both functions the matrix is traversed in rows from top to bottom. There is an outer loop for the columns and an inner loop for the rows. The next paragraph reveals that several alternative functions have been created for analysis and resynthesis, addressing the discussed problems.

For a complete analysis/resynthesis, these one-level functions are called iteratively for each level – as described in paragraph 6.1.1 on p. 51.

A number of wavelet filters have been implemented: Haar, Daubechies, Spline biorthogonal, Battle-Lemarié, Symmlets and Coiflets, most of them in different versions with different number of filter coefficients.

## 6.3 Problems and Solutions

### 6.3.1 Variable-length Arrays

The wavelet coefficients have a different length for each level. Additionally, for most extension schemes, this length depends on the filters. Furthermore, the number of levels depends on the size of the chunks. It is inconvenient and requires some overhead to create one array for each level. Also a dynamic creation of the arrays for each execution of the transform is quite time consuming.

To address this problem, an own class is responsible for handling the variable length arrays of the coefficients. Internally, based on the number of input samples and length of the filter, one large array is created. During analysis, the filter coefficients are written successively to this array: first the details level 0, then details level 1, etc. The remaining approximation level is also written to the array, after all details. The start index and length of each level is stored separately. For wavelet-domain filters and recomposition, functions like `getDetails(level, &count)` provide fast access to the coefficients.

The complete array is never destroyed, so that `new` and `delete` operators do not decrease performance. Only in cases where the array is too small (e.g. the filter length has been increased), it is discarded and a new larger one is created.

### 6.3.2 Filter Coefficients

The filter coefficients need to be provided to the transform functions. These are stored in static arrays. For each wavelet, the low pass filter coefficients are specified in an

array. For biorthogonal wavelets, the high pass coefficients must be specified in a separate array. These arrays are passed to the transform functions.

A list of all available wavelet filters is statically constructed which contains the name, the different filter coefficient arrays with their lengths, and whether it is orthogonal of each wavelet. An initialization function calculates the high pass filters for orthogonal filters with the alternating flip, and the inverse filters of all filters is calculated using the alternating signs pattern. At last, all filters are flipped: As the convolution matrix always uses the filters backwards (last coefficient first), the reconstruction function can be simplified due to this flipped storage.

Most filter coefficients were calculated using the `wfilters` function of the wavelet toolbox of Matlab. For Battle-Lemarié and spline filter coefficients, the `lemarie` and `wspline` functions of the Uvi_Wave toolbox for Matlab [SPG96] were used.

### 6.3.3 Different Length of Biorthogonal Filters

Biorthogonal transforms may have different length for low pass and high pass filter. Handling this is difficult to implement efficiently: by using the optimized block Toeplitz matrix, low pass and high pass filters are calculated in parallel. Different lengths would require a significant amount of checks to be made in the inner loop of calculation. For simplicity, biorthogonal filters are padded with zeros in order to have equal length. This has a negative impact on performance (see paragraph 6.3.5). Further optimization measures may address this point.

### 6.3.4 Boundary Problems

The presented matrices for analysis and resynthesis hide one aspect: how do they end? Apparently, the convolution needs N input samples, but for the last input samples, there are not N following samples. For example, to calculate the last coefficient, convolution needs N-2 input samples after the last input sample. These do not exist. Another view of the same problem is with the wavelet functions: the first and last wavelet lap over outside the original window of input data. To address this problem, several common schemes exist which extend the input data (discussed in [STN96, 340], [MMO96, 6-47]

and [CHE96, ch. 2]). This is not an exhaustive list. However, the presented extension schemes seem to be the most used in wavelet processing.

*Circular convolution* assumes the input vector to be symmetric. The input signal is extended with the first few samples. It can be implemented efficiently, as the last rows of the matrix just wrap around. The major drawback of this solution is that the time information of the wavelet coefficients is inaccurate for the boundaries: modifying a coefficient at the edge has an impact on the other edge when resynthesized. For real time audio processing, this behavior is not acceptable. With small chunks and long filters, it comes close to the periodic behavior of the STFT.

*Zero padding* extends the input signal with N-2 zeros at end and beginning. This causes discontinuities at the borders. There are wavelet coefficients added, as the signal is effectively enlarged. For an input signal of length M samples, the $0^{th}$ level has (M+N-1)/2 details. On reconstruction, the padded values are discarded.

*Symmetric extension* assumes the signal to continue symmetrically at the borders. For this, the signal is mirrored at the boundaries. It does not create discontinuities at the borders and yields relatively low error. Like with zero padding, there are wavelet coefficients added.

*Smooth padding* extrapolates the samples at both boundaries. This is assumed to be good working for smooth signals, like audio signals.

All presented schemes give perfect reconstruction when the wavelet coefficients are not changed. The different types of errors only occur when the wavelet coefficients are modified.

The extension scheme only has an impact on the wavelet coefficients at the boundaries. The inner wavelet coefficients are the same with all extension schemes. Also, extension only plays a role for analysis: on resynthesis, the eventually added paddings are discarded.

In the implementation, the first 3 schemes are implemented as different analysis functions. Additionally, a variation of the symmetric extension using a history has been elaborated. The last N-2 samples and coefficients of the previous chunk are stored in a history object. These values are used as the extension of the left boundary in the next chunk. A parameter controls which extension scheme is used. Unfortunately, the author learned of the smooth padding scheme shortly before finishing this thesis. Therefore, it could not be implemented anymore.

Experimental results favor symmetric extension and history extension as the best-suited scheme from the implemented ones for musical signals: with different signals, they create smaller errors of the boundary samples than with zero padding or circular extension. It depends on the signal, which of the latter 2 schemes yields less error, but the subjective impression is always better with zero padding – the clicks due to the errors are less disturbing than with circular convolution. Also subjectively, symmetric extension and history extension sound best. Interestingly, they produce nearly the same amount of errors and sound equal.

For the circular extension scheme, the function provided in [PTV94, 597] has been retaken. It has been used as a reference implementation. However, it can only handle orthogonal wavelets and the length of the input vectors is restricted to be a power of 2. Due to the bad quality of the circular extension, this function has not been revised for biorthogonal wavelets and arbitrary length of input vector.

A “real” extension could be done by delaying the signal: The last N-2 samples of the current chunk are used as extension. The last N-2 samples of the previous chunk become the first samples of the current chunk. The problem here is that extension needs to be done at each scale of analysis – the input signal would only provide the extension for the $0^{th}$ level.

### 6.3.5 Performance

Like stated earlier, calculation speed of a real time filter is crucial for the performance and possibilities of the entire real-time system. Therefore, an optimized version of an analysis and a resynthesis function have been implemented. For the analysis function, only the zero-padding scheme has been implemented. However, the same optimizations

can be applied to the other analysis functions. As resynthesis is the same for all extension schemes (except for circular convolution), one optimized inverse function is sufficient.

As a first optimization, all increments are implemented with the “++” operator. A “+2”operation is replaced by applying twice the “++” operator.

Secondly, the inner loop is cleaned: in the generic implementation, an if-command checks whether the signal or the extension is taken as input. In the optimized version, there are three outer loops instead of one; one for the left extension, one for the non-extended signal and one for the right extension. This removes completely the inner if-command. A little overhead is added for the 2 additional loop blocks. This is not dramatic: e.g. for a chunk size of 1024 samples and a filter length N=20, a full decomposition executes 23,360 times the inner loop block, while the entire decomposition function is only called 9 times.

Furthermore, calculation of the filter coefficient index is removed: the loop variable of the inner loop is adjusted so that it corresponds exactly the filter coefficient index. This removes 2 additions per inner iteration.

Another optimization is to remove indexed array access: accessing large array elements by an index has a negative impact on performance. Rather, the input signal is accessed with a typed pointer - instead of increasing the index for the array, the pointer is increased. This needs extra variables for the inner loop, but experiments confirm the positive influence. The same applies to the output arrays. Due to their relatively small array indexes, the filter arrays are still accessed with an index. As expected, tests with pointers for the filters showed worse performance.

A test with unrolling the inner convolution loop did not have the desired effect: instead of a `for`-loop, the inner convolution statements are repeatedly copied in a `switch-case` element with fall-through. By the switch-variable it is possible to define where the statements start and thus the number of calculated elements can is defined. So for each inner iteration, one `if`-statement (for checking the `for`-loop condition) and one

conditional jump has been removed. But, unfortunately, this resulted in a slower execution. The optimization of the compiler seems to work well for `for`-loops.

Benchmarks show the effect of these measures: on the test system[40], with chunk size 1024 samples and filter length N=20, executing consecutively the generic forward and inverse transform takes 2.97ms processor time, whereas the optimized functions do it in 2.21ms, a performance increase of about 25%. With a filter length of N=40, 6.25ms and 4.76ms are used, respectively, yielding an optimization of about 24%. These values are approximate mean values: the processor usage varies from chunk to chunk of up to +/- 0.2ms, so an exact benchmark is impossible.

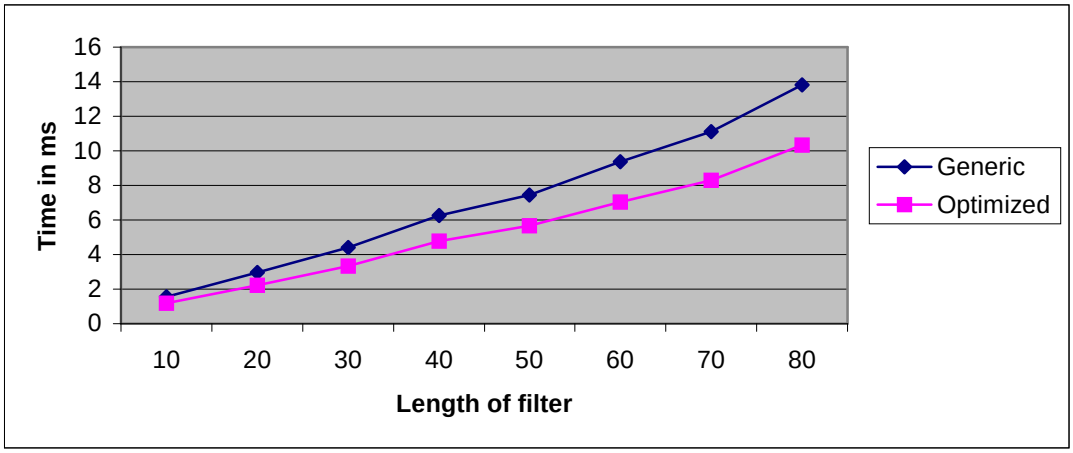



Fig. 23: Performance of the transform algorithm dependent on filter length

Fig. 23 illustrates the processor usage of the transform algorithm. The time is nearly proportional to the length of the filter. This manifests the complexity of O(n) also in respect to the filter length. The diagram does not reveal that the relative performance gain remains approximately constant – at around 24%.

More performance gain can be obtained by hard coding the filter coefficients and unrolling the inner loop as demonstrated in [PTV94, 596]. Further improvements can be made to handle biorthogonal wavelets: firstly, the already mentioned zero padding of filters with different length may be replaced by efficient handling of different lengths. Secondly, the symmetric property can be used to optimize even more.

