

Drum Analysis

Cand.Scient thesis

by

Espen Riskedal (espenr@ii.uib.no)

Department of Informatics
University of Bergen



February 11, 2002

Abstract

This thesis studies drumloops, and the possibility to separate the different drumsounds from each other. Drum onsets and the detection of these are thoroughly discussed. Different approaches for detection and separation are discussed and a measure for correctness is presented.

Acknowledgments

This thesis could not have been made without the help from the following people:

Olli Niemitalo whom without I'd never understand anything about DSP, his IRC sessions taught me a LOT!
Thanks Olli.

Mum for reading the thesis! and encouraging me. Dad for telling me to get it done and buying me my first SoundBlaster! (and a lot of dinners :) Rune for forcing me to become ambidextrous, and finally Svanhild for being great.

Petter B. for being patient and giving me advice.

Stig and KarlTK and generally #klumpfot for giving me pointers on programming.

#musicdsp for giving hints and help on DSP.

Anssi Klapuri for discussing onset extraction and measure of correctness with me.

Core Convergence AS for being understandable in my finishing period.

Boards of Canada, Biosphere, Brothomstates, Haujobb, Thievery Corporation, Jaga Jazzist etc. which all got me into music and DSP coding in the first place.

Florian Bomers for help on wavelets.

Henrik Sundt at NOTAM for discussing the problem and giving hints.

DjFriendly and CSL for providing me with the correct inspirational material (ie. music :D)

God, and all the others I forgot :)

Contents

Acknowledgments	i
Introduction	1
Goal	1
Problem	1
1 Psychoacoustics and drum basics	3
1.1 Drum types	3
1.2 Frequency assumptions	3
1.3 Attack	4
1.3.1 Perceptual Attack Time	4
1.4 Decay	4
1.5 Specific drumsounds	5
1.6 Limits of temporal discrimination	5
1.6.1 Separating close events	5
1.6.2 Weber law	5
2 Digital sound basics	7
2.1 LTI systems	7
2.2 Waveform	7
2.3 FFT and the power spectrum	8
2.4 STFT and the sonogram	10
2.4.1 STFT calculation synopsis	11
2.5 Real-world signals and windowing	11
2.5.1 Amplitude response	11
2.5.2 Windowing	13
2.5.3 Window function tables of merits	15
2.6 Wavelet	16
2.6.1 DWT - discrete wavelet transform	17
3 Testing	18
3.1 Overlapping drumsounds	18
3.1.1 Overlapping in time	18
3.1.2 Overlapping in frequency	19
3.1.3 Overlapping in both time and frequency	20
3.2 Levels	20
3.2.1 Level One - no overlapping in time	20
3.2.2 Level Two - no overlapping in frequency	21
3.2.3 Level Three - weak overlapping in time and frequency	21
3.2.4 Level Four - true overlapping in time and frequency	22
3.3 Measure of correctness for onset extraction	22

3.3.1	What is a correct onset	23
3.3.2	Klapuri correctness	23
3.3.3	Error rate, derived measure correctness	23
3.3.4	Suggested correctness	24
3.3.5	Table of behaviors	24
4	Algorithms and their applications	25
4.1	Finding and defining areas of a drumsound	25
4.1.1	RMS/MAX analysis	25
4.1.2	Frequency-band decibel-threshold analysis	26
4.1.3	Mean decibel-threshold analysis	27
4.2	Separating drums	29
4.3	Finding <i>and</i> separating a drumsound	29
4.3.1	ICA - Independent component analysis	29
4.4	Finding, separating and match finding	30
4.4.1	Cross-correlation	31
4.5	Beat Tracking / Onset extraction algorithms	33
4.5.1	Different approaches	33
4.6	BTS	33
4.6.1	System Description	34
4.6.2	Frequency Analysis	35
4.6.3	Finding onset times	35
4.6.4	Detecting BD and SD	36
4.6.5	Beat prediction	36
4.7	Onset detection using psychoacoustic knowledge	36
4.7.1	System overview	37
4.7.2	Filtering of the bands	37
4.7.3	Onset component detection	38
4.7.4	Intensity of onset components	39
4.7.5	Combining the results	39
4.7.6	Conclusion	39
4.8	Onset detection by wavelet analysis	40
4.8.1	Wavelet analysis	40
4.8.2	Transformations of the modulus plane	41
4.8.3	Highlighting onsets	42
4.8.4	Conclusion	43
5	The analysis program	44
5.1	System overview	44
5.1.1	System flow	44
5.1.2	Class diagram	45
5.2	Key classes	46
5.2.1	DrumSequence	46
5.2.2	Spectrogram	46
5.2.3	Sample	46
5.2.4	DrumToc	46
5.2.5	DrumData	47
5.2.6	DSPTools	47
5.2.7	Error	47
5.2.8	Exception	47
5.2.9	GUIMainWindow	47
5.3	Using the program	47
5.3.1	Loading sample	48

5.3.2	Setting preferences	49
5.3.3	Calculating a spectrogram	49
5.3.4	Viewing frequency bands	50
5.3.5	Calculating onsets	50
5.3.6	The result output	51
5.4	The programming language	51
5.4.1	C++	51
5.4.2	Java	52
5.5	Libraries	52
5.5.1	STL - template library	52
5.5.2	FFTW - FFT library	53
5.5.3	expat - XML parser toolkit	54
5.5.4	Numerical Recipes in C - math functions	55
5.5.5	Qt - GUI toolkit	56
5.5.6	QWT - graph visualization package	56
5.6	Tools	57
5.6.1	Doxygen - documentation system	57
5.6.2	CVS - Concurrent Version control System	57
5.6.3	ElectricFence - memory debugging	58
5.6.4	Qt tools	58
5.7	Experiences	60
5.7.1	Soundfile library	60
5.7.2	DSP library	60
5.7.3	Scientific visualization library	61
5.7.4	Garbage collector	61
6	Results	63
6.1	Onset extraction	63
6.1.1	Time-domain	63
6.1.2	Time-frequency domain	63
6.1.3	Final onset extraction with detailed results	69
6.2	Drumsound separation	80
6.2.1	Blind-Source Separation	80
7	Conclusion	83
7.1	Future work	84
A	Data format for testcases	85
A.1	XML DTD	85
A.1.1	drumtoc element	85
A.1.2	source element	86
A.1.3	drum element	86
A.2	Example of testcase	87
B	Glossary	89
	Bibliography	97

List of Figures

2.1	Waveform of a drumsequence.	8
2.2	Power spectrum of a drumsequence.	9
2.3	Calculation of the power-density spectrum.	9
2.4	Sonogram of the same drumsequence.	10
2.5	DFT positive frequency response due to an N -point input sequence containing k cycles of a real cosine: (a) amplitude response as a function of bin index m ; (b) magnitude response as a function of frequency in Hz. Figure from [Lyo01]:75.	12
2.6	Shows a signal before and after being applied a window function.	13
2.7	Shows the Hanning window function.	14
2.8	Shows the Hamming window function.	15
2.9	Shows the Welch window function.	15
2.10	Shows the difference from using CWT and DWT. Image from www.wavelet.org	17
3.1	Overlapping in time. FFT binsize 256, linear energy plot, Hanning window.	19
3.2	Overlapping in frequency. FFT binsize 256, linear energy plot, Hanning window.	19
3.3	Overlapping in both time and frequency. FFT binsize 256, linear energy plot, Hanning window.	20
4.1	RMS and MAX analysis. RMS is blue, MAX is the dotted red.	26
4.2	Pseudo implementation of the <i>Frequency-band decibel-threshold</i> algorithm.	26
4.3	Shows the results from the <i>Frequency-band decibel-threshold</i> analysis. The yellow lines signifies the estimated onsets of the drums in the drumsequence.	27
4.4	Shows the results from the <i>Mean decibel-threshold</i> analysis. The added red lines show the detected frequency boundaries of the drumsounds detected.	28
4.5	Pseudo implementation of the <i>Mean decibel-threshold</i> algorithm.	28
4.6	Waveform of hihat-sample with zero-padding.	31
4.7	Auto-correlation of hihat-sample.	32
4.8	Cross-correlation between a hihat and a bassdrum.	32
4.9	The figure shows an overview of the BTS system. Image from [GM95].	34
4.10	Explains extracting onset components. Image from [GM95].	35
4.11	Shows detection of BD and SD. Image from [GM95].	36
4.12	System overview. Image from [Kla99].	37
4.13	Processing at each frequency band. Image from [Kla99].	38
4.14	Onset of a piano sound. First order absolute (dashed) and relative (solid) difference functions of the amplitude envelopes of six different frequency bands. Image from [Kla99].	39
4.15	Modulus values from clarinet solo. Image from [TF95].	41
4.16	Detected onsets on modulus plane of clarinet piece. Image from [TF95].	42
4.17	Detected onsets of footsteps with background music. Image from [TF95].	43
5.1	Shows the system flow of the analysis program. The input is an audiofile, the end result is text with the estimated drum onsets (the start of the drums).	44
5.2	The figure shows the inheritance and collaboration of the main (digital signal processing) classes of the analysis program.	45

5.3	The figure shows the inheritance and collaboration of the complete program. The “DSP” classes have all been put in one package.	46
5.4	Main window, also shows the current settings for FFT and smoothing.	48
5.5	File->Load , loads a .wav file into the program.	48
5.6	File->FFT Preferences , sets the preferences for FFT calculation and the optional smoothing of the frequency bands.	49
5.7	Display->Spectrogram , calculates a spectrogram.	49
5.8	Display->Waveform , displays a single frequency band in the spectrogram.	50
5.9	Analyse->Find Drums , extracted onset components.	50
5.10	The result output.	51
5.11	The figure shows how <i>Iterators</i> glue together <i>Containers</i> and <i>Algorithms</i>	53
5.12	1D Real Transforms, Powers of Two UltraSPARC I 167MHz, SunOS 5.6 Image from www.fftw.org	54
5.13	Comparison of six XML parsers processing each test file. Image from [Coo99].	55
5.14	the projects “meta” makefile.	59
6.1	Sonogram of a drumsequence.	64
6.2	Results from the analysis. The yellow lines are detected drum onsets. The circles are drum onsets that went undetected by the algorithm.	65
6.3	Shows the fluctuations of the powerspectrum in a sonogram. The figure shows band 3, around 260 Hz.	66
6.4	The blue graph is the smoothed one using Holt’s method, the green is the original unsmoothed one.	67
6.5	The blue graph is the smoothed one using Savitsky-Golay smoothing filters, the green is the original unsmoothed one.	68
6.6	The yellow lines are the detected onsets, the red lines are the estimated frequency-limits. . . .	69
6.7	Shows the two original drums.	80
6.8	Shows the different mixes of the two drums.	80
6.9	Shows the unmixed drums.	81
6.10	Shows a sonogram of the two original drums. FFT binsize 64, Hanning window, logarithmic power plot, 80dB resolution.	81
6.11	Shows a sonogram of the different mixes of the two drums. FFT binsize 64, Hanning window, logarithmic power plot, 80dB resolution.	81
6.12	Shows a sonogram of the two unmixed drums. FFT binsize 64, Hanning window, logarithmic power plot, 80dB resolution.	82
A.1	DTD for the testcases (drumtoc.dtd).	85
A.2	Example of a drumtoc XML file (hovedfag03_03.dtc).	87
A.3	Example of drums starting at the same time.	87

List of Tables

1.1	Overview of traditional modern drums.	6
2.1	Sidelobe, fall off and other descriptive values for window functions used in the thesis. All the values are from [Bor98] except for Welch window function.	16
2.2	Recommended usage and quality of window function where signal is arbitrary non-periodic. The table taken from [Tecxx].	16
3.1	Table of behaviors.	24
6.1	Onset extraction where no windowing function was used. Average correctness: 18.1%.	70
6.2	Onset extraction using Hamming window. Average correctness: 44.7%.	71
6.3	Onset extraction using Hanning window. Average correctness: 71.1%.	71
6.4	Onset extraction using 50% overlapping in the STFT. Average correctness: 46.0%.	72
6.5	Onset extraction using no overlapping.	72
6.6	Onset extraction with no smoothing functions. Average correctness: 73.3%.	73
6.7	Onset extraction using Savitzky-Golay smoothing. Average correctness: 9.5%.	73
6.8	Onset extraction Savitzky-Golay smoothing and overlapping. Average correctness: 23.4%.	73
6.9	Onset extraction where binsize is 128. Average correctness: 50.4%.	74
6.10	Onset extraction where binsize is 256. Average correctness: 71.1%.	74
6.11	Onset extraction where binsize is 512. Average correctness: 29.8%.	75
6.12	Onset extraction including multiple onsets and a 5ms correctness threshold. Average correctness: 60.2%.	76
6.13	Onset extraction ignoring multiple onsets and with a 5ms correctness threshold. Average correctness: 70.2%.	77
6.14	Onset extraction including multiple onsets and with a correctness threshold of 10ms. Average correctness: 66.5%.	78
6.15	Onset extraction ignoring multiple onsets and with a correctness threshold of 10ms. Average correctness: 77.5%.	79

Introduction

Goal

The goal of this thesis is studying the possibility of developing a system that enables the user to compose a new drum sequence by using an original drumloop as a template to work from. The following functions must be present:

1. loading/saving samples (drumloops)
2. extract the onset of each drumsound
3. extracting the frequency boundaries a drumsound resides in
4. separate the drumsounds from each other
5. find replacement drumsounds from a database of samples if needed
6. rebuild a new drumloop as close as possible to the original one
7. a basic editor for composing new beats

The purpose of such a system is to change an original drumloop in a way not previously possible. You can change the speed¹ of the drumloop without changing its pitch², in other words the BPM³. You could mute selected drums in the loop, or change their position, or even swap a drum altogether. The system is a composition tool where you can use an existing drumloop, separate its elements (the individual drumsounds, e.g. bassdrum/hihat/snare) and re-compose the drumloop until a desired result is achieved. The system can also be used to study at leisure the composition of a drumsequence from for example an educational perspective.

Problem

The areas of research that needs the closest attention in order to make this system work, can be defined by these three main problems:

1. extracting drumsound onsets and defining the frequency boundaries for a drumsound

¹This effect is also called time-stretching or pitch-shifting. There exist good algorithms for doing this, but the algorithms work on the drumloop as a sample (which it is) and not as a composition of different drumsounds.

²The pitch is the key of a sound. Even though drumsounds are not normally stationary signals, they often have a perceived pitch.

³BPM - beats per minute.

2. separating the drumsounds
3. finding matches of similar sounds in a database.

The first task of finding the onset and frequency boundaries of a drumsound was first taken lightly. We thought that a relatively simple algorithm working in the frequency domain would be sufficient, and we also originally combined the onset/frequency problem together with the separation problem.

As this thesis will show the first task was not as easily solved as first anticipated. In addition many of the methods discussed in the thesis show that a clean separation of the problems as outlined above not always is the best approach towards a solution.

Chapter 1

Psychoacoustics and drum basics

1.1 Drum types

The standard classification of instruments following the Sachs-Hornbostel scheme (mentioned in the Ph.D by W. Andrew Schloss) [Sch85], divides instruments into four main categories:

1. **Idiophones** where the vibrating material is the same object that is played (free of any applied tension), e.g. woodblocks, gongs, etc.
2. **Membranophones** where the vibrating material is a stretched membrane, e.g. drums
3. **Chordophones** where the vibrating material is one or more stretched strings e.g. lutes, zithers, etc.
4. **Aerophones** where the vibrating material is a column of air, e.g. flutes, oboes, etc.

We will be looking at sounds produced by only two of the main categories, namely idiophones and membranophones. Mallet instruments (marimba, xylophone, vibraphone, glockenspiel) which might be considered drums are more similar to piano and although being idiophones will not be included here.

1.2 Frequency assumptions

The assumption that theoretically would make this composing system possible, is that you can find some key attributes that define a drumsound and makes it unique.

Initially we were hoping that the frequency distribution would be more sparse for the different drumsounds i.e. bassdrums occurring just in the lower parts of the frequency spectrum, hihats just in the upper parts. This however turned out not to be totally true, as a drumsound usually spreads over a relatively large frequency area, of course including the main area of which it resides in. As Schloss says “The fact is that drums differ enormously in timbre and pitch clarity. Minute differences in shell size and shape, and the thickness and type of membrane, contribute a great deal to the unique sound of different drums” [Sch85].

When describing a particularly recording of Brazilian percussive music, Schloss, in his article on the automatic transcription of percussive music, notes “... the extremely diverse percussion instru-

ments cover an enormous bandwidth, from below 40 Hz to the limit of hearing ¹. Each instrument has a wide spectrum, overlapping the others both in spectrum and in rhythmic patterns.” [Sch85].

One reason for our assumption of a more strict frequency distribution may have been caused by the perceived pitch of certain drums. For example “The acoustics of the timpani have been described briefly by Thomas Rossing, including the reasons these drums elicit a reasonable clear pitch, when in fact they should be quite inharmonic [Ros82]” [Sch85].

The frequency distribution is still an attribute that can be used for describing a drumsound.

1.3 Attack

Another attribute that is normally present in a drumsound is a fast attack. Attack is the time from the start of the sound to the time of peak intensity. Drumsounds are normally generated by hitting surfaces, and the sound produced usually reaches its peak intensity in a short period of time. Sounds that have been extensively changed by signal processing tools (e.g. reversed) may lose this attribute and might be harder to detect.

1.3.1 Perceptual Attack Time

Still there is a problem in musical timing context that should be mentioned - the problem of Perceptual Attack Time (PAT). As Schloss writes “the first moment of disturbance of air pressure is not the same instant as the first *percept* of the sound, which in turn may not necessarily coincide with the time the sound is perceived as a *rhythmic event*. There inevitably will be some delay before the sound is registered as a *new* event, after it is physically in evidence.” [Sch85]:23.

We will not delve any deeper into this problem in this thesis as it actually can be largely ignored. “It turns out that the actual ‘delay’ caused by PAT in the case of drumsounds is quite small, because the slope is typically rather steep.” [Sch85]:24. The test data used for our experiments with different onset algorithms is subjectively crafted using our own perception of the drumsamples. “In trying to define limits on temporal discrimination, researchers have not always agreed on their results; one fact stands out, however - the ‘ear’ is the most accurate sensory system in the domain of duration analysis.” [Sch85]:20. Hence, if the system corresponds with the test data, it will also correlate with our perception of the drumsamples and their onsets.

1.4 Decay

Also, drumsounds normally have a “natural” decay time, unless reverb has been added, it is a special instrument or has been altered in a sample-editor. By decay we mean the time it takes from the peak intensity until the drumsounds disappears, so not like the decay we know from ADSR envelopes in synthesizers or samplers. It is not however easy to give a general rule for the rate of decay for drumsounds as it is with attack. Therefore, making any rules based on this attribute would make the system less general.

¹about 22 kHz when we are young

1.5 Specific drumsounds

In this thesis some specific drumsounds will be mentioned and in order to make it possible for the reader to know what kind of drums they are, here is a brief description with images.

1.6 Limits of temporal discrimination

When studying and trying to mimic the human auditory system it is important to understand the limits of our perception of sound and temporal changes.

1.6.1 Separating close events

First of all for this thesis focused on rhythm and drum onsets in particular it is important to find out what the limits of discriminability is in terms of time. What is the minimum duration between two events that still can be *perceived* by our ear as two distinct events? This is interesting because we want to know what 'resolution' our system for onset detection should operate in.

In a classical study by Ira Hirsch [Hir59] it was found that "it was possible to separate perceptually two brief sounds with as little as 2 msec. between them; but in order to determine the *order* of the stimulus pair, about 15-20 msec. was needed" [Sch85]:22.

In this thesis 5 ms will be used as the lower limit for error in the system, for as Schloss claims: "In the normal musical range, it is likely that to be within 5 msec. in determining attack times is adequate to capture essential (intentional) timing information" [Sch85]:22.

1.6.2 Weber law

Weber's law states that "the perceptual discriminability of a subject with respect to a physical attribute is proportional to its magnitude, that is $\delta x/x = k$ where x is the attribute being measured, and δx is the smallest perceptual change that can be detected. k is called the *Weber ratio*, a dimensionless quantity." [Sch85]:20.

In other words, if you examine something of a certain magnitude, your discriminability will vary proportionally with the magnitude.

It would be interesting to see if this also applies to our hearing. "It turns out that, for very long or very short durations [pauses between rhythmic events], Weber's law fails, but in the area from 200 milliseconds to 2 seconds, a modified version of Weber's law seems to hold, according to Getty [Get75]." [Sch85]:20.

To conclude, the human auditory system works best in the 'musical range' with intervals of 100 - 2000 ms. And people normally have the most accurate sense of temporal acuity in the range of 500 - 800 ms. [Sch85]:21-22.






Name	Description	Image
Snare drum	sharp short sound	
Bass drum	deep bass sound	
Hi-hat	thin light sound	
Cymbal	broad light sound	
Tomtom	different “pitched” sounds	

Table 1.1: Overview of traditional modern drums.

Chapter 2

Digital sound basics

This chapter is a brief introduction in digital sound processing. We give a thorough explanation of the different DSP terms used in this thesis, and we explain and discuss the transformations and their advantages and disadvantages.

2.1 LTI systems

In this thesis we only focus on discrete-time systems, specifically linear time-invariant (LTI) systems.

"A linear system has the property that the output signal due to a linear combination of two or more input signals can be obtained by forming the same linear combination of the individual outputs. That is, if $y_1(n)$ and $y_2(n)$ are the outputs due to the inputs $x_1(n)$ and $x_2(n)$, then the output due to the linear combination of inputs

$$x(n) = a_1x_1(n) + a_2x_2(n) \quad (2.1)$$

is given by the linear combination of outputs

$$y(n) = a_1y_1(n) + a_2y_2(n) \quad (2.2)$$

." [Orf96]:103.

"A *time-invariant system* is a system that remains unchanged over time. This implies that if an input is applied to the system today causing a certain output to be produced, then the same output will also be produced tomorrow if the same input is applied." [Orf96]:104.

2.2 Waveform

A sound presented in the digital domain is called a *sample*. With this we mean an array of numbers representing the form of the sound. A *samplepoint* is one such number in a sample array. The numbers in the array describe the *amplitude* of the sound. Figure 2.1 shows how this looks graphically, and it is called a *waveform*.

By *amplitude* we mean a positive or negative value describing the position of the samplepoint relative to zero. Notice that *magnitude* is very similar, but by *magnitude* we mean a positive value describing the *amplitude* of a signal. In other words magnitude is the amplitude of the wave irrespective of the phase.

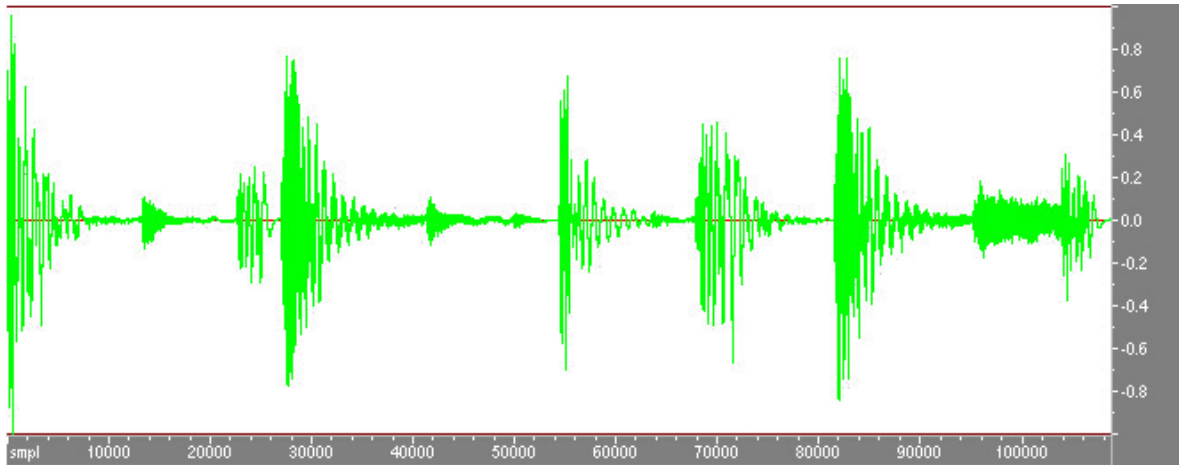


Figure 2.1: Waveform of a drumsequence.

When we look at a waveform we are studying the sample in the *time-domain*. Time is on the horizontal axis and the amplitude is along the vertical axis.

2.3 FFT and the power spectrum

“In 19th century (1822 to be exact), the French mathematician J. Fourier, showed that any periodic function can be expressed as an infinite sum of periodic complex exponential functions. Many years after he had discovered this remarkable property of (periodic) functions, his ideas were generalized to first non-periodic functions, and then periodic or non-periodic discrete time signals.” [Pol01].

The DFT’s (discrete Fourier transform) origin is the CFT (continuous Fourier transform) but since our focus is on discrete-time systems we only discuss the DFT. The DFT equation in exponential form is:

$$X(m) = \sum_{n=0}^{N-1} x(n)e^{-i2\pi nm/N} \quad (2.3)$$

where $x(n)$ is a discrete sequence of time-domain sampled values (basically a sample) of the continuous variable $x(t)$, e is the base of natural logarithms and i is $\sqrt{-1}$. Equation 2.3 can be based on Euler’s relationship $e^{-i\theta} = \cos(\theta) - i\sin(\theta)$ be changed into this rectangular form:

$$X(m) = \sum_{n=0}^{N-1} x(n)[\cos(2\pi nm/N) - i\sin(2\pi nm/N)] \quad (2.4)$$

- $X(m)$: the m th DFT output
- m : the index of the DFT output in the frequency-domain
- $x(n)$: the sample
- n : the time-domain index of the sample
- i : $\sqrt{-1}$

- N : the number of samplepoints in the sample and the number of frequency points in the DFT output [Lyo01]:49-51.

The point of the Fourier transform is to determine the *phase* and *amplitude* of the frequencies in a signal. The result of a DFT is a sequence of complex numbers where the modulus describes the amplitude and the argument describes the phase. If we want to study a sample in the frequency-domain this can be done by performing the DFT (discrete Fourier transform), or its fast implementation FFT (fast Fourier transform), on the sample and by using the result from the transform calculate a power spectrum of the sample, see Figure 2.2.

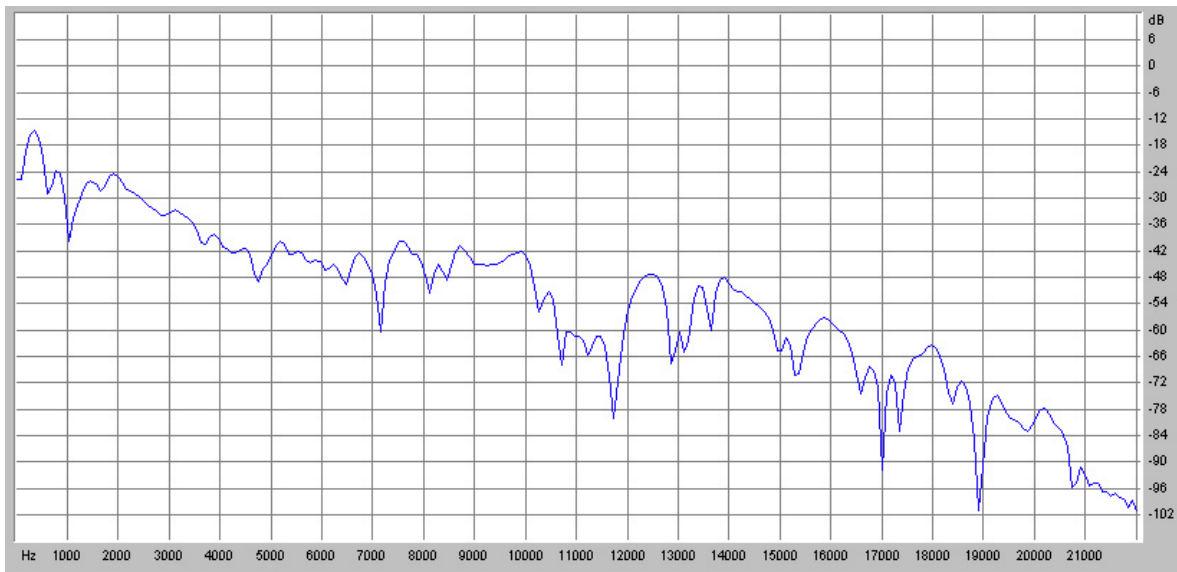


Figure 2.2: Power spectrum of a drumsequence.

In Figure 2.2 the vertical-axis shows the power of the signal in dB and the horizontal-axis shows the frequency in Hz. As we can see the sample has a peak in the lower parts of the spectrum and gradually the power declines towards the upper frequencies. This presentation of the frequency parts of the drumsequence does not tell us anything about their relation in time. If the sample being analyzed was a frequency sweep the power spectrum would not tell us anything but the total amount of power the different frequency components in the sample had.

Ignoring the negative frequencies returned from a DFT, calculating a power density-spectrum with results from the discrete Fourier transform applied on a real (as opposed to complex) signal could be implemented as shown in Figure 2.3.

```
// assuming FFT by four1(buffer-1, N, 1); from Numerical Recipes: www.nr.com
for (int k = 0; k <= N/2; k++) {
    real = buffer[2*k];
    imag = buffer[2*k+1];
    ps.freq[k] = sampleRate*(double)k/(double)N;
    ps.db[k] = 10. * log10( 4.*(real*real + imag*imag) / (double)N*N );
}
```

Figure 2.3: Calculation of the power-density spectrum.

In Figure 2.3 $ps.db[]$ is a vector for the dB calculated, N^1 is the size of the FFT binsize and $buffer[]$ is the vector the power spectrum calculation is performed on (the result returned from the FFT algorithm).

2.4 STFT and the sonogram

STFT (short time Fourier transform) is based upon a series of segmented and overlapped FFTs that are applied along the sample. These segments are often referred to as *windows*. In the STFT, the individual FFTs from these multiple windows are rendered as a 2D plot where the color or intensity represent the power. This is know as a sonogram or spectrogram.

The purpose of using overlapping windows is to produce a time-frequency representation of the data. A high degree of overlapping of the windows can result in a more accurate time-frequency spectrum. Since the size of a FFT binsize decides the frequency resolution there is a fixed frequency resolution when using STFT and the resolution is set mainly by the size of the windows ². The time resolution is also set by the size of the windows. By using small windows one get good time resolution but poorer frequency resolution. And likewise, using larger windows will give better frequency resolution but poorer time resolution. Overlapping can help to produce better time resolution, but only to a certain degree. The window-size thus controls the tradeoff between frequency resolution and time resolution, and it will be constant everywhere in the time-frequency spectrum. The STFT is thus classified as a fixed or single resolution method for time-frequency analysis.

Figure 2.4 shows the result of STFT applied to a drumsequence. The color represent the power of the signal, the horizontal-axis represent the time and the vertical-axis the frequency. Thus we are looking at time-frequency representation of a sample. With this representation we can decide among other things whether a signal is *stationary* or not. A stationary signal is a signal whose average statistical properties over a time interval of interest are constant.

