UNIVERSITEIT ANTWERPEN
departement Wiskunde-Informatica

Sinusoidal Modelling of Polyphonic Audio

Sinusoïdale Modellering van Polyfone Audio

Joachim Ganseman

Proefschrift ter verkrijging van de graad van Licentiaat in de Wetenschappen richting: Informatica

Promotor:
Prof. Dr. Chris Blondia
PATS Research Group - Universiteit Antwerpen
Begeleiding:
Dr. Wim D'haes
Visielab - Universiteit Antwerpen
Academiejaar:
2006-2007

UNIVERSITEIT ANTWERPEN
departement Wiskunde-Informatica

Sinusoidal Modelling of Polyphonic Audio

Sinusoïdale Modellering van Polyfone Audio

Joachim Ganseman

Proefschrift ter verkrijging van de graad van Licentiaat in de Wetenschappen richting: Informatica

Promotor:
Prof. Dr. Chris Blondia
PATS Research Group - Universiteit Antwerpen
Begeleiding:
Dr. Wim D'haes
Visielab - Universiteit Antwerpen
Academiejaar:
2006-2007

Preamble

As a computer scientist student and a passionate amateur musician, I sought out a thesis subject that related my 'job', computer science, with my 'hobby', music. I went searching for people at the University of Antwerp that did research on computer audio, a field on the crossroad of science and arts. About 2 years ago, I was very lucky to find Wim D'haes, working at the VisionLab of the Physics Department of the University of Antwerp, doing his postdoctoral research on audio and music.

The first collaboration that we set up was in the 3rd year of my studies, for that year's large obligatory project work. We investigated alignment algorithms for synchronization of symbolic musical notation and normal audio data [1], and planned an implementation using VST technology [2]. The project was quite ambitious given the limited amount of time, and never really got out of pre-alpha phase. But it did learn me a whole lot of things about digital signal processing, the enormous amount of tricky issues when doing real-time audio computations, the MusicXML [3] and MIDI [4] file formats, C programming, and using VST [2].

This knowledge was further expanded during the work on this thesis. The subject however, is quite different. Where the 3rd year's project was mainly concerned about manipulation of symbolic notation of music (the partition), this time I worked more on the digital signal processing side, and all the complexities that come with it.

A sinusoidal model is one of the preferred techniques to represent an audio segment in a computer. The audio segment is represented in the frequency domain, as a finite sum of sines and cosines (or, more theoretically, complex exponentials). Sinusoidal modelling tries to solve the problem of converting an audio signal to such a signal model, and back again. The model of one single harmonic note is very easy: it contains one fundamental frequency, and a limited number of harmonics. It can be precisely calculated by fitting the model onto the Fourier transform of the real signal, using least-squares methods.

The Ph.D. dissertation of Wim D'haes [5] introduces several important improvements on the calculation of a sinusoidal model of a sound source. These improvements significantly reduce the computational cost that is involved. In this thesis, these improvements are investigated and thoroughly mathematically analyzed. Some tricky issues arise when multiple harmonic

sound sources (several notes played at once) and polyphonic sound sources (several inter-related notes played at once, or chords) are involved, requiring extensions to the algorithms. The properties of an efficient implementation in C will be presented.

The nature of the subject and the used technology requires a significant amount of knowledge that is not always standard in computer science curriculum. Digital signal processing, numerical techniques and optimization, computational complexity theory, concurrency and real-time computation, artificial intelligence, but also elements of music theory and harmony, physics of sound, etc.: all come together when working on a subject like this (or, in general, on any subject involving multimedia processing or encoding).

Detailed information on all of these topics would fill an entire bookshelf. Since there is not too much room here, I will limit myself to an abundant use of references to scientific articles, theses and books. The reader needing some more background information on any of those topics, can certainly find most of the necessary information in there. Several websites are referenced too, and since the internet is a highly dynamic medium, I am obliged to inform that all those websites were checked for the last time on correctness in august 2007.

I have chosen to split this thesis up into 2 large chapters. The first chapter will explain the theory. It presents the theoretical foundations of sinusoidal modelling, the methods developed by Wim D'haes in his dissertation [5], and the extensions needed when working with polyphonic audio sources. The second chapter will present an implementation: an overview of the technologies that have been used, the problems that arise when programming audio processing systems, and some interesting details of an implementation of the algorithms described in the first section.

I hope very much that the reader will enjoy reading this thesis as much as I enjoyed writing it and working on it. At times it has given me severe headaches, but nevertheless it has always been great fun to work on a subject like this.

Acknowledgements

First I wish to thank my supervisor and tutor, Wim D'haes, for willing to guide me in this project. His knowledge of the subject still keeps impressing me, and his patience with me - not the most regular and steadfast student - is priceless. He gave me a clear view on how research in computer audio really works and what it looks like. He often provided me with a place in his office to work on my thesis. He gave me the chance to attend some very interesting conferences and activities. And last but not least, his Ph.D. dissertation forms the basis of this master thesis.

Together with Wim D'haes, I owe the VisionLab at the University of Antwerp and its members my gratitude. They let me use their coffee machine when I was working at Wim's office, and that represents quite an amount of coffee in my case. We had funny and interesting talks during coffee and lunch breaks.

I also wish to thank Prof. Dr. Chris Blondia from the PATS Research Group, who is so kind to be the promotor for this thesis, and granted me total freedom.

Most of my friends, both inside and outside the University of Antwerp, declared me officially crazy for starting a master thesis with a topic like this. But in some stressful periods I couldn't have wished for more comprehensive surroundings. I promise to pay them a round of beers after this work has been finished, for their unconditional support and for the interesting links, references, discussions and ideas they provided me with.

And last but not least I have had the best of support from my parents and my family. They've always believed in me and gave me ample of freedom during the last few years at the university. This thesis is the closing piece of what have been the best years of my life.

Abstract

The Fourier transform of a limited frame of a discrete audio signal is itself a discrete signal, with a limited resolution depending on the frame length. On top of that, the Fourier domain representation suffers from spectral leakage. This master thesis investigates sinusoidal modelling. First we assume that the signal can be described using a model - a set of frequencies and their corresponding amplitudes. This model is optimized from an initial guess to find the exact model parameters which minimize the square difference with the original signal. Building upon the work by D'haes [5], a lot of attention will go to the effects of windowing, the mathematical foundation of the frequency optimization and amplitude estimation algorithms themselves, and the observations that make several orders of speedup possible. The technologies that were used for implementing these algorithms are reviewed, several speed-up improvements on the implementation level are explained, and several concerns for audio application programmers are addressed. The conclusion presents a broad range of unexplored knowledge that waits to be discovered and developed.

Nederlandstalige samenvatting

De discrete Fourier transformatie van een frame van een discreet audiosignaal is zelf discreet, en heeft een beperkte resolutie die afhangt van de framelengte. Bovendien heeft de Fourier domein representatie last van het fenomeen 'spectral leakage'. Dit heeft als gevolg dat de energie van enkele frequentiecomponenten verspreid raakt over het gehele discrete spectrum. Spectral leakage kan aan banden gelegd worden door het toepassen van een vensterfunctie alvorens de Fourier transformatie te berekenen. Er bestaan vele van deze vensterfuncties, elk met hun eigenschappen, en voor de toepassing in de algoritmes in deze thesis zal de Blackman-Harris window functie de meest voor de hand liggende blijken te zijn.

Deze licentiaatsthesis behandelt sinusoïdale modellering. Aangenomen dat een signaal volledig beschreven kan worden als een model - een verzameling frequenties en hun corresponderende amplitudes - gaan we een initiele ruwe benadering van zo'n model optimaliseren om de exacte parameters te vinden voor het model waardoor het kwadratisch verschil met het oorspronkelijk signaal minimaal wordt. Op deze manier bekomen we een model dat het signaal het best mogelijk beschrijft, en alleszins veel nauwkeuriger is dan de discrete Fourier transformatie. De modelparameters zijn reel in plaats van discreet, en hebben dus geen last van een gebrek aan spectrale resolutie.

De algoritmes zijn ontwikkeld door Wim D'haes [5] in zijn doctoraatsthesis, en hierop wordt voortgewerkt. Veel aandacht gaat naar de effecten van het gebruik van een vensterfunctie, want het meenemen van deze functie in de algoritmes laat significante en fundamentele verbeteringen van de tijdscomplexiteit toe. De wiskundige basis van de algoritmes en de verbeteringen erop wordt uitgebreid uit de doeken gedaan. Er wordt ook ingegaan op de problemen die opduiken wanneer men verschillende harmonische signalen of polyfone audio gaat gebruiken, waarvan de frequentiecomponenten elkaar overlappen.

Om deze algoritmes te implementeren werd gebruik gemaakt van verschillende technologien die kort besproken worden. Er zijn verschillende zaken die men in het achterhoofd moet houden wanneer men (real-time) audio applicaties ontwikkeld, de een al wat venijniger dan de andere - een paar worden eruit gelicht. De uiteindelijke implementatie van de algoritmes in C maakt gebruik van enkele optimalisaties op implementatieniveau die ook een woordje waard zijn.

Ter conclusie is er nog een breed veld aan onderzoek en ontwikkeling mogelijk, niet enkel op het gebied van sinusoïdale modellering maar ook daarrond. Een sinusoïdaal model van een audiosignaal is zeer gemakkelijk om te manipuleren en te verwerken. De reden dat het nog maar weinig gebruikt wordt is de relatief hoge tijdscomplexiteit. Nu die sterk gereduceerd is in de algoritmes hierin besproken, staat er niets nog een snelle ontwikkeling van nieuwe toepassingen die van sinusoïdale modellering gebruikmaken in de weg.

Contents

0.1 Introduction

0.1.1 What is sinusoidal modelling?

A sound source, whether monophonic, polyphonic, noisy or not, is most often represented as a function of time. For treatment by computers or other digital equipment, the timeline and the range of possible values is finite and discrete. Today's compact discs use a sampling frequency of 44100 Hz, and have a resolution of 16 bit. Professional recording equipment often uses multiples of these values, to provide even more accuracy in the recordings.

The Fourier transform converts a signal in the time domain to its equivalent in the frequency domain. The equivalence is exact when using continuous, infinite signals and continuous algorithms. In the discrete computer world, the Fourier transform produces a discrete frequency domain representation, and on top of that, artefacts like spectral leakage tend to occur. This might have large influence on the results when the signal is a sound source: humans hear pitch on a quasi-logarithmical scale.