[40] Intel Pentium II, 400MHz with 512KB second level cache.

# 7 Applications of Wavelets in Real Time Digital Audio

This chapter presents the program that has been written to use the wavelet transform for real time audio processing.

## 7.1 Coding Style

The author has a strong programming background in Java. This leads to a Java-like design of the C++ classes. The differences to standard C++ coding style are listed here.

### 7.1.1 Naming Conventions

As in Java, all names in the implementation are a concatenation of words, first letter of each word capitalized. Class names start with a capital letter; fields and variables start in lower case.

### 7.1.2 Interfaces

As C++ does not provide the concept of interfaces, they are implemented as abstract base classes. By multiple inheritage, they are attached to other classes. In this elaboration, these classes are called an interface. While this is not true in a C++ sense, it is true for their function and usage.

### 7.1.3 Object Orientation

The entire framework is designed purely object oriented. To this extent, fields are generally private or protected and get/set methods allow access to them.

## 7.2 The Audio Framework

A set of core classes has been designed to handle real time audio streams. In the following, their main features are presented. Appendix A provides descriptions of all classes and a class inheritance tree of the core classes can be found in Appendix B.

### 7.2.1 Audio Format

The audio framework uses exclusively linear floating-point samples. When data originate from a different encoding (e.g. PCM), they have to be decoded prior to be used in the audio streams.

Furthermore, audio data may come in several channels, e.g. stereo with 2 channels. The implementation can handle any number of channels. Channels are stored in parallel buffers. Interleaved samples (e.g. originating from audio files) are separated.

Another important parameter of the audio format is the sample rate. The framework works well with any sample rate.

The class *AudioFormat* stores the information about number of channels and sample rate. It also provides methods to transform a number of samples to ms and vice versa. For example, this serves to calculate the memory requirements and playing time of a buffer.

### 7.2.2 General Architecture

Streaming of audio data is implemented as a sequence of chunks of audio data. They are managed by the class *SampleBuffer*. The different sources (interface *AudioReader*), modifiers (interface *AudioFilter*) and destinations (interface *AudioWriter*) of the buffers are linked to form a chain. Sample buffers from sources and modifiers are *pulled*, buffers to destinations are pushed. In between is a *synchronizer*, which first pulls all data from the streams and then pushes it to all writers (class *AudioSynchronizer*). The timing reference is provided by a *ticker*, which fires an event when a new buffer needs to be filled (interface *TickProvider*).

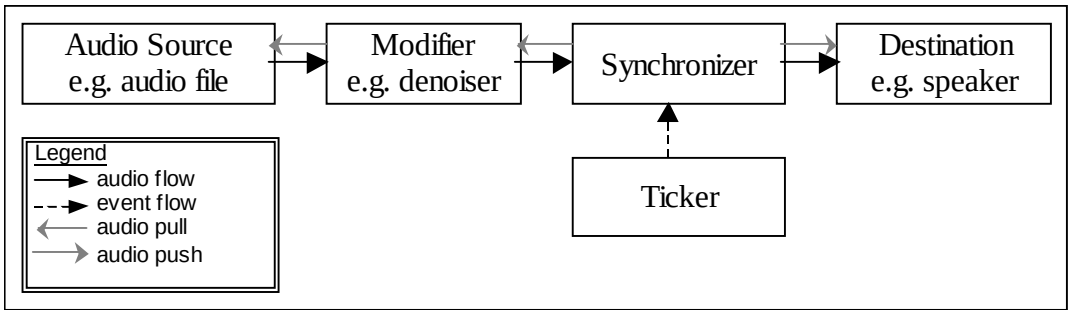



Fig. 24: Audio streaming example

An example of such a chain is presented in Fig. 24. Its architecture plays a file, which is modified using a denoiser. The audio flow arrows show the way of the audio data, push and pull operations are marked as gray arrows. More examples can be found in Appendix C.

The ticker plays a central role to synchronize the audio processing with time. It provides a stable clock, which depends on the length of the sample buffers. An event, the *tick*, is generated at regular intervals of the time of one sample buffer. A tick event is the signal to start filling the next buffer. As the *TickProvider* class is an interface, virtually any class may become a ticker. An useful class for a tick provider is for example the class that outputs data to the soundcard: the ticks can be synchronized with the timing of the soundcard, guaranteeing that no buffer arrives too late or comes too soon.

Once such a tick event arrives at the synchronizer, it initiates the pull chain: it asks the module "before" to fill the provided sample buffer. The asked modifier will ask its previous module, and so on. The first element in a chain finally will fill the passed buffer. Then the chain is proceeded back, in direction of the audio flow, each module filling or modifying the sample buffer. When the sample buffer arrives at the synchronizer, it pushes it to the modules "after" it. In this case, it is only one module, the speaker – on PCs mostly driven by a soundcard device.

One auxiliary class, *AudioMixer*, is provided. It does not really belong to the framework's core, but is useful in any implementation. However, it need not be used, it is provided with the framework for convenience. It is the reference implementation for an *AudioReader* that reads from several inputs and applies modifiers to the stream.

### 7.2.3 Extensions

The framework's core classes only provide the interfaces and some implemented classes like *SoundBuffer*. They do not include real functionality or chained components. All sources, destinations and modifiers have to be implemented as independent extensions[41]. This allows compiling the extensions into dynamically loaded libraries,

[41] Implementing the interfaces *AudioDeviceReader*, *AudioFileReader*, *AudioDeviceWriter*, *AudioFileWriter* and *AudioFilter*.

opening the possibility to extend the audio framework with new extensions without recompiling the main program. Loading of the extensions is not part of the core framework. It has to be provided by the main program, which uses the framework.

An additional extension type exists: encoders and decoders. They have the task to decode from a certain encoding to linear floating-point samples, and encode vice versa. The source and destination extensions can query the available encoders and decoders.

### 7.2.4 Independence of the User Interface

The entire framework, and especially the extensions, is generally independent of a UI and do not implement a user interface[42]. As many extensions (especially filters) may need to be configured by the end user, an interface for parameters is defined. Each filter implements some methods to query the parameters: number of parameters, and for each parameter: name, value range, type of display (slider, checkbox, combo box). How the parameters are displayed to the user depends on the UI. It would be possible to provide all parameters in a console-based UI. Also, automation of parameters can be achieved fairly easy. Furthermore, the filters can deliver feedback to the UI: when a filter changed a parameter by itself, it can send a notification back, so that display of the respective parameter can be updated. With this mechanism, informational parameters are implemented: to the end user, no possibility is given to change this parameter. It only displays a string. In this fashion, the statistics filter (see ch.7.4.6 on p.73) lets the calculated statistics of the audio stream be displayed.

### 7.2.5 Flexibility

Flexibility has been an important aspect for the design of the framework. As noted before, arbitrary sample rates and any number of channels can be used. No limits apply to chaining the modules: audio flow can be split up, combined, interrupted, etc. Of course, this depends also on a clean implementation of the extensions. The existence of the extension mechanism itself also offers a high degree of flexibility.

[42] Unless they provide graphical output, like the wavelet domain display filter (ch.7.4.3, p.70)

### 7.2.6 Performance

As the classes are used in real time, processor usage of the real time functions is to be minimized. Where possible, operations and calculations were implemented in functions that are not called in real time. E.g. temporary memory blocks are allocated once at beginning of real time operation instead of allocating and deallocating during real time usage. More measures to improve performance are detailed in the descriptions of the extensions in chapter 7.3 on pp.67f.

### 7.2.7 Robustness

A major concern of a real time audio framework is its robustness. It must run stable even in exceptional situations. The core system has not a significant impact on robustness: as the actual implementation is done in the extensions, robustness is mostly dependent on them. Chapter 7.3 on pp.67f. discusses the considerations on robustness for the relevant extensions.

Programming errors, which could e.g. lead to null pointer exceptions or memory leakage, are also crucial for robustness. Though, this is not specific to an implementation of an audio framework. Thorough testing and memory checks were made to minimize the probability of remaining errors like these. Also, extensive error handling is necessary for robustness and has been implemented as described below.

### 7.2.8 Error Handling

For consistent error handling, a set of C functions has been retaken from the author's work on the *Studienarbeit* [BOE99]. It provides centralized functions for outputting trace, debug and error messages. All messages are classified with a level. It is configurable, which levels cause a message or not. The output is also configurable: for console applications, it may be output on the console. GUI applications can present a dialog box in case of an error message and list other messages in a list box or similar. Additionally, all messages may be logged to a file. Exceptions are not used to favor portability.

### 7.2.9 Portability

The entire framework and most extensions are written in respect to portability to other platforms. Use of standard C++ should make it fairly easy to use the audio framework on other systems than Windows, where the framework has been developed and tested. Though, there are 2 classes, which need system-dependent implementation: *Lock* and *Thread*. They are declared as external in the core header file. To compile a program, an implementation of these classes must be linked to it. As not all platforms support threads, they must be implemented as dummy versions, which create an error when used. Threads are not used in the core system.

*Thread* provides an encapsulation for the system's mechanism for threads. Like this, an extension, which is system-independent except for the need of threads, becomes platform independent. Methods like `run` and `terminate` allow control of the thread. The `run` method takes as parameter a class that implements the *ThreadRunner* interface. *ThreadRunner* declares only one method: `threadRun`. It is called in the context of the newly created thread. Once it is finished, the thread has terminated.

For thread synchronization, *Lock* allows exclusive execution of a portion of code for only one thread at a time. Two methods, `lock` and `unlock` define the exclusive region. Locks may be implemented with mutex's or semaphores. Only with the use of locks, an application can be made thread-safe. Consequently, it has been used extensively in many classes.

Also one global function with system-dependent implementation is declared in the audio framework: `HighResolutionTime` returns a counter with very high precision. This can be used for performance measurements and benchmarks.

These system-dependent declarations are implemented for the Windows platform. The *Lock* is implemented using a Windows-specific *critical section*. `HighResolutionTime` returns a timer register of Pentium-class processors. Windows disposes of a function that enables to query it. On non-Pentium class systems, it simply returns 0.

## 7.3 Implemented Extensions

Here, the extensions of the framework are detailed. The “filters” – audio modifiers – are presented in the next chapter, although they are also part of the extension framework. Appendix A.3 lists descriptions of all extension classes.

### 7.3.1 Codecs

One codec extension is provided. It handles conversion from and to PCM audio data. The number of bits per sample must be passed to the codec so that it knows how to interpret PCM samples.

### 7.3.2 File IO

Two extensions handling files of the common format “Microsoft wave” are implemented: *WaveFileReader* subclasses *AudioFileReader* and provides an *AudioReader* which gathers the input audio from a file. Conversely, *WaveFileWriter* (derives from *AudioFileWriter*) is an *AudioWriter*, which writes the pushed data to a wave file. For format conversion, they use the codec extensions. Currently, only the PCM codec is supported. This can be enhanced easily. The wave file reader and writer are platform independent (although the file format originates on Windows).