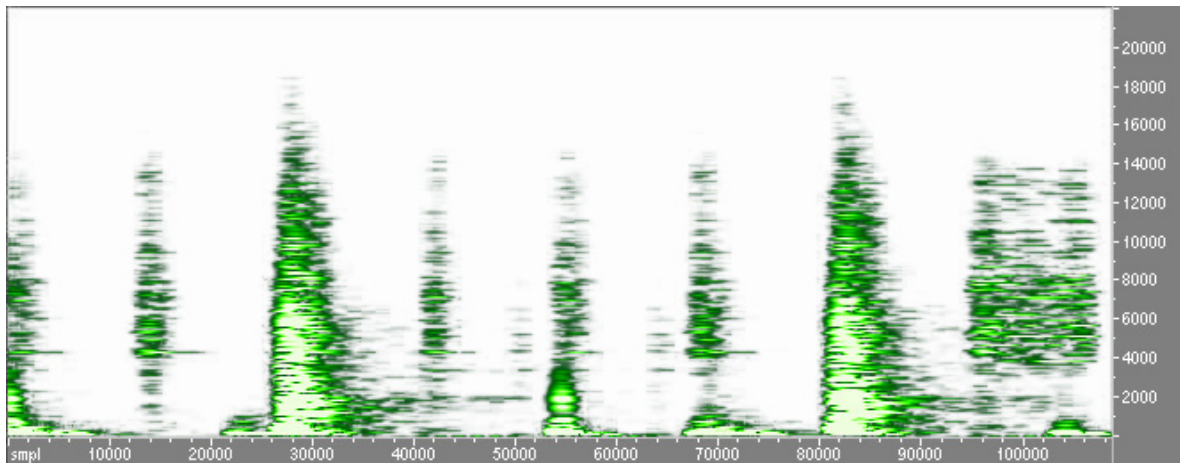


Figure 2.4: Sonogram of the same drumsequence.

¹If the binsize used is say 512 then one gets a resolution of 256, or $N/2$, frequency-bands.

²Zero-padding the windows can change the resolution somewhat

2.4.1 STFT calculation synopsis

The way a sonogram is calculated is thus, block the signal into smaller parts, window these, and do DFT/FFT on them. The number of bins decides the frequency resolution of the sonogram, and the amount of *overlapping* decides the time resolution. The blocking of the signal can be overlapped, so that the second block spans over some parts of the first block, and the third block spans over parts of the second. A usual amount of overlapping is 50%, which means that the blocking advances at steps half the size of the bins.

2.5 Real-world signals and windowing

The DFT of sampled real-world signals gives frequency-domain results that can be misleading. “A characteristics, known as *leakage*, causes our DFT results to be only an approximation of the true spectra of the original input signal prior to digital sampling.” [Lyo01]:71. There are several reasons for this, one being that the input data is not periodic, an actual requirement for the Fourier transform to return the correct frequency components in the signal. Another reason is the fact that the DFT will only produce correct results when the input data contains energy precisely at integral multiples of the fundamental frequency f_s/N where f_s is the samplerate and N is the DFT binsize or size of the window if you will. This means that any sample that contains intermediate frequencies like for example $1.5f_s/N$ will produce incorrect results. Actually “this input signal will show up to some degree in *all* of the N output analysis frequencies of our DFT!” [Lyo01]:73. This characteristics is unavoidable when we perform DFT on real-world finite-length samples.

2.5.1 Amplitude response

If we look at the amplitude response for an N sized DFT in terms of one specific bin, $X(m)$, for a real cosine input having k cycles, it can be approximated by the sinc function:

$$X(m) \approx \frac{N}{2} \cdot \frac{\sin[\pi(k-m)]}{\pi(k-m)} \quad (2.5)$$

In Equation 2.5 m is the bin index. We can use Equation 2.5 to determine how much leakage happens when using DFT.

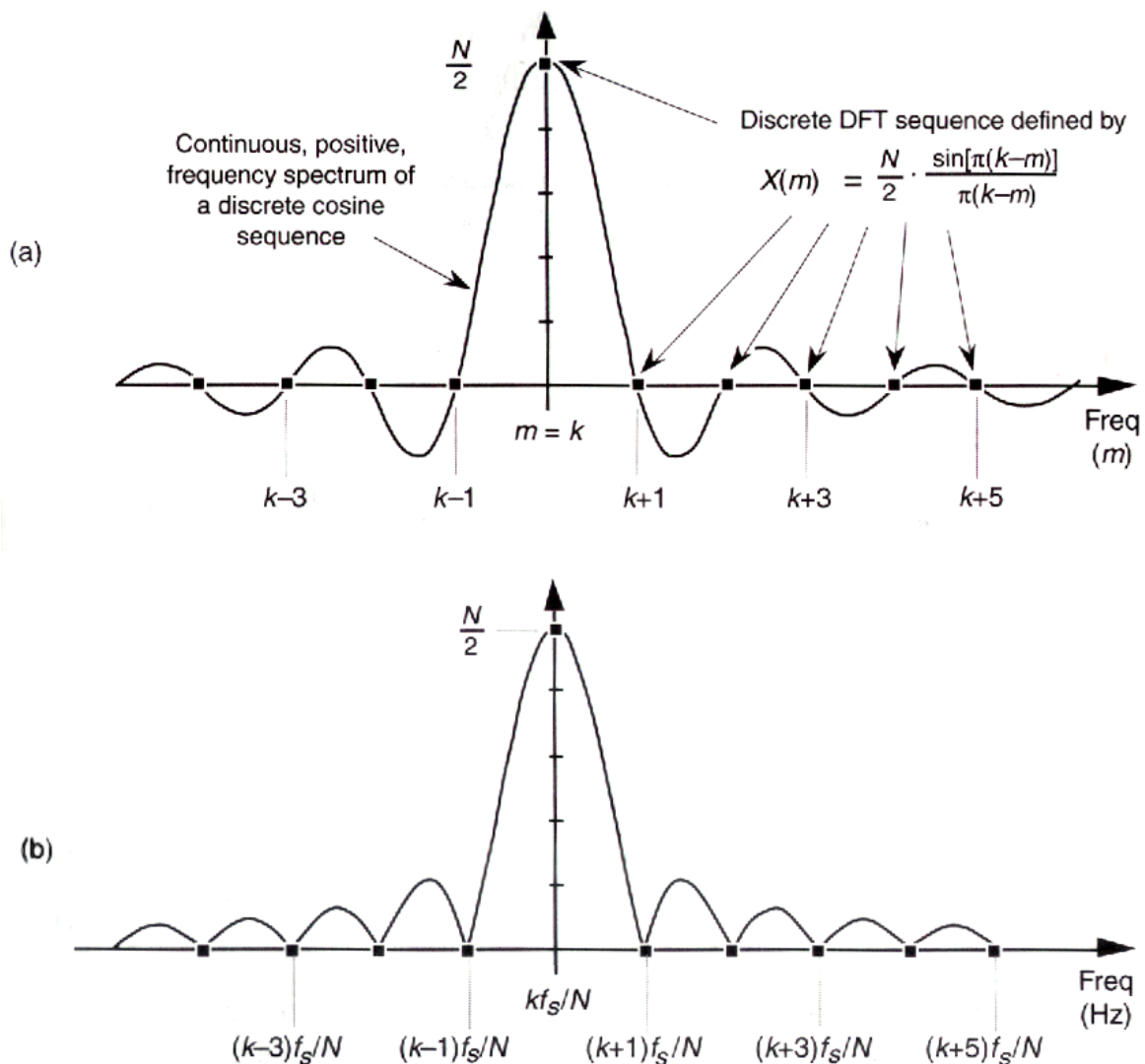


Figure 2.5: DFT positive frequency response due to an N -point input sequence containing k cycles of a real cosine: (a) amplitude response as a function of bin index m ; (b) magnitude response as a function of frequency in Hz. Figure from [Lyo01]:75.

Study Figure 2.5 and notice the main lobe and sidelobes of the curve. Since the DFT is only a sampled version of the continuous spectral curve, the DFT will only give correct analysis “when the input sequence has exactly an integral k number of cycles (centered exactly in the $m = k$ bin)” [Lyo01]:74. When this is the case no leakage occurs, i.e. the sidelobes are zero.

Another characteristic with the DFT is that it leakage also wraps around. “The DFT exhibits leakage wraparound about the $m = 0$ and $m = N/2$ bins. And finally an effect known as scalloping also contributes to the non-linear output from DFT. Consider Figure 2.5 and picture all the amplitude responses for all the bins superimposed on the same graph at the same time. The rippled curve, almost like a picket fence, illustrates the loss DFT has because some (most) frequencies are between the bin frequency centers.

2.5.2 Windowing

DFT leakage is troublesome because it can corrupt low-level signals in neighbouring bins, they will drown in the leakage in not be registered as 'interesting' frequency components.

Windowing is the process of altering the input data in a sequence, a window. In other words windowing is applying a window function to a sequence of input data. The purpose of windowing is to reduce the sidelobes we experienced in Figure 2.5. If these sidelobes are reduced the DFT leakage will also be reduced.

The benefits of windowing the DFT input are:

- reduce leakage
- reduce scalloping loss

The disadvantages of windowing are:

- broader main lobe
- main lobe peak value is reduced (frequency resolution reduced)

However as Lyons puts it “the important benefits of leakage reduction usually outweigh the loss in DFT frequency resolution.” [Lyo01]:83.

Figure 2.6 shows a signal before and after being applied a window function. The signal is not periodic, and this will produce glitches, which again will result in frequency leakage in the spectrum. The glitches can be reduced by shaping the signal so that the ends matches smoothly. By multiplying the signal with the window function we force the ends of the signal to be zero, and then fit together. Starting and ending with the same value is not enough to make the signal repeat smoothly, the slope also has to be the same. The easiest way of doing this is to make the slope of the signal at the ends to be zero. The window function has the property that its value and all its derivatives are zero at the ends.

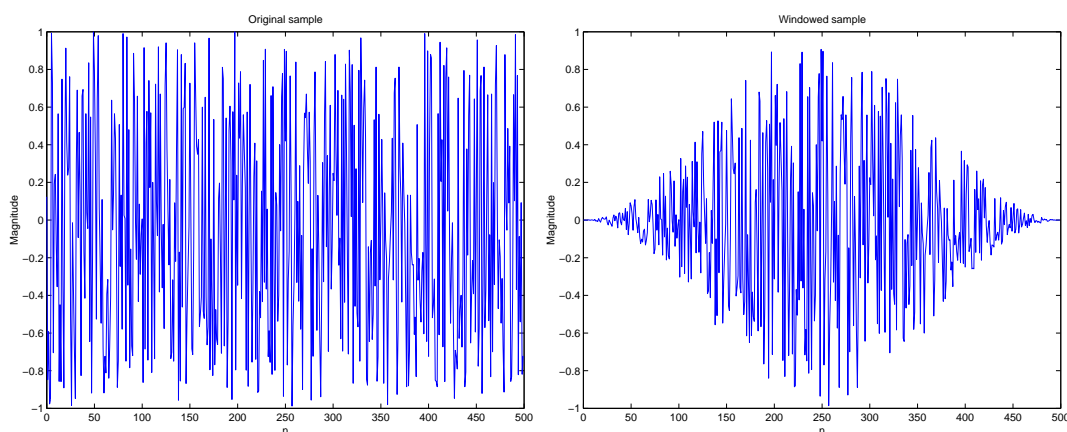


Figure 2.6: Shows a signal before and after being applied a window function.

2.5.2.1 Rectangular window

A rectangular window is a window with no attenuation, or no distortion of the segment it is being applied to. A rectangular window is the window function all other window functions are compared to concerning measures of quality. The rectangular weights (for a rectangular window) are $w(n) = 1$.

The main lobe of the rectangular window is the most narrow, but the sidelobe is only -13 dB below the main peak lobe. The rectangular window is also known as the uniform or boxcar window.

2.5.2.2 Hanning window

The Hanning window, Equation 2.6, is an excellent general-purpose window [Tecxx]. The Hanning window has a reduced first sidelobe, -32 dB below the main peak lobe, and the *roll off* or *fall off*, which means the amount of reduction in dB per octave, is -18 dB [Lyo01], [Bor98]. The Hanning window is also known as raised cosine, Hann or von Hann window and can be seen in Figure 2.7.

$$w(n) = 0.5 - 0.5\cos\left(\frac{2\Pi n}{N}\right), 0 \leq n \leq N - 1 \quad (2.6)$$

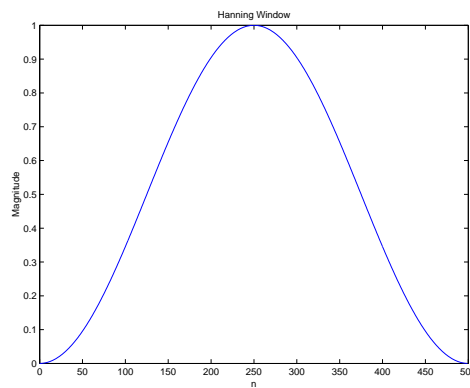


Figure 2.7: Shows the Hanning window function.

2.5.2.3 Hamming window

The Hamming window, Equation 2.7, has an even lower first sidelobe than Hanning, at -43 dB. The fall off on the other hand is only -6 dB, the same as a rectangular window. “This means that leakage three or four bins away from the center bin is lower for the Hamming window than for the Hanning, and leakage a half dozen or so bins away from the center bin is lower for the Hanning window than for the Hamming window” [Lyo01]:84. Figure 2.8 shows the Hamming window, notice how the endpoints do not quite reach zero.

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\Pi n}{N}\right), 0 \leq n \leq N - 1 \quad (2.7)$$

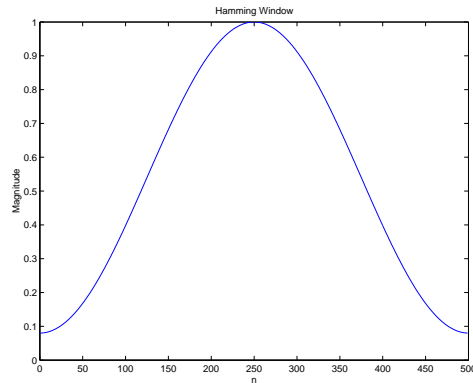


Figure 2.8: Shows the Hamming window function.

2.5.2.4 Welch window

The Welch window, Equation 2.8, is also known as the parabolic window and can be seen in Figure 2.9.

$$w(n) = 1 - \left(\frac{n - \frac{N}{2}}{\frac{N}{2}}\right)^2, \quad 0 \leq n \leq N - 1 \quad (2.8)$$

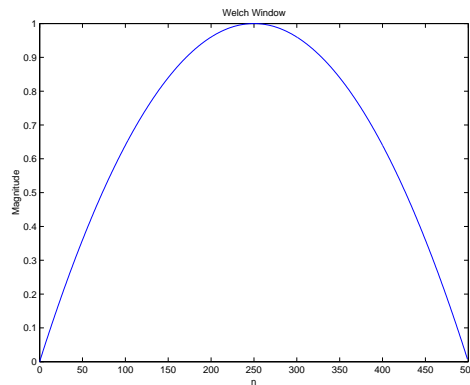


Figure 2.9: Shows the Welch window function.

2.5.3 Window function tables of merits

Table 2.1 shows a comparison between the different window functions used in the analysis program in this thesis.

- Sidelobe: the attenuation to the top of the highest sidelobe
- Fall off: the rate of fall in dB off to the side lobe
- Coherent Power Gain: the normalised DC gain
- Worst Case Processing Loss: the ratio of input signal-to-noise to output signal-to-noise, including scalloping loss for the worst case

Window function	Sidelobe (dB)	Fall off (dB/octave)	Coherent power gain	Worst case processing loss (dB)
Rectangular	-13	-6	1.00	3.92
Hanning	-32	-18	0.50	3.18
Hamming	-43	-6	0.54	3.10
Welch	-26			

Table 2.1: Sidelobe, fall off and other descriptive values for window functions used in the thesis. All the values are from [Bor98] except for Welch window function.

Window function	Best for these signal types	Frequency resolution	Spectral leakage	Amplitude accuracy
Rectangular	Transients & Synchronous Sampling	Best	Poor	Poor
Hanning	Random	Good	Good	Fair
Hamming	Random	Good	Fair	Fair
Welch	Random	Good	Good	Fair

Table 2.2: Recommended usage and quality of window function where signal is arbitrary non-periodic. The table taken from [Tecxx].

Table 2.2 shows a recommended usage of the same window functions as in Table 2.1. As a note the best window to prevent spectral leakage is the Blackman window function, and the best for amplitude accuracy is a Flat Top window [Tecxx], these were not included in the table since they are not used in the analysis program.

2.6 Wavelet

The problem with STFT is finding the correct window function and FFT binsize to use. The size controls both the time and frequency resolution for the whole analysis.

“The problem with STFT is the fact whose roots go back to what is known as the *Heisenberg Uncertainty Principle*. This principle originally applied to the momentum and location of moving particles, can be applied to time-frequency information of a signal. Simply, this principle states that one cannot know the exact time-frequency representation of a signal, i.e., one cannot know what spectral components exist at what instances of times. What one *can* know are the *time intervals* in which certain *band of frequencies* exist, which is a *resolution* problem.” [Pol01].

Choosing the correct binsize and window function is application specific, there are no magic solutions. And as Polikar notes, finding a good binsize and window function “could be more difficult than finding a good stock to invest in” [Pol01]. The wavelet transform solves this dilemma of resolution to some extent.

The biggest difference between the wavelet transform and sliding FFT is that the resolution of the time and frequency axes change for wavelets, as with sliding FFT they stay the same. Generally speaking we can say that the lower frequencies of a sample has poor time resolution but accurate

frequency resolution using the wavelet transform. High frequencies have accurate time resolution but poor frequency resolution. The wavelet transform is because of this called a multiresolution analysis. “Multiresolution analysis is designed to give good time resolution and poor frequency resolution at high frequencies and good frequency resolution and poor time resolution at low frequencies. This approach makes sense especially when the signal at hand has high frequency components for short durations and low frequency components for long durations. Fortunately, the signals that are encountered in practical applications are often of this type.” [Pol01].

2.6.1 DWT - discrete wavelet transform

The DWT’s (discrete wavelet transform) origin is the CWT (continuous wavelet transform). In the CWT, Equation 2.9, the transformed signal is a function of two variables, τ and s , which are the *translation* and *scale* parameters. ψ is the *wavelet basis function* or the *mother wavelet* and $*$ is complex conjugation.

$$CWT_x^\psi(\tau, s) = \Psi_x^\psi(\tau, s) = \frac{1}{\sqrt{|s|}} \int x(t) \psi\left(\frac{t-\tau}{s}\right) dt \quad (2.9)$$

“The wavelet analysis is a measure of the similarity between the basis functions (wavelets) and the signal itself. Here the similarity is in the sense of similar frequency content. The calculated CWT coefficients refer to the closeness of the signal to the wavelet *at the current scale*.” [Pol01]. Figure 2.10 shows the results from a CWT and DWT analysis.

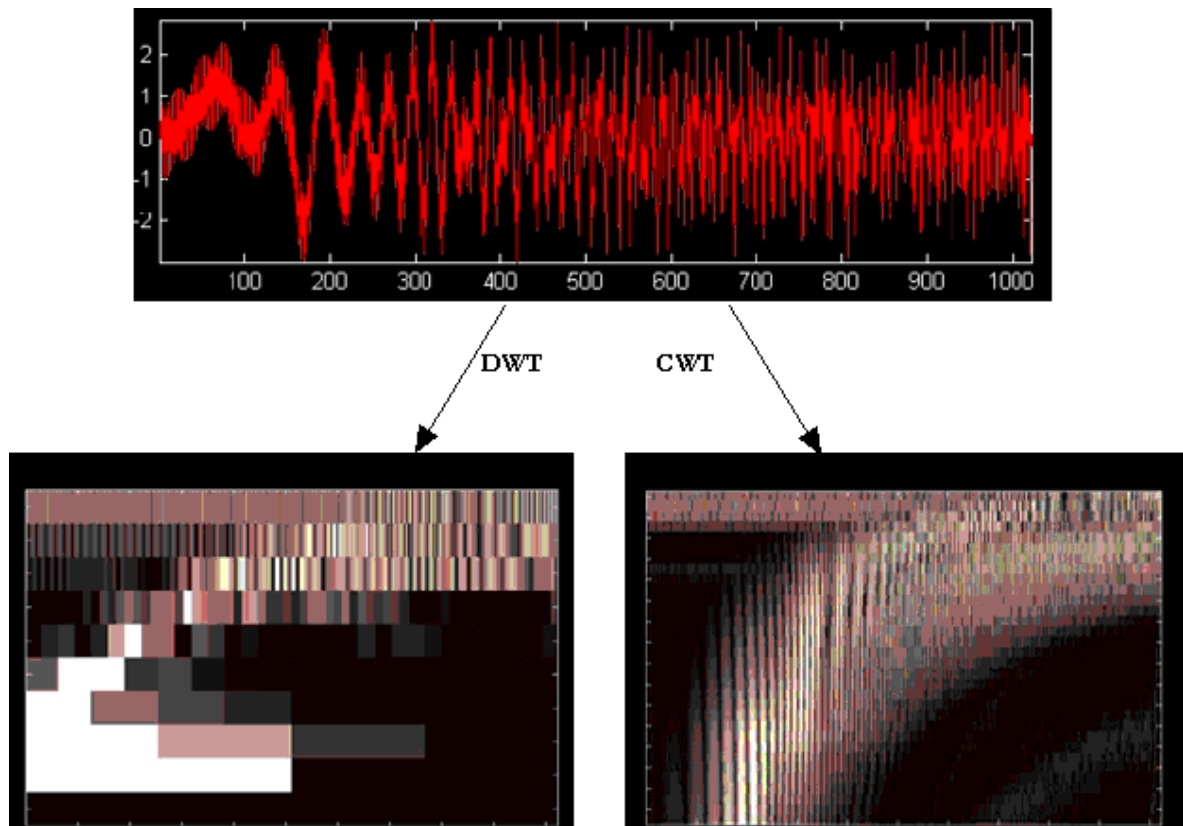


Figure 2.10: Shows the difference from using CWT and DWT. Image from www.wavelet.org.

Chapter 3

Testing

To solve the problem of analyzing and separating a drumsequence, we need to experiment with different algorithms. In order to choose one algorithm or method instead of another, we have to have some kind of measure that tells us which is best.

One way of doing this is to run a number of different drumsequences (custom made, where one knows the correct separation beforehand) through the system, and compare the results from the system with the correct ones. This is a legitimate way of doing this kind of measure, but it would probably yield even better results if a set of questions were defined that determine the effectiveness of the different algorithms. By classifying the test data into different levels and measuring the correctness of each algorithm and how they perform on the different data we can achieve more finely grained test results. One type of leveling is to use the complexity of overlapping as a distribution indicator.

3.1 Overlapping drumsounds

A drumsequence consists of different drumsounds mixed together. These drumsounds can when combined overlap each other in various ways.

1. overlapping in time
2. overlapping in frequency
3. overlapping in time *and* frequency.

3.1.1 Overlapping in time

Overlapping in time means that there are more than one drumsound played simultaneously (e.g. Like a bassdrum and a hihat). The question of separating drumsounds that are only overlapping in time could then be a filtering problem. One would have to define the different areas the drumsounds reside in, and then use the proper filters. Figure 3.1 gives an example of two drumsounds overlapping in time.

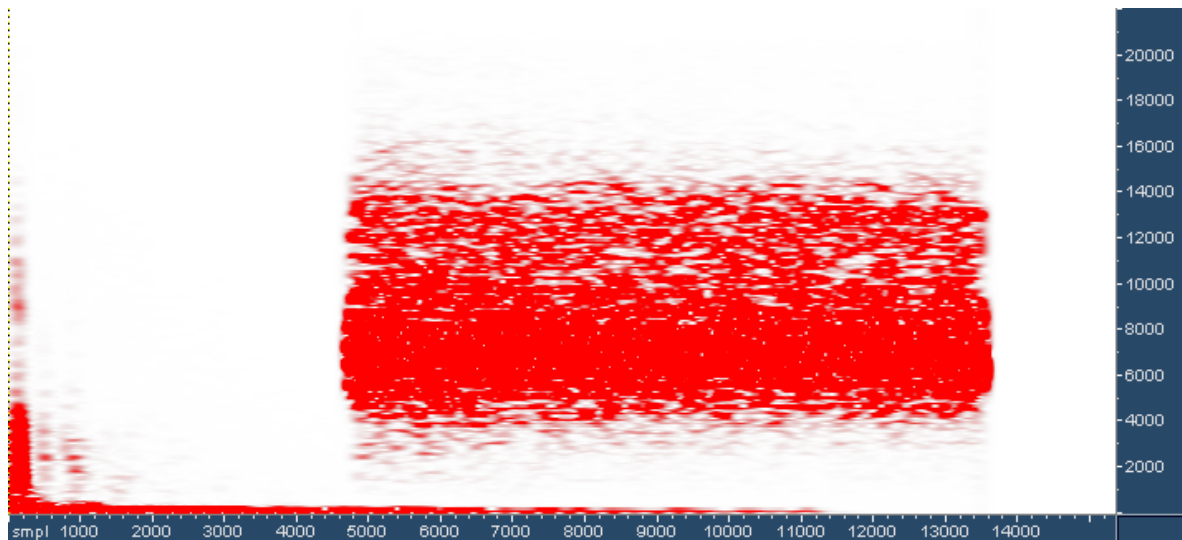


Figure 3.1: Overlapping in time. FFT binsize 256, linear energy plot, Hanning window.

3.1.2 Overlapping in frequency

Overlapping in frequency means that there are two drumsounds that inhabit the same frequencies. These drumsounds are not played simultaneously but exist in the same drumsequence (e.g. The same drumsound played twice, with a pause between). To separate drumsounds that are just overlapping in frequency, one solution is to find the start and stop samplepoints of the drumsounds and just separate these from the original drumsequence. Figure 3.2 gives an example of two drums overlapping in frequency.

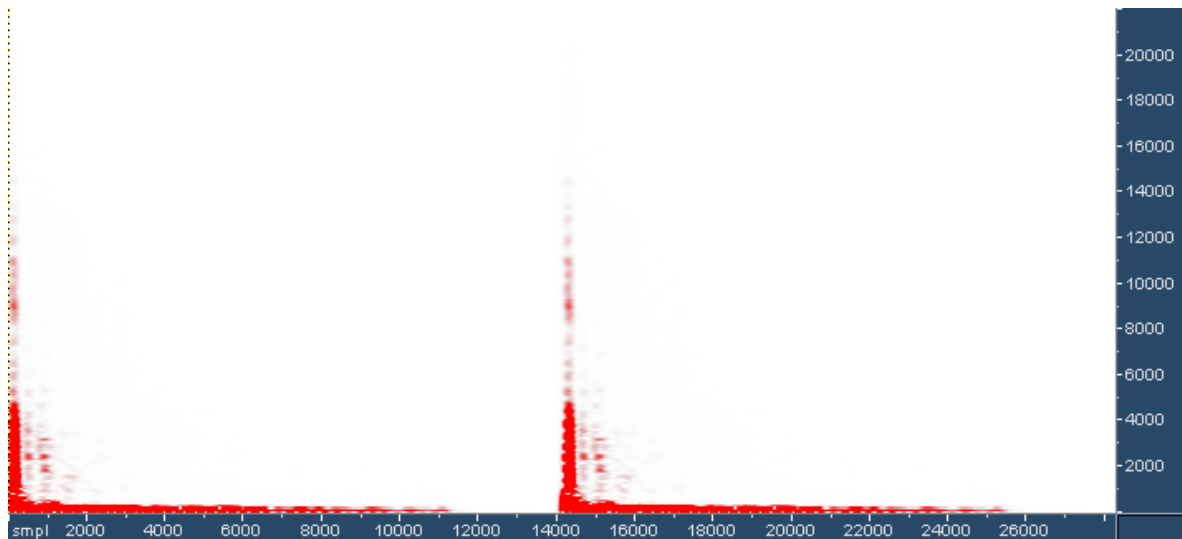


Figure 3.2: Overlapping in frequency. FFT binsize 256, linear energy plot, Hanning window.

3.1.3 Overlapping in both time and frequency

The last kind of overlapping is in both time and frequency. This is the most normal form of overlapping in our problem domain (drumsequences). An example of this is two drumsounds played simultaneously with overlapping frequencies, like a snare-drum and a hihat. Both of these drums are in the higher frequency area. A separation of this kind is not easily solved by either filtering or finding the start and stop samplepoints of the drumsounds involved. Figure 3.3 gives an example of two drumsounds overlapping in both time and frequency.

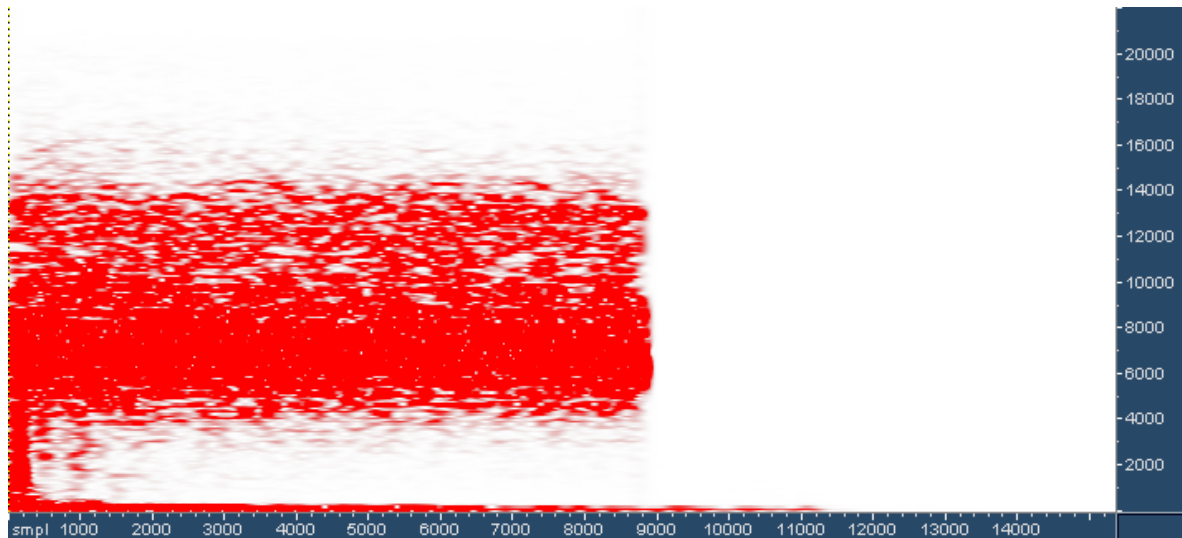


Figure 3.3: Overlapping in both time and frequency. FFT binsize 256, linear energy plot, Hanning window.

3.2 Levels

By using overlapping as a measure of complexity the following levels can be defined.

3.2.1 Level One - no overlapping in time

At this level there is a pause between each drumsound. This means that there might be overlapping frequencies, but this is not an issue since the different drums are in different time-areas of the drumsequence. Questions asked to determine the effectiveness of an algorithm at this level could be:

1. how many drumsounds are separated
2. how accurately is the onset of each drumsound detected
3. how accurately is the end of each drumsound detected
4. how does it perform on weak drumsounds.

Typical problem drumsequences could include weak hihats and reversed hihats¹ etc.

¹To detect drumsounds of this type accurately is not expected by this system.

3.2.2 Level Two - no overlapping in frequency

On the second level we have no overlapping in frequency. This means two drumsounds do not share the same frequency-area of a drumsequence, but they could be played simultaneously (i.e. overlapping in time). It should be noted however that this scenario is quite rare as drumsounds typically occupy a broad frequency-area.

Questions asked to determine the effectiveness of an algorithm at this level could be:

1. how many drumsounds are separated
2. how accurately is the onset of each drumsound detected
3. how accurately is the end of each drumsound detected
4. how accurately are the highest frequency-limits of a drum detected
5. how accurately are the lowest frequency-limits of a drum detected
6. how accurately are the drums separated.

One might think that finding the start and end points of a drumsound on level two would be the same task as it is on level one, this is not necessarily so. On level two we can have drumsounds overlapping in time, this could mean that two sounds are played simultaneously, and an accurate algorithm would distinguish the two drums as separate drums and separate start times.