A sinusoidal model of a single harmonic sound signal consists of one fundamental frequency and a number of harmonics, each with their own amplitude and phase. The fundamental frequency need not be one of the frequencies that is present in the discrete spectrum of the signal, but can lie somewhere in between of the measured values. To estimate the correct fundamental frequency, optimization routines need to be implemented. The amplitude of that frequency can afterwards be calculated.

Rather than representing a sound by its entire time or frequency representation, we can accurately represent it by a model containing only the frequencies, amplitudes and phases of the fundamental frequency and the harmonics. This sinusoidal model is much easier to process further on in any application that needs knowledge of these basic sound properties. It fully defines a single sound and is on top of that more clear and accurate than a complete discrete frequency representation of a sound.

Using a finite set of parameters to describe a sound source is certainly not a new idea. The first attempts at encoding sound in such a way were made with the telephone system in mind. Around 1980, one sought for ways to encode speech in an economical way, to reduce bandwidth and at the same time keep as much of the quality as possible (see for example Flanagan [6]). From 1984 on, the term 'sinusoidal model' as used in this thesis, appears

more and more in publications, (see for example the papers of McAulay and Quatieri, of which especially [7] has been very influential). A short history of the further developments of sinusoidal models with a focus on speech systems can be found in a paper by Bailly [8].

Sinusoidal modelling, as a process, is concerned with fitting a predefined model to a given sound, using parameters like frequency, amplitude and phase. This thesis will analyze and implement the improvements to the sinusoidal modelling algorithm, developed by Wim D'haes in his Ph.D. thesis [5], and also investigate whether this solution for a single harmonic sound source is extensible to multiple sound sources, both non-polyphonic and polyphonic.

0.1.2 Uses of sinusoidal modelling

Sinusoidal modelling has been succesfully used in audio applications like speech analysis and encoding [7], text to speech systems [9], matching algorithms [10], source separation [11], voice effects and resynthesis [12], and so on. In fact, once the sinusoidal model of a sound has been constructed, the parameters can be changed at will. A vast range of sound effects can be developed easily with a sinusoidal model. The manipulated sound can be resynthesized from the model using an inverse Fourier transform.

0.1.3 Pros and Contras

The paper by Syrdal [9] makes an excellent comparison between 2 different approaches for speech encoding: the 'popular' TD-PSOLA (short for Time-Domain Pitch-Synchronous Overlap-Add), and the 'new' Harmonic plus Noise model, which corresponds to a sinusoidal model plus a noise component. The conclusion is that a harmonic model is superior to the other one in all cases, except for computational complexity.

Computational complexity is a large problem in sinusoidal modelling, but with current computer power it is perfectly possible to develop a real-time performing modelling system on an ordinary home PC. As will be shown later on in this thesis, the computational load of the algorithm can be significantly reduced by optimizing the used data fitting algorithm for its task, as developed by D'haes [5]. On top of that, the use of lookup tables and special purpose microprocessor instruction sets like SSE, leads to optimization on

the implementation level.

0.1.4 The aim of this thesis

This thesis investigates the sinusoidal modelling algorithms developed by D'haes in [5]. A first section will elaborate on why these algorithms are needed. The algorithms will be described in detail, and will - where appropriate - also be extended for use with multiple harmonic sounds, and interrelated multiple harmonic sounds (polyphony).

In parallel, an implementation of these algorithms has been made, which is described in the second part of this thesis. The chosen technologies and their properties, assets and drawbacks will be explained. A few implementation level optimizations will be explained in detail. I will also shortly elaborate on the many questions and issues that pop up when developing real-time signal processing applications.

Chapter 1

Sinusoidal Modelling of Polyphonic Audio

1.1 A Digital Signal Processing Crash Course

Any sound wave, when digitized and stored in a computer, has the form of a finite digital signal. It is therefore only logical to give a quick introduction to digital signal processing, before moving on to the real work.

1.1.1 Signal Representation

Time domain representation

Time domain representation of signals is very common and easy to understand. When measuring the local pressure of the medium excited by the signal at the measuring point at any time, a wave propagating through a medium can be described as a function of time.

To encode an analog, real world signal like a sound signal into a computer, the signal has to pass through an analog-to-digital converter (ADC). This device does 2 things. First it performs a sample-and-hold operation, to measure the signal's value at a certain discrete time. Afterwards, this sample gets quantized, that is: the exact analog value is converted (and possibly rounded) to a digital form. This way, any analog continuous-valued signal can be converted to a sampled discrete-valued signal that is fit for storage and processing by computers.

Frequency domain representation

Next to the time domain representation, we can represent signals in the frequency domain. Fourier proved that any periodic signal can be described as a sum of sinusoids (or cosinusoids, which is equivalent), and that this description is unique. The relation between the time domain and frequency domain representation is then given by the Fourier transform.

This Fourier transform has the drawback that it is actually defined for continuous, infinite signals. When handling finite, digital data, some assumptions have to be made for the Fourier transform to be applicable. These assumptions introduce the artefacts that are inherently part of the discrete Fourier transform.

A first assumption that has to be made is that the signal that we wish to transform is infinite or periodical. When our signal only consists of a limited range of measured values, say a frame of 256 samples, the Fourier transform needs to assume that these 256 samples are the period of a conceptually infinite, periodical signal. Probably this is not the case at all, but there is no way around: the computed transform will be that of an infinite signal with the chosen frame as period. The artefact that this assumption introduces is spectral leakage.

A second assumption that has to be made is that the signal is sufficiently sampled. Luckily, the Nyquist-Shannon sampling theorem provides us with a guideline: if a signal contains a maximal frequency f, then the sampling rate must have a frequency of at least $2f$ in order to have the Fourier transform calculate frequency f correctly. If a signal does contain frequencies that are higher than half the sample rate, these will lead to the artefact of aliasing.

1.1.2 The Fourier transform

CFT: the Continuous Fourier Transform

The continuous Fourier transform is defined as:

$$X(f) = \int_{-\infty}^{+\infty} x(t) e^{-2\pi i f t} dt \tag{1.1}$$

Its inverse is:

$$x(t) = \int_{-\infty}^{+\infty} X(f) e^{2\pi i f t} df \tag{1.2}$$

Note that the signal represented in the Fourier domain, $X(f)$, is generally complex: it has a real and an imaginary part. Most people are familiar with a signal in the Fourier domain being a sum of sines and cosines, but that version is equivalent with this complex exponential version. Both are linked by Euler's famous equation:

$$e^{i\phi} = cos(\phi) + isin(\phi) \tag{1.3}$$

Thus, each component of $X(f)$ is a sinusoid with an amplitude $A(f) = |X(f)|$ and phase $\phi(f) = \angle X(f)$. Because complex numbers allow for faster calculations and easier reasoning than sines and cosines, the complex number representation is the most common in today's signal processing applications. For each frequency, a single complex number can represent both amplitude and phase of the sinusoid of that frequency, and thus fully defines that component of the spectrum.

DFT: the Discrete Fourier Transform

The Discrete Fourier Transform of a discrete signal of N samples is defined as:

$$X[k] = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n] e^{\frac{-2\pi i k n}{N}} \tag{1.4}$$

with k = 0, 1, 2, ... N-1.

Its inverse is:

$$x[n] = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X[k] e^{\frac{2\pi i k n}{N}} \tag{1.5}$$

$\frac{1}{\sqrt{N}}$ is a normalization factor, making the transform what is called unitary. It depends on convention whether you want to use it or not - after all, it is only a multiplication of the result with a scalar. The only requirement is that the product of the normalization factors of both the DFT and inverse

DFT is $1/N$.

If the input signal to a DFT is real, then the DFT of that signal is symmetric as follows:

$$X[k] = X[N-k]^* \tag{1.6}$$

The second half of the transform is the mirrored complex conjugate of the first half of the transform. Therefore, when taking the Fourier Transform of 44100 samples (one second of cd quality sound), we obtain a sequence of which only the first half is useful (the other half is redundant), representing the frequencies from 0 to 22050.

The latter shows the most significant drawback of the Discrete Fourier Transform of a discrete signal: it computes a discrete representation in the frequency domain. Each discrete frequency could be said to form a 'channel', representing also the real frequencies around it. These channels are called 'frequency bins'. A frequency bin describes a frequency range. When having the 22050 resulting numbers of the Fourier representation of our signal, the first number of that resulting sequence corresponds to frequency 0 Hz, a constant. The second number represents frequency 1 Hz, but also the surroundings, so actually [0.5 , 1.5 [, and so up to discrete frequency 22049 Hz or its real counterpart, the bin [22048.5, 22049.5[.

FFT: the Fast Fourier Transform

The Fast Fourier transform is an algorithmic implementation of the discrete Fourier transform. Several FFT algorithms exist, but by far the most popular is the one that was developed by Cooley and Tukey in 1965 [13]. Another is for example the prime factor algorithm by Good and Thomas [14]. Several improvements to FFT algorithms are still being developed and proposed today, see for example [15]. The most used ones are probably those that are currently available through the FFTW library.

1.1.3 Spectral Leakage

In this section I will give a short overview of the phenomenon of spectral leakage. We always work with finite, discrete signals and thus with discrete

Fourier transforms. Spectral Leakage and the subsequent necessity for windowing is very well explained on pp. 536-570 of [16] and on pp. 555-563 of [17].

Problematic Properties of the DFT

The DFT calculates from a given signal of N samples, its frequency domain representation consisting of N/2 discrete frequencies. Those frequencies are equally spaced among the bandwidth of the sampled signal (this bandwidth goes from 0 to the Nyquist frequency). The space between the discrete frequencies depends on the length of the input window. That length also forms the period of the first frequency component of the discrete spectrum. Each component of the spectrum calculated by the DFT therefore does not actually represent one single frequency, but can be seen as a small range of frequencies: a frequency channel. The unit of frequency channels is called 'bin' or 'frequency bin'.

The DFT of a discrete signal thus produces a discrete spectrum. The spectrum consists of a finite number of discrete frequencies, each of which is a multiple of the frequency with a period length the size of the signal. Then, what happens if the frequency of a component of the signal does not nicely coincide with one of the frequencies that is represented in the spectrum? This is illustrated on the next pages, both on one-dimensional and two-dimensional signals.