### 7.3.3 Audio Devices

In order to play and record sounds, the 2 extensions *DirectSoundWriter* and *MMEReader* exist. As soundcard access is highly platform-dependent, these work only on Windows.

For device access, latency is a crucial factor for real time digital audio: the slower the soundcard, respectively the more time it needs to process a chunk, the higher the latency of the entire system. Real time behavior may only be achieved with total latency of under some 100ms. Higher latency than 400ms results in a noticeable delay – e.g. changing a parameter of a filter is audible later, which is very inconvenient. Other components of the audio framework do not add significant latency[43]. Consequently, it was the aim to provide minimum-latency access to the soundcard.

[43] except when a delay is implemented by design

Another important aspect is stability: even a 10ms break of the flow of audio data disturbs significantly (and destroys a recordings). So the soundcard must be accessed in a way that minimizes the risk of unwanted breaks. Additionally, all exceptional situations have to be handled cleanly. If the audio framework is too slow to push the chunks in time to the *DirectSoundWriter* (*buffer underrun*), a good reaction has to be implemented Conversely, pushing the chunks too fast (*buffer overflow*) must be considered, too. Underruns and overflows exist in the same sense for the *MMEReader*.

Exception handling needs to minimize audible effects and let the system remain stable. In both classes, a circular buffer is used for handling these exceptions: too many pushed buffers overwrite old ones, so it results in a jump in playback. When the buffers arrive too slowly, the circular buffer contains too little samples for playback and silence is appended. For the *MMEReader*, underruns cause silence to be generated and overflows (i.e. the chunks are not pulled fast enough) cause skipping of recorded audio data.

The extension names reflect the driver model they use: DirectSound is a relatively new model to access the soundcard on a very low level basis. While being quite complicated to implement, it rewards with low latency. The speed of the soundcard is supposed to be used to the maximum extent. DirectSound, initially developed for game sound, evolved continuously with the time. Meanwhile, DirectSound version 8 is about to be released. However, the Windows NT platform only supports up to version 3, therefore in this implementation, only features of version 3 have been used. In general however, DirectSound offers lower latency on Windows 95, 98 or 2000. There, DirectSound offers latency as low as 20ms.

For recording, a DirectSound architecture exists, but unfortunately only in version 5 and later. Therefore, MME, the older driver model of Windows, is used. It dates back to the times of Windows 3.1 and exists on all Windows platforms since then. Highly evolved soundcard drivers provide good performance: with this implementation it is possible to have latency down to 69ms.

Both *DirectSoundWriter* and *MMEReader* implement the *AudioTickProvider* interface. This offers direct synchronization with the soundcard: for playback, a tick event is fired

when one sample buffer may be pushed. For recording, the event is fired every time when the number of samples of one sample buffer has been recorded.

## 7.4 Implemented Filters

In this chapter, the term “filters” does not necessarily stand for frequency modification: it means an audio framework extension for modifying audio – implementing the *AudioFilter* interface. Some of them are auxiliary filters that supported development of this thesis.

The main filter extensions and their implementation are presented in detail in chapters 7.6f. Appendix A.4 provides a list of all classes.

### 7.4.1 Wavelet Transform

These 2 filters, *WTForwardFilter* and *WTinverseFilter* encapsulate the wavelet classes elaborated in chapter 6 on pp.51f. as filter extensions.

*WTForwardFilter* performs the forward wavelet transform on the current chunk. The Parameters allow choosing a wavelet and the extension scheme. Furthermore, an eventually existing optimized version can be selected. In order that following filters can access the wavelet coefficients, a special field in *SampleBuffer* is assigned a *WTFilterInfo* instance, providing information about the effectuated transform and the calculated coefficients in a *WaveletCoeffs* object. Following filters, which need the wavelet domain, can check this field; if it exists (i.e. not equal to NULL), they use the wavelet coefficients, nothing is done.

*WTInverseFilter* applies the inverse WT to the wavelet coefficients, if existent. The resulting time domain samples are written to the *SampleBuffer*. An information field provides the DC offset difference: the last sample of the last buffer is compared to the first sample of this buffer. The lower the difference, the smoother is the resulting boundary. It serves as an estimator for quality: higher difference creates audible discontinuities. To overcome the discontinuities at the boundaries, 2 special modes are implemented. They shift the entire signal to compensate the border discontinuities. Though not guaranteeing continuous $1^{st}$ or higher derivatives, a significant improvement

of clicks at the boundaries can be achieved. However, as the samples are shifted up or down, the DC offset is modified and therefore dynamic range decreased. This is unacceptable for high-quality audio processing; consequently this feature is used for experiments only.

### 7.4.2 Show Wavelet Function

Showing the wavelet function $\psi$ is done by a simple trick: setting all wavelet coefficients to zero, except one single coefficient, will result in the time-domain wavelet function (the grain), created by the inverse transform. This filter allows controlling the scale, temporal position and amplitude of the single non-zero coefficient. By adding a time domain display filter (see ch.7.4.7, p.73) after the inverse WT, the wavelet function can be seen and explored. Changing scale and time shows the effects of dilation and translation.

The *ShowWLFunctionFilter* is an effective means to better understand wavelets and their implementation as filter banks. Furthermore it serves as validation of the inverse wavelet transform and the selected wavelet.

### 7.4.3 Wavelet Domain Display

This filter is specific to Windows platforms. A window is created which displays all wavelet coefficients in the time vs. scale domain (see Fig. 25), also called the scalogram[44]. The amplitude or power of a coefficient is represented as colors. As this is done in real time, every modification of the wavelet coefficients can be monitored visually. The extension schemes (see chapter 6.3.4 on p.56) and their behavior with different wavelets can be compared intuitively. The window can be resized to any wished size.

Many options of the *WaveletDisplayFilter* allow to control display and function. Paddings can be shown optionally, and coloring can be based on absolute amplitude or energy of the coefficients. The color mode parameter allows different color palettes to be used: examples are gray values or fading from red to blue. A good result delivers a color palette, which fades from blue colors for low coefficients over red to yellow.

[44] more information in ch.4.3.4 on pp.31f.

Another parameter controls scaling of colors: e.g. when all coefficients are quite low, scaling the colors up improves visibility of the coefficients. The “vertical scale” parameter chooses the relative height of the different scales of the coefficients’ scales. A vertical scale parameter of 1 shows the *constant*-Q: every scale has the same height. The other extreme value of this parameter, 2, shows linear frequency range for the scales, each scale has double the height as the next scale.

Furthermore, multiple consecutive chunks can be shown at once with the “number of chunks” parameter. 2 parameters control the speed of display: the first allows pausing a


Fig. 25: Wavelet domain display filter (vertical scale=1.5)

specified number of chunks after display of a chunk. This slow motion mode allows studying individual chunks in more detail. It is synchronized with the same parameter in the time domain display (ch.7.4.7 on p.73). Like this, time domain and wavelet domain can be compared efficiently. The second speed control allows switching to “synchronous” display: the window is painted while the chunk is processed. If this parameter is switched off, a message is sent to Windows to initiate a repaint of the display as processor time permits. The latter results in a slower display (i.e. intermediate chunks cannot be seen), but only uses processor resources when available.

### 7.4.4 Noise Generator

This filter adds white noise to the audio stream. Essentially, white noise consists of uniformly distributed random values. They are retrieved by the C function `rand()`. A parameter allows the choice of using directly the normalized random values or to create gaussian noise: it has zero mean, unit variance and a gaussian (or normal) distribution by using the Box-Muller method. The gaussian noise implementation was inspired by [EMB95, 158]. A second parameter lets control the volume of added noise. The filter is system-independent. It is used for validation and testing of the noise reduction filter (chapter 7.6 on pp.78f.).

### 7.4.5 Difference Listener

For comparison of the original and modified audio signal (e.g. modified by the equalizer filter), the difference of both signals can be used. The difference signal contains the amount of samples that have been changed. The amplitude of the difference samples represents the amount of change. This filter is used for 3 main applications:

Listening to the different signal can give instantaneous, intuitive information about the applied modifications. Frequency content and amplitude can be estimated quickly.

Statistics on the difference signal provide important quantitative results. This can be done, e.g. with the statistics filter explained in the next paragraph.

An additional usage of this filter is to reverse the effect of a filter: for example, the difference of a low pass filter results in a high-pass filtered signal.

The implementation is done by 2 filters: *DifferenceBeginFilter* and *DifferenceEndFilter*. The first takes a snapshot of the current audio stream. The *DifferenceEndFilter* calculates the difference of the current signal and the snapshot. Thus, all filters that are to be analyzed with the difference signal need to be applied in between this pair. Both filters do not have any parameters and they are platform-independent.

### 7.4.6 Statistics Filter

This platform-independent filter calculates constantly some statistical data on the audio stream. They are provided as informational parameters and can be displayed by the GUI. 4 values are calculated: minimum sample value, maximum sample value, mean value (DC offset) and energy. An additional parameter lets the user control how often the values are displayed.

The *StatisticsFilter* provides data that can be used for quantitative estimation of the quality of extension filters (with the difference listener) or to validate signals, e.g. from the noise generator.

### 7.4.7 Time Domain Display

A time domain representation of the signal is displayed by the *TimeDisplayFilter* in the dimensions time vs. amplitude. Like for the wavelet domain display (chapter 7.4.3), a separate window is opened – so it is also specific for the Windows platform. Most parameters are analogous to the parameters of the wavelet domain display. Here, the "vertical scale" parameter allows stretching/compressing the display on the vertical axis. Pausing of chunks is synchronized with the wavelet domain display: when both pause the same amount of chunks, they display exactly the same chunk(s). As it is possible to display more chunks at once than the number of paused chunks, a scrolling view of the waveform can be created.

The time domain display is needed to display the wavelet functions with the *ShowWLFunctionFilter* described in chapter 7.4.2.

### 7.4.8 Miscellaneous Filters

2 filters are implemented which were only used for auxiliary purposes.

The *DelayFilter* adds an echo to the signal. Short portions are repeatedly added to the signal, with decreasing volume. 2 parameters allow controlling length of the portion and amount of attenuation for repeated portions. It has been the first developed filter and was used for validation and support of development of the audio framework.

The *BWLowpassFilter* implements an IIR low pass filter by using the Butterworth filter design. The implementation has been taken from [BAL98]. 3 parameters are offered: gain, cut off frequency and resonance. Due to its IIR design, this Butterworth filter has a fairly high steepness. It showed to be helpful for visualizing the amount of separation of the filter bands of the wavelet transform: frequency bands above the cut off frequency should contain zero or small wavelet coefficients. Different wavelets can be compared using the wavelet domain display (see ch. 7.4.3).

## 7.5 The GUI

A GUI has been developed for the Windows platform. It does not use all of the framework's possibilities: a simple chain with 2 readers and 2 writers is used. In-between can be put any number of filters. A diagram of the chain can be seen in Appendix C.1.

The flow of audio data is symbolized with arrows, as can be seen in the screenshot (Fig. 26). With a checkbox, the sources and destinations can be activated. A help button provides basic instructions on how to use the GUI.