Since we have overlapping in time, meaning we have mixed signals, the detection of frequency limits are also measured. We look upon a drumsound here as a rectangle (viewed in a multirate-view) which has an upper and lower frequency limit, and an onset and end point. The detection of the limits might be a separate problem from the actual separation of the different drumsounds. Therefore the quality of the separation of drumsounds are also questioned (e.g. This might be a test of the accuracy of the filters used in extracting the different drums. It is no simple matter to make ideal digital filters.

The reason why the accuracy (or quality if you like) of the separation is not tested on level one is because this will be covered in the start/stop test. Separation on level one is just to extract the data found from drum onset to end without any filtering.

Typical problem drumsequences on level two could include drumsounds that reside in very near frequency areas, or drumsounds that have got weaker frequency-parts that still are crucial to the overall "feel" of the drumsound. Also, drumsounds that have weaker frequency-parts in the middle of their total frequency-span might prove very difficult to separate correctly and should be included in the test set (they might be taken for two drums instead of one).

3.2.3 Level Three - weak overlapping in time and frequency

On the third level we have got what we call weak overlapping in time and frequency. With this we mean that we can "block" the different drumsounds i.e. draw unique polygons along the time and frequency-areas a drumsound occupies, and the polygons will not share the same areas. Here we look on the drumsounds as polygons rather than rectangles. Questions asked to determine the effectiveness of an algorithm at this level could be:

1. how many drumsounds are separated
2. how accurately are the higher frequency-limits of a drum detected

3. how accurately are the lower frequency-limits of a drum detected
4. how accurately are the drums separated.

Testing for the start and end points of a drumsound is not that interesting at this level. What is interesting is to measure the quality of the detection of the frequency-limits, and the separation of the drumsound. What makes it different from level two is that the detection and separation has to handle changing frequency-limits.

Typical problem drumsequences could be the same as used in level two that *also* has weak overlapping.

3.2.4 Level Four - true overlapping in time and frequency

On the fourth level we have true overlapping. With this we mean that we have overlapping in both frequency and time, and it is not possible to "block" the drumsounds in different unique polygons. An example of this could be two similar drums played at the same time. Questions asked to determine the effectiveness of an algorithm at this level could be:

1. how many drumsounds are separated
2. how accurately is the onset of each drumsound detected
3. how accurately is the end of each drumsound detected
4. how accurately are the higher frequency-limits of a drum detected
5. how accurately are the lower frequency-limits of a drum detected
6. how accurately are the drums separated.

At this level it becomes interesting again to look at the start and end points detected. Since we have true overlapping we can have two very similar sounds played almost at the same time. These should be detected as different drums with different start points. This is different from level two, because in level two we didn't have overlapping in frequency (but possibly in time), and the overlapping in frequency adds new complexity to the problem.

Also detection of the limits of a drumsound in the frequency area is highly interesting to measure at this level. Since we have true overlapping the drumsounds will be mangled together and trying to define the silhouette that defines the boundaries of each drumsound will be difficult.

The separation of true overlapped drumsounds will also be very interesting to measure at this level. A goal here is of course to include as little as possible of the other drumsounds occupying the same time and frequency-area.

Typical problem drumsequences will be as mentioned above similar or even identical drumsounds played at the same time. They could also include weak drumsounds mixed on top of strong ones (e.g. a weak hihat played together with a strong snare).

3.3 Measure of correctness for onset extraction

At the moment there are no standard for measuring the correctness of an onset extraction algorithm. Different methods have been proposed and it is natural to use these, and if possible extend them and enhance them to better suit our needs.

In the article "Issues in Evaluating Beat Tracking Systems" by Goto and Muraoka [GM97] they say "Because most studies have dealt with MIDI signals or used onset times as their input, the evaluation of audio-based beat tracking has not been discussed enough".

Also Rosenthal mentions in his paper "Emulation of human rhythm perception" [Ros92a] that evaluating the ideas behind many of the previous systems is not a straightforward task, and that different researchers have worked on different aspects of the problem.

One such aspect is the measure of correctness for onset extraction from audio material.

3.3.1 What is a correct onset

A drum onset is defined as the time where the drumsound starts (or the magnitude reaches a certain peak). A correctly extracted onset has a time equal to the original onset, within a certain threshold. For my test this threshold has been subjectively set to +/- 5 ms unless noted otherwise in the test results.

3.3.2 Klapuri correctness

The measure of correctness used in Anssi Klapuris paper [Kla99] worked by getting a percentage based upon the total numbers of onsets, the number of undetected onsets, and the erroneous extra onsets detected:

$$correct = \frac{total - undetected - extra}{total} \cdot 100\% \quad (3.1)$$

Where *total* is the number of onsets in the original sample, *undetected* is the number of missed detections, *extra* is the number of erroneous detections.

This seems like a reasonable measure but as we look closer at it we find that it is possible to get negative percentages, and even negative percentages below -100%. As an example let us imagine an algorithm tried on a drumsequence with 6 onsets in it. The imagined algorithm extracts 3 correct onsets, misses 3 and also has 4 extra onsets detected. The calculated correctness would in this case be:

$$-16.7\% = \frac{6 - 3 - 4}{6} \cdot 100\% \quad (3.2)$$

Not a really good measure for this example. The onset extraction algorithm does get half of the onsets correct so a *negative* percentage for its correctness seems pretty harsh.

3.3.3 Error rate, derived measure correctness

A measure of error instead of a measure of correctness could be better for certain situations. Error rates are usually allowed to be above 100%, and when the error rates get small enough, it turns out to be more convenient to use error rates, since small numbers are more intuitive, for humans, than numbers very close to 100.

$$error\ rate = \frac{undetected + extra}{total} \cdot 100\% \quad (3.3)$$

Where *total* is the number of onsets in the original sample, *undetected* is the number of missed detections and *extra* is the number of erroneous detections.

It would also be possible to derive a measure of correctness from the error rate:

$$correctness = \left(1 - \frac{undetected + extra}{total}\right) \cdot 100\% \quad (3.4)$$

As we see this is actually exactly the same as the correctness values used by Klapuri above.

3.3.4 Suggested correctness

Borrowing some of the ideas from Klapuri we suggest a new measure of correctness that tries to improve on the scaling of the percentage but still penalize wrong detections:

$$correct = \frac{detected}{onsets} \cdot \left(1 - \frac{extra}{estimated}\right) \cdot 100\% \quad (3.5)$$

Where *detected* is the number of correctly detected onsets in the estimate, *onsets* is the number of actual onsets in the original sample, *extra* is the number of erroneous detected onsets in the estimate (estimated - detected) and *estimated* is the total number of onsets in the estimate.

This equation suits our needs better as it is always in the scale from 0..100% and extra onsets are penalized but not as harshly as in the method used by Klapuri. If we use the same example as above with this method we get a correctness of 21.4%.

3.3.5 Table of behaviors

Below is a table showing the behavior of the different methods mentioned above when given varied inputs:

row	onsets	estimated	detected	undetected	extra	Klapuri %	suggested %
1	6	6	6	0	0	100	100
2	6	0	0	6	0	0	0 ¹
3	6	6	0	6	6	-100	0
4	6	6	1	5	5	-66.7	2.8
5	6	7	3	3	4	-16.7	21.4
6	6	6	3	3	3	0	25
7	6	5	3	3	2	16.7	30
8	6	20	3	3	17	-233.3	7.5
9	30	25	20	10	5	50	53.3
10	30	35	20	10	15	16.7	38.1

Table 3.1: Table of behaviors.

It is interesting to notice how the suggested measure rewards more restrictive onset extraction algorithms. If we look at the example where we have the same number of correct detections, but variations in the total estimated number of onsets (row 5,6 and 7) we see a variation from 21.4% to 30%.

Also if the number of estimated onsets gets very large in comparison to the actual number of onsets we still get a positive percentage but a small one (7.5%), even though half of the onsets in the original sample has been detected (row 8).

¹division by zero

Chapter 4

Algorithms and their applications

There are different ways of attacking the problem and trying to solve it. The previous chapters have tried to separate the problem in different stages, and in order to get an understanding of the problem this can be beneficial. But, as stated before, some methods do not follow this strict classification and attack the problem from different angles. Here we classify the different approaches into different categories.

4.1 Finding and defining areas of a drumsound

This problem is a combination of both finding the start and end points of a drumsound, and finding its frequency boundaries. To do this one has to set some practical limits to when a drumsound actually starts, and how weak a signal can become before it is looked upon as neglectable/silent.

4.1.1 RMS/MAX analysis

RMS analysis¹ works on the amplitude of the drumsequence (i.e. in the time domain, on the waveform) and describes the variation of the magnitude in the sample. It calculates the RMS value for small parts of the sample and produces a new array, a vector, that can be used in predicting where a drum starts and ends. For a vector v with N elements (which are samplepoints from the whole sample) the RMS value would be:

$$RMS = \sqrt{\frac{v_1^2 + v_2^2 + \dots + v_N^2}{N}} \quad (4.1)$$

This is calculated for each buffer, and the end result is a new vector.

By MAX analysis we have the same approach as with the RMS that we step through the whole sample buffer-wise, but instead of calculating the RMS value, we find the element in v with the highest value and return this. The resulting MAX vector, which has the same length as the RMS vector has a faster attack and is more sensitive to changes in the magnitude. Figure 4.1 show the result on doing MAX and RMS analysis on a drumsequence.

¹Root Mean Square.

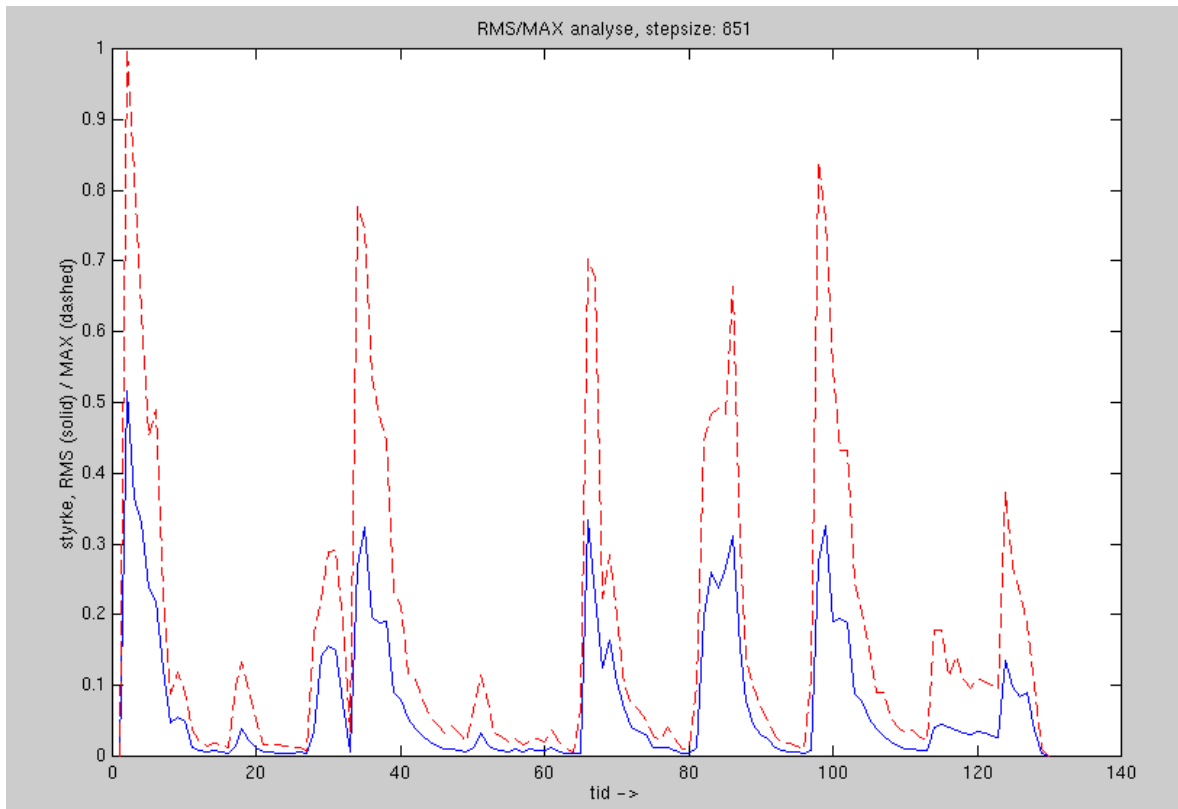


Figure 4.1: RMS and MAX analysis. RMS is blue, MAX is the dotted red.

4.1.2 Frequency-band decibel-threshold analysis

The RMS/MAX analysis operate in the time-domain of a sample. We could also use a similar analysis in the time-frequency domain, operating on the power-density spectrum produced when calculating the sonogram of a sample.

Instead of looking at the amplitude of the sample, the decibel² value for each frequency-band is analyzed. The climb in dB between the previous and current value is measured, and if above a certain threshold, notifies a possible drum being played. The number of hits at each time interval (a sonogram is divided into time and frequency) is counted, and if above a threshold, signifies that a drum *has* been hit. Figure 4.2 shows a pseudo implementation of the analysis.

```

for (i=0;i<bands;i++){
  for (j=0;j<length_of_band;j++) {
    if (sgram[i][j] > dBThreshold)
      if ((sgram[i][j] - prev_sgram_value) > dbClimbThreshold) regHit(j);
  }
}
traverses_hits {
  if (hits_at_same_time > hitThreshold) regDrum(time);
}

```

Figure 4.2: Pseudo implementation of the *Frequency-band decibel-threshold* algorithm.

²Decibel, dB, describes the intensity of a signal.

This method is discussed in Chapter 6. Figure 4.3 shows a result from the analysis, the yellow line is the onset of drumsounds predicted by the analysis.

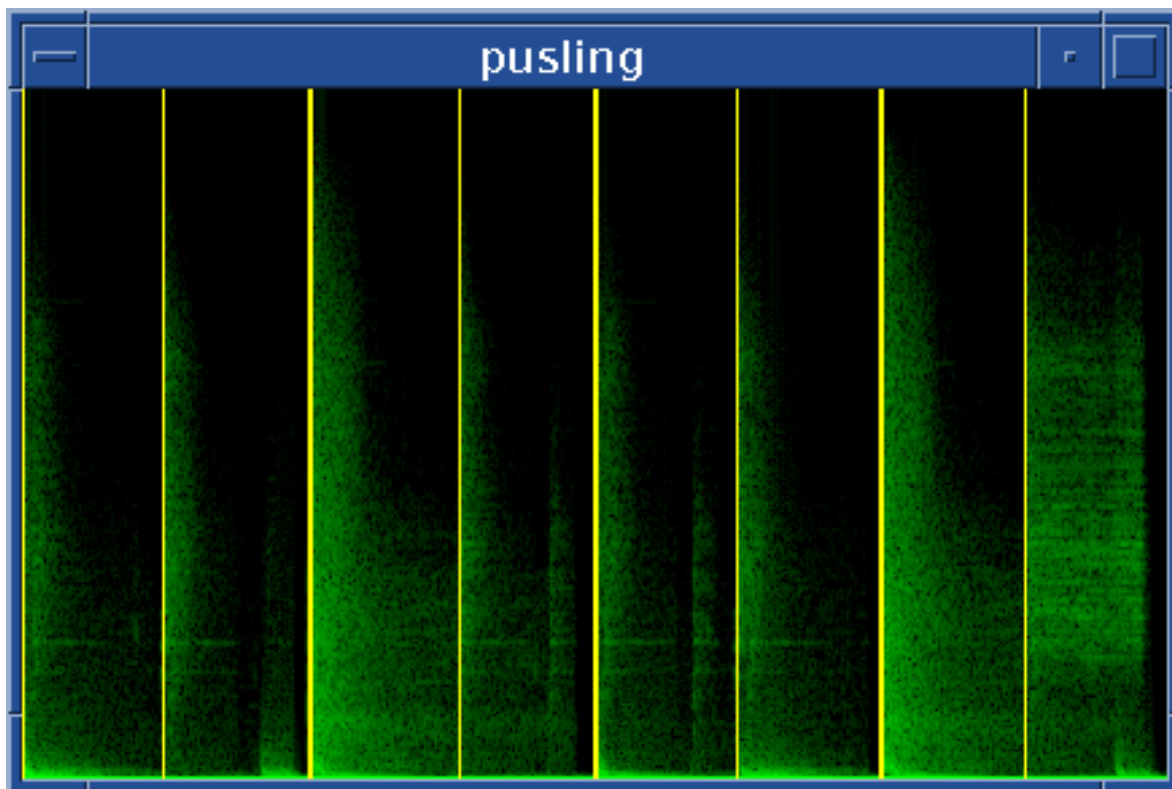


Figure 4.3: Shows the results from the *Frequency-band decibel-threshold* analysis. The yellow lines signifies the estimated onsets of the drums in the drumsequence.

If you study the green areas displaying the power of the signal at a specific time and frequency, you will see that some weaker parts are not detected as drumsounds.

4.1.3 Mean decibel-threshold analysis

Both the *RMS/MAX* analysis, and the *Frequency-band decibel-threshold analysis* fail to detect two drums hit at the same time, they will register only as one drum. As an extension to the *Frequency-band decibel-threshold analysis* we have tried to make an algorithm that also detects multiple drums using the finds from it. It calculates mean values of smaller buffers at the position a drum is reported to have been hit. In other words, it iterates over the whole frequency spectrum at a specified position, if it finds sections above a certain threshold, it signifies that a drum occupies this area.

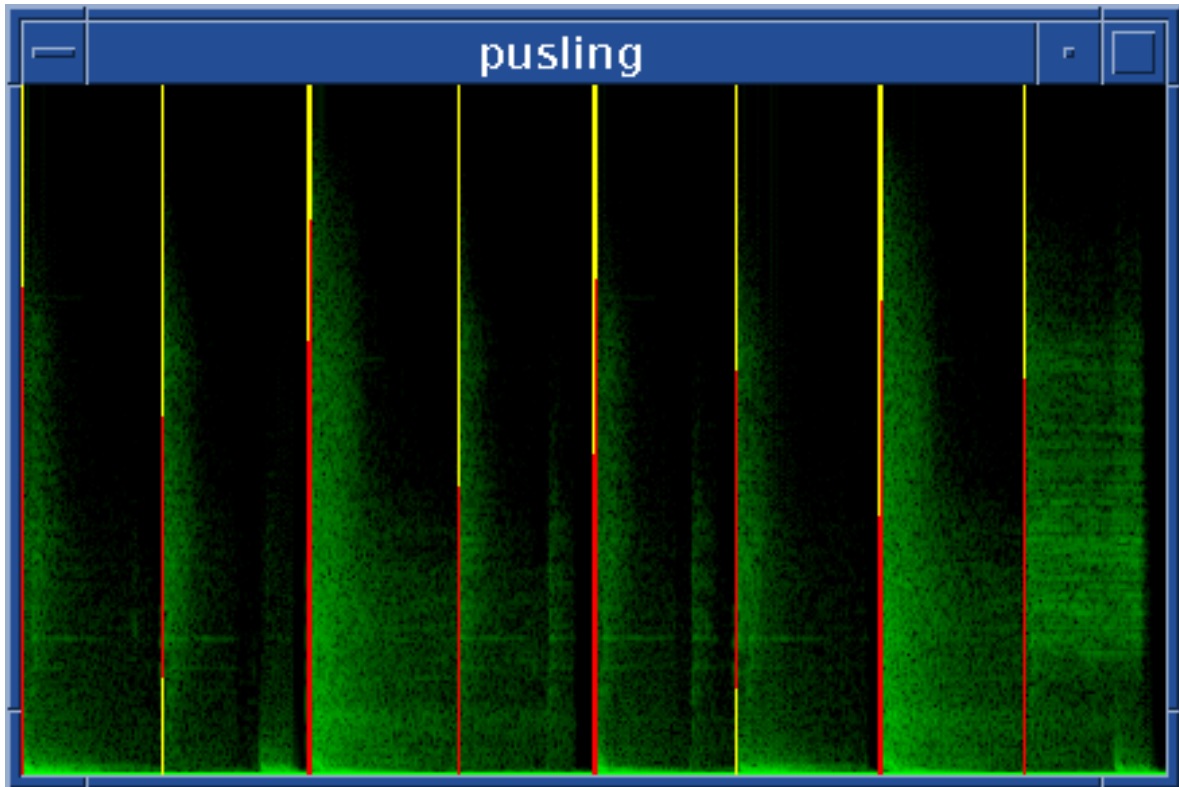


Figure 4.4: Shows the results from the *Mean decibel-threshold* analysis. The added red lines show the detected frequency boundaries of the drumsounds detected.

In Figure 4.4 the red lines signify the finds of the Mean decibel-threshold analysis. The lines visualize the estimated upper and lower frequency limits of the drumsounds. The results shown here are not accurate, as an example study the latest detected onset. The frequency limit of this drumsound is detected lower than it is, and the upper frequency limit is also detected inaccurately.

In Figure 4.5 follows a pseudo-implementation of the *Mean decibel-threshold analysis*:

```

for (i=0;i<frequencies-bufferLength;i++) {

    // calculates meanvalue of a small buffer
    calculateMeanValue(i,bufferLength);

    // define start of lower frequency limit for drum, else define start of
    // upper frequency limit of drum
    if ((meanValue > dBMeanThreshold) && (!drumStart)) startDrum(i);
    else if ((meanValue < dBMeanThreshold) && (drumStart))
        endDrum(i+bufferLength);
}

```

Figure 4.5: Pseudo implementation of the *Mean decibel-threshold* algorithm.

Chapter 6 includes a discussion of Frequency-band decibel-threshold analysis and Mean decibel-threshold analysis. The conclusion for both are simple, they are too simple and need to be improved.

4.2 Separating drums

If one want to separate the drums found in the above mentioned methods, there are different ways of doing this. If the drumsounds are not overlapping in time, it is only a question of finding the start/end-points and separation is performed by extracting the wanted parts in the drumsequence.

For more complex overlapping filtering is a solution. A filter works like a barrier that keeps out certain frequencies, and let others pass. Filtering can be done in either the time or the frequency-domain. In the time-domain there are two usual methods of filtering, IIR and FIR filters. IIR stands for *infinite impulse response*, FIR stands for *finite impulse response*. Linear time-invariant (LTI) systems can be classified into FIR or IIR types depending on whether their impulse response has finite or infinite duration [Orf96]:98. The impulse response of a system (or a filter as in this case) is the output of the system when given a unit impulse³. A FIR filter has impulse response $h(n)$ that extends only over a finite time interval, say $0 \leq n \leq M$, and is identically zero beyond that [Orf96]:108:

$$\{h_0, h_1, h_2, \dots, h_M, 0, 0, 0, \dots\}$$

An IIR filter has no such finite duration, hence infinite impulse response. The n th output for a general M -tap FIR filter is described in Equation 4.2 [Lyo01]:165. $x(n)$ is the original sample, $h(m)$ are filter coefficients.

$$y(n) = \sum_{m=0}^{M-1} h(m)x(n-m) \quad (4.2)$$

Equation 4.3 describes the equation for IIR filters, where $x(n)$ is the original sample, $y(n)$ previous filter output and $b(m)$ and $a(k)$ are filter coefficients.

$$y(n) = \sum_{m=0}^{M-1} b(m)x(n-m) + \sum_{k=0}^{K-1} a(k)y(n-k) \quad (4.3)$$

In the frequency-domain filtering is done in another way. In the time-domain convolution is used to filter. Now because of the transformation to frequency domain, multiplying in the frequency-domain is the same as convolution in the time domain. Hence, multiplying the data in the frequency domain is the same as filtering.

4.3 Finding and separating a drumsound

The two problems of defining the onset and frequency limits of a drum, and then separating it, can be done in one step. The method below does not find the different drums as the methods described above, it just separates what it believes to be different signals.

4.3.1 ICA - Independent component analysis

Blind source separation (BSS) is a known academic problem, and actually a problem related to the drum-separation problem: Given two linear *mixes* of two signals (which we know to be independent), find the signals using these two mixes. Put another way, imagine two people talking together and this is digitally recorded (sampled) using two microphones. The voice of each person is recorded by both microphones. Using the two sampled recordings, separate each voice from each other.

³ $\delta(n) \equiv \begin{cases} 1, & \text{for } n=0 \\ 0, & \text{for } n \neq 0 \end{cases}$ also know as the *unit sample vector* [PM96]:45.

"Source signals are denoted by a vector

$$s(t) = (s_1(t), \dots, s_n(t))^T, t = 0, 1, 2, \dots$$

and there is the assumption that each component of $s(t)$ is independent of each other." "Without loss of generality, we assume the source signal $s(t)$ to be zero mean. Observations are represented by

$$x(t) = (x_1(t), \dots, x_n(t))^T$$

they correspond to the recorded signals. In the basic blind source separation problem, we assume that observations are linear mixtures of source signals:

$$x(t) = As(t)$$

where A is an unknown linear operator. A Typical example of linear operators is an $n \times n$ real valued matrix." "The goal of blind source separation is to find a linear operator B such that the components of the reconstructed signals

$$y(t) = Bx(t)$$

are mutually independent, without knowing the operator A and the probability distribution of source signal $s(t)$. Ideally we expect B to be the inverse of operator A " [MIZ9x]:2-3.

"Popular approaches of solving the blind source separation problem using independent component analysis are based on the following three strategies:

1. factoring the joint probability density of the reconstructed signals by its margins
2. decorrelating the reconstructed signals through time, that is, diagonalizing the covariance matrices at every time, and
3. eliminating the cross-correlation functions of the reconstructed signals as much as possible" [MIZ9x]:3.

The difference between this thesis problem and the blind source separation problem is that you usually operate with more mixes in BSS. In this thesis we operate on mono or stereo samples this gives at most two mixes. With ICA you need at least as many mixes as the signals that you want to extract. In other words with ICA it would only be possible to at most extract two different drums from a drumsequence, and this is not enough for complete drumsequences. For *parts* of a drumsequence where we suspect that the sample might contain only two distinct drumsounds ICA could still be applied.

Overcomplete independent component analysis tries to alleviate this problem, and tries to solve the BSS problem when there are more sources than mixes [LS98]. Common for both overcomplete and normal ICA is that you have to have at least two mixes. For drumsequences this is still interesting if you have samples recorded in stereo and Overcomplete ICA could possibly be used as the analysis tool for complete drumsequences.

4.4 Finding, separating and match finding

If you were to substitute the drumsounds a drumsequence consist of with similar drumsounds from a database, one way of doing this could be with cross-correlation.

4.4.1 Cross-correlation

Cross-correlation between two vectors (samples) x and y can be defined as a sequence r

$$r_{xy}(l) = \sum_{i=1}^n x(i)y(i-l), l = 0, \pm 1, \pm 2, \dots, \pm n \quad (4.4)$$

A maybe easier way to understand cross-correlation is to picture yourself two vectors x and y which are shifted in relation to each other, one to the left, the other to the right. For each shift, the corresponding elements of each vector are multiplied and added.

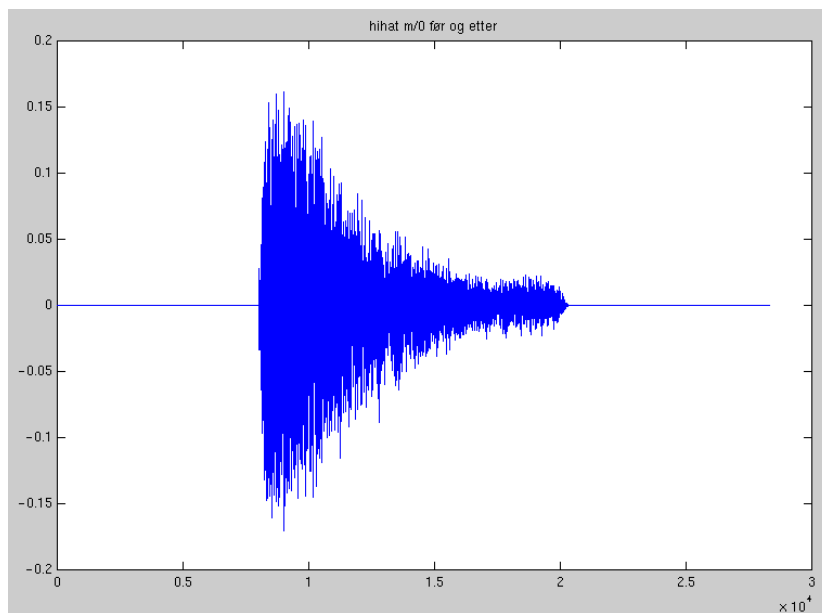


Figure 4.6: Waveform of hihat-sample with zero-padding.

In Figure 4.6 we have the waveform of a hihat with zero-padding before and after the actual signal. This is auto-correlated⁴, and as we can see in Figure 4.7 there is a distinct peak at the position where the vectors are aligned to each other.

⁴It means that the vector being correlated is being correlated with itself.

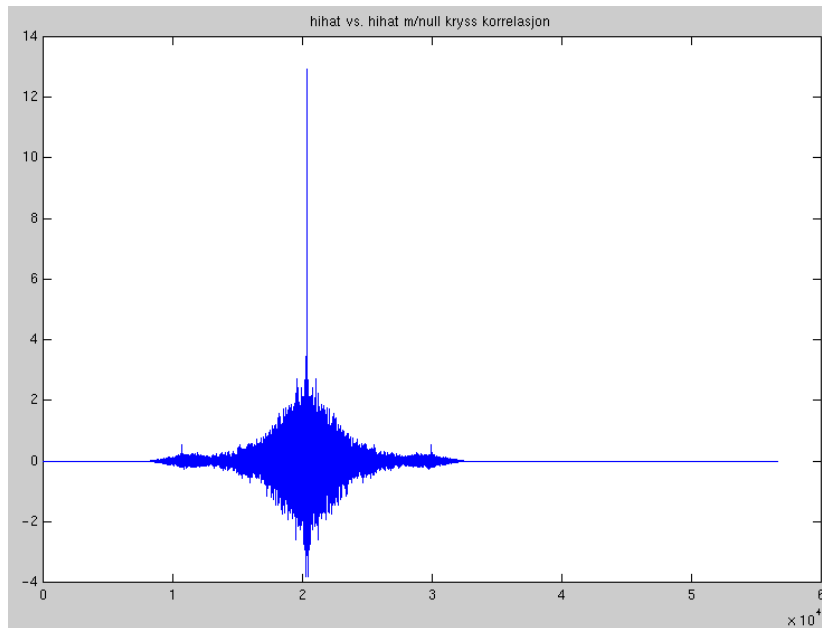


Figure 4.7: Auto-correlation of hihat-sample.

This distinct peak is what is being used to determine which of the different samples from the database that are the closest match. An example of two drumsounds that are not too similar, the Figure 4.8 shows the cross-correlation between a hihat and a bassdrum.