Spectral leakage in 1D signals

When an input signal f has a frequency that haphazardly coincides with the center of one of the frequency channels, all the energy of that frequency in the signal will concentrate into that one channel. So, the strength of the frequency of the signal is accurately measured. An example is shown here: a signal of 250Hz at a sample rate of 8000 Hz, has exactly 32 periods per second. Taking a 256-point DFT, we compute the strength of a series of frequency bins that are each $4000/128 = 31.25$ Hz wide. The 8th frequency bin is thus centered around the frequency of $8 * 31.25 = 250$ Hz, so it matches exactly the frequency that is present in the input signal.

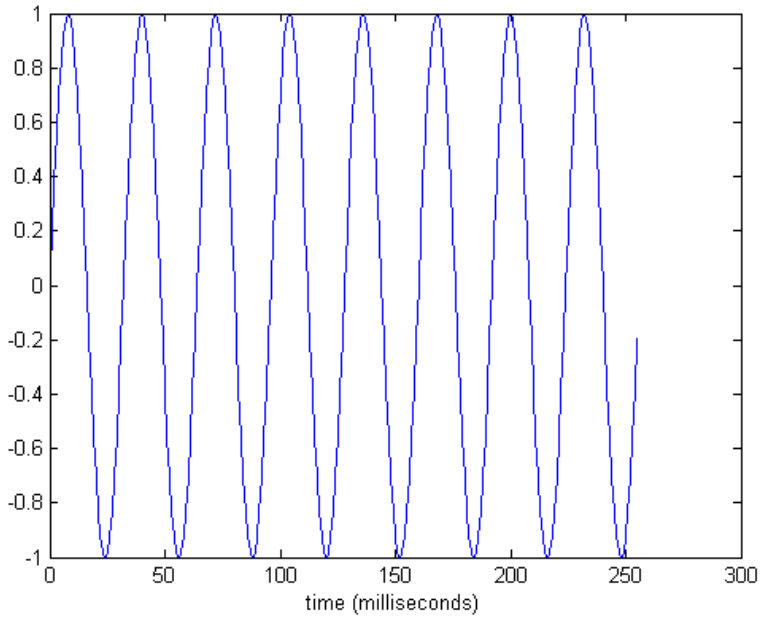

Figure 1.1: 256 samples of a 250Hz signal sampled at 8000Hz

When we lower the frequency of the input signal a little bit, the signal will not coincide anymore with one of the analysis channel centers. Energy of this signal is spread out across all other frequencies, but the largest concentration is still in the neighborhood of the closest channel. The spectrum as we calculate it is less reliable: we see all kinds of frequencies emerge at different strengths, while in reality there is only one frequency present. This phenomenon is called spectral leakage.

Spectral leakage is still manageable when the input signal only contains a few frequencies. But in real life applications, this is rarely the case: most signals consist of an abundance of frequencies, and most of those will not nicely coincide with the center of the analysis channels. The DFT of such a signal is polluted with spectral leakage for almost every occurring frequency. Spectral leakage tends to spread very wide, across the whole spectrum even, and so interferes with other parts of the spectrum.

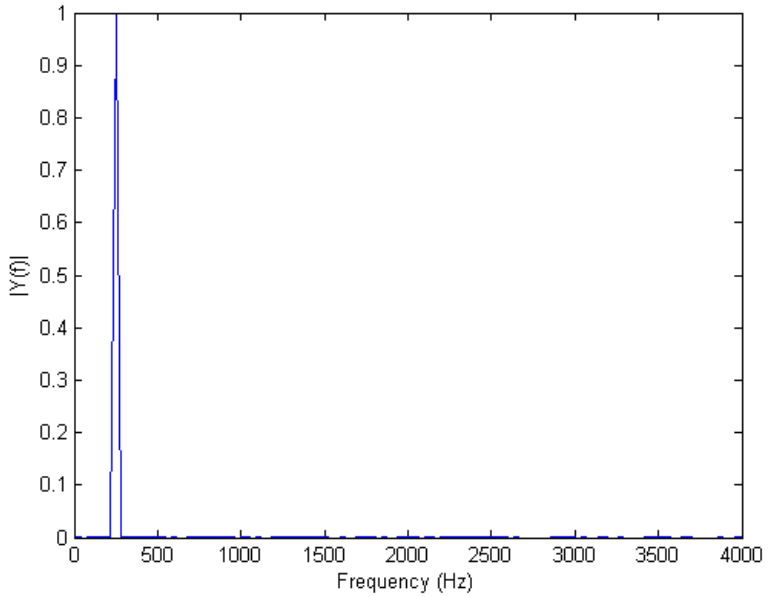

Figure 1.2: The 256-point DFT of Fig. 1.1

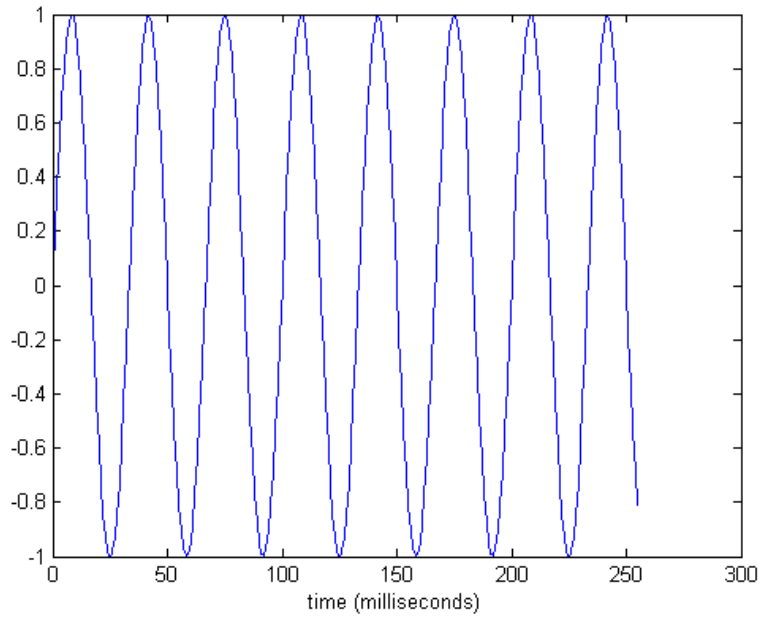

Figure 1.3: 256 samples of a 240Hz signal sampled at 8000Hz

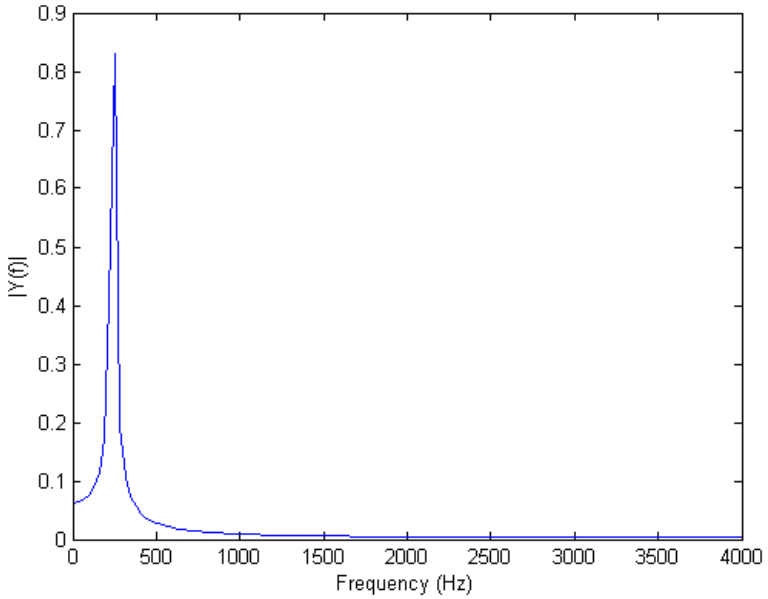

Figure 1.4: The 256-point DFT of Fig. 1.3

Spectral leakage in 2D signals

An image can be viewed as a two-dimensional finite signal. Reworking the previously presented digital signal processing methods to more dimensions would bring us too far here, but a light introduction on the DFT of images and filtering can be found in [18].

In image processing, a clear Fourier transform with as few leakage as possible is also advantageous: it focuses the attention on those spectral components that are really important in the image, and thus can be of use when designing filters for the frequency domain, or even editing the frequency domain.

Figure 1.5 consists of a series of black and white bands, rotated 45 degrees. It can be proved that the Fourier transform of a rotated image corresponds to the rotated Fourier transform of the original non-rotated image. So we expect the Fourier transform (figure 1.6) to consist of a periodic series of points perpendicular on the band direction. The points are equally spaced and seem to go on infinitely. However, when computing this Fourier transform, the resulting spectrum is a true work of art (figure 1.7).

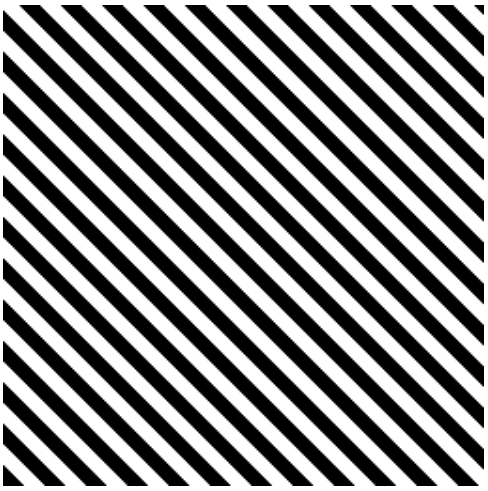

Figure 1.5: Test image

Figure 1.6: The expected FFT of Fig. 1.5

In figure 1.7, we can still distinguish the most important components of the signal: the clear white points on the diagonal. The spectral leakage formed is visible as the geometric patterns emanating from these points. But in bad lighting, unclearly printed, or with a bad monitor, it could be difficult to point out the diagonal band of white points as the most significant data in the calculated spectrum. Also, software that would rely on this computed spectrum for further processing could be fooled into thinking that the other geometric patterns are of more importance.