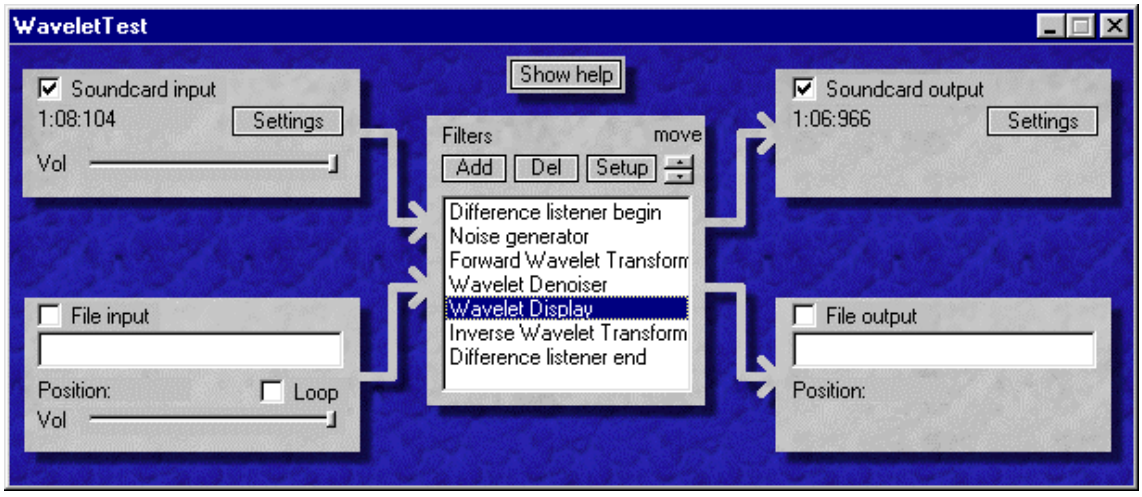



Fig. 26: Screenshot of the GUI

The screenshot shows an example of using the wavelet denoiser: noise is added to the audio signal with the noise generator (ch.7.4.4). In the wavelet domain (wavelet filters, ch.7.4.1), the signal is denoised using the denoise filter (see ch.7.6). The wavelet domain is visualized with the wavelet domain display filter (ch.7.4.3), where the effect of different denoising parameters can be followed visually. Finally, the entire set of

filters is surrounded by the difference listener (ch.7.4.5). This allows to hear the difference signal of original and denoised signal. If denoising were perfect, the difference signal should be complete silence. Any existing sound in the difference signal corresponds to unwanted modifications done by the noise reduction filter. This setup allows finding the right parameters for the denoiser efficiently. A slightly different setup, with the "Difference Listener Begin" filter applied after the noise generator, would allow listening to the noise, which has been removed by the denoiser.

### 7.5.1 Soundcard IO

Audio data can be retrieved and played using the soundcard. The extensions *MMEReader* and *DirectSoundWriter* are used for that. For the soundcard input, a slider lets the user control the volume of the audio data that is fed into the system. While running, the running time of the respective reader or writer is displayed in minutes:seconds:ms (see Fig. 27).

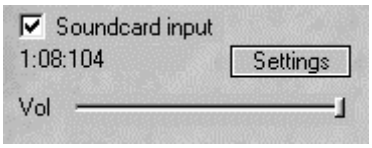



Fig. 27: Soundcard input

Internally, the soundcards are used in 16bit resolution, 44100Hz sample rate and in stereo. Consequently, the entire application runs at 44100Hz, with 2 channels.

A setup dialog (see Fig. 28) lets the user choose the soundcard (if several soundcards are installed in the computer) and define the buffer size. The buffer size determines directly the latency. Furthermore, it displays when buffer underruns or overflows occur. The setup dialog is invoked by pressing on the respective "Settings" button. The setup dialogs for input and output have the same layout and functionality.

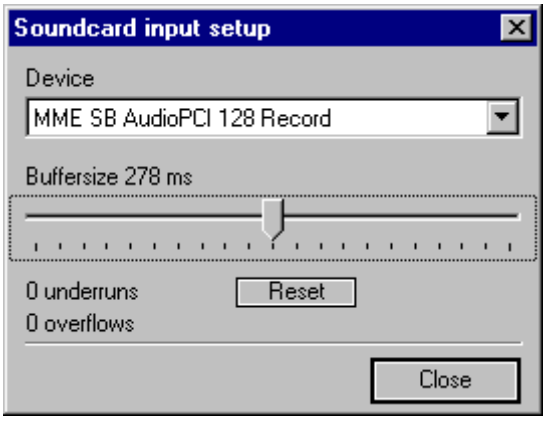



Fig. 28: Soundcard setup dialog

The tick source (see ch.7.2.2 on p.62) is assigned dynamically: if available, the soundcard output (*DirectSoundWriter*) is used, otherwise, the soundcard input (*MMEReader*) is used.

By activating both soundcard input and soundcard output, an audio loop is created: the signal that comes into the soundcard is immediately routed to the soundcard output, with eventual processing in between. This corresponds to the behavior of a stand-alone effects generator: it can be used for live presentations, recordings, etc. For example, if a user wishes to copy a record to CD using a stand-alone CD recorder. Music from the record shall be denoised using the noise reduction filter (see ch.7.6). The user would connect the record player to the soundcard input, and the soundcard output to the CD recorder. On the computer runs this application with soundcard in and out activated. Like this, the CD recorder burns a CD with the denoised music.

### 7.5.2 File IO

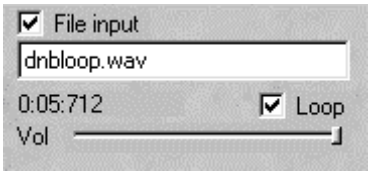


Fig. 29: File input

To use a file as input, either exclusively or in addition to soundcard input, a filename has to be provided in the text field, and the file input checkbox has to be activated. This feeds the audio data of the file into the flow of the system. As for the soundcard input, a slider lets the user control the volume. The "loop" checkbox lets the file be played repeatedly. Fig. 29 shows an example: the file "dnbloop.wav" is played repeatedly with full volume. The current playing position is 5 seconds and 712ms.

The file output works analogously: when a filename is entered in the text field, activating it causes a wave file to be created to which is written continuously the output of the system. When the file output is deactivated, the file is closed and can be used further: e.g. one can enter the filename in the file input box and use it as input audio stream, or it can be burned to a CD.

For file IO, the extensions *WaveFileReader* and *WaveFileWriter* are used (see ch.7.3.2, p.67). Consequently, only audio files in the format Microsoft Wave are handled. Furthermore, as the program uses exclusively 44100Hz sample rate, only files with this sample rate are accepted as input; generated files will have this sample rate. Mono input files are transformed into a stereo stream.

### 7.5.3 Filters

Any number of filters can be inserted in the audio flow. A list displays the integrated filters. Internally, an *AudioMixer* class is used (see ch.7.2.2 on p.62). This class also mixes the 2 input streams from soundcard and file input.

Buttons control the chain of filters: the “Add” button presents a dialog, which contains all filters, registered as an extension (Fig. 30). The name of each filter is retrieved by querying the filter object. Clicking on a filter name displays a short description of the filter. Also the description is provided by the filter itself. Pressing “OK” appends the selected filter to the list of filters. One filter type can be added any number of times.

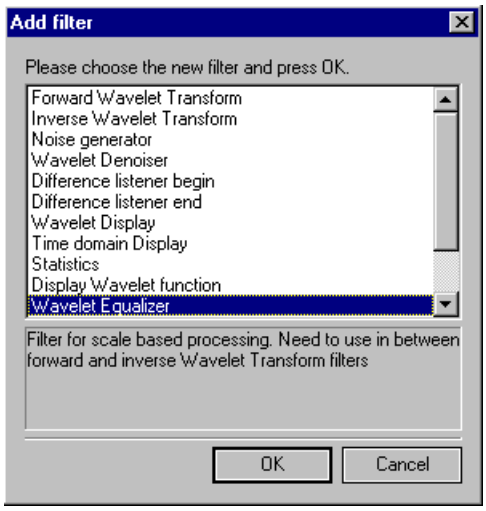



Fig. 30: Add filter dialog

The ”Del” button removes a selected filter from the list. The filters are applied in the order in which they appear in the list. To change the position of a filter, the up/down buttons are used.

The “setup” button displays a dialog, which contains all parameters of the selected filter. The dialog is created dynamically: the respective filter object is queried for the attributes of the parameters. For each parameter, the corresponding GUI element is created: a slider, a checkbox, a combo box, or only an informational field. Fig. 31 shows the setup dialog for the “wavelet domain display” filter, demonstrating the first three types of GUI elements. Additionally, for each setup dialog, a checkbox on top left lets control whether the corresponding filter is active (this is not a parameter of the filter). On top right of

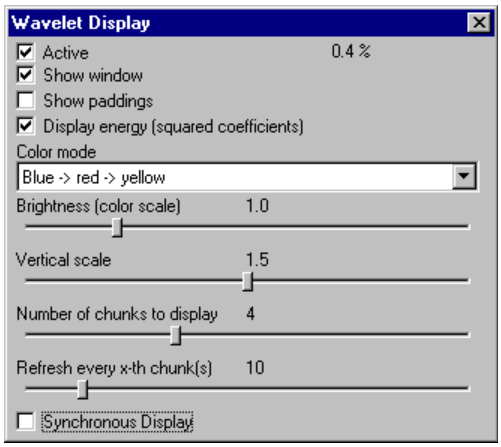



Fig. 31: A filter setup dialog

the dialog, the relative processor usage is displayed: it is calculated with the `HighResolutionTime` function (see ch.7.2.9 on p.66).

As the dialogs are generated dynamically, several dialogs can be displayed at once, one dialog for each filter.

## 7.6 Noise Reduction

### 7.6.1 Overview

Noise is an unwanted, often wideband sound, which is generally unwanted. It reduces the dynamic range of a system. Many different types exist; correlated and uncorrelated to the signal's amplitude or its frequency, continuous "noise floor" or noise transients. The most frequent is a wideband noise floor. It occurs in many elements of an electronic audio system: analog circuits like amplifiers, ADC's and DAC's add quantization noise, tape recordings have "tape hiss". Many elements like cables reduce the signal's amplitude, reducing signal-to-noise ratio (SNR). To compensate this, the signal needs to be amplified, which then adds noise [EMB95, 158].

Wideband noise can be "filtered" by the ear, up to a certain degree. Humans can concentrate on the non-noisy parts of the signal, and do not notice the noise floor. However, in pauses, where only noise is audible, it is better noticed. Also, different types of noise decrease perceptual quality. As noise reduces the signal-to-noise ratio, it reduces overall fidelity. Consequently, noise is unwanted.

### 7.6.2 Conventional Noise Reduction

Much research has been done to reduce noise. Tape hiss[45] can be lowered by using special noise reduction systems like the ones developed by Dolby. They dynamically compress the signal before recording. On playback, the signal is expanded to its original amplitude distribution. The recorded signal on tape has a limited dynamic range, so the low dynamic regions, where tape hiss occurs, are not used. However, these systems alter

[45] A special kind of noise that originates on analog recordings on magnetic tapes.

slightly certain parts of the signal and can have hearable distortions sometimes [ROA96, 396].