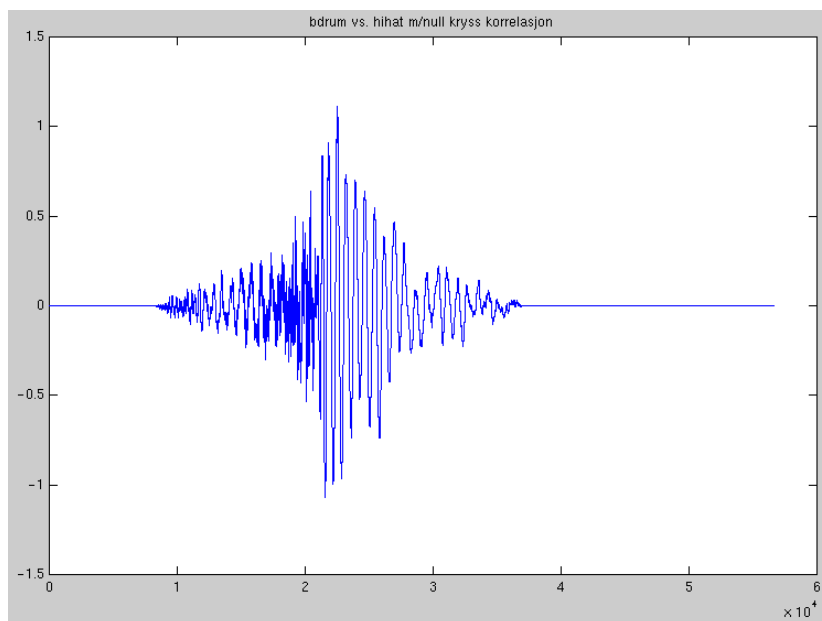


Figure 4.8: Cross-correlation between a hihat and a bassdrum.

As we can see this peak is not as distinct as the above from the auto-correlation.

4.5 Beat Tracking / Onset extraction algorithms

One area of research that is closely related to the beat extraction problems studied in this thesis is *beat tracking*. Beat tracking⁵ is what we humans do when we listen to a piece of music and follow it by hand-clapping or foot-tapping. "Identifying the temporal location of downbeats is a fundamental musical skill that comes naturally to most people. Even people without musical training can tap their foot with the beat when they hear a musical performance and can generally perform this task with ease and precision even in the face of unusual rhythms, musical expressiveness, and imprecise performances." [AD90]

It is important to understand that even though we can perform this task with ease, making a computer system doing the same thing is difficult. "In a reasonable model of human auditory processes, many other processes act on the incoming auditory data. These include: parsing the music into separate events, separating the music into streams, noting repeated patterns, parsing the harmonic structure, recognizing instruments, and so on." [RGM94].

The main difference between beat extraction/beat separation and beat tracking is that with beat extraction the system doesn't need to "understand" the rhythm. The whole point with a beat tracking system is to be able to predict when the next beat will occur. This is not so with a beat extraction system - here the goal is to find and separate the different sounds that a drumloop consist of. Of course, it could be argued that "understanding" the rhythm might help in the extraction process.

The similarities between the problems are that both beat tracking and beat extraction need to find the onset times of drumsounds. In order to try to predict the rhythmic structure of an audio signal one needs to know the onset times of the drums⁶, and in order to separate out the different drumsounds one needs to know at least where they start.

4.5.1 Different approaches

Various beat tracking related systems have been undertaken in recent years [DMR87]; [DH89]; [AD90]; [Dri91]; [Ros92b]; [DH94]; [Ver94]; [Lar95]; [GM95]; [GM96]; [Sch98]. These differ mainly in two categories:

1. What form of signals they use as their input data (audio or MIDI⁷).
2. If the systems operate in real-time or not.

Now obviously the systems working on audio signals are the most interesting in our case, this is because these systems will run into more of the same problems as with beat extraction. Whether the system runs in real-time or not isn't that important, but strangely enough the systems operating on audio signals often also are real-time systems [GM96] and [Sch98].

4.6 BTS

Masataka Goto and Yoichi Muraoka has developed a beat tracking system BTS that tracks the beat for popular music in which drums maintain the beat. In their experiment using music from commercially distributed compact discs, the system correctly tracked beats in 40 of 42 popular songs [GM95]. The main features of their system are:

⁵or Rhythm tracking.

⁶or other instruments depending on what kind of music one is processing.

⁷or similar derivatives thereof.

1. The system manages multiple agents that maintain multiple hypotheses of beats.
2. The system use musical knowledge stored as drum patterns to make context-dependent decisions.
3. All processes are performed based on how reliable detected events and hypotheses are.
4. Frequency-analysis parameters for onset detection are dynamically adjusted.

4.6.1 System Description

The BTS system assumes that the time-signature of the music is 4/4 and that the tempo of the beat is from 65 M.M.⁸ To 185 M.M. Also, the system expects the beat to be almost constant.

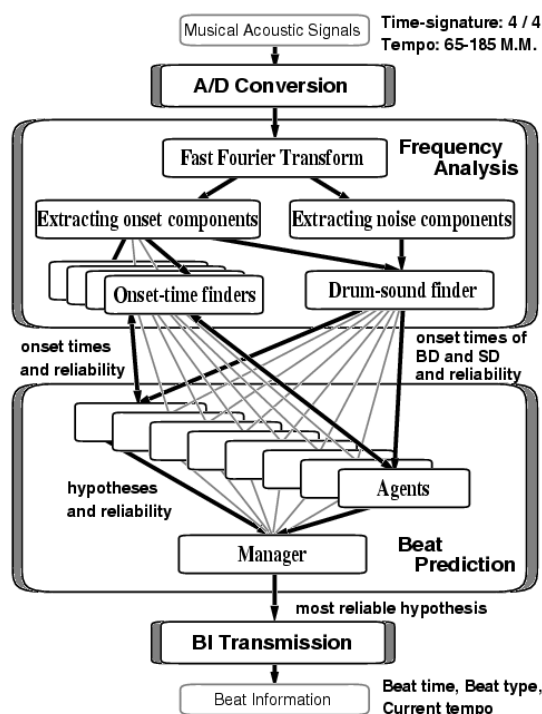


Figure 4.9: The figure shows an overview of the BTS system. Image from [GM95].

Figure 4.9 show the main features of the BTS system. The two main stages in the system is *Frequency Analysis* and *Beat Prediction*. In the Frequency analysis stage onset time is extracted in different frequency bands. Also, the onset time for bass drum (BD) and snare drum (SD) is also found. In the Beat Prediction stage the data from the previous stage is used by multiple agents that have different hypotheses which try to predict the beat. These agents are run in parallel. Finally the beat predicted by the most reliable hypothesis is then transmitted to other applications who want to use the information.

⁸The number of quarter notes pr. minute.

4.6.2 Frequency Analysis

Finding the onset times are obtained by the following process. The frequency spectrum (the power spectrum) is calculated by FFT. Each time the FFT is applied to the digitized signal, the window of the FFT is shifted to the next time position. The frequency resolution is determined by the window size. The time resolution is determined by the time shift of each FFT window.

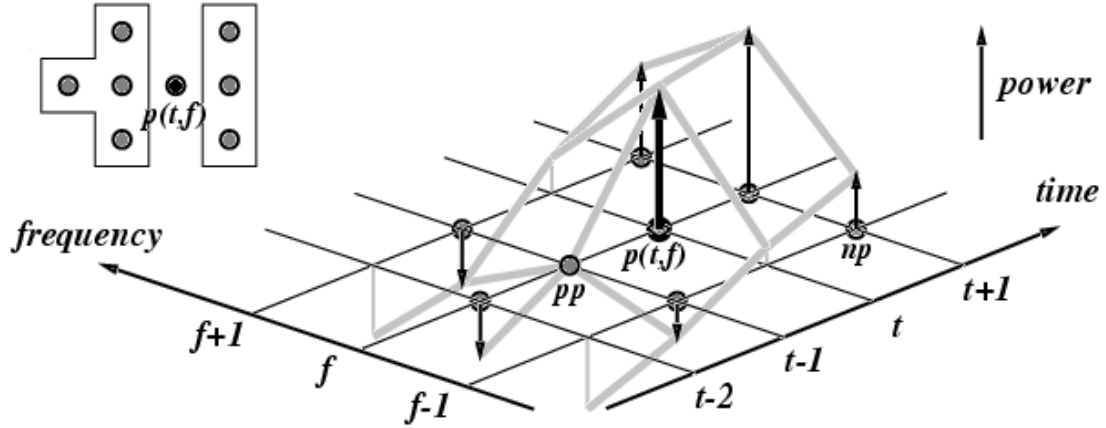


Figure 4.10: Explains extracting onset components. Image from [GM95].

The onset components and their degree of onset (rapidity of increase in power) are extracted from the frequency spectrum. The way this is done in BTS is that one regards the frequency component $p(t, f)$ that fulfills the conditions:

$$\begin{cases} p(t, f) > pp \\ np > pp \end{cases}$$

where $p(t, f)$ is the power of the spectrum of frequency f at time t , pp and np are given by:

$$\begin{aligned} pp &= \max(p(t-1, f), p(t-1, f \pm 1), p(t-2, f)) \\ np &= \min(p(t+1, f), p(t+1, f \pm 1)). \end{aligned}$$

In effect, these conditions extract components whose power have been increasing. Figure 4.10 gives a graphical presentation of the analysis.

The degree of onset $d(t, f)$ is given by:

$$d(t, f) = p(t, f) - pp + \max(0, p(t+1, f) - p(t, f)).$$

4.6.3 Finding onset times

The onset time is found by peak-finding in $D(t)$ along the time axis. $D(t)$ is the sum of the degree of onset from $d(t, f)$:

$$D(t) = \sum_f d(t, f).$$

$D(t)$ is linearly smoothed by convolution kernel before the peak time and peak value of $D(t)$ are calculated. The onset time is given by this peak time.

4.6.4 Detecting BD and SD

The BTS system finds the bassdrums and the snaredrums in the analyzed signal by examining the frequency histogram for the onset components. The characteristic frequency for BD is given by the lowest peak of the histogram, and the SD is given by the maximum peak of the histogram above the frequency of BD, see Figure 4.11.

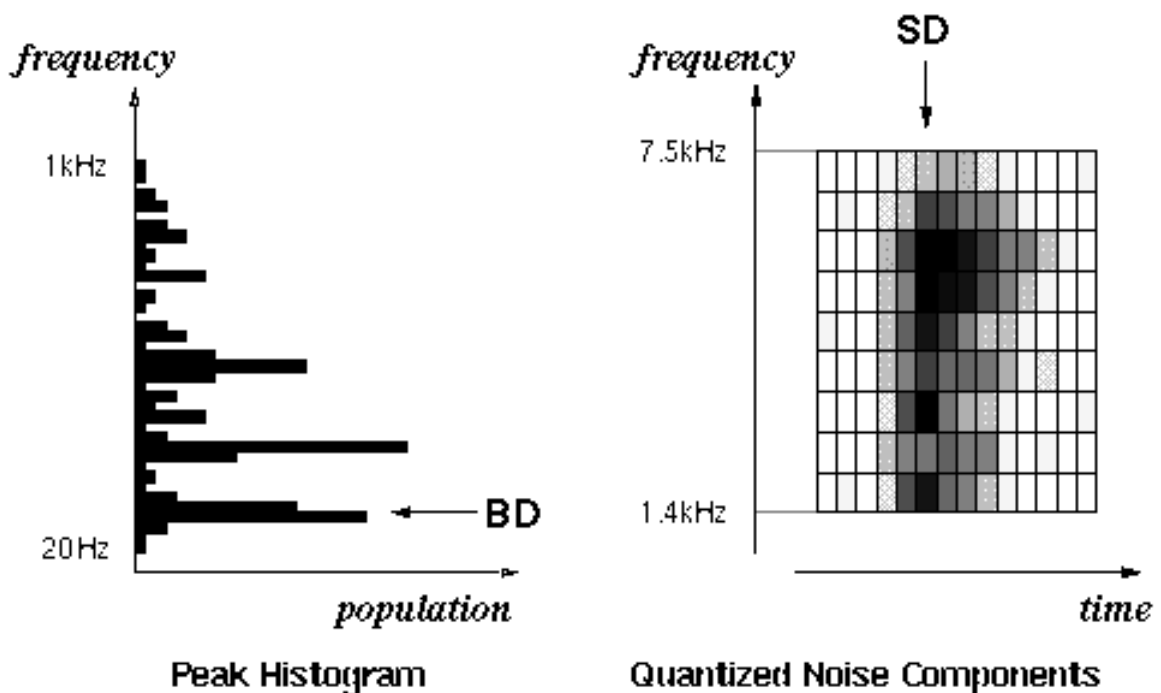


Figure 4.11: Shows detection of BD and SD. Image from [GM95].

4.6.5 Beat prediction

Using the results from the frequency analysis stage BTS uses multiple agents that interpret the results and maintain their own hypotheses, each of which consists of next time predicted, its beat type, its reliability and current IBI⁹.

The end result is a system capable of following a beat without losing track of it, even if some hypotheses become wrong. The interesting parts of this system for this thesis is especially the way onset components are found.

4.7 Onset detection using psychoacoustic knowledge

Inspired by the works of Goto [GM95];[GM96] and Scheirer [Sch98] Anssi Klapuri designed a system where he uses psychoacoustic knowledge to perform onset detection. The system utilizes band-wise

⁹The inter-beat-interval is the temporal difference between two successive beats.

processing and a psychoacoustic model of intensity coding to combine the results from the separate frequency bands [Kla99].

4.7.1 System overview

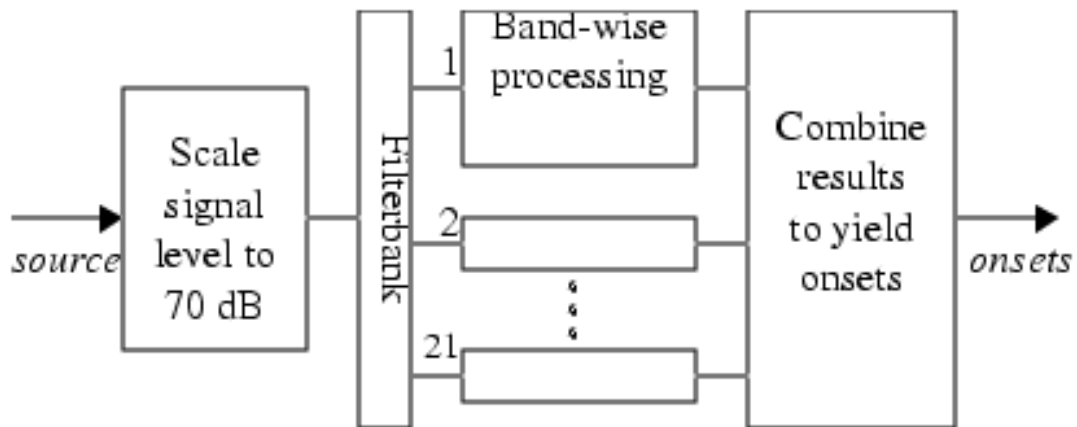


Figure 4.12: System overview. Image from [Kla99].

As we can see from Figure 4.12, the signal is first normalized to 70 dB using the model of loudness proposed Moore et al. [MGB97]. Then a filterbank divides the signal into 21 non-overlapping bands. These bands are processed and *onset components* are determined. Finally the onset components are combined to yield onsets.

4.7.2 Filtering of the bands

Klapuri uses a filterbank with nearly critical-band filters which cover the frequencies from 44 Hz to 18 kHz. The lowest three among the required 21 filters are one-octave band-pass filters. The remaining eighteen are third-octave band-pass filters. All subsequent calculations can be done one at a time.

To ease the computations each filter is full-wave rectified¹⁰ and then decimated by 180¹¹. Amplitude envelopes are calculated by convolving the band-limited signals with a 100 ms half-Hanning (raised cosine) window. This window system performs much of the same energy integration as the human auditory system, preserving sudden changes, but masking rapid modulation [Sch98],[McA92]. Figure 4.13 shows the band-wise processing.

¹⁰ $out(x) = abs(filter(x))$.

¹¹Every 180th sample value is chosen.

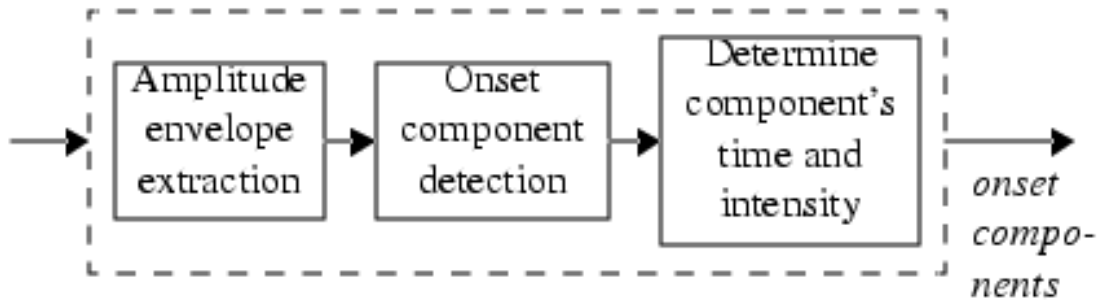


Figure 4.13: Processing at each frequency band. Image from [Kla99].

4.7.3 Onset component detection

What is interesting in Klapuri's system is that he tries to answer the problems met in other systems of a similar kind. These systems determine onset detection based on the calculation of a first order difference function of the signal amplitude envelopes and taking the maximum rising slope as an onset or an onset component.

Klapuri discovered in simulations that the first order difference function reflects well the loudness of an onsetting sound, but its maximum values fail to mark the time of an onset precisely. This he argues is due to two reasons. "First, especially low sounds may take some time to come to the point where their amplitude is maximally rising, and thus that point is crucially late from the physical onset of a sound and leads to an incorrect cross-band association with the higher frequencies. Second, the onset track of a sound is most often not monotonically increasing, and thus we would have several local maxima in the first order difference function near the physical onset" [Kla99].

To handle these problems Klapuri began by calculating the first order difference function $D(t)$:

$$D(t) = \frac{d}{dt}A(t) \quad (4.5)$$

where $A(t)$ denotes the amplitude envelope function. $D(t)$ is set to zero where the signal is below a minimum audible field. Then the first order difference function is divided by the amplitude envelope function to get a first order *relative difference function* W , (the amount of change in relation to the signal level). This is the same as differentiating the logarithm of the amplitude envelope,

$$W(t) = \frac{d}{dt}\log A(t). \quad (4.6)$$

This $W(t)$ is then used both for detecting onset components, and determining their onset time. Referring Moore [Moo95] it is claimed that the smallest detectable change in intensity is approximately proportional to the intensity of the signal. This holds for intensities from about 20 dB to about 100 dB. Thus the onset components are detected by a simple peak picking operation, which looks for peaks above a global threshold in the relative difference function $W(t)$.

"The relative difference function effectively solves the above mentioned problems by detecting the onset times of low sounds earlier and, more importantly, by handling complicated onset tracks, since oscillations in the onset track of a sound do not matter in relative terms after its amplitude has started rising" [Kla99]. This is shown graphically in the Figure 4.14.

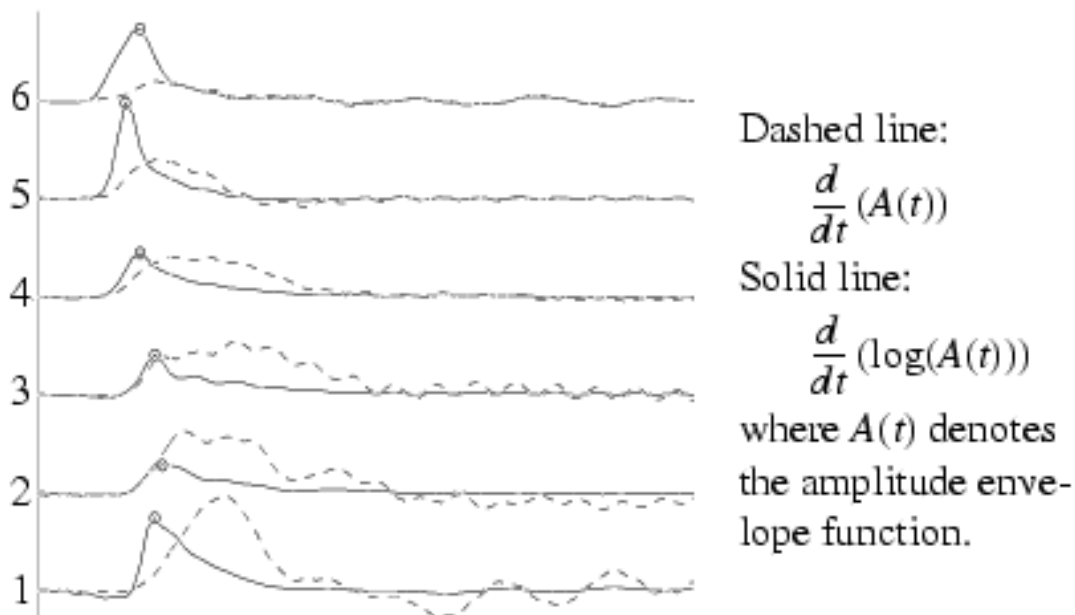


Figure 4.14: Onset of a piano sound. First order absolute (dashed) and relative (solid) difference functions of the amplitude envelopes of six different frequency bands. Image from [Kla99].

4.7.4 Intensity of onset components

To determine the intensity of an onset component the intensity is picked from the first order difference function $D(t)$ multiplied by the center frequency of the band.

Components that are closer than 50 ms to each other are dropped out based on their intensity. The one with the highest intensity remains.

4.7.5 Combining the results

The final onsets for the overall signal was calculated as follows.

First the onset components from the different bands were sorted in time order, and were regarded as actual sound onset candidates. Then each onset candidate was assigned a loudness value based on a simplified loudness model from Moore et al. [Kla99]. Doing this for each onset candidate yielded a vector of candidate loudness' as a function of their times.

Candidates under a certain threshold were dropped and candidates in the vector closer than 50 ms to a louder candidate were also dropped. Among equally loud but too close candidates the middle one (median) was picked and the others dropped. Using a simple peak picking operation on the remaining vector gave the final onset detections.

4.7.6 Conclusion

The method described above seems to be giving robust and accurate onset detections in a diverse set of real-world music samples. "Experimental results show that the presented system exhibits a significant generality in regard to the sounds and signal types involved. This was achieved without higher-level logic or a grouping of the onsets." [Kla99].

Of the 10 musical pieces tested on the system all but two were over 84% correct in their onset detections. The remaining two were symphony orchestra pieces where individual onsets often are smoothed out. Very strong amplitude modulations were also present in these pieces and this confused the system.

The onset detection method here seems even more fitted than the one described by Goto and Muraoka [GM95] for their BTS system and should be tested in this thesis. Also, the suggested measure of the correctness by Klapuri has been analysed and has inspired the measure of correctness we chose for this thesis, see Chapter 3.

4.8 Onset detection by wavelet analysis

Another approach in the same area has been done using wavelet analysis. The problem is the same, namely detecting the onset of sounds and also this system needs no prior knowledge of the input signal, and do not use higher-level logic to calculate their detections.

Crawford Tait and William Findlay uses a semitone-based wavelet analysis "to generate a time-frequency plane of modulus values, that is then transformed according to metrics derived from the study of auditory perception. The plane is then viewed as a series of vectors, and calculation of the distance between groups of vectors adjacent in time shows peaks in the distance function at onset locations. The final stage of interpretation involves detecting peaks in this function, and classifying the peak as onsets, or otherwise" [TF95].

The reason for their approach is that they felt that the existing systems were either too specialized and made generalization difficult [GM95], or that they relied on pitch-detection which might be a potentially more complex problem and rendered detection of unpitched sounds difficult. Also time-frequency decomposition of audio signals has traditionally been achieved using Fourier analysis, and "its linear division of the frequency scale does not correspond well with our perception of pitch" [TF95].

4.8.1 Wavelet analysis

The analysis is based solely on the modulus plane, and uses the harmonic wavelet analysis of Newland [New95]. It allows a semitone division of the frequency scale, which makes it suited for musical input.

The computed modulus values are mapped to dot densities and plotted against time as shown in Figure 4.15.



Figure 4.15: Modulus values from clarinet solo. Image from [TF95].

4.8.2 Transformations of the modulus plane

The modulus plane is first mapped to a logarithmic scale to enhance significant features.

A weighting is applied to each level of the scale based on the work of Stevens [Ste72]. This will compensate for the human auditory systems varying sensitivity to different frequencies.

Finally, adaptive normalization of the modulus plane is performed. This is necessary because the quieter notes in a passage tend to be dwarfed out by the louder notes [TF95].

The resulting new modulus plane can be studied in Figure 4.16.

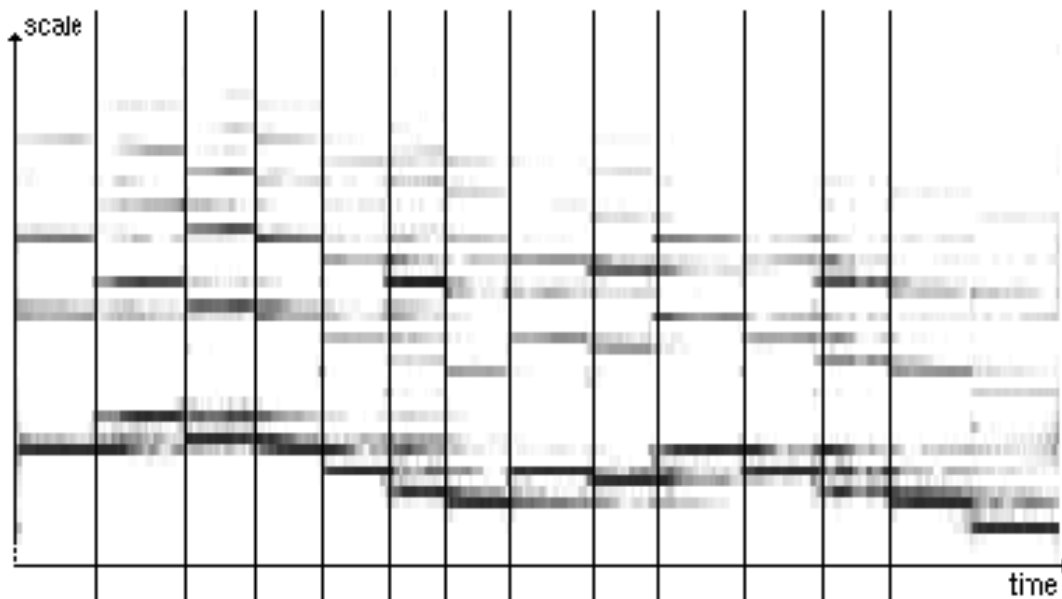


Figure 4.16: Detected onsets on modulus plane of clarinet piece. Image from [TF95].

4.8.3 Highlighting onsets

“In order to investigate time varying behavior, the plane of modulus values is divided into vectors, each constituting a slice through the plane at the highest time resolution. Vectors are compared by treating them as points in N dimensional space (if N semitone bands are being analyzed), and considering the distance between them as given by the Euclidean norm:

$$d_{ij} = \sqrt{\sum_{k=0}^{N-1} (M_{ik} - M_{jk})^2} \quad (4.7)$$

where M_{ik} is the modulus value for semitone k in the vector at time i , and semitone levels 0 to $N - 1$ at times i and j are being considered. A vector distance will thus exist if a change in frequency and/or amplitude takes place between the two point in time under consideration." [TF95].

Further, the average between two vectors in adjacent sliding windows is calculated. They represent the state of the modulus plane for a short interval, and peaks in the calculated function are evident even when only gradual changes occur between notes. The output is somewhat smoothed, but even further smoothing is applied to aid in peak detection.

"The peaks are detected by simply locating points at which series of successive increases are followed by a series of successive decreases. A small threshold is also introduced, and peaks must be separated by at least the minimum note length." [TF95].

This method demands further purging because spurious onsets are detected. Some offsets are detected as onsets, and some erroneous onsets are also detected. This is done by classifying each peak based on the behavior of the modulus plane around its location in time. The resulting onsets finally detected can be studied in the figure above. All onsets for the clarinet piece are detected except the last one.

4.8.4 Conclusion

This system has not been tested with as many real-life musical piece samples, but shows promise. "The harmonic wavelet transform provides a useful tool for analysis of all kind of sound, but more work is required in its practical application ... We feel that the onset detection method mentioned herein performs well on a variety of cases, however the classification of peaks could be improved." [TF95].

Also the fact that this method does not rely on pitch-detection (none of the methods described in this thesis does so) might possibly make it suitable for inharmonic sounds. "The benefit of not relying on pitch perception, widens the scope of possible applications considerably." [TF95]. To show this an onset detection of footsteps with increasing music in the background was performed, see Figure 4.17.

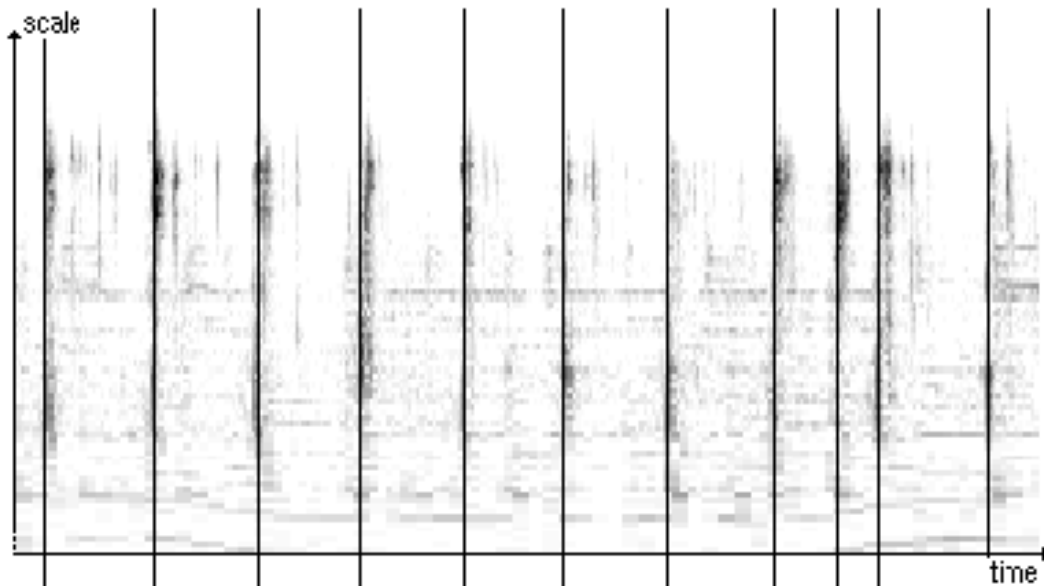


Figure 4.17: Detected onsets of footsteps with background music. Image from [TF95].

Still, this method seems to rely a bit more on experimental values for the different test cases to perform optimally. Further the smoothing performed might contribute to skewing the detected onset times as it does for STFT using Savitzky-Golay smoothing, see Chapter 6.

Chapter 5

The analysis program

5.1 System overview

The analysis system consist of two parts. We are trying to follow the MVC paradigm with a clear separation of the model, the view and the controller in the implementation. The one part is the actual analysis and processing classes, they also contain the data structure needed to store the samples, spectrograms and so on. The other part is the view and controller part. This is the graphical user interface and it is implemented in Qt.

5.1.1 System flow

The main operations in the analysis program are described in the flowchart below, Figure 5.1.