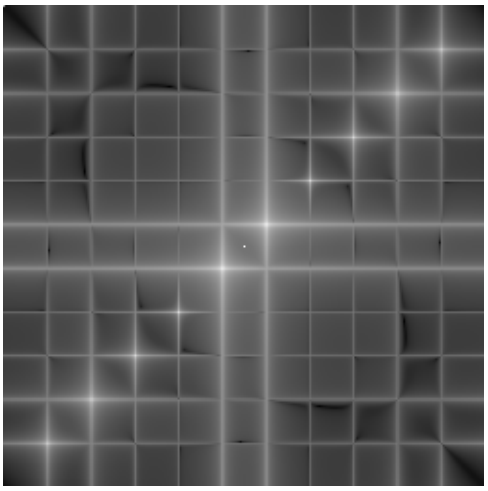

Figure 1.7: The calculated FFT of Fig. 1.5

The cause of spectral leakage

If we still want to make a trustworthy analysis of a spectrum of such a signal, we have to devise methods to reduce the spectral leakage. Therefore we must first analyse the cause of the leakage. The cause is that the DFT assumes that the input signal is infinite and periodic. That is, it will not calculate the spectrum of only the finite input signal, but rather the spectrum of the input signal that is repeated eternally. This assumption is inherited from the definition of the continuous Fourier transform: the continuous Fourier transform is defined on infinite signals.

In practical terms: the perfect transform, like in figure 1.2 , can only be computed if the finite input signal only consists of components with a positive integer number of complete periods. In our example one-dimensional signal this would be 8 complete periods. In Figure 1.3 the only signal component that is present does not contain an integer number of periods. The transform calculated will actually be the transform of an endless stream of these short signals.

As the beginning and the end of the signal do not coincide (as is the case when a positive integer number of periods is reached), there will be glitches at those points where the signal is glued together. These glitches, where the signal restarts before being able to complete its component's periods, are responsible for the leakage: to make up for the glitches, energy is added to lower and higher frequencies around the component's real frequency.

In the 2-dimensional case, this can easily be visualized. Consider figure 1.8, where this assumption of infinity has been worked out. It is clearly visible that the image is not continued through the horizontal and vertical bands, but is just tiled. The tiles introduce irregularities in the image that are visible in the spectrum as leakage.

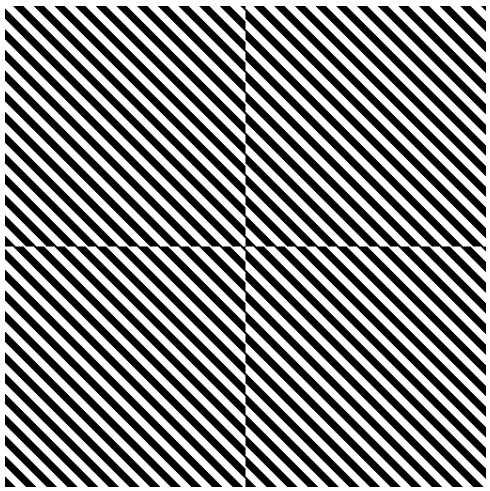

Figure 1.8: The reason of leakage: glitches when repeating

1.1.4 Window Functions

The solution is quite simple: if the problem lies with the beginning and the end of the finite signal not fitting neatly together, just make them fit neatly together. We can apply several window functions (also called apodization functions) to a signal, to make sure that the signal converges to 0 at the beginning and the endpoints, thus making the ends fit nicely together when the signal is repeated. This reduces most spectral leakage, but unfortunately also has side effects: by multiplying the signal with a window function, we are actually computing the DFT of another function than the real signal. A very recent and comprehensive starter text on windowing is [19].

What are window functions?

Window functions are special functions that a signal has to be multiplied with prior to the calculation of the DFT. They can have any form or shape,

as long as they are defined within the borders of the signal. Several types of window functions exist, and most of them are very well explained in the very significant papers by Harris [20] and Nuttall [21]. Multiplying a signal with a window function has the effect that the signal is "squeezed" into the frame defined by the window function.

When a signal is Fourier transformed without a window function having been applied, actually, a rectangular window function has been applied. It is bordered at the left and right hand sides by the starting point and endpoint of the signal, and has a constant value of 1.

Properties and examples

Multiplication by a window function in the time domain, corresponds to convolution with the Fourier transform of the window in the frequency domain. In order to view the actual spectrum of the signal, we want a convolution with another function that resembles as much as possible the Dirac delta function (a single peak with value 1 at the origin, the other values 0). Convolution with that function is an identity operation of the convolution operator.

And immediately, it becomes clear why a rectangular window (or 'no window') is not so good a choice for a window function. A rectangular window has as its Fourier transform a sinc function, which does not resemble the Dirac delta function at all:

$$sinc(x) = \frac{sin(x)}{x} \tag{1.7}$$

And even worse, this function has a very slow asymptotical behaviour on the sides (in this context also called the sidelobes), so that even far away from the origin, the function still has relatively significant values. In 2-dimensional signal processing, this gives rise to the effect of ringing.

The situation is much better for the 3-term Blackman function, which is defined as :

$$w(n) = 0.42 - 0.5cos(\frac{2\pi n}{N-1}) + 0.08cos(\frac{4\pi n}{N-1}) \tag{1.8}$$

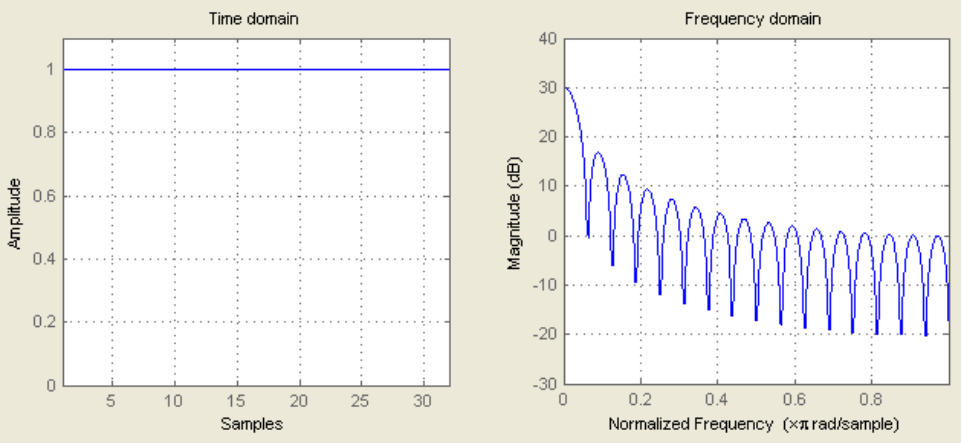

Figure 1.9: The rectangular window and its FFT

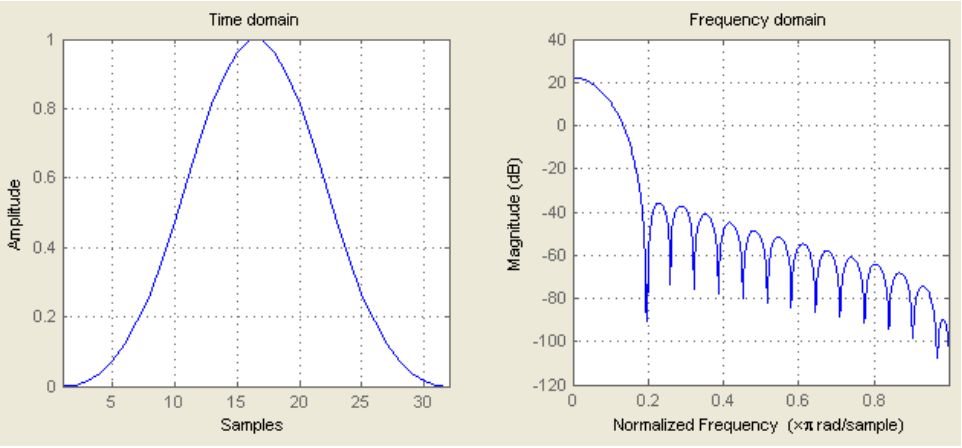

Figure 1.10: The Blackman window and its FFT

The result of multiplying our 240Hz example with this window function is shown in Fig. 1.11. The DFT of the windowed signal is shown in Fig. 1.12. Compared to the DFT of the non-windowed (or rectangular-windowed) function, this one is much cleaner, and only contains a significant amount of spectral energy very close around the real frequency.

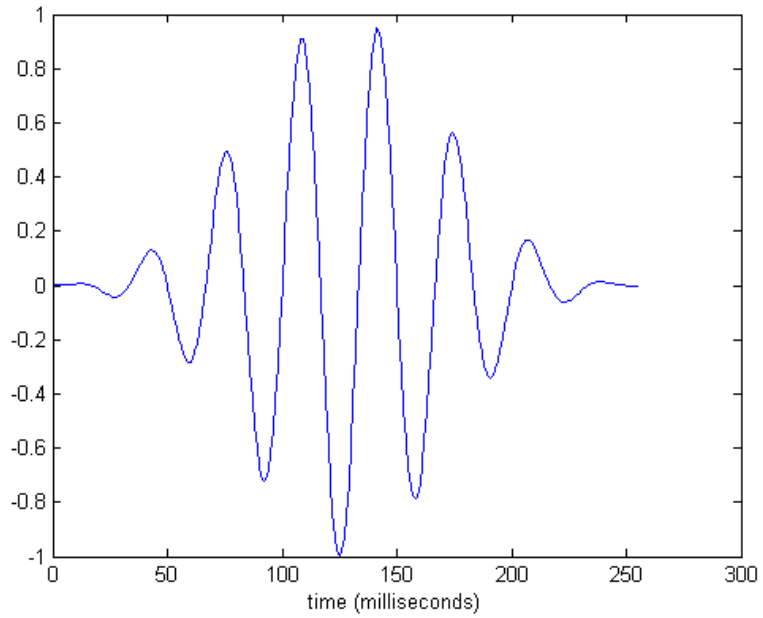

Figure 1.11: Fig. 1.3 with a Blackman window applied

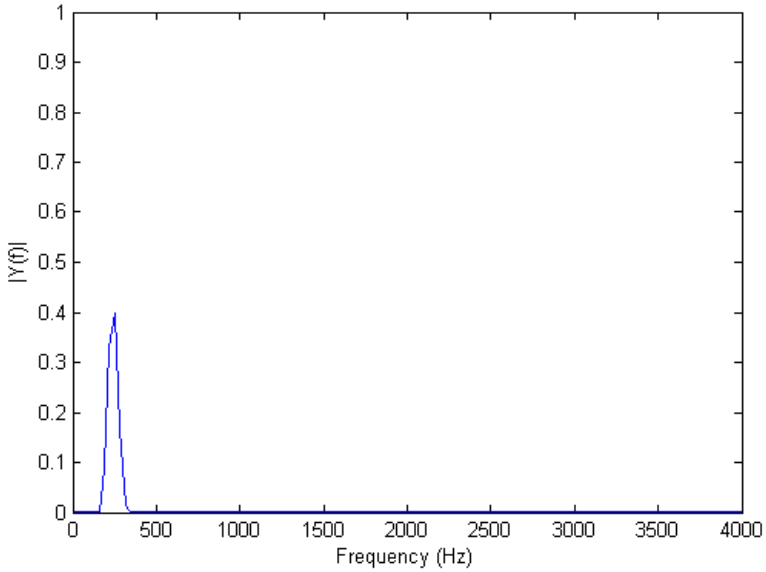

Figure 1.12: The 256-point DFT of Fig. 1.11

The difficult trade-off

Applying a window function has its implications, though. The original signal, at the endpoints of the time frame, is being reduced to nearly 0. This means that a whole lot of information on the signal is lost. In the DFT of the windowed signal, this results in a loss of spectral resolution: an error is introduced in the input for the DFT calculations and the resulting spectrum is "blurred". For well windowed signals, the spectrum will contain almost no leakage, but the spectral peaks that are present will be broader and more spread out to the nearest frequency bins.