Another way to reduce noise perception is a *noise gate*. It monitors the amplitude of the signal. Once the signal's amplitude falls below a certain threshold, the signal is muted completely. The noise gate does not remove noise; it only eliminates the noisy pauses, where noise is perceived especially well.

A common technical for digital reduction of noise is done by special filtering. At first, the pure noise is analyzed. Its frequency response is used to construct a filter, which removes the frequency components of the analyzed noise [EMB95, 158]. In this case, where the frequency distribution is known, also FFT-based algorithms are very effective [RKK00, 2], but have less applicability on non-stationary parts of the signal. It is difficult to use these algorithms for real-time noise reduction: the frequency analysis can only be done with a noise-only signal. When the type of noise changes, the filter needs to be recalculated. Additionally, the filters also reduce frequency components of the original signal that lie in the noise spectrum.

### 7.6.3 Wavelet-based Algorithm

Noise reduction using the wavelet transform has been initially researched by Donoho [GRA95, 12]. Using the WT for denoising is superior compared to conventional techniques explained above, because denoising is done in different scales with different time resolution.

Two major wavelet-based denoising algorithms have been developed by Donoho: linear denoising, where noise is assumed to consist of high frequency components. The corresponding scales (the finer scales) are set to zero [RKK00, 2].

The second algorithm, *non-linear denoising*, or *wavelet shrinkage*, assumes noisy data to have low energy in the wavelet domain. This corresponds with the ability of the wavelet transform to extract the main (correlated) "features" of a signal. Two variants were developed by Donoho: *hard thresholding* sets all coefficients below a certain threshold to zero, maintaining all others. *Soft thresholding* also sets the low coefficients

to zero, but reduces all other coefficients by the threshold. Like this, the wavelet coefficients are smoother, but the energy of the signal is heavily modified [LGO95, 4].

Non-linear denoising proved to work well for many different types of signals: 1D medical, 2D geophysical and synthetic aperture radar signals [LGO95, 1]. Images can be denoised successfully, too. As non-linear denoising is dependent on the signal (i.e. it does not remove anything when all coefficients are greater than the threshold), it is even used in fields where not noise is to be removed: e.g. for removing blocking artifacts of JPEG compressed images [LGO95, 1].

Non-linear denoising can remove many kinds of noise; it is not necessary to know the type of noise as for conventional algorithms.

Many refinements of non-linear denoising have been developed. Most notably, algorithms exist to find the optimal threshold. Other algorithms use a non-decimated WT. For further information, the reader is referred to [LGO95], [RKK00], [COI94], [GSB97], [MMO96, 6-82f.].

### 7.6.4 Implementation

The implementation is done as a filter extension in class *DenoiseFilter*. The 2 types of non-linear denoising, soft and hard thresholding, can be chosen with a parameter. Other parameters are the threshold, how many levels are to be denoised, and whether the paddings are included for thresholding. The parameters allow flexible control about the denoising process.

In tests it has been found out that denoising low levels is not very effective. It results in phase distortions of lower frequencies, which cause discontinuities at the chunk borders. These can be perceived as clicks. Additionally, denoising of lower scales did not improve perceptual amount of noise reduction. Therefore, the number of levels to be denoised can be chosen. Wavelet shrinkage of the first 5 levels showed to work well with many kinds of musical signals.

The implementation of the non-linear noise reduction algorithm has been straightforward: the wavelet coefficients are examined one after another. When its

absolute value falls below the threshold, it is set to zero. For soft thresholding, additionally all other coefficients are reduced by the threshold.

### 7.6.5 Results

In general, noise reduction gave good results. With low noise levels, denoising worked very effective. The soft thresholding method showed to work well, hard thresholding created audible artifacts. Many types of noise can be removed from many different audio signals. The audio part of the accompanying CD-ROM presents some audible examples how to denoise recordings (see appendix D.7 for more details).

Testing has been done with synthetic white gaussian noise created by the noise generator filter (see ch.7.4.4 on p.72) and with real signals that have a significant amount of noise. The accompanying CD-ROM contains some audible examples of original pieces of music and their denoised version (see Appendix D).

Still, some problems occurred. The threshold parameter effectively controls the amount of noise to be removed. With a sufficient high threshold, it is possible to remove also high noise levels. However, a significant amount of the musical signal is removed, too, especially high-frequency components tend to be affected.

Additionally, noise is not removed uniformly: with lower threshold levels, which do not affect the signal much, spurious noise components remained. These do not correspond to a noise floor – they are quite audible and sound grainy. Therefore, parameterization is crucial, and a tradeoff between amount of removed noise and modification to the original has to be done.

For quantitative estimation of efficiency, selected pieces of music have been denoised and an error estimation has been calculated. This method is described in [RKK00, 8], and works only when the original signal is known. The error estimation is $\hat{E} / E$, where $\hat{E}$ is the square root of the energy of the difference of original signal and denoised signal, and $E$ is the square root of the energy of the added noise. The condition $\hat{E} / E < 1$ guarantees successful removal of noise [RKK00, 8].

For all tests, a Battle-Lemarié wavelet has been used with 49 filter coefficients. The denoiser has been set to denoise the first 5 levels, paddings inclusive, using soft thresholding. The piece of music[46] has a wide frequency usage and some short transients.

At first, the energy of pure noise at different amplitudes has been collected to calculate $E$. Then, two series of measurements of $\hat{E}$ were done: in the first, the threshold has been set to yield a minimum error[47]. Secondly, the threshold has been set by ear: where the least noise was noticeable, with acceptable modification of the music.

| Noise amplitude[48] | Series 1: minimum error | | Series 2: threshold by ear | |
|---|---|---|---|---|
| | threshold | $\hat{E} / E$ | threshold | $\hat{E} / E$ |
| -37dB (1.4%) | -58dB | 0.956 | -50dB | 1.121 |
| -34dB (2.0%) | -50dB | 0.977 | -47dB | 1.063 |
| -32dB (2.5%) | -50dB | 0.921 | -45dB | 1.012 |
| -30dB (3.2%) | -51,5dB | 0.896 | -43.5dB | 0.940 |
| -27dB (4.5%) | -44.5dB | 0.840 | -40dB | 0.871 |

Table 1: Noise reduction measurements

The quantitative error estimation shows that the implemented algorithm is able to remove successfully white gaussian noise in a real time environment – without specifically adapting the algorithm to the type of noise. $\hat{E} / E$ stays below one in all measurements in series 1, even for higher noise levels. However, when listening to the denoised music from series 1, noise is still audible.

When the threshold is adjusted by ear, the error estimate is greater, sometimes even above 1. Listening to the output of series 2, noise is effectively removed. However, also music is removed, which explains the increased the error estimate. This also shows that numbers are not sufficient for evaluating the performance of audio filters, listening to the output is always necessary, too.

[46] Dnbloop.wav – on the CD-ROM in directory “Program”.
[47] found by trial and error
[48] the value in parenthesis gives the linear volume

## 7.7 Equalizing

### 7.7.1 Overview

An *equalizer* or *spectrum shaper* changes frequency power of selected bands. The term originates from one of its first applications: to compensate irregularities of the playback medium. For example, a concert hall with a resonant boom at 150Hz needs attenuation at this frequency to compensate the hall's exaggeration of it [ROA96, 194]. 2 special kinds of equalizers are widely used, e.g. in mixing consoles, amplifiers or consumer hi-fi products. A *graphic equalizer* has one control each for a frequency band. Most graphic equalizers have a fixed Q[49]. The control allows attenuation or boosts of its respective frequency band. A *parametric equalizer* allows changing the center frequency of each frequency band. Some also offer adjustment of the gain and bandwidth of the filter [ROA96, 194]. In the following, the term "equalizer" is used as synonym for "graphic equalizer".

### 7.7.2 Conventional Equalization

Digital equalization systems are conventionally built using a filter bank. Each filter of the filter bank is a narrow band pass filter. The output of the filters is combined to give the equalized signal [ROA96, 193].

### 7.7.3 Wavelet-based Algorithm

Even if using the wavelet transform for equalizing involves filter banks, it is a different approach: the output of the filter bank is not used to form the output signal, rather, the output of the inverse transform produces the equalized output. The WT is adequate for wide band equalization, as it has constant-Q filter bands; each scale corresponds to one frequency band [CHE96, ch.5].

The algorithm uses a set of factors, one for each scale. The wavelet coefficients of each scale are multiplied with the factor of the corresponding scale. A factor of 1 leaves the coefficients unchanged, lower values attenuate the scale's coefficients and higher values

[49] see paragraph 4.2 on p.25

amplify it. As the scales correspond to frequency bands, the corresponding frequency components of the signal are changed according to the factors.

### 7.7.4 Implementation

The wavelet equalizer is implemented as a filter extension in class *EqualizerFilter*. As all of the wavelet filters, it needs to be surrounded by the wavelet filters (ch.7.4.1 on p.69). With a parameter for each scale, the factors of the respective scale can be modified.

It exists a different operating mode: the addition mode *adds* or *subtracts* a value per scale from the coefficients, instead of multiplying them. While not having a direct musical application, it has been shown to be very useful for experimental research on the wavelet domain. Applied to music, a slight increase of all coefficients of a scale results in added noise. When applied to silence, the pure sum of all wavelets of one scale can be heard: depending on the used wavelet, more or less clean sounding tones result. The scale determines the pitch of the tone, which is, not surprisingly, in intervals of octaves between 2 scales.

### 7.7.5 Results

The equalizer filter works as expected. With a wavelet of good frequency band separation and therefore a sufficient filter length, the amount of aliasing is very low. The frequency bands of the scales can well be boosted or lowered. The equalizer can be listened to in the third audio example (track 4) in the audio part of the accompanying CD-ROM (see appendix D.7 for more details).

Two problems occur when the factors are set to extreme values, like setting some factors to 0. Firstly, aliasing occurs. This originates in the non-perfect separation of the frequency bands. The used 49-tap filter (Battle-Lemarié with 49 filter coefficients) creates very little aliasing compared to shorter wavelets, still it can be heard with these extreme settings. Secondly, the boundaries do not match, especially when low scales are changed heavily. The symmetric extension scheme improves the boundary errors compared to circular or zero padding. But in all cases, the discontinuities at the boundaries can be heard as little clicks when factors are set to extreme values.

For experiments on wavelets, this filter extension provides many applications. For example, by setting all factors to 0 except one, the output of one single scale can be heard. The addition mode provides material for new experiments, too. Like explained above, the results of adding to or subtracting from the coefficients can be explored. The uniformly arranged wavelets (when all wavelet coefficients of one scale are set to a certain value, by applying the addition mode to a silent signal) provide conclusions about the frequency of a scale of different wavelets.

# 8 Conclusion

This thesis analyzed the usage of the wavelet transform for real time digital audio. By providing theoretical background and presenting important aspects that apply for using wavelets for signal processing, it has been shown that wavelets are an efficient technique of analysis, processing, and resynthesis of the time-scale representation.