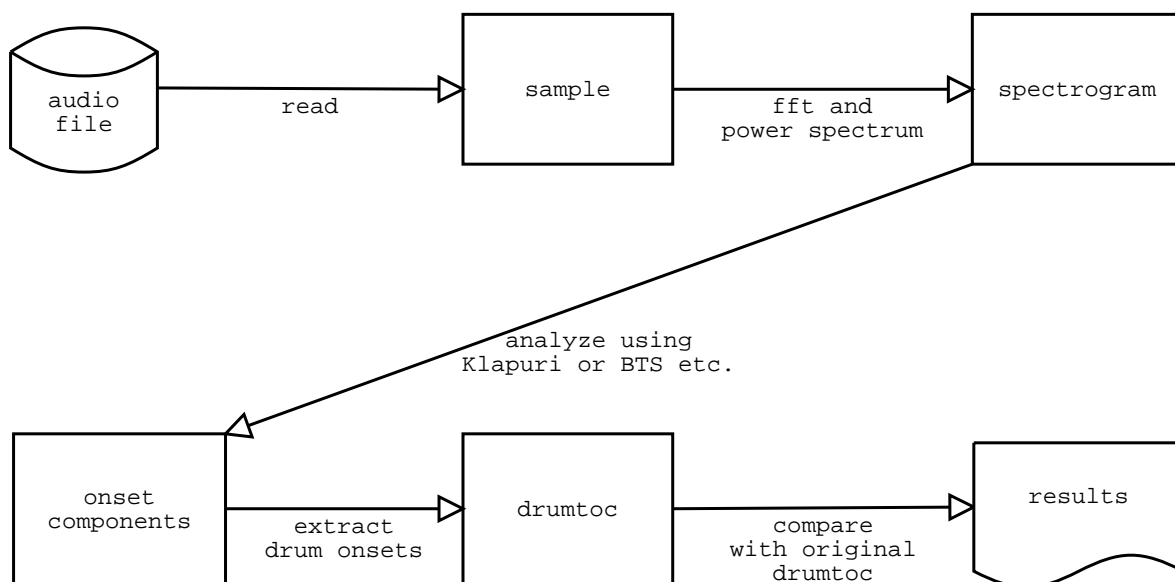


Figure 5.1: Shows the system flow of the analysis program. The input is an audiofile, the end result is text with the estimated drum onsets (the start of the drums).

An audiofile is read from the harddisk and turned into a sample. Then we apply a Fourier transformation on the sample and get a spectrogram. This spectrogram is used as the input for analysis that

try to extract onset components. From these onset components the actual drum onsets are extracted. And finally the estimated drum onsets are compared to the original ones (read from an XML file) and the result is printed on the screen. The result is the number of drums, their onsets, and a measure of correctness.

5.1.2 Class diagram

The main classes in the analysis program are presented in Figure 5.2 in a class diagram. We use UML figures and symbols.

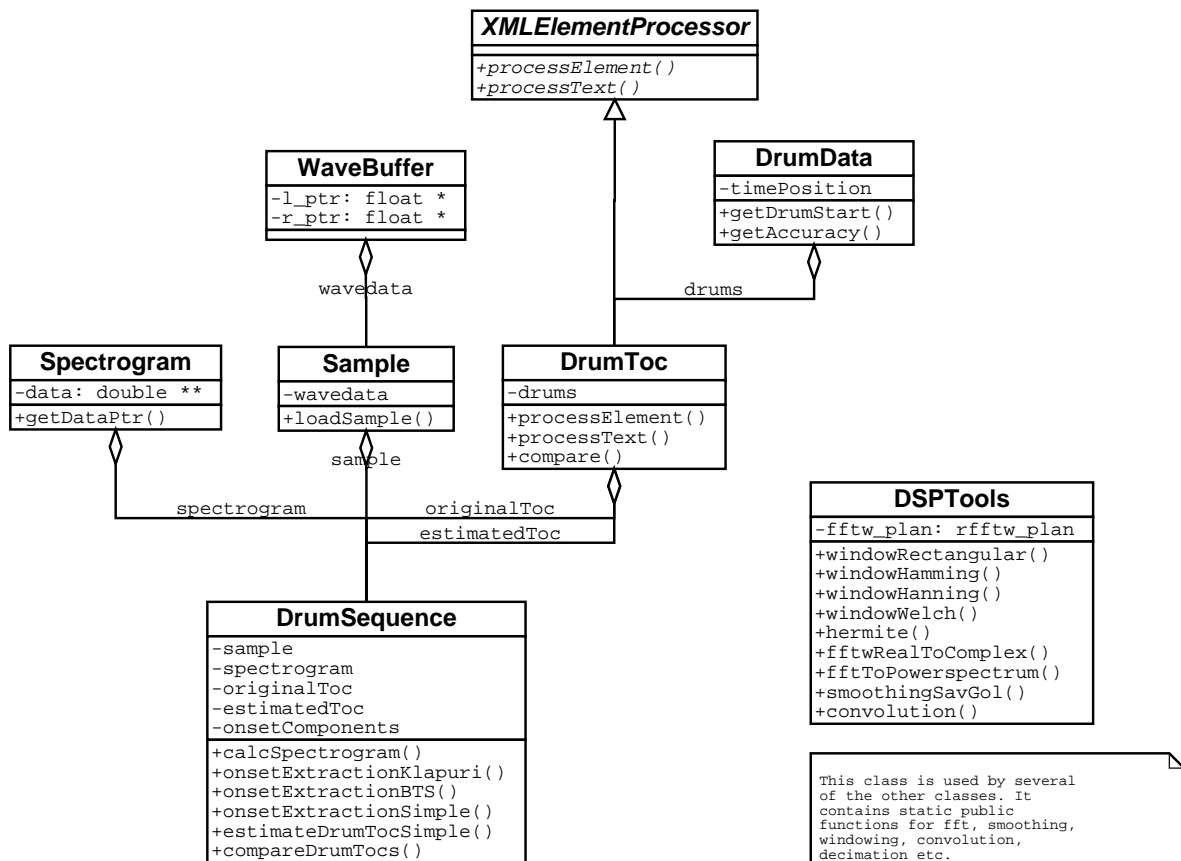


Figure 5.2: The figure shows the inheritance and collaboration of the main (digital signal processing) classes of the analysis program.

These classes take care of the different calculations and analysis performed on the audiofile.

The classes mainly concerned with the user interface work in collaboration with the “DSP” classes. In Figure 5.3 we see the relationship between the user interface and the “DSP” classes.

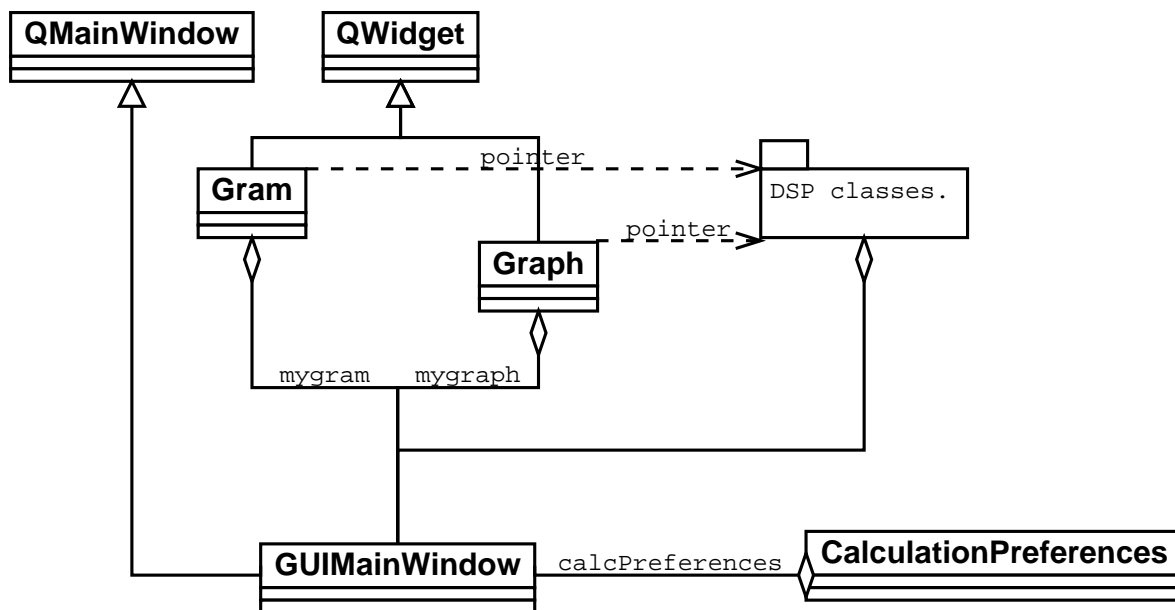


Figure 5.3: The figure shows the inheritance and collaboration of the complete program. The “DSP” classes have all been put in one package.

5.2 Key classes

Here is a brief description explaining the data structure and behavior of the key classes in the analysis program.

5.2.1 DrumSequence

The DrumSequence has a sample, can produce a spectrogram, and perform certain analysis on the spectrogram. It does the comparison of the estimated drum onsets and the original ones. GUIMainWindow uses mainly functions in DrumSequence to control the program.

5.2.2 Spectrogram

The Spectrogram has a pointer to a 2D table of power spectrum data produced by doing Fourier transformations on the Sample. It also deliver an interface for doing operations on a Spectrogram.

5.2.3 Sample

The Sample contains the actual sampled data. When the audiofile is read a Sample object is made and the sampled data stored in it.

5.2.4 DrumToc

The DrumToc is the TOC (table of contents) for an audiofile. Basically it contains a vector of drum onsets. It also offers functions for comparing drum onsets.

5.2.5 DrumData

The DrumData contains the data describing a drumsound read in by the XML file and also data extracted by the analysis. It contains the drums position in time, if the drum has been 'found' by the analysis program and with what accuracy.

5.2.6 DSPTools

The DSPTools class is a collection of DSP functions needed by several classes in the program. DSPTools uses Numerical Recipes in C and FFTW for calculations.

5.2.7 Error

Error reporting and logging need some extensions in C++. We implemented our own error class that offers basic but needed ERROR and LOG macros that, when called, will report the file and line number where the error or logging occurred. Sadly in C++ it is not easily possible to support the automatic printing of the function names the logging or errors occur in, but this would have been possible for C¹.

5.2.8 Exception

Unlike Java, C++ has no default exception handler taking care of uncaught or unexpected exceptions. We felt that this functionality was needed and implemented a standard exception class that offered this. The Exception class makes sure the programmer (or user) is notified of which exception was uncaught, and where it was thrown from. It uses much of the same functionality as the Error class.

5.2.9 GUIMainWindow

GUIMainWindow is the controlling class for the whole application. It implements QMainWindow and handles the drawing of the user interface and also the handling of the menu selections.

5.3 Using the program

Using the program is pretty straight forward. First you load a sample, then you choose your preferences for FFT and smoothing and calculate the spectrogram. If you want to you can study the power of the different frequency bands in a graph view. Finally you can extract the onsets using the Analyze menu.

¹Some C++ compilers support it but not the one used in this implementation.

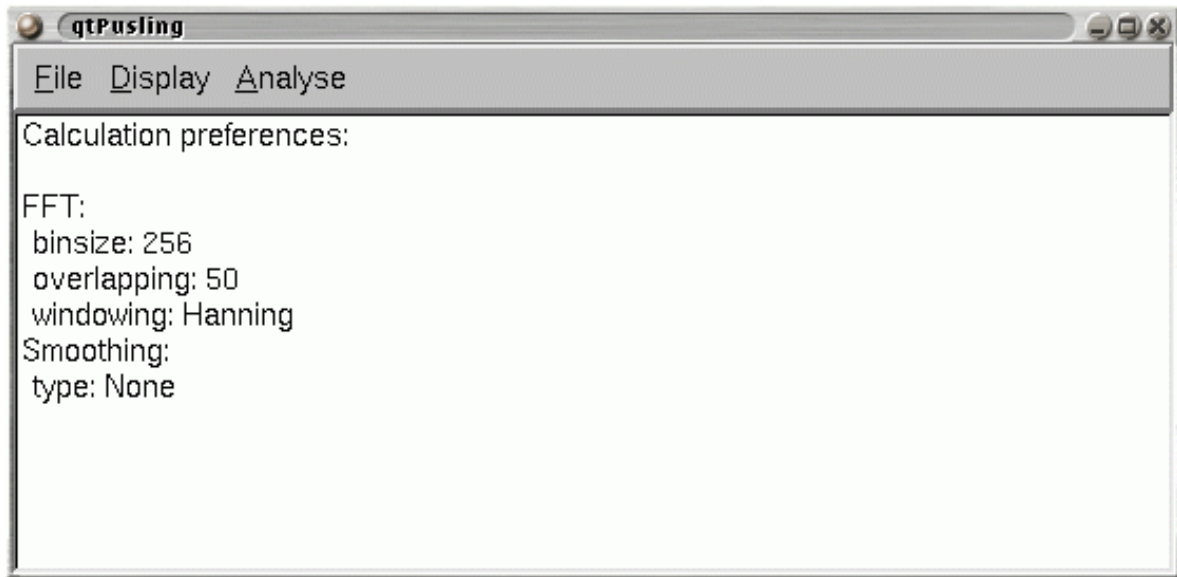


Figure 5.4: Main window, also shows the current settings for FFT and smoothing.

The main application-window in Figure 5.4 shows the current settings and also offers through the menus the different operations and visualisations available in the analysis program.

5.3.1 Loading sample

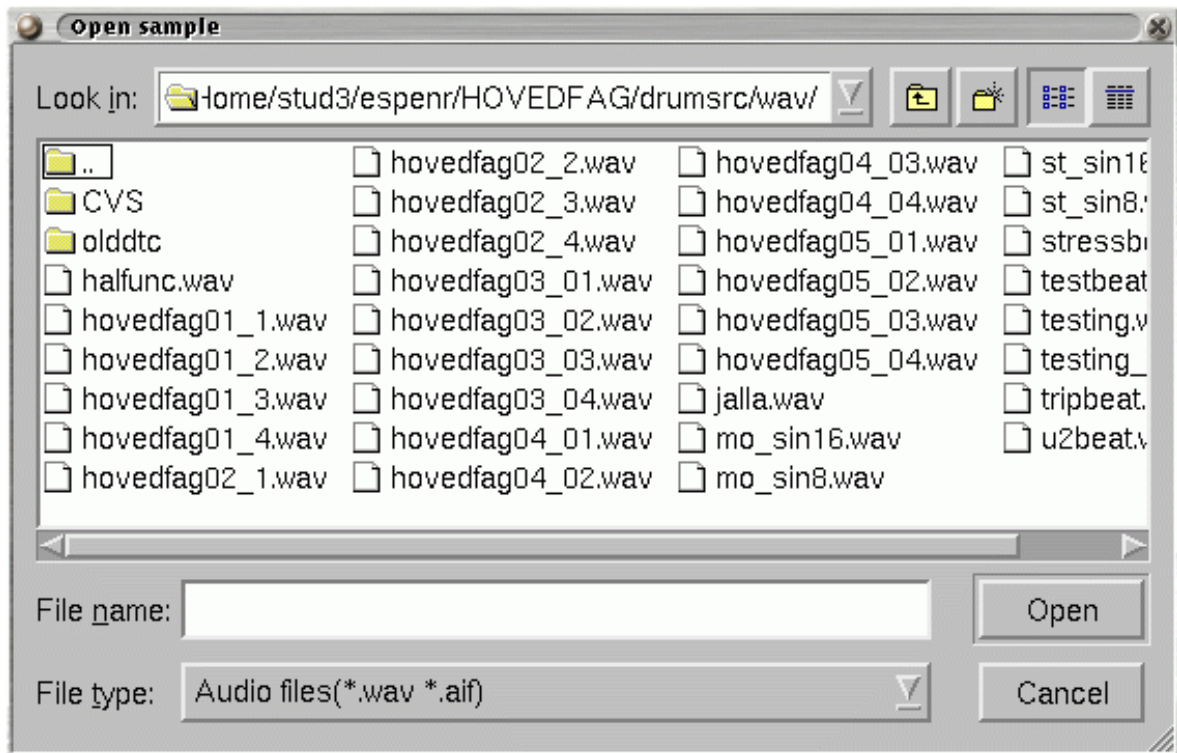


Figure 5.5: **File->Load**, loads a .wav file into the program.

The program loads stereo samples, but uses only one of the channels for analysis. At the moment only WAV files are supported. The .drc files, the TOC for the drumsequences is loaded automatically. See Figure 5.5.

5.3.2 Setting preferences

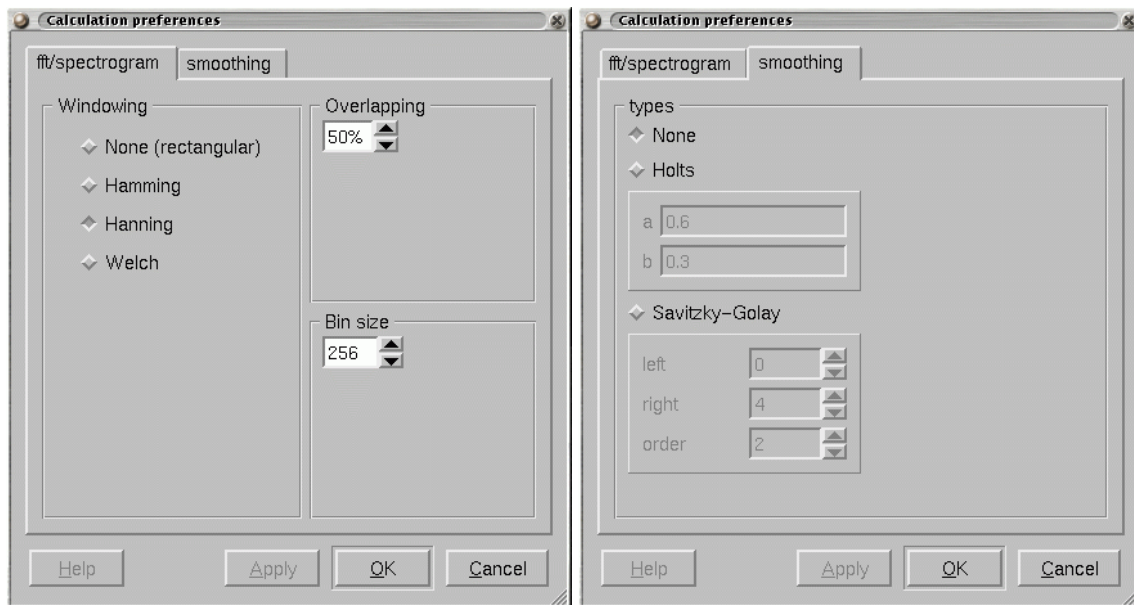


Figure 5.6: **File->FFT Preferences**, sets the preferences for FFT calculation and the optional smoothing of the frequency bands.

Figure 5.6 shows the preferences dialog. The FFT binsize ranges from 256 to 2048. But it is possible to insert arbitrary values for other binsizes. The overlapping ranges from 1 to 99 percent, so some overlapping will always be present. The smoothing values from Holts are the trend and level smoothing constants. The Savitzky-Golay values are the number of samples on the left and right side of the smoothed samplepoint. The last value is the polynomial order of the Savitzky-Golay smoothing function.

5.3.3 Calculating a spectrogram

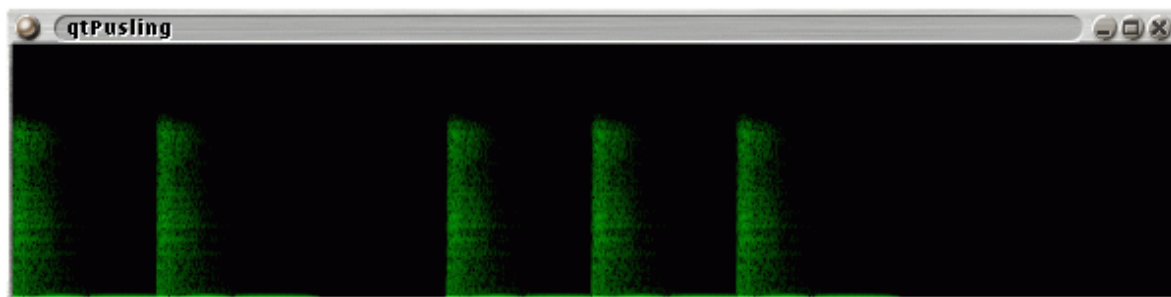


Figure 5.7: **Display->Spectrogram**, calculates a spectrogram.

The spectrogram is calculated and showed with colors representing dB values, see Figure 5.7. The image presented is a one to one correlation to the spectrogram, no scaling or transformation of the actual spectrogram data is done.

5.3.4 Viewing frequency bands

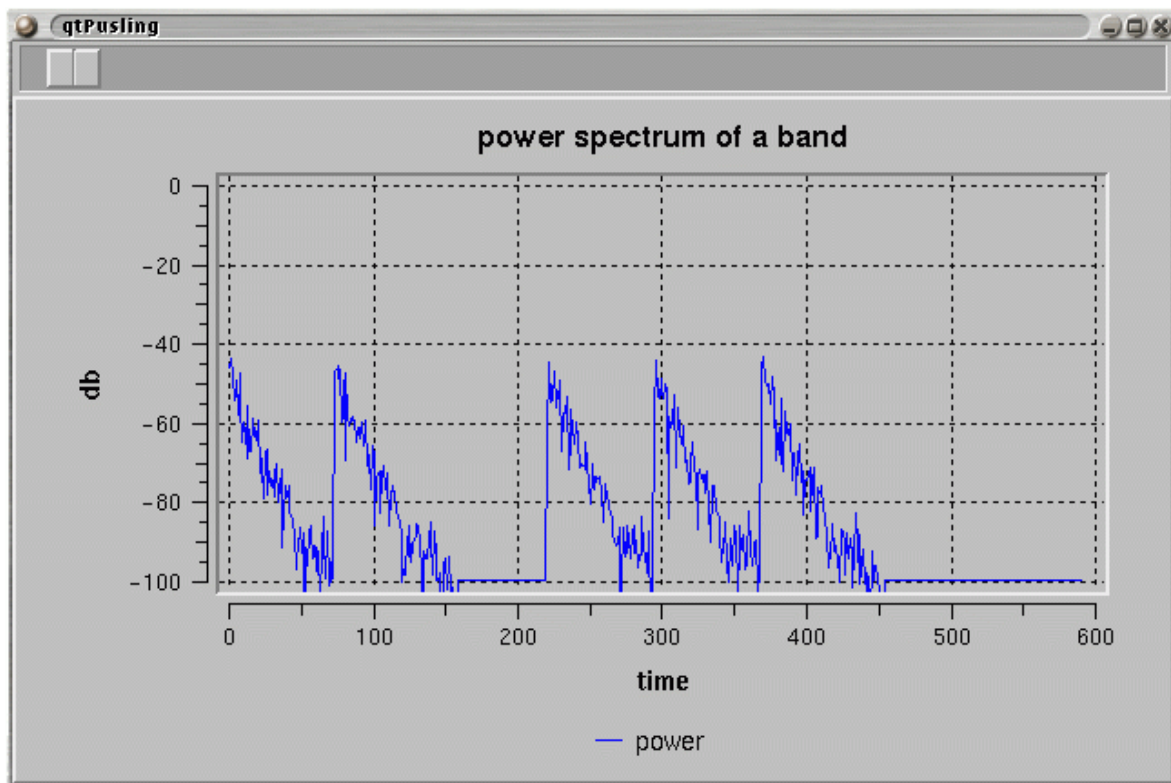


Figure 5.8: **Display->Waveform**, displays a single frequency band in the spectrogram.

To study the effect of smoothing the frequency bands it is possible to view each individual band in the spectrogram, see Figure 5.8. The slider determines which band you are viewing.

5.3.5 Calculating onsets

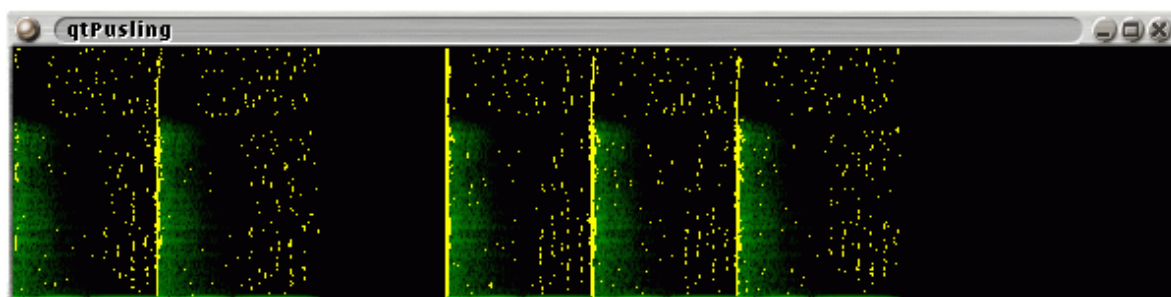


Figure 5.9: **Analyse->Find Drums**, extracted onset components.

By selecting to find drums, you start an analysis of the spectrogram, and the onsets are compared to the ones loaded from the .drc file, see Figure 5.9. The result is presented in the standard output as shown in Figure 5.10.

5.3.6 The result output

```
sample.cpp(116): LOG - loadSample(/Home/stud3/espenr/HOVEDFAG/drumsrc/wav/hovedfag01_1.wav)
drumsequence.cpp(322): LOG - found: 6
drumdata.cpp(45): LOG - Accuracy: 0.418311
drumdata.cpp(45): LOG - Accuracy: 0.511721
drumdata.cpp(45): LOG - Accuracy: 0.0471217
drumdata.cpp(45): LOG - Accuracy: 0.11365
drumdata.cpp(45): LOG - Accuracy: 0.256798
drumtoc.cpp(80): LOG - nmbr of drums in original: 5
drumtoc.cpp(81): LOG - nmbr of drums in estimate: 6
drumtoc.cpp(82): LOG - hits : 5
drumtoc.cpp(83): LOG - missed : 0
drumtoc.cpp(84): LOG - extra : 1
drumsequence.cpp(339): LOG - Drumtoc for: Roland R8
Drum start: 0.10 Detected!
Drum start: 214.40 Detected!
Drum start: 642.90 Detected!
Drum start: 857.30 Detected!
Drum start: 1071.50 Detected!
Drumtoc for: /Home/stud3/espenr/HOVEDFAG/drumsrc/wav/hovedfag01_1.wav
Drum start: 2.90 Detected!
Drum start: 211.88 Detected!
Drum start: 638.55 Detected!
Drum start: 853.33 Detected!
Drum start: 986.85
Drum start: 1068.12 Detected!
guimainwindow.cpp(178): LOG - correctness: 83.3333
```

Figure 5.10: The result output.

5.4 The programming language

5.4.1 C++

The reason for choosing C++ when making the analysis program was based on the speed and calculation requirements expected from the program. There was no need for the different algorithms to produce their onset estimates realtime, but it should be possible to relatively quickly test out different settings for the algorithms and get the results.

There was of course also a need of a relatively mature object oriented programming language so that making the needed abstractions was possible, hence languages like C was left out.

5.4.2 Java

Other programming language alternatives were considered. Java is a consistent but a bit simplified language compared to C++, but it has a huge and very usable collection of packages. At the start of the implementation there was no or little support for sound programming in Java, this has changed and might have caused the language decision to conclude with Java instead.

With JMF and JavaSound Java has actually grown into a good platform to do sound programming. There exist software synthesizers and even voice recognition systems in Java [Mic02]. But, it is an interesting note that the 'engine' of the JavaSound package is written entirely in C.

As for the speed requirement, that the calculations should terminate in a relatively short period of time, Java still might be inferior. Tests show that it is possible to make Java programs that compare or even in some cases challenge the speed of their C++ counterparts. Lutz Prechelt did a study where the same program was implemented using C, C++ and Java by several different developers. There was 24 programs written in Java, 11 in C++, and 5 in C. All the programs had the same functionality. From this study he found that "As a result, an efficient Java program may well be as efficient as (or even more efficient than) the average C or C++ program for the same purpose." [Pre99] however, he also found that "The three fastest Java programs are about twice as fast as the median C/C++ program and 10 times slower than the three fastest C/C++ programs." [Pre99]. Lutz concluded that "it is wise to train programmers to write efficient programs" (or choosing good developers) instead of focusing only on the "benchmarks" of the language.

5.5 Libraries

The decision to use C++ was also based on the vast amount of different libraries developed for it.

5.5.1 STL - template library

The C++ standard library provides a set of common classes and interfaces that greatly extend the core C++ language. At the heart of the C++ standard library, the part that influenced its overall architecture, is the *standard template library (STL)*. The STL is a generic library that provides solutions to managing collections of data with modern and efficient algorithms. Generally speaking the STL builds a new level of abstraction upon C++ [Jos99]:73.

Figure 5.11 shows the relationship between the key components in STL. The components are:

- **Containers** used to manage collections of various kinds. Typical examples are: vectors, lists, sets and maps.
- **Iterators** are used to step through, or iterate, elements in a collection of objects (usually containers). The advantage of iterators is that they offer a small but common interface for any arbitrary container.
- **Algorithms** are used to process the elements of collections. Examples are: search and sort. Since algorithms use iterators they only have to be implemented once to work for any container type [Jos99]:74.

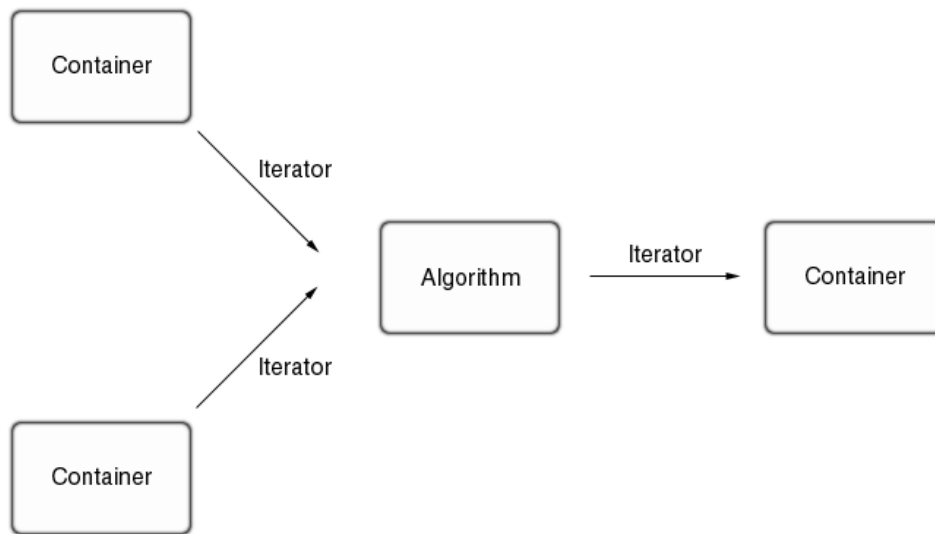


Figure 5.11: The figure shows how *Iterators* glue together *Containers* and *Algorithms*

5.5.2 FFTW - FFT library

As sonograms and calculations in the frequency domain were going to be a key element in the analysis program, an FFT library was needed. There was initially also some focus on parallel programming for the analysis program, so choosing libraries supporting this would be beneficial.

FFTW (the Fastest Fourier Transformation in the West) was developed at MIT by Matteo Frigo and Steven G. Johnson. “FFTW is a C subroutine library for computing the Discrete Fourier Transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. Moreover, FFTW’s performance is portable: the program will perform well on most architectures without modification.” [FJ01].

FFTW supports three ways of usage in a parallel environment:

1. **Threading** “Any program using FFTW can be trivially modified to use the multi-threaded routines. This code can use any common threads implementation, including POSIX threads.”
2. **MPI** “[FFTW] contains multi-dimensional transforms of real and complex data for parallel machines supporting MPI. It also includes parallel one-dimensional transforms for complex data.”
3. **Cilk** “FFTW also has an experimental parallel implementation written in Cilk, a C-like parallel language developed at MIT and currently available for several SMP platforms.” [FJ01].

Two other reasons for choosing FFTW were the thorough documentation and the benchmarks for the transformations on various platforms. We only do real one-dimensional transforms in the analysis program, so this was the most interesting part of the benchmarks.