In general there is always a trade-off to make when choosing a window function, between making the spectrum peaks as narrow as possible, and making the outer sides of the leakage fall off as fast as possible. There exists a whole range of windows functions, each with their own properties. The most important ones that are still used today are listed in [20] and [21].

Which one is best to use really depends on the situation: the signal and the user. For one application, one might want to use a Kaiser window with a low or high coefficient, while for a really large signal, a Blackman window is computationally much more interesting, and yields nearly the same results. Depending on the signal, a Bartlett window might give an excellent result, while a Gaussian window really messes up the spectrum. In most professional applications, it is customary to offer the user a choice between different windows, so that he can compare the differences him/herself and choose the one that provides him/her with the most satisfactory results.

The Blackman-Harris window function

Nuttall, Blackman-Harris and Blackman-Nuttall windows are all variations on the same theme: they are the same function but they differ in constants. This function is:

$$w(n) = a_0 - a_1 cos\frac{2\pi n}{N-1} + a_2 cos\frac{4\pi n}{N-1} - a_3 cos\frac{6\pi n}{N-1}; \qquad (1.9)$$

The coefficients are:

For Nuttall: $a_0 = 0.355768$, $a_1 = 0.487396$, $a_2 = 0.144232$, $a_3 = 0.012604$.
For Blackman-Harris: $a_0 = 0.35875$, $a_1 = 0.48829$, $a_2 = 0.14128$, $a_3 = 0.01168$.

For Blackman-Nuttall: $a_0 = 0.3635819$, $a_1 = 0.4891775$, $a_2 = 0.1365995$, $a_3 = 0.0106411$.

As such they form an extension to the Blackman window that we have seen earlier. The constants are different for each of these windows. The differences are minimal, but the effects are quite large. I won't elaborate on them, though, since that would bring us too far outside the scope of this thesis.

Further in this thesis, especially the Blackman-Harris window will be used very often. The reason for that is that of all windows that are currently known, it has the best spectral leakage suppression properties (as also described in [20]. This property will be of extreme importance when reviewing the algorithms of D'haes from [5]. The loss of spectral resolution is an acceptable disadvantage, because optimization algorithms are used for finding the real frequencies anyway.

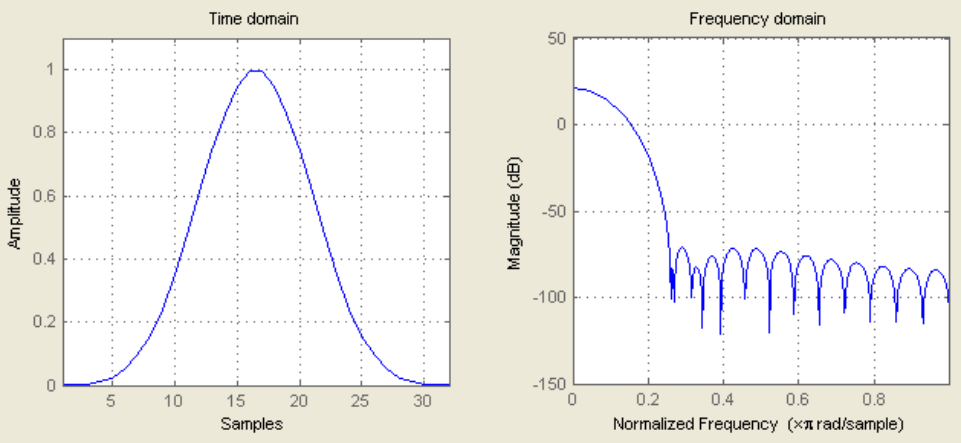

Figure 1.13: The Blackman-Harris window and its FFT

1.2 Polyphonic Audio

To better understand what will come, it is necessary to introduce the basics of classical western music theory.

1.2.1 Terminology

When a musical instrument like a piano makes a sound, it makes the air vibrate with a certain frequency. The human ear is able to sense these vibrations and perceive the tone height of the sound, which is called a pitch. When placing the note in its musical context (a tonality), it is called a tone. When written down on paper, it is called a note.

Instruments may produce a sound at the same pitch, but it may sound completely different: the timbre of all musical instruments differs. The reason for this is that the generated sound consists of not only one fundamental frequency but also of a series of harmonics: multiples of that fundamental frequency. Differences in how the amplitudes of the harmonics relate to each other can make us tell the instruments apart.

A generated sound need not be pitched. Percussion instruments like drums, generate sounds that will appear as consisting of a vast range of frequencies, however, none of these frequencies relate to eachother in a distinguishable way, or pop out as special. These sounds are unpitched or noisy. As with pitched sounds, also different kinds of noise exist. Often they are named like colors: white noise for example has equal power at all frequencies of the spectrum - a uniform spread.

The distance between two distinct pitches is an interval. In western music the basic interval is a semitone. More than 2 pitches that function together, form a chord. A scale is an ordered sequence of notes serving as the basis for any musical piece. Scales come in different modes depending on the intervals between the notes. Eastern music often works with pentatonic scales, consisting of only 5 notes. Indian raga music or arabian maqamat music uses scales that have intervals smaller than the traditional western semitone. Also in modern western music, large scales that work with microtones have been developed.

1.2.2 Classical Western Music Theory

Several musical theories exist that define rules on how the notes and chords should relate to each other in music. This is very much a matter of taste, more than physics or math. For a western ear, arabic music sounds very much out of tune, and vice versa. To keep things simple, I will only explain the basics of classical western music theory here, as developed and used around the years 1600-1900. The fact that western people are so used to it and that it is relatively simple, makes also popular western music mostly adhere to this classical system - albeit with a little more freedom, like jazz influences.

Classical western music uses a 7 note so-called diatonic scale as basis. Each note of this scale gets a functional name, the most important ones being the tonic (1st note), the subdominant (4th note) and the dominant (5th note). On top of every note, one can build a triad chord consisting of 2 thirds (a third is an interval consisting of 3 - a minor third - or 4 - a major third - semitones). These chords are designated with a roman number, indicating the position of the note on which the chord is built in the scale: the triad on the tonic, subdominant and dominant are designated as I, IV, V respectively.

Each classical western piece of music has an underlying harmonic structure that progresses with time, which is traditionally described in the form of chord progressions: sequences like I - IV - I - IV - V - I . At any time, the notes of the current chord serve as guide notes for the music that has been built upon it. These notes will often occupy the most important places in the music, like the beginning of a measure or the bass line. There are rules for connecting the different chords together, but the composer is of course free to make variations, additions, elaborations, omissions, decorations, permutations and combinations of several elements, at his own taste.

As long as the underlying harmonic structure is distinguishable, everything still sounds correct to our ears. But there is a lot of overlap: different chords can have several notes in common (I and IV have the tonic in common, I and V have the dominant in common). So, seeing a tonic, is the underlying chord a I or a IV? The next note, is it part of a real basic chord or is it just a decoration of the composer to fill up an empty space? This must be derived from the context, and this context can take many forms. As can be seen, in even the simplest examples of western music theory, lots of ambiguities exist. Also, each composer has his own style, which makes developing automated harmonical analysis algorithm a daunting task - though sometimes attempts are being made, like [22].

1.2.3 Musical Harmony and Mathematics

Traditional western music is very much related to mathematics. Pythagoras is credited with the discovery that, when the string length of a stringed instrument is halved, the resulting note sounds exactly one octave (one full scale span) higher. Also, the string length ratio between tonic and dominant was found to be 3/2, producing a perfect fifth interval, and between the tonic and subdominant 4/3, producing a perfect fourth interval. By artificially stacking 3/2 string ratio's on top of each other, the so-called circle of fifths, a 12-tone scale can be constructed. When reordered into one octave, a scale consisting of 12 semitones appears. The circle of fifths forms the basis of classical harmony, as can be seen in our example: the chords IV - I - V each have an interval of a perfect fifth between them.

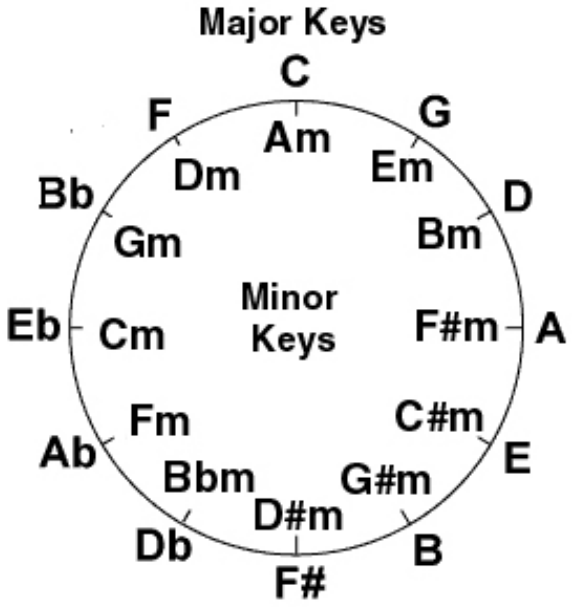

Figure 1.14: The Circle of Fifths

There is a good reason to choose the 3/2 interval as basis. As seen earlier, each instrument that plays a note, does not only produce a fundamental frequency but also harmonics: frequencies that are an integer multiple of the fundamental frequency. The first, and often most powerful, harmonics of a 100 Hz tone are tones of 200 Hz (mark the ratio 2:1, an octave), 300 Hz (mark the ratio 3:2, a fifth), and 400 Hz (mark the ratio 4:3, a fourth).