The presented implementation is a suitable base for development of wavelet-based processing in real time. The GUI and the realized extensions enable exploration and further research on wavelets and the filter bank wavelet transform. The real time aspect adds a new dimension to existent research on wavelets.

Denoising and graphical equalization have been successfully implemented as a wavelet-domain filter. Further improvements are possible for future work to eliminate the side effects. Especially the choice of the wavelet remains a critical aspect. Future research should be directed on finding suitable wavelets with minimum phase distortion and maximum separation of frequency bands to further eliminate aliasing. The clicks due to the boundary problem (see ch. 6.3.4 on p.56) should be completely removed by using the smooth padding scheme and eventually applying an overlap-and-add algorithm.

This thesis provides a rich base to continue the research on wavelet-based signal processing in real time.

# Bibliography

In this section, all referenced documents are listed. The documents marked with “CD” can be found on the accompanying CD-ROM. See Appendix D for more details on the CD-ROM.


| | |
|---|---|
| ALT96<br>CD | Altmann, Joshua: Surfing the wavelets,<br>http://www.monash.edu.au/cmcm/wavelet/, 1996 |
| BAL98<br>CD | baltrax@hotmail.com (author contacted, but no reply): Resonant low pass filter design, http://www.harmony-central.com/Computer/Programming/resonant-lp-filter.c, 1998 |
| BOE99<br>CD | Bömers, Florian: Modem Access Server, unpublished Studienarbeit (study thesis), Universität Mannheim, 1999 |
| BRS89 | Bronstein, Ilja N.: Taschenbuch der Mathematik, Verlag Harri Deutsch, 1989 |
| CDS96<br>CD | Calderbank, A. R. / Daubechies, I. / Sweldens, W. / Boon-Lock, Y.:Wavelet transforms that map integers to integers, http://cm.bell-labs.com/who/wim/papers/integer.pdf, 1996 |
| CHA99<br>CD | Chaplais, F.: A wavelet tour of signal processing by Stéphane Mallat – a short presentation, http://cas.ensmp.fr/~chaplais/Wavetour_presentation/, 1999 |
| CHE96<br>CD | Cheng, Corey: Wavelet Signal Processing of Digital Audio with Applications in Electro-Acoustic Music,<br>http://www.eecs.umich.edu/~coreyc/thesis/thesis_html, 1996 |
| COH92<br>CD | Cohen, Jack K.: Wavelets – A new orthogonal Basis,<br>http://www.cwp.mines.edu/wavelets/, 1992 |
| COH93<br>CD | Cohen, Jack K.: Battle-Lemarié Wavelets,<br>http://www.cwp.mines.edu/wavelets/, 1993 |
| COI94<br>CD | Coifman, Ronald: Adapted Waveform Analysis and Denoising,<br>ftp://pascal.math.yale.edu/pub/wavelets/papers/, 1994 |
| CRO98<br>CD | Cross, Don: Time domain filtering techniques for digital audio,<br>http://www.intersrv.com/~dcross/timefilt.html, 1998 |
| DEW97<br>CD | Deighan, Andrew / Watts, Doyle: Ground-roll suppression using the wavelet transform, http://seg.org/publications/geoarchive/1997/nov-dec/deighan.html, 1997 |
| EFF98<br>CD | Effelsberg, Wolfgang: Lecture notes Multimedia-Technik, Universität Mannheim, 1998 |

| | |
|---|---|
| EMB95 | Embree, Paul M.: C-Algorithms for Real-Time DSP, Prentice Hall PTR, 1995 |
| FIS99 CD | Fisher, Tony: Interactive filter design, http://www-users.cs.york.ac.uk/~fisher/mkfilter, 1999 |
| FRJ00 CD | Frigo, Matteo / Johnson, Steven G.: The Fastest Fourier Transform in the West, http://www.fftw.org/, 2000 |
| GRA95 CD | Graps, Amara: An introduction to wavelets, http://www.amara.com/current/wavelet.html, 1995 |
| GSB97 CD | Ghael, S. / Sayeed, A. / Baraniuk, R.: Improved Wavelet Denoising via Empirical Wiener Filtering, http://www.dsp.rice.edu/publications/pub/spie97_akira.ps.Z, 1997 |
| ISO93 CD | ISO/IEC: Coding of moving pictures and associated audio for digital storage media at up to about 1.5 mbit/s, part 3: audio. ISO/IEC 11172-3, first edition, 1993 |
| JAS94 CD | Jawerth, B. / Sweldens, W.: An Overview of Wavelet based multiresolution analysis, http://cm.bell-labs.com/who/wim/papers/papers.html, 1994 |
| KIE97 | Kientzle, Tim: A programmer's guide to sound, Addison Wesley Developers Press, 1st printing, 1997 |
| LGO95 CD | Lang, M. / Guo, H. / Odegard, J. / Burrus, C.: Nonlinear processing of a shift invariant DWT for noise reduction, http://www.dsp.rice.edu/publications/pub/CML9503.ps.Z, 1995 |
| MAA00 CD | Jansen, M.: Wavelet Thresholding and Noise reduction, http://www.cs.kuleuven.ac.be/~maarten/publications/PhD/index.html, 2000 |
| MAL98 | Mallat, Stéphane: A wavelet tour of signal processing, Academic Press, 1998 |
| MMO96 CD | Misiti, M. / Misiti, Y. / Oppenheim, G. / Poggi, J.: The Wavelet Toolbox for use with Matlab, User's guide of Matlab, version 1, 1996 |
| OPS85 | Oppenheim, A.V. / Schafer R.W.: Elaborazione numerica dei signali, Franco Angeli Editore, Milano/Italy, 3rd edition, 1985 |
| PPR91 | De Poli, Giovanni / Piccialli, Aldo / Roads, Curtis: Representations of musical signals, The MIT Press, 1991 |
| PTV94 CD | Press, William H. / Teukowsky, Saul A. / Vetterling, William T. / Flannery, Brian P.: Numerical Recipes in C, Cambridge University Press, 2nd printing, 1994 |
| RKK00 CD | Roy, Manojit / Kumar, V. Ravi / Kulkarni, B.D.: Simple denoising algorithm using the wavelet transform, sent by R. Kumar by email: ravi@che.ncl.res.in, 2000 |

| | |
|---|---|
| ROA96 | Roads, Curtis et al.: The computer music tutorial, The MIT Press, 1996 |
| SCH97 CD | Scherer, Karl: Splines und Wavelets, Universität Bonn, 1997 |
| SPG96 CD | Sanchez, S. / Prelcic, N. / Galan, S.: Uvi_Wave V3.0 Wavelet toolbox for Matlab, ftp://ftp.tsc.uvigo.es/pub/Uvi_Wave/matlab/, 1996 |
| STN96 | Strang, Gilbert / Nguyen, Truong: Wavelets and filter banks, Wellesley-Cambridge Press, 1996 |
| THO99 | Thorwirth, Niels.: Copyright protection for mp3 audio, unpublished master's thesis, Universität Mannheim, 1999 |
| TOC98 CD | Torrence, Christopher / Compo, Gilbert P.: A practical guide to wavelet analysis, Bulletin of the American Meteorological Society, Vol. 79, No. 1, January 1998 |
| UYW99 CD | Uytterhoeven, G. / Van Wulpen, F.: WAILI - Wavelets with Integer Lifting, http://www.cs.kuleuven.ac.be/~wavelets/, 1999 |
| VAL99 CD | Valens, C.: A really friendly guide to wavelets, http://perso.wanadoo.fr/polyvalens/clemens/clemens.html, 1999 |
| VEK95 | Vetterli, Martin / Kovacevic, Jelena: Wavelets and Subband coding, Prentice Hall PTR, 1995 |

# Appendix A - Class Description

Here, all classes that were implemented are shortly described. Inside a paragraph, they are ordered by declared header file and logical coherency.

## A.1 Wavelet classes

| Name: | WaveletCoeffs |
|---|---|
| Declared in: | wavelets/wavelets.h |
| Child of: | none |
| Description: | Manages a set of wavelet coefficients in different levels/scales. |

| Name: | WaveletTransform |
|---|---|
| Declared in: | wavelets/wavelets.h |
| Child of: | none |
| Description: | Provides methods for calculating forward and inverse wavelet transform. |

## A.2 Framework Core Classes

| Name: | CFOURCC |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | none |
| Description: | Class for handling a four-letter code. It is used as identifier for various types, like the encoding type. |

| Name: | AudioDeviceFormat |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioFormat |
| Description: | Base class for AudioFormat of audio devices. In addition to AudioFormat, it has the attribute bits per sample. |

| Name: | AudioFileFormat |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioDeviceFormat |
| Description: | Base class for AudioFormat of files. It has additional attributes for file type, file name extension, encoding, whether samples are signed and whether samples are in little or big endian. |

| Name: | AudioStream |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | None |
| Description: | Interface for an audiostream. It has attributes position and audio format. Methods allow to open/close and start/stop the stream. |

| Name: | AudioReader |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioStream |
| Description: | An AudioStream from which can be read samples. |

| Name: | AudioWriter |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioStream |
| Description: | An AudioStream to which can be written samples. |

| Name: | AudioExtension |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | none |
| Description: | Base interface for all extensions. It contains methods for retrieving information of the extension like name, description and its author. |

| Name: | AudioFile |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioExtension |
| Description: | Base interface of an extension that handles audio files. |

| Name: | AudioFileReader |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioFile, AudioReader |
| Description: | Interface for an audio file reader. It is based on AudioFile and on AudioReader. |

| Name: | AudioFileWriter |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioFile, AudioWriter |
| Description: | Interface for classes that write audio files. It is based on AudioFile and on AudioWriter. |

| Name: | AudioTickCallback |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | none |
| Description: | Interface for classes that receive events of a ticker. |

| Name: | AudioTickProvider |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | none |
| Description: | Interface for classes that provide ticks - that are tickers. |

| Name: | AudioExtensionTickProvider |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioExtension, AudioTickProvider |
| Description: | Interface for tick provider extensions. |

| Name: | AudioMessageReceiver |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | none |
| Description: | Interface for classes that can receive audio messages. Audio messages are currently only 2 integer values. |

| Name: | AudioMessageSender |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | none |
| Description: | Interface for classes that send audio messages. Any number of AudioMessageReceivers can register to receive the events. |

| Name: | AudioDevice |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioExtension, AudioTickProvider, AudioMessageSender |
| Description: | Interface for audio devices. It is based on AudioExtension. |

| Name: | AudioDeviceReader |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioDevice, public AudioReader |
| Description: | Interface for an audio device that reads audio data from the device. This means, e.g. recording. |

| Name: | AudioFilterCallback |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | none |
| Description: | Interface for classes that wish to receive events from AudioFilters. Like this an audio filter can notify when a parameter changed. |

| Name: | AudioFilter |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioExtension |
| Description: | Interface for audio filter extensions. Audio filters have an audio format, and the most important "work" method, where the actual processing of the audio data is done. |

| Name: | AudioCodec |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | AudioExtension |
| Description: | Interface for an extension that provides encoding/decoding capabilities. |

| Name: | Lock |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | none |
| Description: | Interface for a class that provides thread synchronization. It needs a system-dependent implementation. |

| Name: | Thread |
|---|---|
| Declared in: | core/audioclasses.h |
| Child of: | none |
| Description: | Interface for creating a thread and managing it. It can be stopped and the priority can be changed. |

| Name: | Clist |
|---|---|
| Declared in: | core/audioutils.h |
| Child of: | none |
| Description: | Class that handles a list of elements of pointer type. Elements can be added, deleted, moved, etc. Push and pop methods let it work as a stack. |

| Name: | WinLock |
|---|---|
| Declared in: | core/windows/winsynchro.h |
| Child of: | Lock |
| Description: | Implementation of the Lock interface for the Windows platform. |

| Name: | WinThread |
|---|---|
| Declared in: | core/windows/winsynchro.h |
| Child of: | Thread |
| Description: | Implementation of the Thread interface for the Windows platform. |

| Name: | AudioMixer |
|---|---|
| Declared in: | audioimpl.h |
| Child of: | AudioReader |
| Description: | An AudioReader implementation that reads from any number of input AudioReaders. The input streams are mixed. Any number of filter extensions can be applied to the mixed stream. |

| Name: | AudioSynchronizer |
|---|---|
| Declared in: | audioimpl.h |
| Child of: | AudioTickCallback |
| Description: | Reference implementation of a synchronizer that reads data from an AudioReader and writes data to any number of AudioWriters. On every tick event, a chunk is read from the reader and written to the writers. |