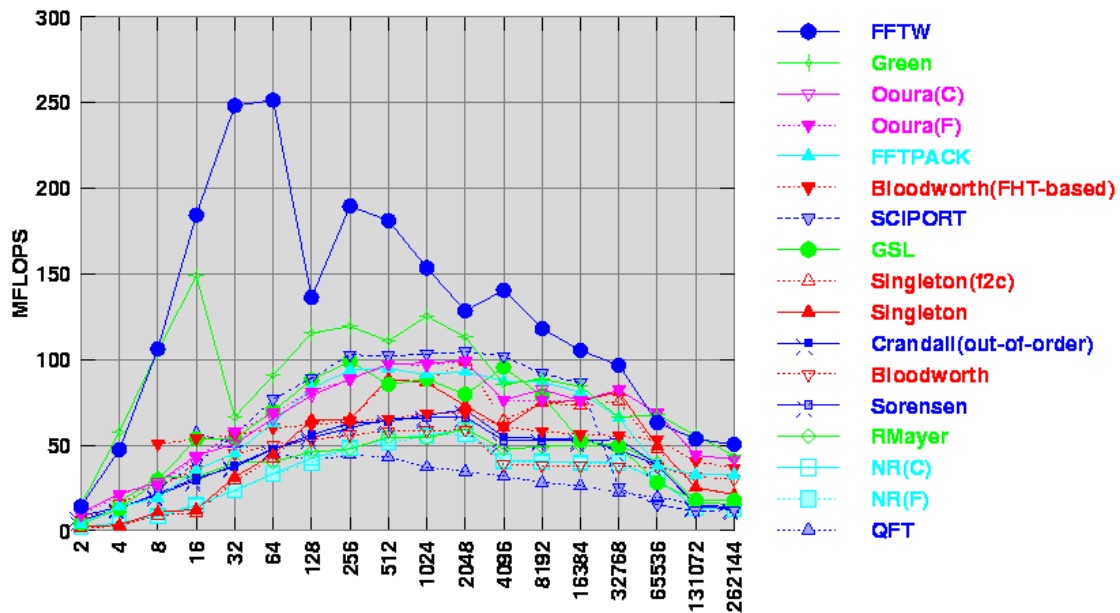


Figure 5.12: **1D Real Transforms, Powers of Two** UltraSPARC I 167MHz, SunOS 5.6 Image from www.fftw.org.

As we can see from Figure 5.12 FFTW is equal to or above most of the other libraries. The situation is the same for other platforms, and also for complex or real multi-dimensional transforms. “We have compared many C and Fortran implementations of the DFT on several machines, and our experiments show that FFTW typically yields significantly better performance than all other publicly available DFT software. More interestingly, while retaining complete portability, FFTW is competitive with or faster than proprietary codes such as Suns Performance Library and IBM’s ESSL library that are highly tuned for a single machine.” [FJ98].

5.5.3 expat - XML parser toolkit

For parsing XML *expat* was chosen because of its size and speed. *expat* is the underlying XML parser for the open source Mozilla project, perl’s XML::Parser, and other open-source XML parsers. The C library was written by James Clark, who also made *groff* (an nroff look-alike), *Jade* (an implementation of ISO’s DSSSL stylesheet language for SGML), *XP* (a Java XML parser package), and *XT* (a Java XSL engine).

In an article by Clark Cooper [Coo99] *expat* was compared, see Figure 5.13, to other XML libraries written in both C, perl, Python and Java.

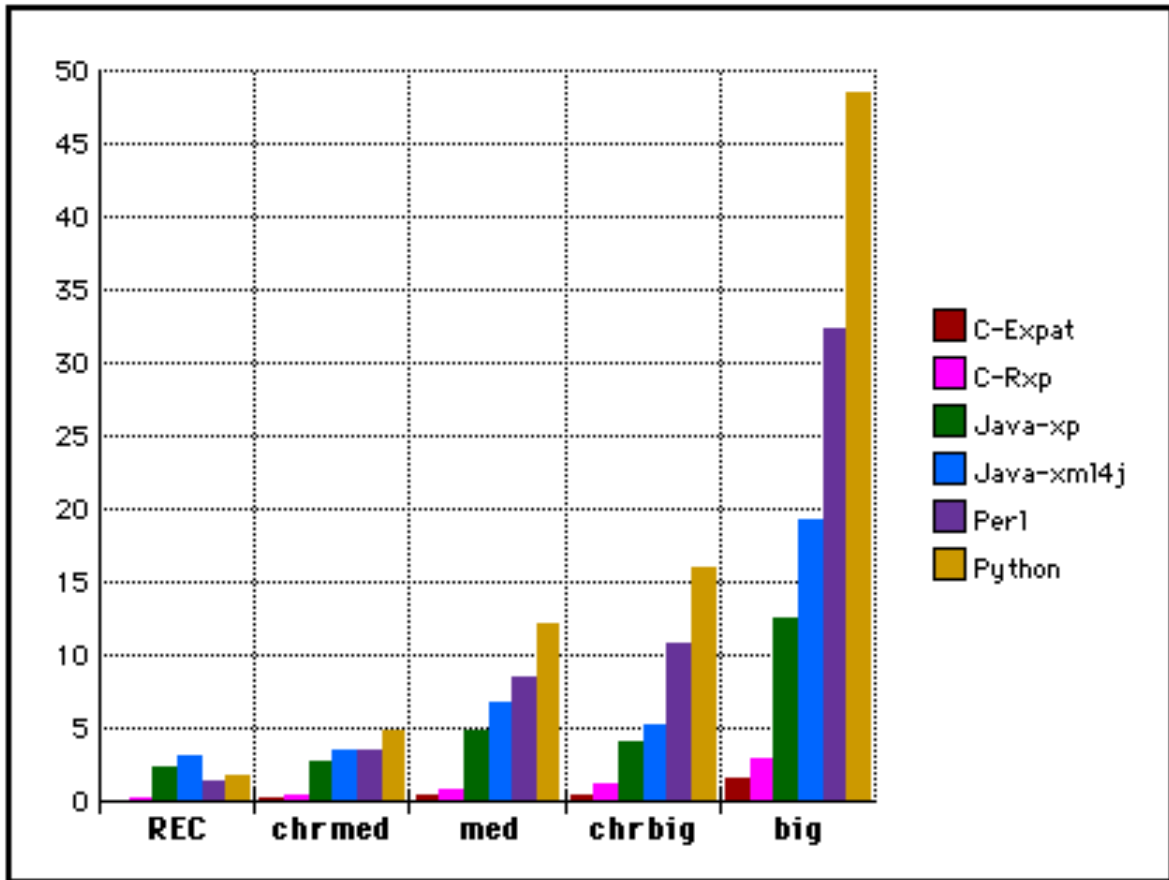


Figure 5.13: Comparison of six XML parsers processing each test file. Image from [Coo99].

rec.xml is the file used for benchmarking, it is about 160K in size. The med.xml is 6 times the size of rec.xml and big.xml is 32 times that size. Literally the the rec-file is repeated 6 and 32 times in these files. chrmed.xml and chrbig.xml contain just the text contents of rec.xml repeated 6 and 32 times.

“There aren’t really many surprises. The C parsers (especially Expat) are very fast, the script-language parsers are slow, and the Java parsers occupy a middleground for larger documents.” [Coo99].

For our analysis program the XML files are usually very small, so this focus on execution performance as the only feature might be a bit misplaced. The expat library has a limited functionality and is probably harder to use in a comparison to the other libraries programming-wise. “These tests only measure execution performance. Note that sometimes programmer performance is more important than parser performance. I have no numbers, but I can report that for ease of implementation, the Perl and Python programs were easiest to write, the Java programs less so, and the C programs were the most difficult.” [Coo99].

5.5.4 Numerical Recipes in C - math functions

Numerical Recipes in C is a book and library explaining and implementing numerical computing algorithms in C². For the analysis program several different numerical functions were needed, and

²There also exist Fortran versions.

to use a library that was well tested and well documented would greatly ease the implementation. Numerical Recipes specifically has functions for smoothing data (Savitzky-Golay Smoothing Filters), convolution using FFT and the wavelet transform which all were used in our program. “Our aim in writing the original edition of Numerical Recipes was to provide a book that combined general discussion, analytical mathematics, algorithmics, and actual working programs.” [PTVF92]. And for the second edition the goal remained the same but also not to make it grow too much in size. Reading chapters in the book and using parts of the library the general feeling is that this blend of discussions, example code, mathematics and reference to the API does not work. To provide hints and understanding about certain numerical algorithms it does its job, but as a reference to a programmer it fails.

5.5.5 Qt - GUI toolkit

Qt is a cross-platform C++ GUI application framework. It provides application developers with all the functionality needed to build graphical user interfaces. Qt is fully object-oriented, easily extensible, and allows true component programming.

The key modules available in Qt³ are:

1. **Tools** platform-independent Non-GUI API for I/O, encodings, containers, strings, time & date, and regular expressions.
2. **Kernel** platform-independent GUI API, a complete window-system API.
3. **Widgets** portable GUI controls.
4. **Dialogs** ready-made common dialogs for selection of colors, files, etc.
5. **Network** advanced socket and server-socket handling plus asynchronous DNS lookup.
6. **XML** a well-formed XML parser with SAX interface plus an implementation of the DOM Level 1.

The analysis program needed to visualize the samples, their waveforms, and the drum onsets. It needs to give the user understandable information and provide him with controls to easily change the settings of the algorithms to test out new calculations.

Qt is well tested and offers a GUI designer, Qt Designer, to easily design the layouts needed in a program. Qt is also very well documented and has an active user community. Qt has also been used by companies like AT&T, NASA and IBM. [Tea01].

5.5.6 QWT - graph visualization package

Qt provides no scientific visualization widgets. There are no graph or histogram widgets, so these have to be implemented on top of Qt.

QWT (Qt Widgets for Technical Applications) is an extension to Qt that provides widget and components that can display multiple graphs in a 2D plot.

QWT was used in visualizing waveforms and spectral data returned from FFT and power spectrum calculations. Overall QWT worked nicely but it is not mature enough to satisfy all the needs as a package for scientific visualization.

³Qt ver. 2.3.1 and later 3.0 was used for the implementation.

5.6 Tools

Several tools were used in the development of the analysis program. Of these *Doxygen*, *CVS* and *tmake* gave the most benefits.

5.6.1 Doxygen - documentation system

Doxygen[vH01] is a documentation system for C++, IDL (Corba, Microsoft and KDE-DCOP flavors) and C. Doxygen is developed on Linux, but is set-up to be highly portable. Doxygen is distributed under the terms of the GNU General Public License.

Doxygen offers these features:

1. **Generate** an on-line documentation browser (in HTML) and/or an off-line reference manual (in \LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), Postscript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.
2. **Extract** the code structure from undocumented source files. This can be very useful in order to quickly find your way in large source distributions. The relations between the various elements are visualized by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

In addition to this, Doxygen also indexes all the classes, functions and members, and can perform searches through a web frontend that finds these.

For the analysis program both the search and the automatic generation of dependency graphs, inheritance diagrams and collaboration diagrams helped in the development. As the analysis program was developed "on and off" during the studies, these automatically generated documents and graphs helped to get faster back into the correct mindset.

One of the allowed documentation syntaxes in Doxygen is the same as the one used in Javadoc⁴. This also helped and encouraged the use of Doxygen as previous experiences with Javadoc had been similarly rewarding, and we had already learned the syntax used in Javadoc.

5.6.2 CVS - Concurrent Version control System

CVS is an acronym for the *Concurrent Versions System* and was developed by Brian Berliner⁵.

CVS is a "Source Control" or "Revision Control" tool designed to keep track of source changes made by a developer or groups of developers working on the same files, allowing them to stay in sync with each other as each individual chooses.

CVS keep track of collections of files in a shared directory called "The Repository". Each collection of files can be given a module name, which is used to checkout that collection. After checkout, files can be modified, then committed back into the repository and compared against earlier revisions. Collections of files can be tagged with a symbolic name for later retrieval. You can add new files, remove files you no longer want, ask for information about sets of files, produce patch "diffs" from a base revision and merge the committed changes of other developers into your working files.

⁴Javadoc is the tool from Sun Microsystems for generating API documentation in HTML format from doc comments in source code[Mic01].

⁵Other contributors are: Jeff Polk, David D. Zuhn and Jim Kingdon.

Even for a single developer CVS is very useful as it enables development on different clients. The source repository is available, for example using SSH, and as long as the developer keeps committing his changes a fresh repository can always be checked out. Another positive aspect of using revision control in general is that the developer grows bolder in his refactoring of code. Since the previous versions of the source code are easily available, the fear of losing working code disappears.

5.6.3 ElectricFence - memory debugging

C++ has no inherit bounds checking in the language. This causes a lot of actual errors in C++ programs to go unnoticed until they finally manifest themselves into spurious crashes and strange errors in arbitrary operations during execution of the program.

Electric Fence helps detect two common programming bugs: software that overruns the boundaries of a `malloc()` memory allocation, and software that touches a memory allocation that has been released by `free()`. Unlike other `malloc()` debuggers, Electric Fence will detect read accesses as well as writes, and it will pinpoint the exact instruction that causes an error. It has been in use at Pixar⁶ since 1987, and at many other sites for years.

Electric Fence uses the virtual memory hardware of the computer to place an inaccessible memory page immediately after (or before, at the user's option) each memory allocation. When software reads or writes this inaccessible page, the hardware issues a segmentation fault, stopping the program at the offending instruction. It is then trivial to find the erroneous statement in a debugger. In a similar manner, memory that has been released by `free()` is made inaccessible, and any code that touches it will get a segmentation fault.

By simply linking the analysis program with Electric Fence library were able to detect most `malloc` buffer overruns and accesses of free memory. As `new` and `delete` (at least in our development environment) just wrapped `malloc` and `free`, Electric Fence was sufficient even for C++ development.

The only drawback with using Electric Fence is the huge leap in memory usage while running the program. A doubling in memory use was not uncommon, and since both the analysis part and the graphical user interface already are relatively memory dependent this caused a performance hit.

Still the usage of Electric Fence enabled us to find relatively intricate errors, and by fixing these to make the program much more stable.

5.6.4 Qt tools

In addition to their application framework Qt offers a set of tools that helps the development of graphical user interfaces, and applications in general.

5.6.4.1 `tmake`

`make` and `makefiles` are a wonderful tool for automating the process of building and compiling programs. None the less to write a `makefile` for a non-trivial project is not easy.

`tmake` is a `makefile` generator that creates and maintains `makefiles` for software projects, and also simplifies the task of making `makefiles` for cross platform development. As an added bonus `tmake` also simplifies the task of compiling Qt classes and generating user interfaces from the Qt Designer XML files, which would need special treatment in traditional `makefiles`.

⁶Well known animation company.

This tool helped a lot in the development of the analysis program. It makes it possible for the developers to focus on the actual development instead of menial tasks such as setting up proper makefiles.

Thus the meta-makefiles one has to maintain with tmake is smaller and more to the point. It defines the header and source files used in the program and also which libraries to link with, and what include paths to use. In addition one can specify the various UI (XML Qt Designer) files used.

Figure 5.14 shows the actual tmake project file used for the analysis program ⁷:

```
HEADERS = drumsequence.H drumsequenceexception.H dsptools.H gram.H
graph.H guidr umdialog.H guimainwindow.H guispectrogramdialog.H
sample.H sampleexception.H spectrogram.H wavebuffer.H drumdata.H
guicalculationpreferencesimpl.h calculationpreferences.H
onsetcomponent.H xmlelementprocessor.H xmlelement.H xmlprocessorcrea
tor.H xmlparser.H drumtoc.H error.H exception.H

SOURCES = drumsequence.cpp dsptools.cpp gram.cpp graph.cpp
guidrumdialog.cpp gui mainwindow.cpp guispectrogramdialog.cpp
sample.cpp spectrogram.cpp wavebuffer.cpp drumdata.cpp
guicalculationpreferencesimpl.cpp calculationpreferences.cpp onse
tcomponent.cpp xmlparser.cpp xmlprocessorcreator.cpp drumtoc.cpp
error.cpp exception.cpp

TARGET = qtPusling

INTERFACES = guicalculationpreferences.ui

CONFIG = qt debug warn_on

INCLUDEPATH =
$HOME/include/recipes_c-ansi;$HOME/libs/fftw/include;$HOME/i
nclude;$QWTDIR/include;

unix:TMAKE_CFLAGS = -DNRANSI
unix:TMAKE_CXXFLAGS = -DNRANSI
unix:LIBS = -L$HOME/libs/fftw/lib -L$HOME/lib -lfftw -lfftw -lqwt
-lrecipes_c -lexpat
```

Figure 5.14: the projects “meta” makefile.

As we can see there are no make rules or dependencies defined, all we have listed are the header and source files, the name of the target, the interface used, some parameters specifying the build of the target and finally the includepath, compiler flags and which libraries to link with.

tmake supports several compilers and operating-systems. This makes it possible to use the exact same project file, or “meta-makefile”, on different platforms.

5.6.4.2 Designer

Implementing the layout and presentation logic of a graphical userinterface can be tedious and slow if done by hand. Placement and alignment of widgets are not easily perceived when looking at source

⁷This snapshot was from late 2001.

code. A much faster and intuitive way is to “draw” the graphical userinterface using an application that offers the different widgets as “brushes” much in the same as any normal drawing application. With this tool one can also to some extent define the interaction and behavior in the graphical userinterface.

Qt Designer is a tool that offers the ability to define the layout and the interaction of a graphical userinterface. With similar tools we often felt one loses the overview and control of what is really happening in the code. Qt Designer uses a very simple mechanism in any object-oriented language, inheritance, to keep the code clean and understandable. Qt Designer generates a class A which includes the defined layouts and eventhandlers. Eventhandlers in class A that are not part of the Qt classes used in the layout gets implemented in A as skeleton functions. The developer then implements a class B that inherits A and which overrides the skeleton functions and implement actual behavior in them. This way new versions of A can be generated and unless they change the interface of the class (like adding or changing eventhandlers) B can stay the same. ⁸

5.7 Experiences

In the development of the analysis program we have made some experiences that might be interesting to include. Although we have reused several libraries and tools in the development of the program an even greater use of this would have been beneficial.

5.7.1 Soundfile library

For the analysis program we wrote our own soundfile loader. Importing soundfiles should have been done reusing other developers code. This way the analysis program would have supported more formats. Bit manipulation is often needed when importing various formats as some use big-endian others use little-endian. As an example Microsoft WAV format use little-endian for their short and long and Apple/SGI AIFF use big-endian. To support these formats and make the code portable using an already mature soundfile-library would be better.

5.7.2 DSP library

For writing graphical userinterfaces there exist a multitude of toolkits and frameworks. The same applies to 3D graphical programming. It is not so for audio or DSP programming and it is in our opinion badly needed. A library providing classes for samples, filters, readers/writers, recording, playback, FFT and wavelets would greatly cut the development time and leave more room for experimentation concerning the algorithms and methods one wants to have tested.

5.7.2.1 SPUC - Signal Processing using C++ library classes

The library SPUC seems to be a step in the right direction. “The objective of SPUC is to provide the Communications Systems Designer or DSP Algorithm designer with simple, efficient and reusable DSP building block objects. Thus allowing a transition from System design to implementation in either programmable DSP chips or hardwired DSP logic.” [KK02]. Still it lacks important transformations such as FFT and wavelets and there is no functionality for reading/writing soundfiles or playback.

⁸This is of course only partially true. Changing parts of the internal numbering of radiobuttons for example, might produce strange results if the inherited class was not also updated to reflect the changes.

The way we decided to develop this analysis program was to choose specialized libraries and combine them to deliver the functionality we needed. This was successful to a certain degree but also time-consuming.

5.7.3 Scientific visualization library

An important part of any experimentation is to present the experiment results in an understandable format. For waveforms and power spectrums presenting them graphically makes the result highly human-readable.

As with DSP libraries, there is also a lack of alternatives for scientific visualization libraries. To our knowledge there exist no mature library that focus on scientific visualization for C++. There exist a lot of GUI widget libraries (Qt, GTK, wxWindows) but no library that on top of these develop widgets for showing graphs, histograms, 3D figures, spectrograms etc.

5.7.4 Garbage collector

A garbage collector is a part of a language's runtime system, or part of the program as an add-on library, that automatically determines what memory a program is no longer using, and recycles it for other use. Garbage collection is not part of the C++ language, but there are several garbage collection libraries available. The greatest benefits when using garbage collection are:

- Development effort: memory management is (programmer-)time consuming and error prone.
- Reusability: since memory management is handled automatically programmers do not have to restrict and specialize their designs for the task at hand.
- Functionality: In general, it is considerably easier to implement and use sophisticated data structures in the presence of a garbage collector.
- Safety: it is possible to trick the garbage collector into not missing parts or failing, but not likely. Hans-J. Boehm (one of the makers of the Boehm-Demers-Weiser conservative garbage collector) puts it this way "In our experience, the only examples we have found of a failure with the current collector, even in multi-threaded code, were contrived." [Boe93].
- Debugging: A garbage collector can eliminate premature deallocation errors. Also, memory leaks induced by failing to deallocate unreferenced memory may be hard to trace and eliminate. Garbage collectors automatically eliminate such leaks.

There are disadvantages with garbage collection as well. The main issues are an increase in memory usage, increase in CPU load, higher latency in an application and paging locality (issues with virtual memory system). Still these are just potential performance hits and problems, and need not be experienced at all. [Boe93].

By using ElectricFence in parts of the development of the analysis program we were able to experience some of the benefits when using a garbage collector, namely leak detection. This was a positive experience but we were not satisfied with the increase in memory usage and, as result of that, increased latency that this caused. By deciding to use a garbage collector we think the application would have become more stable and, because of less time spent in debugging, the analysis program could have included a larger amount of algorithms.

The Boehm-Demers-Weiser conservative garbage collector [Boe02] can be used as a garbage collecting replacement for C malloc or C++ new. The collector is not completely portable, but the

distribution includes ports to most standard PC and UNIX platforms. This garbage collector seems as a logical choice if we were to use a garbage collector in C++. Among the traditional garbage collection algorithms the Boehm-Demers-Weiser conservative garbage collector uses a mark-sweep algorithm. It provides incremental and generational collection under operating systems which provide the right kind of virtual memory support. [Boe02].

Chapter 6

Results

6.1 Onset extraction

As we suggested in the introduction, finding onsets for drumsounds in a drumsequence is not a trivial problem. Most research in this area has been for the purpose of beat tracking [Sch98] [GM95] [AD90]. In beat tracking the onsets are used as low level information that is interpreted and transformed into higher level information (typically meter, time signature, tempo etc.) and therefore finding all onsets in a drumsequence is not paramount.

However in our analysis system we needed the best onsets extraction algorithm available as it is the basis for developing the kind of composition tool we wanted. Using the same algorithm to detect *onset components* as in the BTS system [GM95], described in Chapter 4, and using our own algorithm to combine them and extract onsets, we were able to develop a system that produced encouraging results.

Before we decided to use this onset extraction algorithm several other approaches were explored. Chapter 4 describes both time-domain and time-frequency domain algorithms.

6.1.1 Time-domain

RMS/MAX analysis on the waveform of the sample use only the amplitude of the sample to try to determine if an onset has occurred. This kind of analysis soon proved to be insensitive to weak drumsounds being overshadowed by neighbouring strong onsets. Another drawback about time-domain analysis is that since there is no frequency information available, handling multiple onsets seems difficult.

6.1.2 Time-frequency domain

Based on our experiences from time-domain analysis we decided that operating in the time-frequency domain would better solve the problem of onset extraction. The reason for this was that subtleties in the drumsequence would not be easily discovered with methods that ignore frequency and only consider the amplitude.

6.1.2.1 Frequency-band decibel-threshold analysis

As explained in Chapter 4, the *frequency-band decibel-threshold analysis* looks at variations in the decibel of each band a sonogram produces. If the current decibel value leaps over a certain threshold compared to the previous decibel, this is registered as a possible hit. If enough hits are registered at the

same time in different bands, a drum onset is found. Refer to Figure 4.2 for a pseudo implementation of the analysis.

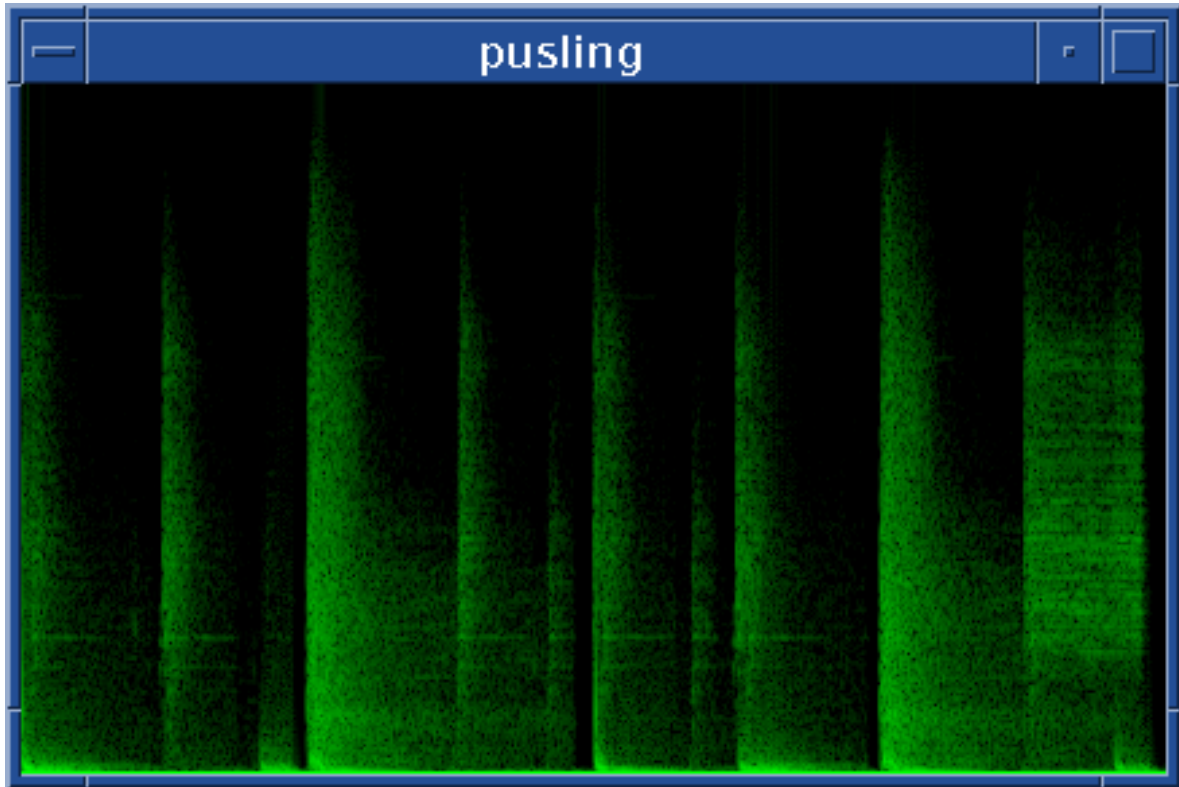


Figure 6.1: Sonogram of a drumsequence.

In Figure 6.1 the sonogram was calculated using a Hamming window, the FFT binsize was 512, overlapping 50%. The ideal interpretation of this sonogram from the analysis would be to detect only the distinct vertical "lines" in the sonogram. These lines show a change of signal power in most of the frequency-bands which, with the broad frequency spectrum drumsounds usually have, would mean a drum was hit. Figure 6.2 shows the output from the onset extraction using the frequency-band decibel-threshold analysis.

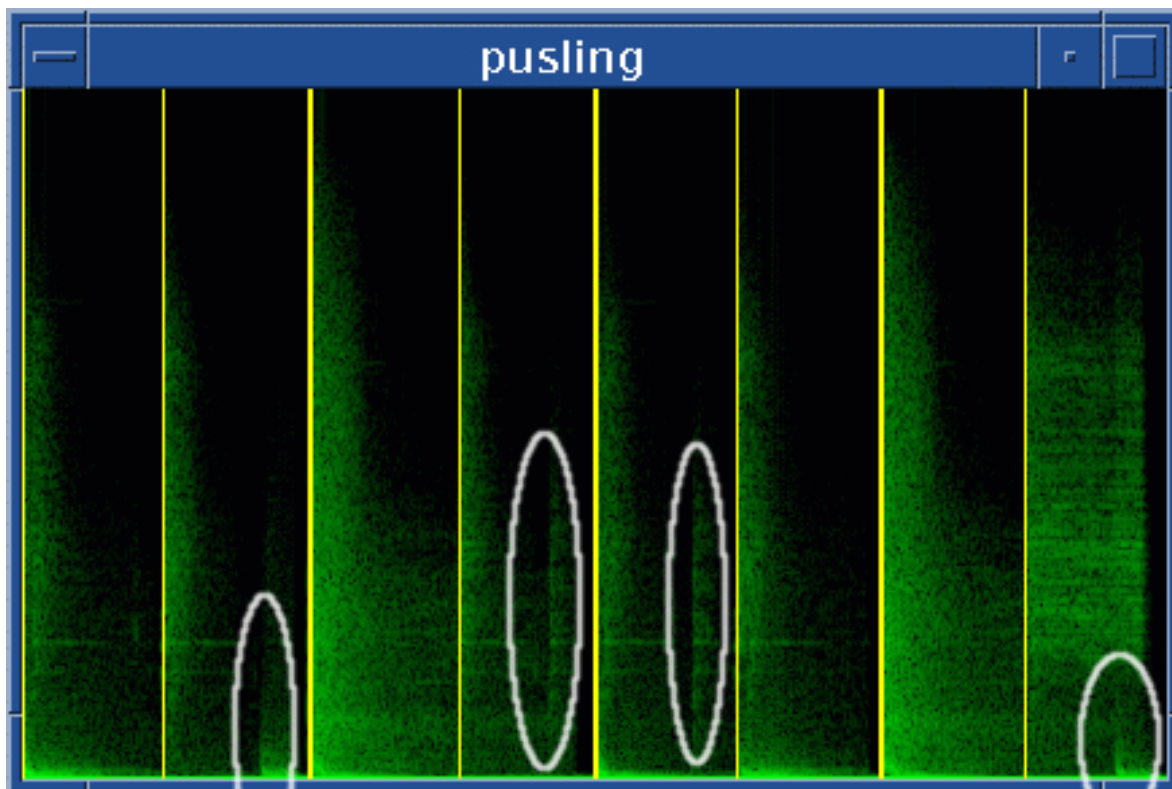


Figure 6.2: Results from the analysis. The yellow lines are detected drum onsets. The circles are drum onsets that went undetected by the algorithm.

The yellow vertical lines are the output from the system. For this specific test we were using a *dBThreshold* (everything below is ignored) of -54 dB, the *dBClimbThreshold* (the amount of change in power in a band) was set to 20 dB and for an onset to be detected there had to be 5% hits (i.e. at least 5% of the frequency-bands had to have an onset component at that specific time).

Studying Figure 6.2 we see that some of the yellow lines are wider than others. This is because there are actually 2 or 3 lines adjacent to each other because of bad detection by the algorithm (in other words, the algorithm thinks it has detected more than one drum). The algorithm also fails to detect some drums altogether (look at the drawn circles above).

What this qualitative study shows is that the algorithm has problems detecting weaker drumsounds (the circled areas) and correctly handle the drumsounds it does detect (the multiple detections of the same drumsounds).

Figure 6.2 is the best 'clean' result we were able to achieve with the frequency-band decibel-threshold analysis. By lowering the climb threshold we should be able to detected the undetected drums too, but this would result in an even greater number of double/triple detections of already detected drums.

One of the reasons for the *dBClimbThreshold* to be so high, is because of the fluctuation in the decibel values in the frequency-bands. The DFT does not as one would hope, calculate the phase and amplitude for a *broad* band around the frequency the bin is centered about, but only a narrow band, this is discussed in Chapter 2 (and this is the same effect that causes the scalloping loss). Hence, the decibel calculated for the different bands will fluctuate because the frequencies in a drumsound are

not stationary¹. Thus, when using a low climb-threshold, the fluctuations were interpreted as possible onset components.