When playing a note of 100Hz and the note 3/2 above it, 150 Hz, several harmonics of these notes coincide with each other and reinforce each other: the 300 Hz harmonics and their multiples. This appears to humans as pleasingly consonant. In general, when the mathematical relationship between physical frequencies is a very simple ratio, so a lot of harmonics coincide, these tones are regarded as consonant or well sounding.

There is one problem however: stacking 3/2 ratios on top of each other, does not lead us back to the beginning. The whole cycle of fifths, 12 intervals in total, should encompass exactly 7 full octaves. Suppose we start from a frequency of 1 Hz, or 2^0, we expect to come out at 128 Hz, or 2^7. But, $(3/2)^{12} = 129.7463...$ Result: a reordered cycle of fifths creates a nice approximation to the 12-semitone octave, but is far from exact. To solve the problem, in a Pythagorean tuning, the last frequency of 129.7463 Hz is quite brutely replaced with what it should be ideally: 128 Hz.

If every fifth interval is in perfect tune except the last one (the so called wolf interval), this exception can of course be heard by the audience. Ancient composers simply did not use it, but by the end of the Renaissance this rude 'patch' was no longer satisfiable. Several new tunings for the 12-semitone were developed, among others by Johannes Kepler. Nowadays, a so called equal tempered tuning is used: the distances between the 12 semitones are taken to be equal, each with a factor $2^{1/12}$ different from the previous semitone.

The equal tempered tuning however, breaks the Pythagorean string ratios. A perfect fifth interval, 7 semitones wide, does not have a ratio of 3/2 but one of $2^{7/12}/2 = 2.9966.../2$. However, this difference is so small that it is only distinguishable for the well trained ear, or for people with a good absolute hearing. The other perfect interval, the forth, has the same problem. For the fundamental frequency, this discrepancy is still small, but in the harmonics of each tone, which are multiples of the fundamental frequencies, these small inexactnesses are also multiplied and become stronger.

On certain instruments which produce lots of strong harmonics, like some kinds of flutes, the increasing differences in harmonics between two sounding tones can give rise to an effect known as phantom fundamentals: at a certain point, the divergence between harmonics is so large that the brain cannot tell any more which harmonic belongs to which fundamental frequency, and so it assumes that another, more fitting fundamental should be present to explain these harmonic ratios: we hear notes that are not there.

The classical western harmony has firm roots in mathematics, but also on a higher level, classical musical structures often have a very strict definition that could be seen as mathematically. Mathematics has often been of inspiration to composers that were developing musical structures, sometimes even providing in raw material like melodies. The introduction of computers allowed calculation-intensive mathematics to be used in compositions, pioneered by, amongst others, Xenakis. The correspondences between mathematics and music can also be analyzed from a more philosophical point of view, the most known work probably being the popular-scientific book by Hofstadter on Gödel, Escher and Bach [23].

1.2.4 Harmony? Polyphony?

The frame in which we will work further in this thesis still needs to be sketched. Therefore, we need some more definitions and assumptions.

The very fact that a sinusoidal model will be used, implies that the described system is only useful for sounds that are harmonic. That is, every tone that is present must be pitched, which means that every tone has a fundamental frequency and a series of harmonics. We will assume that any frequencies that are present in the sound and are not directly related to one of the tones, are just noise components.

Polyphony is attained when the tones that are present in the sound, are interrelated in a consonant way. The definition of consonance is very subjective, and so is actually polyphony. A relatively reliable metric of consonance could be the number of coinciding harmonics in the sounding tones, give or take a certain error induced by well-tempered tuning.

1.3 Sinusoidal Modelling

1.3.1 A sinusoidal model

For a signal containing one or more tones with a limited number of frequencies and corresponding amplitudes, storing the signal using all time information or frequency information is overkill. Instead, we can reconstruct the whole signal if we only store those few frequencies and their amplitudes, that are actually present in the signal. There is hardly any use in storing the 0 parts of the spectrum, they do not contribute to the signal in any way.

A signal is composed of a sum of sinusoids (or cosinusoids, which is esentially the same), each with their own frequency, amplitude and phase. Then we can approximate any real signal x_n of length N with K components, by defining a sinusoidal model $\tilde{x}_n$ as follows:

$$\tilde{x}_n = \sum_{k=0}^{K-1} a_k \cos\left(\frac{2\pi\omega_k n}{N} + \phi_k\right) \tag{1.10}$$

The window that is applied to the signal has its center at the origin of the timescale. Also, to compute the DFT of a signal, we need to center the signal around the origin of the timescale. This will ensure that the computed spectrum is not phase shifted. Centering the signal happens by substituting $n - n_0$ for n, where $n_0 = \frac{N-1}{2}$.

So, the windowed and centered sinusoidal model of a signal can be written as:

$$\tilde{x}_n = w_n \sum_{k=0}^{K-1} \left(a_k cos\left(2\pi\omega_k \frac{n - n_0}{N} + \phi_k\right)\right) \tag{1.11}$$

We are mainly concerned with finding the best frequencies ω_k and corresponding amplitudes a_k, so that $\tilde{x}_n$ approximates x_n the best.

Using Euler's Formula

$$e^{i\phi} = cos\phi + isin\phi \tag{1.12}$$

we can reformulate the model using complex exponentials. These complex exponentials are what is actually computed by most FFT routines, and they allow for more elegant reasoning in the Fourier domain.

The cosine translates to complex exponentials as:

$$cos(x) = \frac{e^{ix} + e^{-ix}}{2} \tag{1.13}$$

This is directly derivable from Euler's Formula. Substituting this in eq. 1.11 yields:

$$\begin{aligned}
\tilde{x}_n &= w_n \sum_{k=0}^{K-1} \left(a_k \frac{e^{i\left(2\pi\omega_k \frac{n-n_0}{N} + \phi_k\right)} + e^{-i\left(2\pi\omega_k \frac{n-n_0}{N} + \phi_k\right)}}{2} \right) \\
&= \frac{1}{2} w_n \sum_{k=0}^{K-1} \left(a_k e^{\left(\frac{2\pi i\omega_k (n-n_0)}{N} + i\phi_k\right)} + a_k e^{\left(\frac{-2\pi i\omega_k (n-n_0)}{N} - i\phi_k\right)} \right) \\
&= \frac{1}{2} w_n \sum_{k=0}^{K-1} \left(a_k e^{\frac{2\pi i\omega_k (n-n_0)}{N}} e^{i\phi_k} + a_k e^{\frac{-2\pi i\omega_k (n-n_0)}{N}} e^{-i\phi_k} \right)
\end{aligned} \tag{1.14}$$

The complex amplitude of a sinusoid is defined as: $A_l = a_l e^{i\phi_l}$. It contains information about both the amplitude and the phase of a sinusoid. Substitution in the last equation yields:

$$\tilde{x}_n = \frac{1}{2} w_n \sum_{k=0}^{K-1} \left(A_k e^{\frac{2\pi i\omega_k (n-n_0)}{N}} + A_k^* e^{\frac{-2\pi i\omega_k (n-n_0)}{N}} \right) \tag{1.15}$$

where A_k^* denotes the complex conjugate of A_k.

Every complex amplitude A_l can be split up into its real and imaginary components: $A_l = A_l^r + iA_l^i$. Using Euler's Formula, the real and imaginary components of the complex amplitude relate to the amplitudes of the sinusoidal representation as:

$$\begin{aligned}
A_l^r &= a_l cos(\phi_l) \\
A_l^i &= a_l sin(\phi_l)
\end{aligned}$$

It is now easy to go back to the first definition of our model in Eq 1.10, taking the complex amplitude with us. Also remind that $cos(-x) = cos(x)$ and $sin(-x) = -sin(x)$. Starting from Eq 1.15, we find:

$$\begin{aligned}
\tilde{x}_n &= \frac{1}{2} w_n \sum_{k=0}^{K-1} \left((A_k^r + iA_k^i) e^{\frac{2\pi i \omega_k (n-n_0)}{N}} + (A_k^r - iA_k^i) e^{\frac{-2\pi i \omega_k (n-n_0)}{N}} \right) \\
&= \frac{1}{2} w_n \sum_{k=0}^{K-1} \left((A_k^r + iA_k^i) \left(cos\left(2\pi\omega_k \frac{n-n_0}{N} \right) + isin\left(2\pi\omega_k \frac{n-n_0}{N} \right) \right) \right. \\
&\quad \left. + (A_k^r - iA_k^i) \left(cos\left(\frac{-2\pi\omega_k (n-n_0)}{N} \right) + isin\left(\frac{-2\pi\omega_k (n-n_0)}{N} \right) \right) \right) \\
&= \frac{1}{2} w_n \sum_{k=0}^{K-1} \left(2A_k^r cos\left(2\pi\omega_k \frac{n-n_0}{N} \right) - 2A_k^i sin\left(2\pi\omega_k \frac{n-n_0}{N} \right) \right) \\
&= w_n \sum_{k=0}^{K-1} \left(A_k^r cos\left(2\pi\omega_k \frac{n-n_0}{N} \right) - A_k^i sin\left(2\pi\omega_k \frac{n-n_0}{N} \right) \right) \qquad (1.16)
\end{aligned}$$

Minimizing the error

The eventual goal is to make the model resemble the real input as good as possible. To express this, we define an error function $\chi(\overline{A}; \overline{\omega})$, as the square difference between the real signal and its model. Given a set of frequencies, which are obtained by an iterative algorithm, the error function is defined with a vector of complex amplitudes as variables. This error needs to be minimized. This involves calculating the partial derivatives and putting them to zero.

$$\chi(\overline{A}; \overline{\omega}) = \sum_{n=0}^{N-1} (x_n - \tilde{x}_n)^2 \qquad (1.17)$$

Calculating the frequencies must happen through an optimization process. Since we only have a discrete spectrum at our disposal using the DFT, we have no knowledge of the real frequencies that are around. We have to

start with a crude sinusoidal model containing some estimate of the frequencies that are present, which can be chosen either manually by peak picking, or automatically by some kind of spectrum analysis. We can calculate the amplitudes and phases (taken together in the complex amplitude) that make this crude model resemble the signal the best way possible, through a process of amplitude estimation.

Afterwards, we can perform an optimization of the chosen frequencies, to make the sinusoidal model converge to the real situation. At every optimization step, the amplitudes need to be re-estimated, so we can see whether we are optimizing in the correct direction. In the next sections of this thesis, we will view in detail how both amplitude estimation and frequency optimization can be carried out, making full use of theoretical results involving windowing, that allow for significant speedup.