## A.3 Framework Extensions

| Name: | PCMCodec |
|---|---|
| Declared in: | codecs/pcmcodec.h |
| Child of: | AudioCodec |
| Description: | Generic implementation of a simple PCM codec. It converts PCM samples to floating point samples. |

| Name: | WaveFileReader |
|---|---|
| Declared in: | fileio/wavefile.h |
| Child of: | AudioFileReader |
| Description: | Extension that reads Microsoft WAVE files. It implements the AudioFileReader interface. |

| Name: | WaveFileWriter |
|---|---|
| Declared in: | fileio/wavefile.h |
| Child of: | AudioFileWriter |
| Description: | Extension that writes Microsoft WAVE files. It implements the AudioFileWriter interface. |

| Name: | DirectSoundWriter |
|---|---|
| Declared in: | devices/windows/directsound.h |
| Child of: | AudioDeviceWriter, ThreadRunner |
| Description: | Device writer extension for DirectSound devices. |

| Name: | MMEReader |
|---|---|
| Declared in: | devices/windows/mmewave.h |
| Child of: | AudioDeviceReader, ThreadRunner |
| Description: | A device reader extension that uses MME for capturing audio data from the sound card. |

## A.4 Filters

| Name: | EqualizerFilter |
|---|---|
| Declared in: | filters/wlscale.h |
| Child of: | AudioFilter |
| Description: | A wide band graphical equalizer in the wavelet domain. |

| Name: | ShowWLFunctionFilter |
|---|---|
| Declared in: | filters/wlscale.h |
| Child of: | AudioFilter |
| Description: | Allows showing the wavelet function with a time domain display. |

| Name: | DelayFilter |
|---|---|
| Declared in: | filters/delays.h |
| Child of: | AudioFilter |
| Description: | Filter extension that adds a delay (echo) effect to the audio stream. |

| Name: | DenoiseFilter |
|---|---|
| Declared in: | filters/denoise.h |
| Child of: | AudioFilter |
| Description: | Filter extension that denoises the audio stream in the wavelet domain. |

| Name: | BWLowpassFilter |
|---|---|
| Declared in: | filters/lowpass.h |
| Child of: | AudioFilter |
| Description: | Filter extension that applies an IIR low pass filter to the audio stream. It uses the Butterworth filter design method. |

| Name: | NoiseFilter |
|---|---|
| Declared in: | filters/quality.h |
| Child of: | AudioFilter |
| Description: | Adds white gaussian noise to the audio stream. |

| Name: | DifferenceBeginFilter |
|---|---|
| Declared in: | filters/quality.h |
| Child of: | AudioFilter |
| Description: | Takes a snapshot of the audio stream. |

| Name: | DifferenceEndFilter |
|---|---|
| Declared in: | filters/quality.h |
| Child of: | AudioFilter |
| Description: | Uses the snapshot of DifferenceBeginFilter and calculates the difference to the current stream. |

| Name: | StatisticsFilter |
|---|---|
| Declared in: | filters/quality.h |
| Child of: | AudioFilter |
| Description: | Provides some statistical data of the audio stream. |

| Name: | WTFilterInfo |
|---|---|
| Declared in: | filters/wlfilter.h |
| Child of: | none |
| Description: | This class is stored as WaveletInfo in the SampleBuffer to provide the wavelet coefficients to following filter extensions. |

| Name: | WTForwardFilter |
|---|---|
| Declared in: | filters/wlfilter.h |
| Child of: | AudioFilter |
| Description: | Calculates the forward wavelet transform of the audio stream. The wavelet coefficients are stored in an WTFilterInfo instance in the SampleBuffer so that following filters can access the wavelet domain. |

| Name: | WTInverseFilter |
|---|---|
| Declared in: | filters/wlfilter.h |
| Child of: | AudioFilter |
| Description: | Applies the inverse transform to the wavelet coefficients stored in the SampleBuffer. |

| Name: | WaveletDisplayFilter |
|---|---|
| Declared in: | filters/windows/wldisplay.h |
| Child of: | AudioFilter |
| Description: | Displays a scalogram of the wavelet filter coefficients. This filter extension needs Windows. |

| Name: | TimeDisplayFilter |
|---|---|
| Declared in: | filters/windows/timedisplay.h |
| Child of: | AudioFilter |
| Description: | Shows a window that paints the audio stream in time vs. amplitude planes. This filter extension is for Windows platforms. |

## A.5 Windows GUI

| Name: | StreamInfo |
|---|---|
| Declared in: | wingui/winmain.h |
| Child of: | none |
| Description: | Base class for storing information about audio stream objects and their visual representation. |

| Name: | DeviceInfo |
|---|---|
| Declared in: | wingui/winmain.h |
| Child of: | StreamInfo, AudioMessageReceiver |
| Description: | Base class for storing information about device objects and their visual representation. |

| Name: | DeviceReaderInfo |
|---|---|
| Declared in: | wingui/winmain.h |
| Child of: | DeviceInfo |
| Description: | Class for storing information about device reader instances and their visual representation. |

| Name: | DeviceWriterInfo |
|---|---|
| Declared in: | wingui/winmain.h |
| Child of: | DeviceInfo |
| Description: | Class for storing information about device writer instances and their visual representation. |

| Name: | FileInfo |
|---|---|
| Declared in: | wingui/winmain.h |
| Child of: | StreamInfo |
| Description: | Base class for storing information about file reader/writer instances and their visual representation. |

| Name: | FileReaderInfo |
|---|---|
| Declared in: | wingui/winmain.h |
| Child of: | FileInfo |
| Description: | Class for storing information about file reader instances and their visual representation. |

| Name: | FileWriterInfo |
|---|---|
| Declared in: | wingui/winmain.h |
| Child of: | FileInfo |
| Description: | Class for storing information about file writer instances and their visual representation. |

| Name: | FilterParamDescr |
|---|---|
| Declared in: | wingui/winmain.h |
| Child of: | none |
| Description: | Informational class for one parameter of a filter extension. |

| Name: | FilterParamList |
|---|---|
| Declared in: | wingui/winmain.h |
| Child of: | Clist |
| Description: | A CList-based class for handling lists of parameters. |

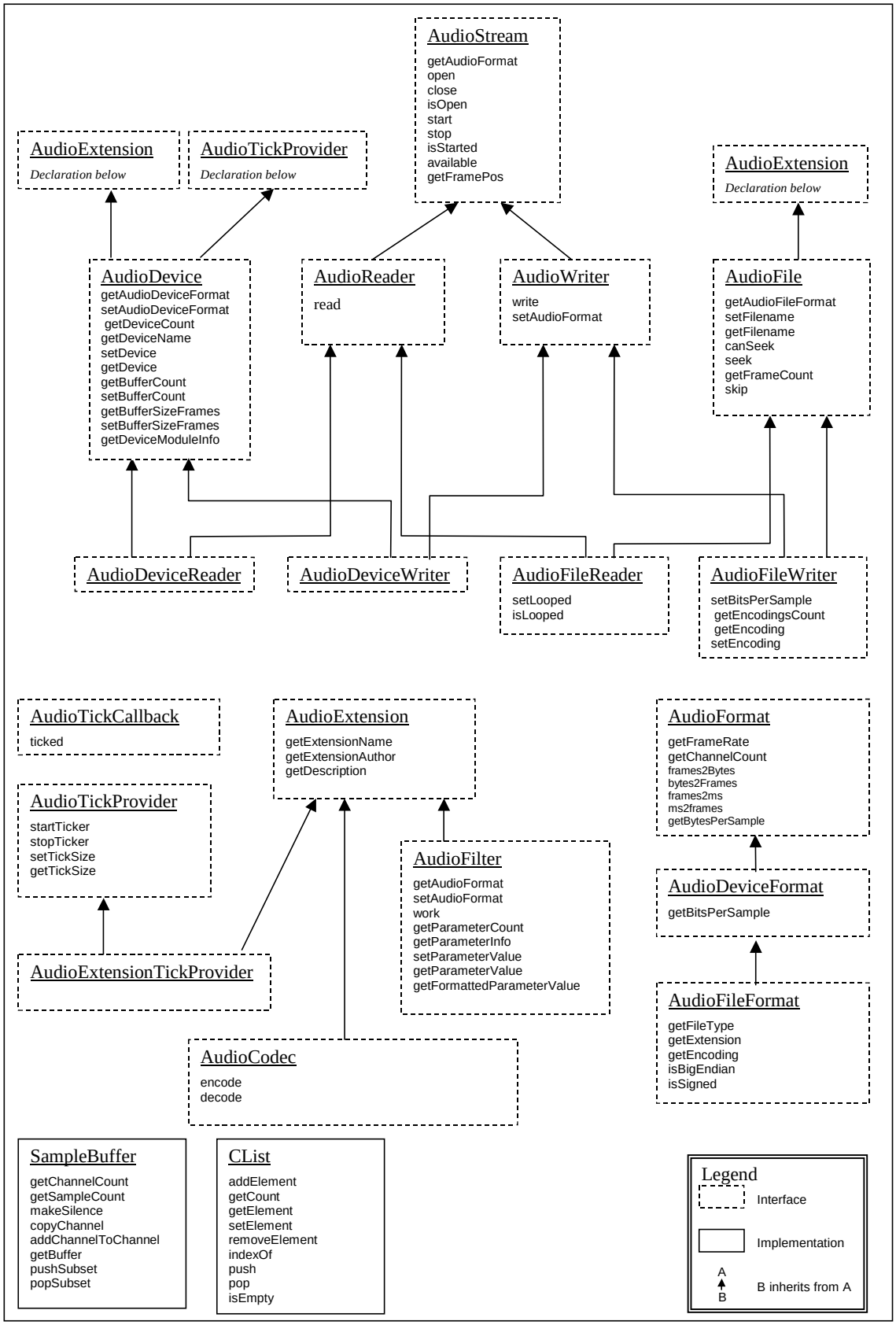
| Name: | FilterInfo |
|---|---|
| Declared in: | wingui/winmain.h |
| Child of: | AudioFilterCallback |
| Description: | Class for storing information about filter extension instances and their visual representation. |

| Name: | FilterInfoList |
|---|---|
| Declared in: | wingui/winmain.h |
| Child of: | CList, FilterTimingCallback |
| Description: | List of FilterInfo instances. The visual representation is handled also. |