Figure 6.3 gives an example of the fluctuations in the bands by showing the third frequency-band (around 260 Hz) in the sonogram. The graph is far from smooth and we understand how algorithms can be mistaken when only considering the signal power from one time-position compared to another.

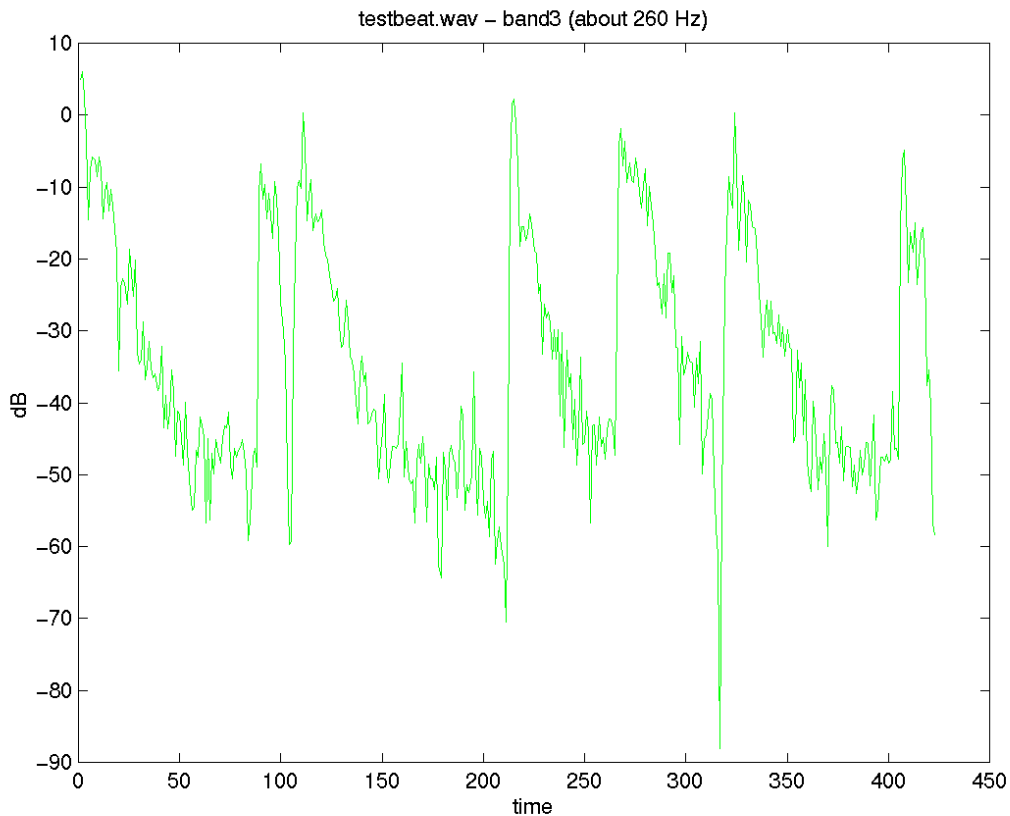


Figure 6.3: Shows the fluctuations of the powerspectrum in a sonogram. The figure shows band 3, around 260 Hz.

6.1.2.2 Smoothing the FFT output

The problem with the fluctuations in the frequency-bands lead us to investigate ways of smoothing the data without distorting it too much. The important properties we want to preserve is the height and width of the peaks and their position.

A basic moving average filter² will not suffice because "a narrow spectral line has its height reduced and its width increased" [PTVF9x]:650.

One possible smoothing method could be *linear exponential smoothing* (Holt's method) shown in Figure 6.4.

¹A signal whose statistical properties do not change over time.

²Filter that calculates the mean of small parts of the signal, example: $y(n) = (x(n-1) + x(n) + x(n+1))/3$.

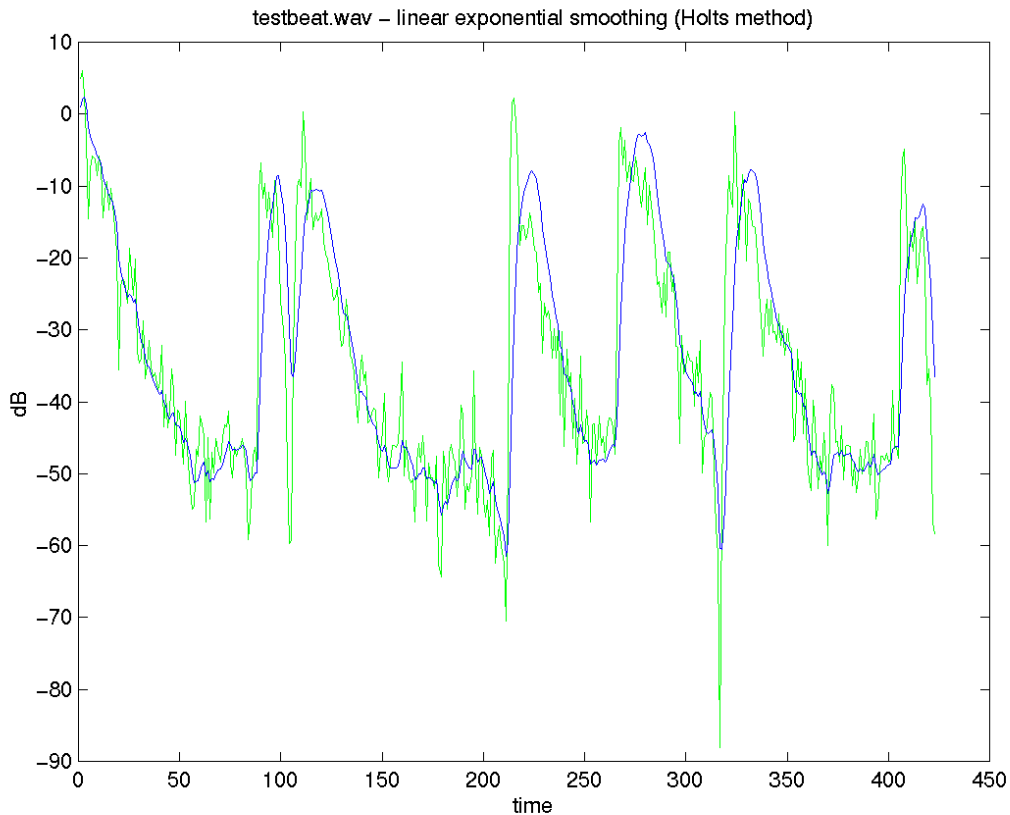


Figure 6.4: The blue graph is the smoothed one using Holt's method, the green is the original unsmoothed one.

This method works well if the data contains a trend but no cyclic pattern.

$$\hat{y}_k = ay_k + (1 - a)(\hat{y}_{k-1} + t_{k-1}) \quad (6.1)$$

$$t_k = b(\hat{y}_k - \hat{y}_{k-1}) + (1 - b)t_{k-1} \quad (6.2)$$

where a is the level smoothing constant, and b is the trend smoothing constant. For the example above a smoothing level of 0.2 was used and a trend smoothing of 0.1. The resulting graph is a much smoother version of the original and the peaks are relatively well preserved although their maxima have been skewed.

Another maybe even more suited smoothing algorithm is the Savitsky-Golay smoothing filters. "Savitsky-Golay filters were initially (and are still often) used to render visible the relative width and heights of the spectral lines in noisy spectrometric data" [PTVF9x]:605. Figure 6.5 shows a graph smoothed using Savitsky-Golay smoothing.

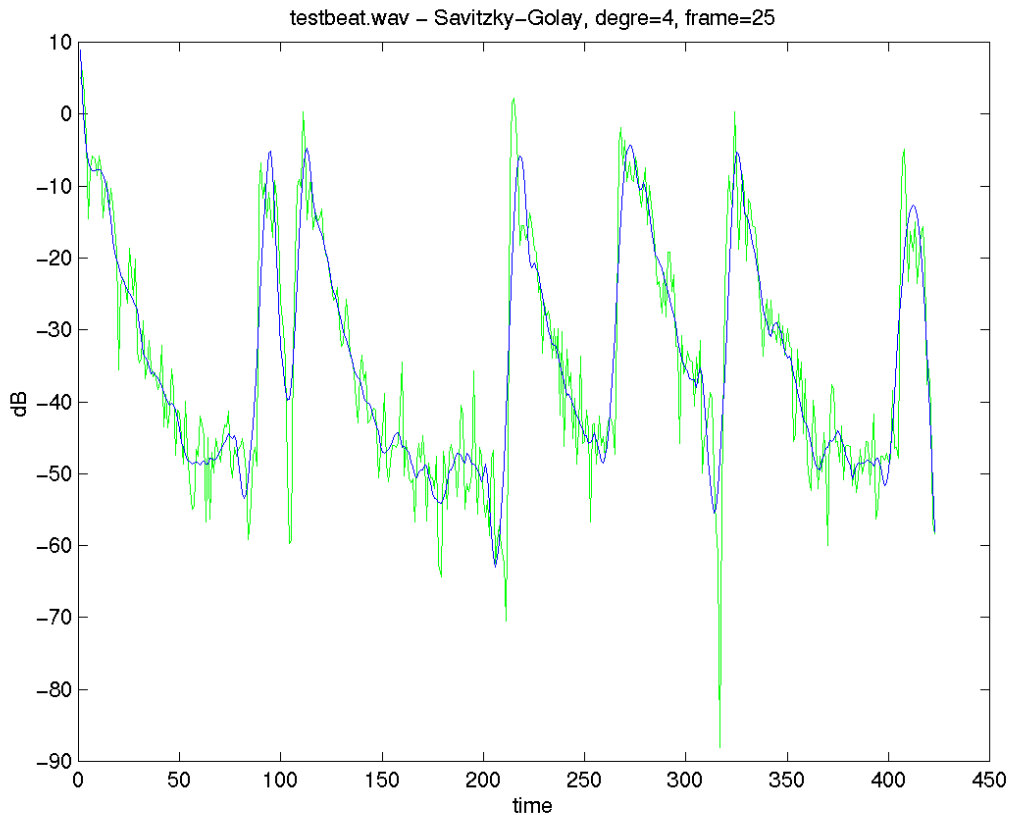


Figure 6.5: The blue graph is the smoothed one using Savitsky-Golay smoothing filters, the green is the original unsmoothed one.

With the Savitsky-Golay smoothing, we see that the height of the peaks are about the same as with the linear exponential smoothing, but that position of the peaks are more true to the original signal (are not that skewed). Also the peaks are more pointed than the ones from linear exponential smoothing. With a smaller frame size, even better similarity to the peaks would have been produced, but then the 'noisy parts' would not have been as subdued. The Savitsky-Golay smoothing was tested in our final onset extraction system, refer to Section 6.1.3.3.

6.1.2.3 Mean decibel-threshold analysis

Even though the frequency-band decibel-threshold onset extraction algorithm was scrapped we did some attempts to develop a method trying to predict the *frequency-limits* (the lower and upper frequency-limits) of the detected drumsounds.

Mean decibel-threshold analysis scans along the spectral-line (a vertical line in the sonogram) of an onset and estimates the frequency-limits of the detected sample(s) by a simple thresholding of the signal power. This way we hoped multiple drumsounds could be detected if they had different frequency distributions and this would be reflected as peaks in the spectral-line at the time of the onset. Figure 6.6 shows the result of one such analysis.

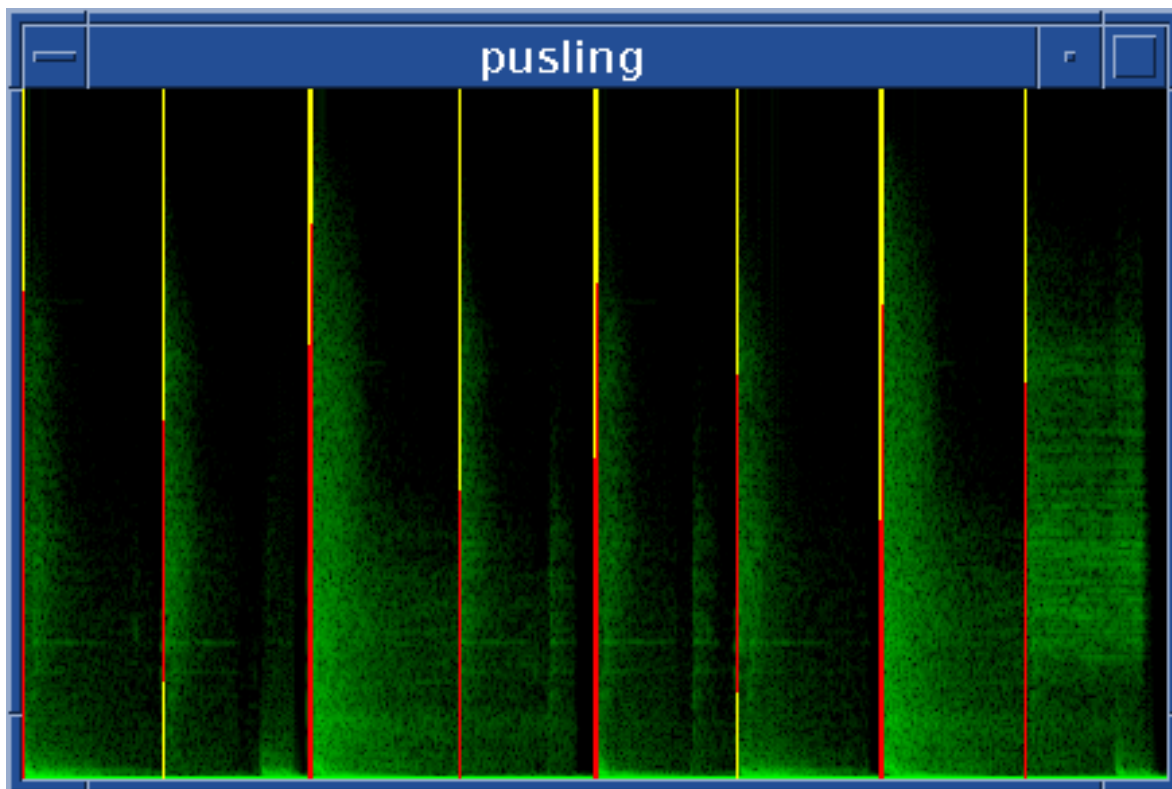


Figure 6.6: The yellow lines are the detected onsets, the red lines are the estimated frequency-limits.

The mean decibel-threshold analysis is far too basic. The results show that it does not predict the frequency-limits of a drumsound accurately. As an example; if we look at the second estimated onset, we can see that the actual drumsound spans a greater frequency-area then the red line predicts. Trying to determine the frequency-limits of a drumsound based on only the onset spectral-line, will not yield good enough results.

6.1.3 Final onset extraction with detailed results

The onset extraction algorithm we chose was a combination of the BTS [GM95] onset components analysis and our own algorithm for interpreting the onset components into drumsound onsets.

What we do with the onset components is that we give them individual weights based on how certain we are this onset component reflects an actual onset. This is done by finding the number of neighbour onset components an onset component has. If an onset component has two neighbours (one in the frequency band above and one below) we give it the highest weight, if the onset component has no neighbours we give it the lowest weight. Based on these weights we extract the estimated onsets.

In the following tests we use a testset of 20 samples of drumsequences that has been subjectively studied and has the original onsets stored in a corresponding XML file. We load the samples and let the analysis program estimate the onsets of the drumsounds in the samples, and compare them to the original onsets stored in the XML files. The samples are all 16bit, mono and with a 44.1kHz samplerate. The drumsequences include both weak drumsounds together with strong ones, overlapping drumsounds both in time and frequency and multiple onsets. The testsets are grouped into 5 different groups where a sample number in the group signifies the complexity of the drumsequence. So `hovedfag01-1.wav` is a rather simple drumsequence while `hovedfag01-4.wav` is more complex.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	18	8	1	10	39.51
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	9	17	5	4	12	16.34
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	9	28	5	4	23	9.92
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	11	34	5	6	29	6.68

Table 6.1: Onset extraction where no windowing function was used. Average correctness: 18.1%.

In order to thoroughly test the onset algorithm we have experimented with changing different settings for the STFT and the thresholds for correctness (how many milliseconds we define as the upper limit for a correct onset detection). The results are discussed in the following sections. We use the measure of correctness that we developed and discussed in Chapter 3.

6.1.3.1 Windowing

As we discussed in Chapter 2, DFT on real-world signals will result in heavy leakage and the possible swamping of neighbouring frequency components unless we perform some kind of windowing. We need to select the best window function for our application.

For all these tests the FFT binsize was 256, there was no overlapping, no smoothing, and the correctness threshold was 5ms. Multiple onsets were ignored. The only variable that varies is the windowing function used.

In Table 6.1 the average correctness is 18.1% when no windowing function was used (or what is also called a rectangular window).

Table 6.2 clearly shows the improvement we get by using a windowing function for the onset extraction. An average correctness of 44.7% using a Hamming window, shows the potential of improvement by choosing the correct window function.

Using the Hanning window clearly gives the best result for our application, see Table 6.3. An average correctness of 70.1% compared to respectively 44.7% and 18.1% is far superior.

6.1.3.2 Overlapping

The amount of overlapping can to some extent control the time resolution we get in a sonogram. With an overlapping of 50% we move only with half the 'speed' along the time-axis when calculating the STFT. One side effect is that the changes in the power of the sonogram gets a bit smeared out as they are repeated (overlapping), changes becomes more gradual.

Onset extraction results in this section are calculated with an FFT binsize of 256, a Hanning window, no smoothing, and with multiple onsets ignored. The correctness threshold is still 5ms. Here we are looking at the effect of overlapping and what it does to our onset results.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	10	8	1	2	71.11
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	9	15	8	1	7	47.41
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	9	17	7	2	10	32.03
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	11	26	9	2	17	28.32

Table 6.2: Onset extraction using Hamming window. Average correctness: 44.7%.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	8	8	1	0	88.89
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	9	9	8	1	1	79.01
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	9	11	7	2	4	49.49
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	11	11	9	2	2	66.94

Table 6.3: Onset extraction using Hanning window. Average correctness: 71.1%.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	11	5	4	6	25.25
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	9	9	6	3	3	44.44
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	9	9	6	3	3	44.44
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	11	13	10	1	3	69.93

Table 6.4: Onset extraction using 50% overlapping in the STFT. Average correctness: 46.0%.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	8	8	1	0	88.89
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	9	9	8	1	1	79.01
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	9	11	7	2	4	49.49
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	11	11	9	2	2	66.94

Table 6.5: Onset extraction using no overlapping.

In Table 6.4 we have an overlapping of 50%. The average correctness is 46.0%. If we do the same onset extraction using no overlapping as in Table 6.5 we get an average correctness of 71.1%³. As we can see, the algorithm clearly favours sonograms calculated without overlapping. What we are probably experiencing is that the smearing in the power distribution results in poorer onset detection by the algorithm.

6.1.3.3 Smoothing

We hoped that using a smoothing function would help in the detection of onsets for the previously discussed time-frequency algorithms. We also experimented with what kind of results this gave for this onset extraction algorithm.

³The same as in Table 6.3 as they are in effect identical analysis.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag01-2.wav: Roland R8	7	8	6	1	2	64.29
hovedfag01-3.wav: Roland R8	8	8	7	1	1	76.56
hovedfag01-4.wav: Roland R8	9	9	8	1	1	79.01

Table 6.6: Onset extraction with no smoothing functions. Average correctness: 73.3%.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag01-2.wav: Roland R8	7	7	3	4	4	18.37
hovedfag01-3.wav: Roland R8	8	9	2	6	7	5.56
hovedfag01-4.wav: Roland R8	9	10	2	7	8	4.44

Table 6.7: Onset extraction using Savitzky-Golay smoothing. Average correctness: 9.5%.

In Table 6.6, the FFT binsize is 256, there is no overlapping, no smoothing and the correctness threshold is 5 ms. Multiple onsets are ignored and the window function is Hanning.

In Table 6.7, the analysis values are the same as for Table 6.6 but with Savitzky-Golay smoothing added. The number of leftward (past) data points is 2, the number of rightward (future) datapoints is 4. The order of the smoothing polynomial is 2. The average correctness drops from 73.3% to 9.5% so in this case Savitzky-Golay smoothing did not help.

Largely maintaining the same analysis values as in Table 6.7 except changing the leftward data points to 0 and adding 50% overlapping helps but the result is not an *improvement* on the extraction result using no smoothing. This gives an average correctness of 23.4%, see Table 6.8. Generally speaking we found no combination of settings where we were able to improve the correctness of the onset extraction algorithm by smoothing the FFT output. We suspect that a similar effect as when using overlapping happens when using Savitzky-Golay smoothing.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag01-2.wav: Roland R8	7	25	6	1	19	20.57
hovedfag01-3.wav: Roland R8	8	20	6	2	14	22.50
hovedfag01-4.wav: Roland R8	9	20	7	2	13	27.22

Table 6.8: Onset extraction Savitzky-Golay smoothing and overlapping. Average correctness: 23.4%.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	9	8	1	1	79.01
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	9	15	8	1	7	47.41
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	9	15	7	2	8	36.30
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	11	19	9	2	10	38.76

Table 6.9: Onset extraction where binsize is 128. Average correctness: 50.4%.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	8	8	1	0	88.89
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	9	9	8	1	1	79.01
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	9	11	7	2	4	49.49
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	11	11	9	2	2	66.94

Table 6.10: Onset extraction where binsize is 256. Average correctness: 71.1%.

6.1.3.4 Binsize

Varying the FFT binsize affects both the number of frequency bands in the resulting sonogram and also the timing resolution of the sonogram. This means that finding the correct binsize for the analysis is very important.

The windowing function is Hanning, and the correctness threshold is 5ms. Multiple onsets are ignored. The results in this section uses a varying FFT binsize but otherwise have no overlapping or smoothing.

Table 6.9 shows the onset extraction results when the binsize is set to 128. The average correctness is 50.4%.

Table 6.10 shows the results when performing onset extraction using a binsize of 256. The results are much improved: 71.1%.

Finally Table 6.11 shows onset extraction where the binsize is set to 512. The average correctness

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	8	4	5	4	22.22
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	9	8	4	5	4	22.22
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	9	7	4	5	3	25.40
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	11	9	7	4	2	49.49

Table 6.11: Onset extraction where binsize is 512. Average correctness: 29.8%.

is 29.8% and this result is the worst of these three. As we can see the FFT binsize that gives the best result is 256, this gives the best tradeoff between frequency and time resolution.

6.1.3.5 Correctness threshold and multiple onsets

Until now multiple onsets have been filtered out from the test cases. This was because we wanted to focus specifically on certain issues and not let the result get too distorted by other factors. Now we test how the onset extraction algorithm performs when multiple onsets are included compared to when they are ignored. We also experiment with the correctness threshold and how this affects the correctness measure we get from the analysis.

The FFT binsize for this analysis is 256, and no smoothing or overlapping has been used. The window function is Hanning. In these tests we include *all* the 20 samples to get a more complete view on how the onset extraction performs on different drumsequences.

Table 6.12 shows the result from onset extraction where we include multiple onsets and have a 5 ms correctness threshold. The average correctness is 60.2%. Comparing this to Table 6.13 which has the same settings except in this case we ignore multiple onsets, we see a difference of about 10%. Ignoring multiple onsets we get an average measure of correctness of 70.2%. As the onset extraction algorithm delivers no special functionality for multiple onsets the difference between these two numbers is greatly influenced by the number of multiple onsets present in the drumsequences. The greater number of simultaneously multiple onsets the lower the measure of correctness.

When we lower the threshold for correctness to 10 ms instead of 5 ms and perform the same tests, we get when including multiple onsets an average measure of correctness of 66.5%. When ignoring multiple onsets we get a correctness of 77.5%. Recall from Chapter 1 a discussion on limits of temporal discrimination. We chose 5 ms as the threshold because humans generally have this lower limit to capture essential intentional timing information. 10 ms is thus a bit above but still usable in an application of our type. An *average* measure of correctness of 77.5% for a varied selection of drumsequences we feel is a very good result.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag01-1.wav: Roland R8	5	5	4	1	1	64.00
hovedfag01-2.wav: Roland R8	7	8	6	1	2	64.29
hovedfag01-3.wav: Roland R8	11	8	7	4	1	55.68
hovedfag01-4.wav: Roland R8	13	9	8	5	1	54.70
hovedfag02-1.wav: Roland R8	7	8	7	0	1	87.50
hovedfag02-2.wav: Roland R8	9	8	5	4	3	34.72
hovedfag02-3.wav: Roland R8	12	10	8	4	2	53.33
hovedfag02-4.wav: Roland R8	14	11	9	5	2	52.60
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	8	8	1	0	88.89
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	11	9	8	3	1	64.65
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	14	11	7	7	4	31.82
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	17	11	9	8	2	43.32
hovedfag04-01.wav: Roland TR606	8	7	7	1	0	87.50
hovedfag04-02.wav: Roland TR606	13	10	9	4	1	62.31
hovedfag04-03.wav: Roland TR606	14	11	9	5	2	52.60
hovedfag04-04.wav: Roland TR606	18	12	10	8	2	46.30
hovedfag05-01.wav: Roland R8	16	15	15	1	0	93.75
hovedfag05-02.wav: Roland R8	16	21	16	0	5	76.19
hovedfag05-03.wav: Roland R8	16	10	10	6	0	62.50
hovedfag05-04.wav: Roland R8	4	8	3	1	5	28.12

Table 6.12: Onset extraction including multiple onsets and a 5ms correctness threshold. Average correctness: 60.2%.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag01-1.wav: Roland R8	5	5	4	1	1	64.00
hovedfag01-2.wav: Roland R8	7	8	6	1	2	64.29
hovedfag01-3.wav: Roland R8	8	8	7	1	1	76.56
hovedfag01-4.wav: Roland R8	9	9	8	1	1	79.01
hovedfag02-1.wav: Roland R8	7	8	7	0	1	87.50
hovedfag02-2.wav: Roland R8	7	8	5	2	3	44.64
hovedfag02-3.wav: Roland R8	10	10	8	2	2	64.00
hovedfag02-4.wav: Roland R8	12	11	9	3	2	61.36
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	8	8	1	0	88.89
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	9	9	8	1	1	79.01
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	9	11	7	2	4	49.49
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	11	11	9	2	2	66.94
hovedfag04-01.wav: Roland TR606	8	7	7	1	0	87.50
hovedfag04-02.wav: Roland TR606	10	10	9	1	1	81.00
hovedfag04-03.wav: Roland TR606	10	11	9	1	2	73.64
hovedfag04-04.wav: Roland TR606	11	12	10	1	2	75.76
hovedfag05-01.wav: Roland R8	16	15	15	1	0	93.75
hovedfag05-02.wav: Roland R8	16	21	16	0	5	76.19
hovedfag05-03.wav: Roland R8	16	10	10	6	0	62.50
hovedfag05-04.wav: Roland R8	4	8	3	1	5	28.12

Table 6.13: Onset extraction ignoring multiple onsets and with a 5ms correctness threshold. Average correctness: 70.2%.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag01-1.wav: Roland R8	5	5	5	0	0	100.00
hovedfag01-2.wav: Roland R8	7	8	6	1	2	64.29
hovedfag01-3.wav: Roland R8	11	8	7	4	1	55.68
hovedfag01-4.wav: Roland R8	13	9	8	5	1	54.70
hovedfag02-1.wav: Roland R8	7	8	7	0	1	87.50
hovedfag02-2.wav: Roland R8	9	8	7	2	1	68.06
hovedfag02-3.wav: Roland R8	12	10	10	2	0	83.33
hovedfag02-4.wav: Roland R8	14	11	11	3	0	78.57
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	8	8	1	0	88.89
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	11	9	8	3	1	64.65
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	14	11	7	7	4	31.82
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	17	11	9	8	2	43.32
hovedfag04-01.wav: Roland TR606	8	7	7	1	0	87.50
hovedfag04-02.wav: Roland TR606	13	10	9	4	1	62.31
hovedfag04-03.wav: Roland TR606	14	11	9	5	2	52.60
hovedfag04-04.wav: Roland TR606	18	12	10	8	2	46.30
hovedfag05-01.wav: Roland R8	16	15	15	1	0	93.75
hovedfag05-02.wav: Roland R8	16	21	16	0	5	76.19
hovedfag05-03.wav: Roland R8	16	10	10	6	0	62.50
hovedfag05-04.wav: Roland R8	4	8	3	1	5	28.12

Table 6.14: Onset extraction including multiple onsets and with a correctness threshold of 10ms. Average correctness: 66.5%.

samplename	org. onsets	est. onsets	detected	undetected	extra	correctness
hovedfag01-1.wav: Roland R8	5	5	5	0	0	100.00
hovedfag01-2.wav: Roland R8	7	8	6	1	2	64.29
hovedfag01-3.wav: Roland R8	8	8	7	1	1	76.56
hovedfag01-4.wav: Roland R8	9	9	8	1	1	79.01
hovedfag02-1.wav: Roland R8	7	8	7	0	1	87.50
hovedfag02-2.wav: Roland R8	7	8	7	0	1	87.50
hovedfag02-3.wav: Roland R8	10	10	10	0	0	100.00
hovedfag02-4.wav: Roland R8	12	11	11	1	0	91.67
hovedfag03-01.wav: Alesis HR16B / Thievery Corporation	9	8	8	1	0	88.89
hovedfag03-02.wav: Alesis HR16B / Thievery Corporation	9	9	8	1	1	79.01
hovedfag03-03.wav: Alesis HR16B / Thievery Corporation	9	11	7	2	4	49.49
hovedfag03-04.wav: Alesis HR16B / Thievery Corporation	11	11	9	2	2	66.94
hovedfag04-01.wav: Roland TR606	8	7	7	1	0	87.50
hovedfag04-02.wav: Roland TR606	10	10	9	1	1	81.00
hovedfag04-03.wav: Roland TR606	10	11	9	1	2	73.64
hovedfag04-04.wav: Roland TR606	11	12	10	1	2	75.76
hovedfag05-01.wav: Roland R8	16	15	15	1	0	93.75
hovedfag05-02.wav: Roland R8	16	21	16	0	5	76.19
hovedfag05-03.wav: Roland R8	16	10	10	6	0	62.50
hovedfag05-04.wav: Roland R8	4	8	3	1	5	28.12

Table 6.15: Onset extraction ignoring multiple onsets and with a correctness threshold of 10ms. Average correctness: 77.5%.

6.2 Drumsound separation

6.2.1 Blind-Source Separation

Tests have been done using ICA (independent component analysis). By modifying the publicly available "Basic Source Separation Code" by Tony Bell⁴, and changing it to better read samples, we loaded two drumsounds, mixed them together to form two different mixes of the two sounds, and then using methods in Bells code, unmixed the two drumsounds from each other.