The reason to choose for an iterative optimization method instead of a quadratic interpolation of the spectrum in order to find the correct frequencies, is that we wish to handle overlapping frequency responses: several frequencies very close together influence each other in the discrete spectrum, making quadratic interpolation yielding useless results. Several optimization methods can be used for this kind of task, among others gradient descent, Newton's method, or Gauss-Newton optimization.

1.3.2 Amplitude Estimation

Once a new set of frequencies has been calculated in a next iteration of the frequency optimization routine, we have to calculate the amplitudes of all those frequencies, in order to make the model resemble the input as good as possible.

The mathematical workout

Since our amplitudes are complex numbers which can be separated into their real and imaginary parts (see eq. 1.16), we can view those parts as separate variables. Thus, we can minimize the error by putting all partial derivatives with respect to one of the unknown complex amplitude components to 0:

$$\frac{\partial \chi(\overline{A};\overline{\omega})}{\partial A_l^r} = 0$$
$$\frac{\partial \chi(\overline{A};\overline{\omega})}{\partial A_l^i} = 0 \tag{1.18}$$

Working out the first equation from the previous formulas 1.17 and 1.16, this comes down to calculating the following partial derivative with respect to A_l^i:

$$\frac{\partial}{\partial A_l^r} \sum_{n=0}^{N-1} \left(x_n - w_n \sum_{k=0}^{K-1} \left(A_k^r cos\left(2\pi\omega_k \frac{n-n_0}{N} \right) - A_k^i sin\left(2\pi\omega_k \frac{n-n_0}{N} \right) \right) \right)^2 \tag{1.19}$$

Working out the inner brackets gives:

$$\frac{\partial}{\partial A_l^r} \sum_{n=0}^{N-1} \left(x_n - w_n \sum_{k=0}^{K-1} A_k^r cos\left(2\pi\omega_k \frac{n-n_0}{N} \right) + w_n \sum_{k=0}^{K-1} A_k^i sin\left(2\pi\omega_k \frac{n-n_0}{N} \right) \right)^2 \tag{1.20}$$

And putting all summations in front, we get the following to solve:

$$\frac{\partial}{\partial A_l^r} \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} \left(x_n - w_n A_k^r cos\left(2\pi\omega_k \frac{n-n_0}{N} \right) + w_n A_k^i sin\left(2\pi\omega_k \frac{n-n_0}{N} \right) \right)^2 \tag{1.21}$$

Using the well known chain rule, we first work out the exponent, and then multiply by the derivative of the base:

$$\begin{aligned} \frac{\partial \chi}{\partial A_l^r} &= 2 \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} \left(x_n - w_n A_k^r cos\left(2\pi\omega_k \frac{n-n_0}{N} \right) + w_n A_k^i sin\left(2\pi\omega_k \frac{n-n_0}{N} \right) \right) \\ & \quad w_n cos\left(2\pi\omega_l \frac{n-n_0}{N} \right) \end{aligned} \tag{1.22}$$

The reason that the derivative of the base is only a cosine function, is that the term $w_n A_k^r cos\left(2\pi\omega_k \frac{n-n_0}{N}\right)$ can be regarded as a constant for all $k \neq l$, and thus it can be discarded in derivation. The same goes for all A^i, since we are deriving with respect to A_l^r. For $k = l$, A_l^r is just a normal variable, thus $w_n A_l^r cos\left(2\pi\omega_l \frac{n-n_0}{N}\right)$ can be regarded as a linear function, the derivative to A_l^r being $w_n cos\left(2\pi\omega_l \frac{n-n_0}{N}\right)$.

This partial derivative needs to be put to 0. We can do away with the factor 2 in front, since this is only a constant. The multiplication that resulted from the chain rule can be worked out and put inside the summation:

$$\begin{aligned} 0 \;=\; & \sum_{n=0}^{N-1}\sum_{k=0}^{K-1}\left(x_n w_n cos\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right. \\ & -w_n^2 A_k^r cos\left(2\pi\omega_k\frac{n-n_0}{N}\right) cos\left(2\pi\omega_l\frac{n-n_0}{N}\right) \\ & \left.+w_n^2 A_k^i sin\left(2\pi\omega_k\frac{n-n_0}{N}\right) cos\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right) \end{aligned} \tag{1.23}$$

Note that this yields that in some terms, the window function w_n gets squared. This will be a useful property in the future. Splitting up the summations and reordering the terms, the requirement that the partial derivative be 0 comes down to the following result:

$$\begin{aligned} & \sum_{k=0}^{K-1}\left(A_k^r\sum_{n=0}^{N-1}\left(w_n^2 cos\left(2\pi\omega_k\frac{n-n_0}{N}\right) cos\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right)\right) \\ - & \sum_{k=0}^{K-1}\left(A_k^i\sum_{n=0}^{N-1}\left(w_n^2 sin\left(2\pi\omega_k\frac{n-n_0}{N}\right) cos\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right)\right) \\ & = \sum_{n=0}^{N-1}\left(x_n w_n cos\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right) \end{aligned} \tag{1.24}$$

For the partial derivatives with respect to the imaginary parts of the complex amplitudes, the calculations are analog, yielding the following result:

$$
\begin{aligned}
&-\sum_{k=0}^{K-1}\left(A_k^r\sum_{n=0}^{N-1}\left(w_n^2 cos\left(2\pi\omega_k\frac{n-n_0}{N}\right)sin\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right)\right)\\
&+\sum_{k=0}^{K-1}\left(A_k^i\sum_{n=0}^{N-1}\left(w_n^2 sin\left(2\pi\omega_k\frac{n-n_0}{N}\right)sin\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right)\right)\\
&\qquad\qquad = -\sum_{n=0}^{N-1}\left(x_n w_n sin\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right) \qquad (1.25)
\end{aligned}
$$

For a model with K components, we have for each of those components these 2 equations to solve: one for the real and one for the imaginary part of the complex amplitude of that component. Having 2K equations in 2K unknowns, this can be written in matrix form: a 2K by 2K matrix with 2K unknowns. We define the following K by K submatrices B, the vector of unknowns A and the constant terms C as follows:

$$
\begin{aligned}
B_{l,k}^{1,1} &= \sum_{n=0}^{N-1}\left(w_n^2 cos\left(2\pi\omega_k\frac{n-n_0}{N}\right)cos\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right)\\
B_{l,k}^{1,2} &= -\sum_{n=0}^{N-1}\left(w_n^2 sin\left(2\pi\omega_k\frac{n-n_0}{N}\right)cos\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right)\\
B_{l,k}^{2,1} &= -\sum_{n=0}^{N-1}\left(w_n^2 cos\left(2\pi\omega_k\frac{n-n_0}{N}\right)sin\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right)\\
B_{l,k}^{2,2} &= \sum_{n=0}^{N-1}\left(w_n^2 sin\left(2\pi\omega_k\frac{n-n_0}{N}\right)sin\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right)\\
C_l^1 &= \sum_{n=0}^{N-1}\left(x_n w_n cos\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right)\\
C_l^2 &= -\sum_{n=0}^{N-1}\left(x_n w_n sin\left(2\pi\omega_l\frac{n-n_0}{N}\right)\right) \qquad (1.26)
\end{aligned}
$$

Here, k is the index of the component (the row index in B), and l the index of the component with respect to which the partial derivative was taken (the column index in B, and therefore also the row index in A and C). Hence, the problem of finding the amplitudes of the given frequencies now comes down to solving the 2K by 2K matrix equation:

$$\begin{bmatrix} B_{l,k}^{1,1} & B_{l,k}^{1,2} \\ B_{l,k}^{2,1} & B_{l,k}^{2,2} \end{bmatrix} \begin{bmatrix} A_l^r \\ A_l^i \end{bmatrix} = \begin{bmatrix} C_l^1 \\ C_l^2 \end{bmatrix} \tag{1.27}$$

The D'haes Optimization

In order to show how this matrix can be set up quickly, the behavior of the sinusoidal model in Fourier space must be investigated. Recall the definition of a sinusoidal model in equation 1.15 . Taking the Fourier transform of this model, gives us the spectrum of the model:

$$\tilde{X}_m = \frac{1}{2} \sum_{n=0}^{N-1} \left(w_n \left[\sum_{k=0}^{K-1} \left(A_k e^{2\pi i \omega_k \frac{n-n_0}{N}} + A_k^* e^{-2\pi i \omega_k \frac{n-n_0}{N}} \right) \right] e^{2\pi i m \frac{n-n_0}{N}} \right) \tag{1.28}$$

Working out the inner brackets and the multiplication of exponentials:

$$\tilde{X}_m = \frac{1}{2} \left(\sum_{k=0}^{K-1} \left(A_k \sum_{n=0}^{N-1} \left(w_n e^{-2\pi i (m-\omega_k) \frac{n-n_0}{N}} \right) \right) + \sum_{k=0}^{K-1} \left(A_k^* \sum_{n=0}^{N-1} \left(w_n e^{-2\pi i (m+\omega_k) \frac{n-n_0}{N}} \right) \right) \right) \tag{1.29}$$

Which, for ease of use, we can rewrite as:

$$\tilde{X}_m = \frac{1}{2} \left(\sum_{k=0}^{K-1} \left(A_k W(m-\omega_k) + A_k^* W(m+\omega_k) \right) \right) \tag{1.30}$$

with

$$W(m) = \sum_{n=0}^{N-1} \left(w_n e^{-2\pi i m \frac{n-n_0}{N}} \right) \tag{1.31}$$

This last substitution comes in very handy: W(m) is the definition of the Fourier transform of the window w(n). Thus, it is proved that the DFT of a sinusoidal model is actually a linear combination of DFTs of the window

function that is used, shifted over ω_k and weighted by complex amplitude A_k. This is a major advantage: the Fourier transform of the Blackman-Harris window function is very strongly bandlimited, as can be seen in figure 1.13, with the sidelobes having an attenuation of -92dB [20]. Though squaring the window function enlarges the bandwidth of the main lobe, the squared window also still is bandlimited. We can define the Fourier transform of the squared window as follows:

$$Y(m) = \sum_{n=0}^{N-1} \left(w_n^2 e^{-2\pi i m \frac{n-n_0}{N}} \right) \tag{1.32}$$

The normal and squared window functions and their DFTs are known beforehand and can be implemented in software using scaled and oversampled lookup tables, so these values can be regarded as constants in calculations.