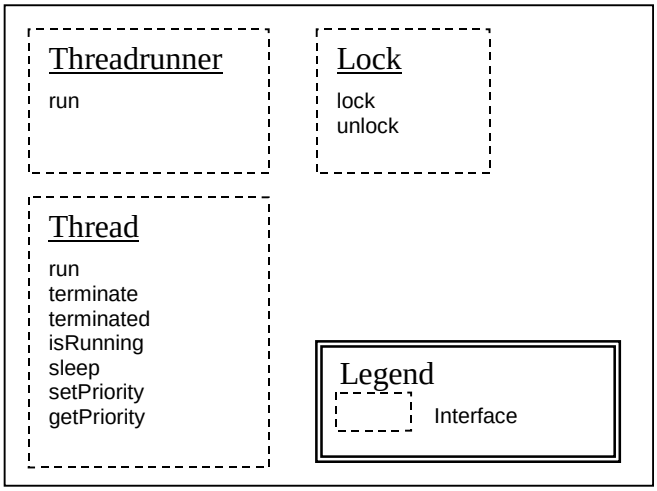
# Appendix B– Class Inheritance Trees

In this appendix, the most important interfaces and classes of the audio framework (i.e. no extensions or the GUI program) are presented in inheritage trees. The diagrams use a notation similar to UML. The included methods are a representative selection, they are not exhaustive; especially utility methods and overridden methods are not included.

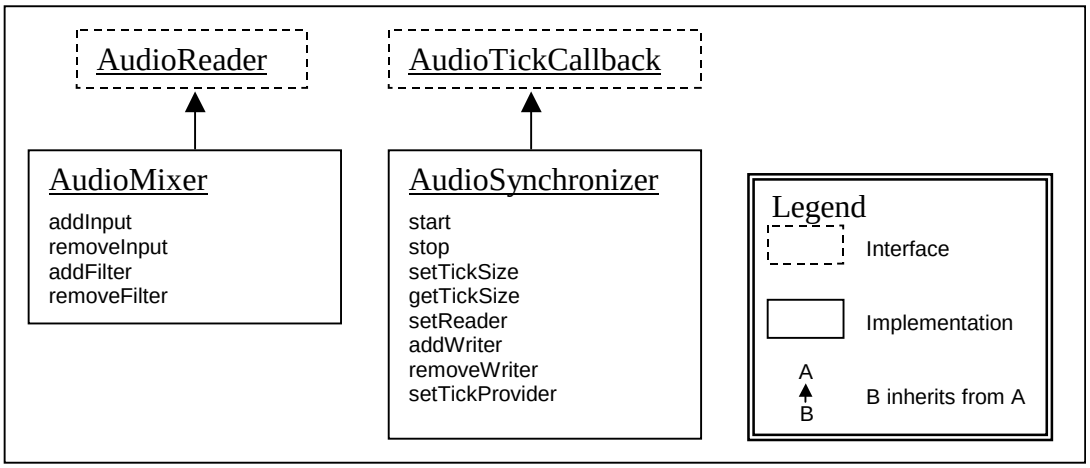
## B.1 Core Interfaces and Classes



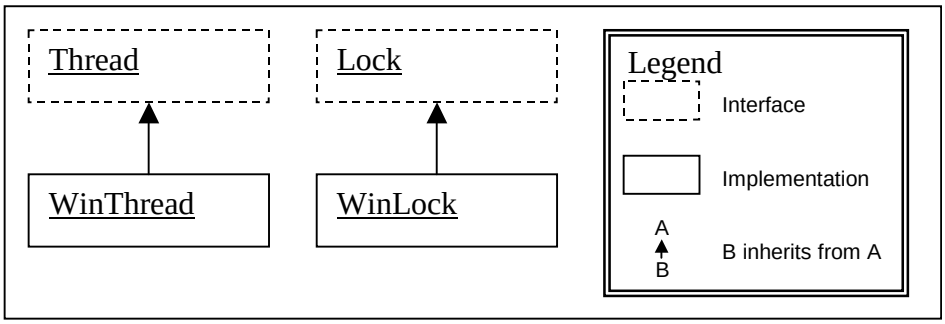
## B.2 Platform-dependent Interfaces




## B.3 High-level Classes



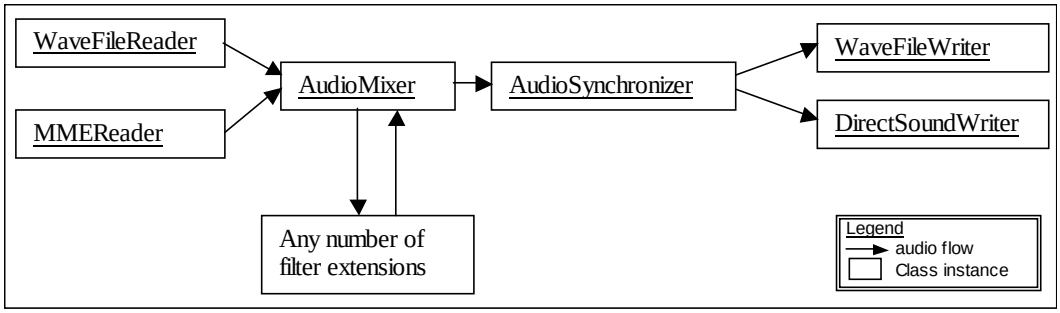

## B.4 Windows Implementation



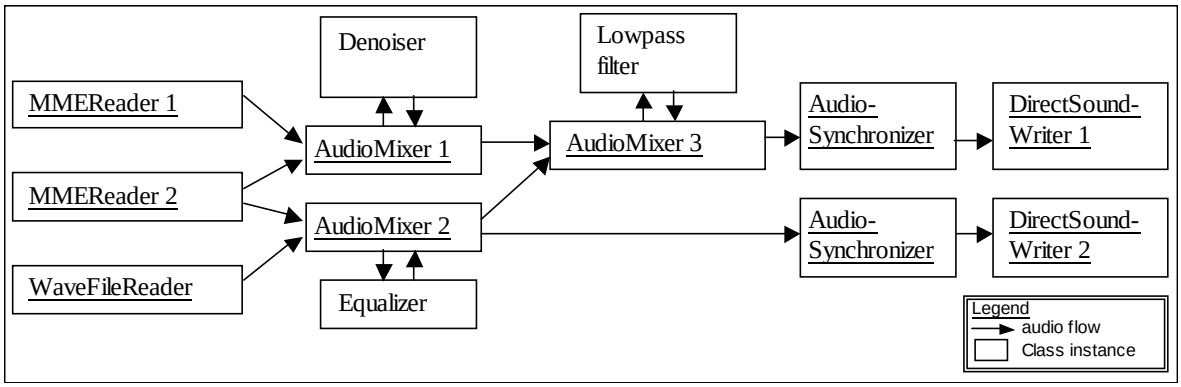
# Appendix C– Audio Framework Chains

Here, 2 audio chains are presented to show the possibilities of the audio framework.

## C.1 Audio Chain of the GUI




## C.2 An example Audio Chain



# Appendix D– the CD-ROM

This thesis is accompanied by a CD-ROM. It contains the implemented program as source and as executable files, some sound examples, this document in PDF format, the bibliography documents (where available) and the necessary programs to view the files. In the audio part, some audible presentations are provided.

The CD-ROM is created with the Joliet file system. This allows long filenames and can be read on major operating systems. To incorporate both data and audio, it is a mixed mode CD-ROM.

## D.1 Directory “Bibliography”

In this directory, there are all referenced documents, as far as they are available in electronic format. The documents are placed in sub directories, which are named after the abbreviation of the reference, as used in this thesis. The files are in the formats PostScript (extension .ps), HTML (extension .html or .htm), PDF (extension .pdf) or Microsoft Word (extension .doc). In the sub directories, a file “web.txt” or “email.txt” (or both) contains the source URL or email address from where the document has been obtained.

## D.2 Directory “Program”

The executable program for Windows can be found here, it is called “WaveletTest.exe”. It runs on Windows 95 (with DirectX 3.0 or higher installed), Windows 98, Windows NT 4 (with service pack 3 or higher) and Windows 2000. Also some audio files with the extension .wav are in this directory. They can be used in the application.

## D.3 Directory “Readers”

Here, the applications for reading the various document types can be found. In the respective sub directories, there are Adobe Acrobat Reader for PDF files, Microsoft Internet Explorer 5 and Netscape Communicator 4.7 for HTML files and

GhostScript/GhostView for PostScript files. All these programs are for the Windows platform and they are in English language.

## D.4 Directory “Source”

This directory contains all source files. They have the extension “.h” for header files and “.cpp” for c++ source code. The “build” sub directory contains project and workspace files for Microsoft Visual C++ version 6. In the “lib” directory, all interfaces and classes of the audio framework, the extensions and the wavelet classes can be found. The WinGUI directory contains the source code of the example application for Windows.

## D.5 Directory “Thesis”

In this directory, the thesis (this document) can be found in PDF, PostScript and Word 97 (or 2000) format. A sub directory contains the images of the document. The Word document needs the image directory, as it reads the images from there.

## D.6 Directory “Unsorted Info”

This directory contains more documents found in the Internet that are related to wavelets. Like for bibliography documents, a “web.txt” file references the origin.

## D.7 Audio part

The audio part of the CD-ROM can be listened to with ordinary CD players or by using a CD player program on the computer. Some examples demonstrate the usage and sound of the implemented application. They are completely created with the application.

The first track is the CD-ROM part and should not be played back. Four tracks explain some possibilities of the implemented program: the first example (track 2) uses the denoiser to reduce the noise of a historic sonata recording. A more technical example is track 3: denoising of artificially created noise is demonstrated as well as usage and sound of the difference listener. The third example in track 4 explains the wavelet equalizer filter extension and shows some of its capabilities. The combination of

denoising and equalizing is presented in the last example (track 5): the quality of a home recording is improved with the 2 filter extensions.

# Ehrenwörtliche Erklärung

Ich versichere, daß ich die beiliegende Diplomarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quelle und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommen Stellen als solche kenntlich gemacht habe.
Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ich bin mir bewußt, daß eine falsche Erklärung rechtliche Folgen haben wird.

Mannheim, im Mai 2000