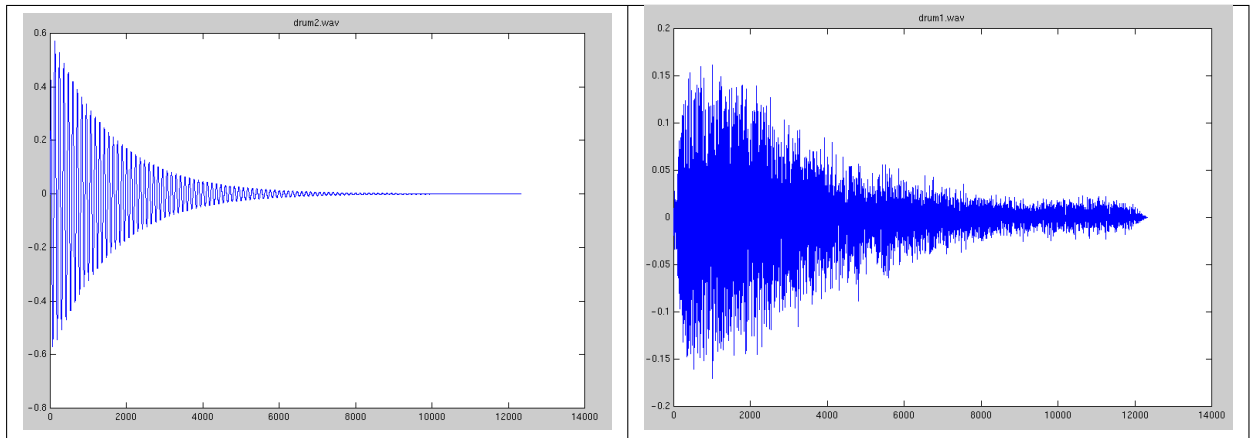


Figure 6.7: Shows the two original drums.

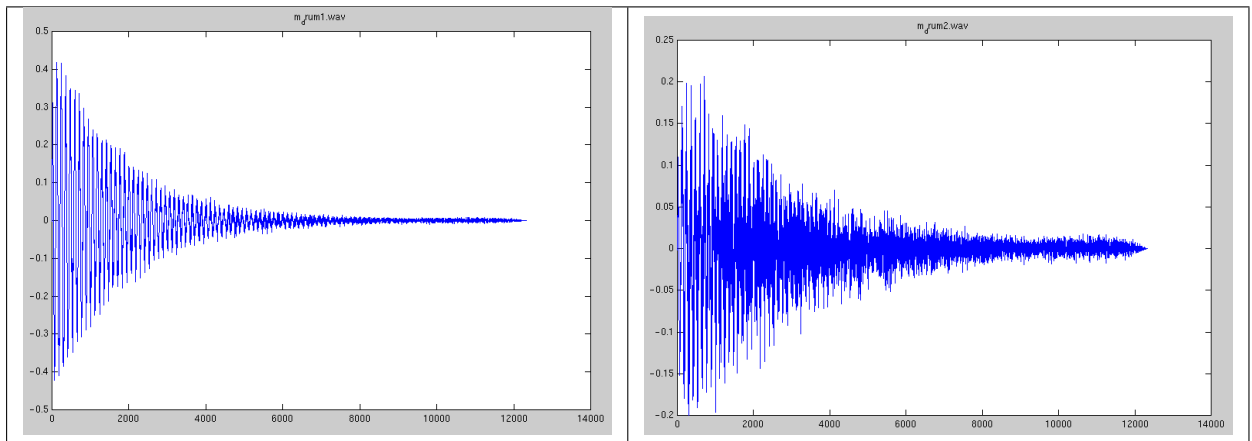


Figure 6.8: Shows the different mixes of the two drums.

⁴Tony Bell Ph.D., CNL, Salk Institute (now at Interval Research, Palo Alto).

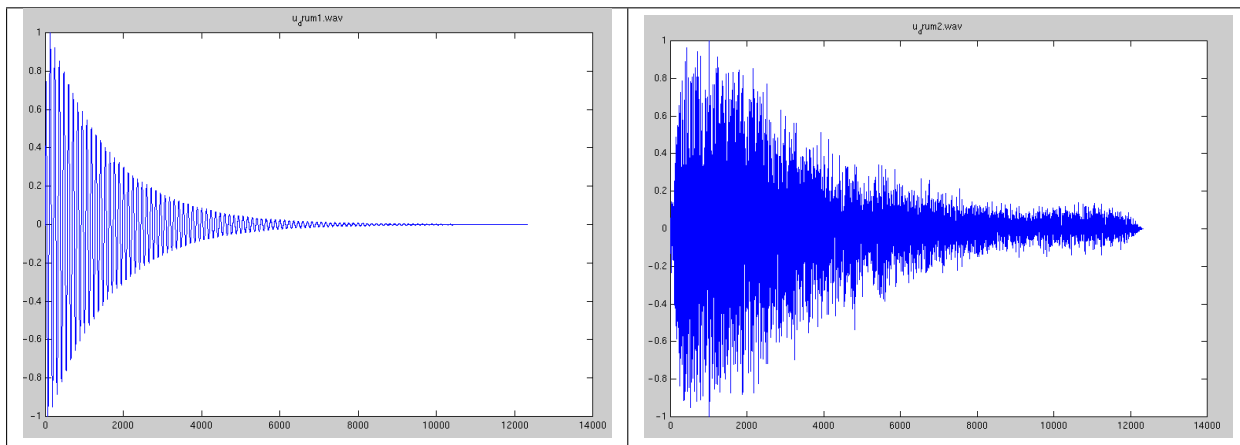


Figure 6.9: Shows the unmixed drums.

Figures 6.7, 6.8 and 6.9 show the start drumsounds, the mixes and the end separated drumsounds. It is hard to determine if the ICA methods actually perform well or not, so we also present sonograms of the same figures. Figure 6.10, 6.11 and 6.12 show the sonograms and here it becomes clearer that the ICA actually does a good job of separating the mixed drumsounds.

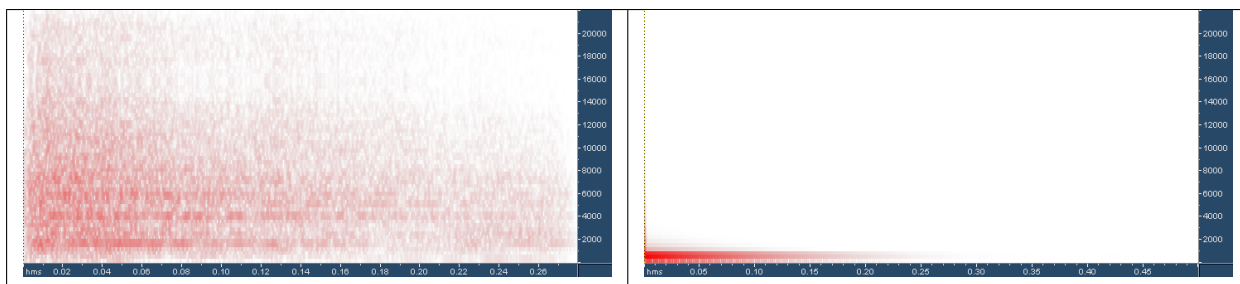


Figure 6.10: Shows a sonogram of the two original drums. FFT binsize 64, Hanning window, logarithmic power plot, 80dB resolution.

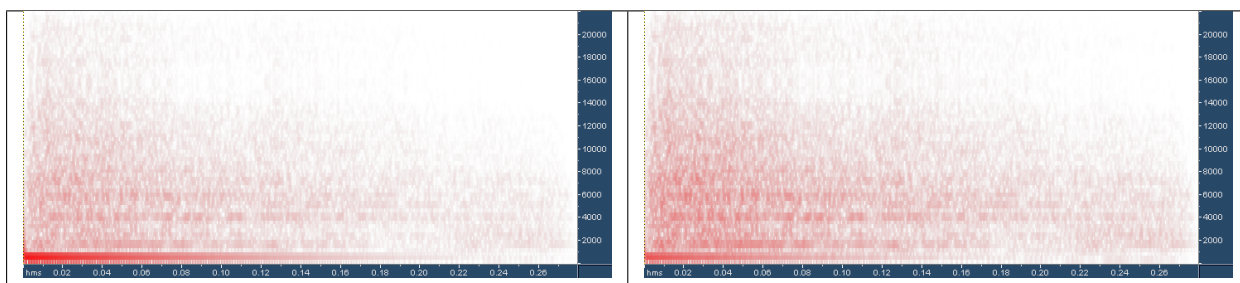


Figure 6.11: Shows a sonogram of the different mixes of the two drums. FFT binsize 64, Hanning window, logarithmic power plot, 80dB resolution.

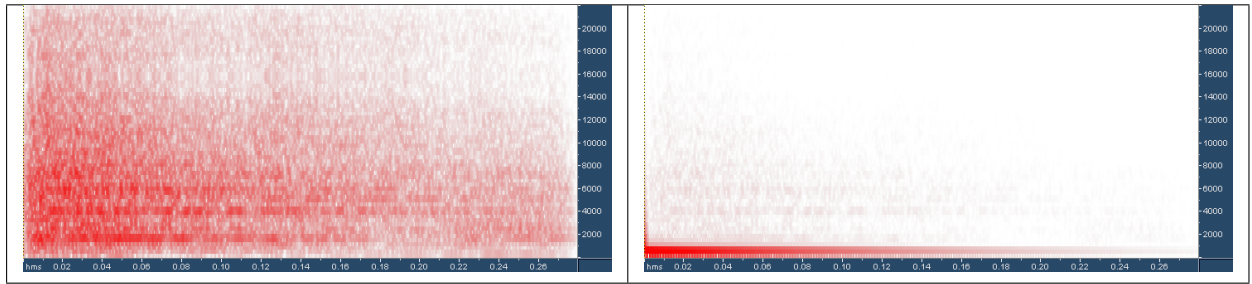


Figure 6.12: Shows a sonogram of the two unmixed drums. FFT binsize 64, Hanning window, logarithmic power plot, 80dB resolution.

The resulting unmixed drumsounds, Figure 6.12, are very similar to the original ones, Figure 6.10, and this might suggest that using ICA to separate drumsounds from each other is a viable option to look further into. However, there have to be as many (or more) mixes of the drumsequence as there are drums to be separated and this is a very limiting restriction. We can not assume there are more than two mixes of the drumsequence (a stereo sample), and can therefore, with normal ICA (not overcomplete), separate more than two drums. This could still be used in small parts of the drumsequence where there are no more than two drums played simultaneously.

Chapter 7

Conclusion

We have studied the possibility to make a composing tool (sequencer) that breaks down the different parts of a drumsequence into the original or close to the original samples it was composed of. We have experimented in the following research areas:

1. extracting drumsound onsets and defining the frequency boundaries for a drumsound
2. separating the drumsounds.

Concerning onset extraction we have developed based on similar research an algorithm that for singular onsets is robust and with a high degree of accuracy. This onset extraction algorithm could be the basis for higher level algorithms or analysis that focused more on extracting meta-information from the samples (such as meter, time signature etc.). We have also showed that it is possible, despite the resolution issues when using STFT, to obtain such accurate results.

The analysis program or parts of the program we have developed could already at this stage be incorporated in various applications. Examples are automatic transcription, video/music synchronization and audio applications such as ReCycle etc.

For the onset extraction testing we developed an XML format that could also be used by others for similar research. The XML format is functional and extendable. The testset of samples we used during the testing of the onset algorithms is also available for validation of our findings or for use in similar tests.

We have also developed a measure of correctness to be used for measuring the correctness of onset extraction. We feel this measure is an improvement over the ones in use today.

Concerning the area of drumsound separation, the results has not been as conclusive and complete as with the onset extraction. We have studied and experimented with analysis such as ICA and traditional filters but in this area no definitive conclusion could be reached.

Concerning the area of finding similar matches of a drumsound in a database, the experimentation and research has been greatly limited. This is largely because the two research areas of onset extraction and drumsound separation needed answers before finding matches in a database became important.

We feel the scope of work for this thesis has been huge. We have needed to develop an analysis framework and application, learn digital signal processing *and* discover and understand the research being done in the beat-tracking and sound-separation community. This thesis has been done from a computer-science perspective focusing on software development, not as a mathematical one focusing on digital signal processing. Ideally, libraries for onset extraction and signal separation should have been available in order for us to keep that focus.

7.1 Future work

We feel that a natural continuation of this work would be in these three areas:

1. broader and more varied set of tests
2. comparing with other onset algorithms and changing time-frequency transformation
3. experiment and explore drumsound separation more thoroughly.

The test cases used in this thesis are all made using a computer. One natural extension of these would be to include live recordings of drumsequences. It would also be interesting to produce statistics on what type of drumsound (like hihat, snare etc.) that is most often detected correctly and similar statistics.

It would be highly interesting to do an analysis where the different available onset extraction algorithms were compared to each other. It would also be interesting to change the time-frequency transformation, the STFT, to a multiresolution transformation like the DWT.

Finally, recent articles and papers have described quite interesting algorithms for drumsound separation and rhythm decomposition. One of these papers is “Riddim” a thesis by Iroro Orife that use independent subspace analysis [Ori01]. It would be natural to study and experiment with these algorithms in order to try and find a solution for the separation problem.

Appendix A

Data format for testcases

One of the problems with research in automatic transcription and beat-tracking systems operating on real-world test and data sets, has been the lack of a uniform form or format for the testdata. XML is a relatively simple standard describing a markup-language that makes it easy to make structured data formats that are easy to understand and parse.

There exist a lot of tools for XML processing, so the possibility for other researchers to read and process the same datasets used in this thesis, should be relatively good.

A.1 XML DTD

The DTD, see Figure A.1, describes the rules for XML files defining onset information about a drum-sequence. The actual sample is an audio file with the same name as the XML file, but with a .wav extension.

```
<!-- Describes the drums present in a sample -->
<!ELEMENT drumtoc (source?, drum+)>

<!-- where the drumsound is from CD/Live recording/Drummachine -->
<!ELEMENT source (#PCDATA)>

<!-- the actual drumsound -->
<!ELEMENT drum EMPTY>
<!ATTLIST drum start CDATA #REQUIRED
              stop  CDATA #IMPLIED
              high  CDATA #IMPLIED
              low   CDATA #IMPLIED
              type  (bdrum | snare | clhihat | ophihat | cymbal | tom | other) #IMPLIED
              force (weak | normal | strong) #IMPLIED
              name  CDATA #IMPLIED>
```

Figure A.1: DTD for the testcases (drumtoc.dtd).

A.1.1 drumtoc element

This is the top element in the XML tree. There can only be one *drumtoc* element in an XML file. *drumtoc* is a *Table Of Contents* for a sample. And it and its subelements describe the drum onsets

found in the sample. The two subelements allowed in a *drumtoc* element are the *source* and *drum* elements.

A.1.2 source element

This element describes the actual source, as in recording or equipment, the sample originates from. This element can be either specified zero or once in a *drumtoc* element. The *source* element has no attributes and its content is #PCDATA, meaning text.

A.1.3 drum element

This element describes a drum onset. This element must occur one or more times inside the *drumtoc* element. This element has no content but one required and several implied attributes.

- **start** This attribute is required and specifies the exact position in milliseconds compared to the start of the sample.
- **stop** This attribute is optional and describes the stop of the drumsound. This is the moment when the power of the drumsound has decayed and is no longer perceivable.
- **high** This attribute is optional and describes the upper limit of the frequency spectrum the drumsound occupies.
- **low** This attribute is optional and describes the lower limit of the frequency spectrum the drumsound occupies.
- **type** This attribute is optional and defines what class of drum the drumsound belongs to. This attribute uses predefined types as its only legal values.
- **force** This attribute is optional and describes in subjective values the loudness of the drumsound in relation to the other drumsounds in the sample.
- **name** This attribute is optional and gives the drumsound a name.

A.1.3.1 drum type attribute

The *type* attribute of the *drum* element describes the class the perceived drumsound belongs to. This is also a subjective measure but should map pretty uniformly even when the XML files are written by different persons. Here is a description of the different types and what drumsounds usually belongs to them:

- **bdrum** This is bass drums. That means 'low' frequency drums that usually maintains the base rhythm in a drumloop.
- **snare** This is snare drums. That means usually 'high' frequency drums with a snappy characteristic.
- **clhihat** This is closed hihats. This is hihat sounds that have a short decay
- **ophihat** This is open hihats. This is hihat sounds with a long decay.

- **cymbal** This is cymbals. Any kind of cymbal being crashcymbal, ridecymbal etc. Cymbals have long decays and a wide frequency spectrum.
- **tom** This is tom-toms. Low mid or high tom-toms fall into this category.
- **other** This is everything else. Examples are bells, 'talking drums' etc.

It would be better for the *type* attribute to follow a more rigorous and formalized system of classification but this would also require the listener that fills in the XML file testsets to have a more formal education. As it now is specified anybody with basic musical knowledge can do a decent job of classifying the drumsounds present in a drumsequence.

A.2 Example of testcase

Figure A.2 shows an example of an XML file describing the different onsets present in the sample *hovedfag03_03.wav*. As we can see the DTD is specified in the DOCTYPE of the XML file.

```
<?xml version="1.0"?>
<!DOCTYPE drumtoc SYSTEM "drumtoc.dtd">
<drumtoc>
<source>Alesis HR16B / Thievery Corporation</source>
<drum start="0.2" type="clhihat"/>
<drum start="0.2" type="bdrum"/>
<drum start="214.4" type="clhihat"/>
<drum start="428.7" type="clhihat"/>
<drum start="429.0" type="snare"/>
<drum start="535.9" type="clhihat"/>
<drum start="643.0" type="clhihat"/>
<drum start="643.0" type="bdrum"/>
<drum start="857.7" type="ophihat"/>
<drum start="1071.7" type="clhihat"/>
<drum start="1071.9" type="snare"/>
<drum start="1285.9" type="clhihat"/>
<drum start="1500.6" type="ophihat"/>
<drum start="1500.4" type="snare"/>
</drumtoc>
```

Figure A.2: Example of a drumtoc XML file (*hovedfag03_03.dtc*).

According to this file there are 14 drum onsets in *hovedfag03_03.wav*. They belong to 4 different drumtypes: *bdrum*, *ophihat*, *clhihat* and *snare*. Also from this file we can see that there are several drums being hit at once, like with the first two onsets, see Figure A.3.

```
<drum start="0.2" type="clhihat"/>
<drum start="0.2" type="bdrum"/>
```

Figure A.3: Example of drums starting at the same time.

It says that after 0.2 ms there are two drums being hit. One is of the type *clhihat* (closed hihat) and the other is a *bdrum* (bass drum). What other characteristics these drumsounds have we do not know. But as we have seen the XML DTD makes it possible to specify other attributes, but it is not required.

The *source* tag in the XML file tells us that this specific sample has drumsounds both from a drum machine, the *Alesis HR16B*, and from a recording of the band *Thievery Corporation*. This information is not critical for operations like automated comparisons of onset extraction algorithms but can be interesting as documentation of the actual testsets.

Appendix B

Glossary

Adaptive Differential Pulse Coded Modulation (ADPCM)

A speech compression algorithm that adaptively filters the difference between two successive PCM samples. This technique typically gives a data rate of about 32 Kbps.

adaptive filter

A filter that can adapt its coefficients to model a system.

aliasing

The effect on a signal when it has been sampled at less than twice its highest frequency.

amplitude

A value (positive/negative) describing the position of the samplepoint relative to zero.

amplitude modulation

A communications scheme that modifies the amplitude of a carrier signal according to the amplitude of the modulating signal.

applet

A small *Java* program running inside a web browser.

attack

Describes the time it takes before a sound reaches its peak intensity (e.g. Gunfire has a very fast attack, but low rumbling thunder will have a slower attack).

attenuation

Decrease, typically concerning magnitude.

autocorrelation

The correlation of a signal with a delayed version of itself.

bandpass filter

A filter that only allows a single range of frequencies to pass through.

bandstop filter

A filter that removes a single range of frequencies.

bandwidth

The range of frequencies that make up a more complex signal.

biquad

Typical 'building block' of IIR filters - from the bi-quadratic equation.

BPM

Beats per minute. Popular music is usually in the are of 80-140 BPMs.

butterfly

The smallest constituent part of an FFT, it represents a cross multiplication, incorporating multiplication, sum and difference operations. The name is derived from the shape of the signal flow diagram.

BSS

Blind Source Separation. The problem of filtering out certain sounds or signals in a noisy environment, like a specific persons voice at a cocktail party.

coherent power gain

Because the window function attenuates the signal at both ends, it reduces the overall signal power. This reduction in signal power is called the Coherent Power Gain.

convolution

An identical operation to Finite Impulse Response filtering.

correlation

The comparison of two signals in time, to extract a measure of their similarity.

decay

Describes the time it takes from the point of peak intensity until it disappears.

decibel

Decibal or *dB* is defined as a unit for a logarithmic scale of magnitude.

Discrete Fourier Transform (DFT)

A transform that gives the frequency domain representation of a time domain sequence.

discrete sample

A single sample of a continuously variable signal that is taken at a fixed point in time. Also called *sample*.

drumloop

A sample of a percussive loop

drumsequence

A sample of a percussive part or loop

drumsound

A sample of a drum

Fast Fourier Transform (FFT)

An optimised version of the DFT.

fall off

The rate of fall off to the side lobe. Also known as roll off.

Finite Impulse Response (FIR) filter

A filter that typically includes no feedback and is unconditionally stable. The impulse response of a FIR filter is of finite duration.

frequency

The rate of completed cycles in a signal. Usually measured in Hz (cycles/second).

frequency domain

The representation of the amplitude of a signal with respect to frequency.

gain

Amplification or increase, typically concerning magnitude.

GPL, GNU General Public Licence

A software license that provides a high degree of freedom in a collaborative software development effort.

highpass filter

A filter that allows high frequencies to pass through.

ICA

Independent Component Analysis.

Infinite Impulse Response (IIR) filter

A filter that incorporates data feedback. Also called a recursive filter.

Impulse Response

The result vector when feeding a system with a signal.

JavaScript

A (usually) client-side scripting language executed in web browsers.

Java

Object oriented programming language. Targeted at embedded devices, application servers, and 'classical' applications.

lowpass filter

A filter that allows low frequencies to pass through.

LTI

Linear time-invariant. An LTI-system has the property that the output signal due to a linear combination of two or more input signals can be obtained by forming the same linear combination of the individual outputs

magnitude

A positive value describing the *amplitude* of a signal. The magnitude is the amplitude of the wave irrespective of the phase.

MAX

A highly complex sequencing program for composing rule-based music.

MIDI

Musical Instrument Digital Interface. MIDI is a standard that lets your MIDI device (like your MIDI keyboard) communicate with other MIDI devices.

modulation

The modification of the characteristics of a signal so that it might carry the information contained in another signal.

multirate

As opposed to STFT wavelets are considered a multirate analysis because the resolution of the analysis is not static or fixed.

onset

The start of an event. In this context the start of a drumsound in a sample.

onset detection

The process of finding onsets in a sample.

onset components

Indicators found based on frequency or time-domain calculations used to determine onsets.

passband

The frequency range of a filter through which a signal may pass with little or no attenuation.

phase

A particular stage or point of advancement in a cycle; the fractional part of the period through which the time has advanced, measured from some arbitrary origin often expressed as an angle (phase angle).

pitch

The key of a sound, dependent primarily on the frequency of the sound waves produced by its source.

pole

Artefact leading to frequency dependent gain in a signal. Generated by a feedback element in a filter.

power intensity spectrum

A spectrum showing (usually) the decibel of signal in relation to its frequencies.

processing loss

Processing Loss measures the degradation in signal to noise ratio due to the window function. It is the ratio of Coherent Power Gain to Equivalent Noise Bandwidth.

Pulse Code Modulation (PCM)

The effect of sampling an analog signal.

recursive filter

See *Infinite Impulse Response filter*.

sample

A digital representation of a physical sound. Also called a *discrete sample*.

samplepoint

A samplevalue in a sample

sampling

The conversion of a continuous time analog signal into a discrete time signal.

samplerate

The inverse of the time between successive samples of an analog signal.

server

Any machine or system delivering 'services' to clients.

sidelobe

Lobes in magnitude beside the mainlobe.

smoothing

The process of evening out irregularities in processed data.

sonogram

An 'image' showing the variation over time of the frequency and power (decibel) of a signal.

spectrogram

In this thesis used as the same as a *sonogram*.

spectrum analyser

An instrument that displays the frequency domain representation of a signal.

stationary

A signal whose statistical properties do not change over time.

stopband

The frequency range of a filter through which a signal may NOT pass and where it experiences large attenuation.

TCP

Transmission Control Protocol (TCP) provides a reliable byte-stream transfer service between two endpoints on the Internet.

time domain

The representation of the amplitude of a signal with respect to time.

UDP

User Datagram Protocol (UDP) provides an unreliable packetized data transfer service between endpoints on the Internet.

waveform

A graphical presentation of a sample, often also used as the same as *sample*.

wavelet

Wavelets are a way to analyze a signal using base functions which are localized both in time (as Diracs, but unlike sine waves), and in frequency (as sine waves, but unlike Diracs). They can be used for efficient numerical algorithms and many DSP or compression applications.

window

A function/vector usually describing an attenuation of a signal. Often used in *FFT*.

windowing

The process of applying a *window* to a vector.

XML

Extensible Markup Language. A universal format for structured documents and data.

z-domain

The discrete frequency domain, in which the $j\omega$ axis on the continuous time s-plane is mapped to a unit circle in the z-domain.

zero

Artifact leading to frequency dependent attenuation in a signal. Generated by a feedforward element in a filter.

Bibliography

- [AD90] P. E. Allen and R. B. Dannenberg. Tracking musical beats in real time. *Proceedings of the 1990 ICMC*, pages 140–43, 1990.
- [Boe93] Hans-Juergen Boehm. Advantages and disadvantages of conservative garbage collection. http://www.hpl.hp.com/personal/Hans_Boehm/gc/issues.html, 1993.
- [Boe02] Hans-Juergen Boehm. A garbage collector for c and c++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/, 2002.
- [Bor98] Chris Bores. FFT window functions - Limits on FFT analysis. <http://www.bores.com>, 1998.
- [Coo99] Clark Cooper. Benchmarking xml parsers. <http://www.xml.com/lpt/a/Benchmark/article.html>, 1999.
- [DH89] P. Desain and H. Honing. The quantization of musical time: A connectionist approach. *Computer Music Journal*, (13(3)):56–66, 1989.
- [DH94] P. Desain and H. Honing. Advanced issues in beat induction modeling: syncopation, tempo and timing. *Proceedings of the 1994 ICMC*, pages 92–94, 1994.
- [DMR87] R. B. Dannenberg and B. Mont-Reynaud. Following an improvisation in real time. *Proceedings of the 1987 ICMC*, pages 240–248, 1987.
- [Dri91] A. Driesse. Real-time tempo tracking using rules to analyze rhythmic qualities. *In Proceedings of the 1991 ICMC*, pages 571–581, 1991.
- [FJ98] Matteo Frigo and Steven G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. *ICASSP conference proceedings*, 3:1381–1384, 1998.
- [FJ01] Matteo Frigo and Steven G. Johnson. FFTW. <http://www.fftw.org/>, 2001.
- [Get75] David J. Getty. Discrimination of short temporal intervals: A comparison of two models. *Perception and Psychophysics*, 18(1):1–8, 1975.
- [GM95] M. Goto and Y. Muraoka. Music understanding at the beat level - real-time beat tracking for audio signals. *IJCAI-95*, 1995.
- [GM96] M. Goto and Y. Muraoka. Beat tracking based on multiple-agent architecture - a real-time beat tracking system for audio signals. *ICMAS-96*, 1996.

- [GM97] M. Goto and Y. Muraoka. Issues in evaluating beat tracking systems. *IJCAI-97 Workshop on Issues in AI and Music - Evaluation and Assessment*, 1997.
- [Hir59] Ira J. Hirsch. Auditory perception of temporal order. *Journal of the Acoustical Society of America*, 31(6):759–767, 1959.
- [Jos99] Nicolai M. Josuttis. *The C++ Standard Library - A Tutorial and Reference*. Addison Wesley, 1999.
- [KK02] Tony Kirke and Julius Kusuma. Signal processing using c++. <http://spuc.sourceforge.net/>, 2002.
- [Kla99] Anssi Klapuri. Sound onset detection by applying psychoacoustic knowledge. *ICASSP-99*, 1999.
- [Lar95] E. W. Large. Beat tracking with nonlinear oscillator. *Working Notes of the IJCAI-95 Workshop on Artificial Intelligence and Music*, pages 24–31, 1995.
- [LS98] M. S. Lewicki and T. J. Sejnowski. Learning overcomplete representations. *Submitted to Neural Computation*, 1998.
- [Lyo01] Richard G. Lyons. *Understanding Digital Signal Processing*. Prentice-Hall, 2001.
- [McA92] T. McAulay. The auditory primal sketch: A multiscale model of rhythmic grouping. *Journal of New Music Research*, (23):25–70, 1992.
- [MGB97] B. Moore, B. Glasberg, and T. Baer. A model for the prediction of thresholds, loudness and partial loudness. *J. Audio Eng. Soc.*, 45(4):224–240, 1997.
- [Mic01] Sun Microsystems. Javadoc. <http://java.sun.com/j2se/javadoc/index.html>, 2001.
- [Mic02] Sun Microsystems. Java Speech API. <http://java.sun.com/products/java-media/speech/>, 2002.
- [MIZ9x] N. Murata, S. Ikeda, and A. Ziehe. An approach to blind source separation based on temporal structure of speech signals. 199x.
- [Moo95] B. Moore, editor. *Hearing. Handbook of Perception and Cognition*. Academic Press, 2 edition, 1995.
- [New95] D. Newland. Signal analysis by the wavelet method. *Technical Report CUED/C-MECH/TR.65*, 1995.
- [Orf96] S. J. Orfanidis. *Introduction to Signal Processing*. Prentice-Hall, 1996.
- [Ori01] Iroro Orife. Riddim: A rhythm analysis and decomposition tool based on independent subspace analysis. 2001.
- [PM96] John G. Proakis and Dimitris G. Monoakis. *Digital Signal Processing - Principles, Algorithms, and Applications*. Prentice-Hall, 3 edition, 1996.
- [Pol01] Robi Polikar. The engineer’s ultimate guide to wavelet analysis - the wavelet tutorial. <http://engineering.rowan.edu/~polikar/WAVELETS/WTtutorial.html>, 2001.

- [Pre99] Lutz Prechelt. Comparing Java vs.C/C++ Efficiency Differences to Interpersonal Differences. *COMMUNICATIONS OF THE ACM*, 42:109–112, 1999.
- [PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C The Art of Scientific Computing*. Cambridge University Press, 2 edition, 1992.
- [PTVF9x] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical recipes in C: The art of scientific computing. 199x.
- [RGM94] D. Rosenthal, M. Goto, and Y. Muraoka. Rhythm tracking using multiple hypotheses. *Proceedings 1994 in ICMC*, 1994.
- [Ros82] Thomas Rossing. The physics of kettledrums. *Scientific American*, 1982.
- [Ros92a] D. Rosenthal. Emulation of human rhythm perception. *Computer Music Journal*, (16(1)):64–76, 1992.
- [Ros92b] D. Rosenthal. Machine rhythm: Computer emulation of human rhythm perception. ph.d. dissertation. 1992.
- [Sch85] W. Andrew Schloss. On the automatic transcription of percussive music – from acoustic signal to high level analysis. 1985.
- [Sch98] E. D. Scheirer. Tempo and beat analysis of acoustic musical signals. *Journal of the Acoustical Society of America*, 103(1), 1998.
- [Ste72] S. S. Stevens. Perceived level of noise by mark vii and decibels (e). *Journal of the Acoustical Society of America*, 51(2):575–601, 1972.
- [Tea01] The Qt Team. Qt. <http://www.trolltech.com/products/qt/index.html>, 2001.
- [Tecxx] Nicolet Technologies. Windowing for dynamics measurements. <http://www.niti.com/files/Windowing.pdf>, 19xx.
- [TF95] F. Tait and W. Findlay. Wavelet analysis for onset detection. *International Computer Music Conference*, pages 500–503, 1995.
- [Ver94] B. Vercoe. Perceptually-based music pattern recognition and response. *Proc. Of the Third Intl. Conf. for the Perception and Cognition of Music*, pages 59–60, 1994.
- [vH01] Dimitri van Heesch. Doxygen. <http://www.stack.nl/~dimitri/doxygen/>, 2001.