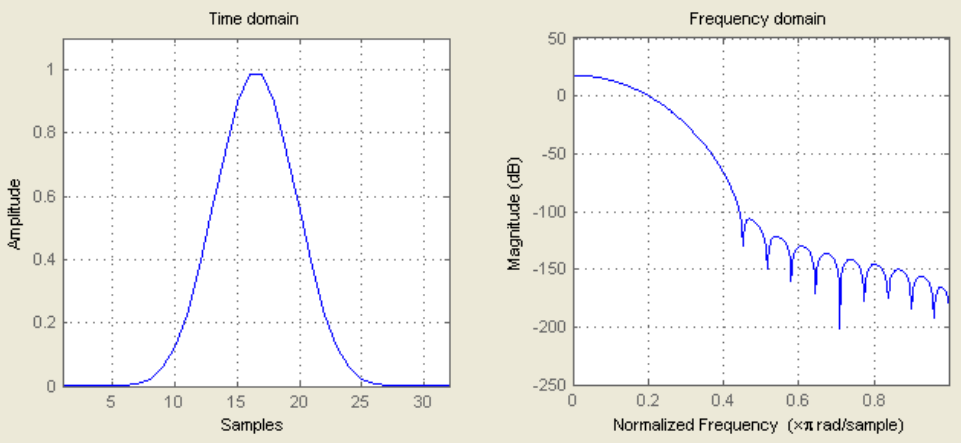

Figure 1.15: The squared Blackman-Harris window and its FFT

Going back to matrix equation 1.27, we can further simplify its components that were written out in 1.26. For example, $B_{l,k}^{1,1}$ can be written in function of $W(m)$, using the Simpson formulas of trigonometry:

$$\begin{aligned} B_{l,k}^{1,1} &= \frac{1}{2} \sum_{n=0}^{N-1} \left(w_n^2 \left(cos\left(2\pi(\omega_k + \omega_l)\frac{n-n_0}{N} \right) + cos\left(2\pi(\omega_k - \omega_l)\frac{n-n_0}{N} \right) \right) \right) \\ &= \frac{1}{2} \left(\Re\left(Y(\omega_k + \omega_l) \right) + \Re\left(Y(\omega_k - \omega_l) \right) \right) \end{aligned} \tag{1.33}$$

And the same goes for the other components of matrix B:

$$\begin{aligned} B_{l,k}^{1,2} &= -\frac{1}{2}\left(\Im\left(Y(\omega_k+\omega_l)\right)+\Im\left(Y(\omega_k-\omega_l)\right)\right) \\ B_{l,k}^{2,1} &= -\frac{1}{2}\left(\Im\left(Y(\omega_k+\omega_l)\right)-\Im\left(Y(\omega_k-\omega_l)\right)\right) \\ B_{l,k}^{2,2} &= -\frac{1}{2}\left(\Re\left(Y(\omega_k+\omega_l)\right)-\Re\left(Y(\omega_k-\omega_l)\right)\right) \end{aligned} \tag{1.34}$$

Though these results were developed using complex arithmetic, we are handling in practice a symmetric window. The squared Blackman-Harris window is also symmetric. Thus, the Fourier transform of the squared Blackman-Harris window is real, meaning that the imaginary components in the matrix equations are 0. This means that submatrices $B_{l,k}^{1,2}$ and $B_{l,k}^{2,1}$, consisting only of imaginary components, are both 0.

It is then possible to split up the large 2K by 2K matrix equation in 2 separate K by K matrix equations, that we can solve separately using a standard Gaussian elimination solver. More details on some implementation issues will be provided in the second part of this thesis.

$$\begin{aligned} A^r &= SOLVE(B^{1,1}, C^1) \\ A^i &= SOLVE(B^{2,2}, C^2) \end{aligned} \tag{1.35}$$

Taking a closer look at the structure of the matrix $B^{1,1}$, we see that its computation requires elements from 2 other matrices: one formed by $Y(\omega_k+\omega_l)$, denoted Y^+ for short, and one formed by $Y(\omega_k-\omega_l)$, or Y^-. Those matrices have a fixed structure, for example, given a model with 4 frequencies:

$$Y^- = \begin{bmatrix} Y(0) & Y(\omega_1-\omega_0) & Y(\omega_2-\omega_0) & Y(\omega_3-\omega_0) \\ Y(\omega_0-\omega_1) & Y(0) & Y(\omega_2-\omega_1) & Y(\omega_3-\omega_1) \\ Y(\omega_0-\omega_2) & Y(\omega_1-\omega_2) & Y(0) & Y(\omega_3-\omega_2) \\ Y(\omega_0-\omega_3) & Y(\omega_1-\omega_3) & Y(\omega_2-\omega_3) & Y(0) \end{bmatrix} \tag{1.36}$$

The DFT of the squared window is bandlimited and has its maximal value at the center, so, $Y(0)$ is maximal. Now, if the 2 frequencies ω_k and ω_k lie close to eachother, their difference will be close to 0. In the squared window DFT, this will correspond to a value that is close to the center, thus a large value. If those 2 frequencies are far away from eachother, their difference yields a value far from 0, which, filled in in Y, returns a very low value.

Since the squared window is also bandlimited, these low values will occur everywhere in the matrix where the frequencies ω_k and ω_l are far from eachother. We can even calculate how far they have to be from eachother to yield a small value: the width of the main lobe is our criterium. Recall that the spectrum of our model is a linear combination of shifted DFTs of the window function. The window function has a broad main lobe, and insignificant sidelobes. If a frequency is present that ends up in the main lobe of one of the other frequencies, this will lead to a large value in the matrices Y^+ and Y^-.

The other way round, if the difference between 2 frequencies is large enough, they fall in each other's sidelobes. Since the sidelobes of the Blackman-Harris function attain an attenuation of -92dB , the squared Blackman-Harris reaches at least that amount of attenuation - though the main lobe is wider. Values that small can be discarded and replaced by 0 without any noticeable effect.

If the frequencies are ordered, the matrices $B^{1,1}$ and $B^{2,2}$ thus tend to become band diagonal, which allows much faster solving. If all frequencies are very well separated, the matrices are purely diagonal. If some frequencies are close together, small squares of larger values tend to grow alongside the main diagonal, indicating that the main lobes of their frequency responses in Fourier space overlap.

The fact that the resulting matrix equations are band diagonal, is a key element of this optimization of amplitude estimation and can only be derived by explicitly including a window function with a very good sidelobe behaviour.

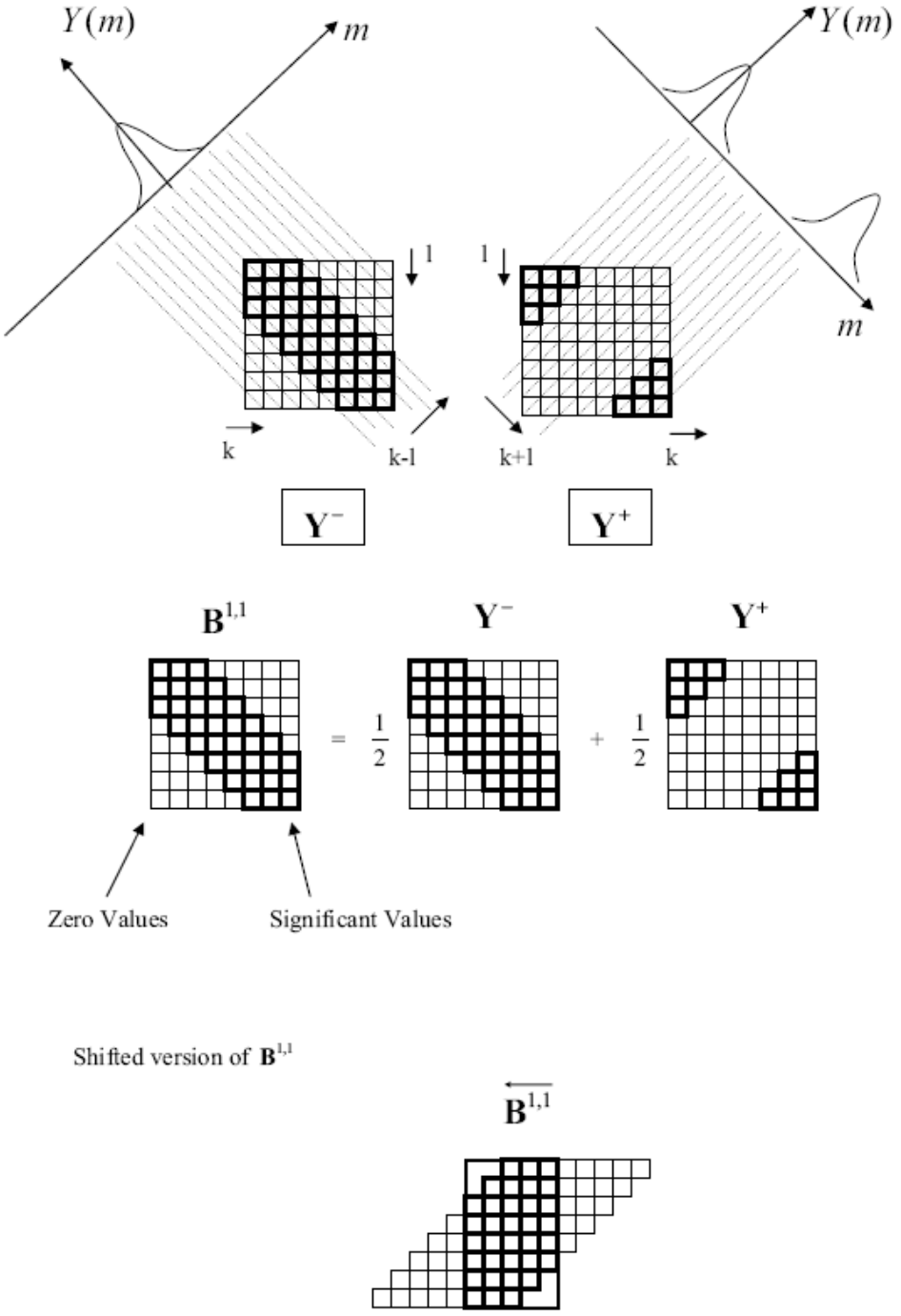

Figure 1.16: Structure of the band diagonal matrix B and storage as a shifted matrix. Image © Wim D'haes.

The elements of B, which only depend on the known DFT of the window function, can be computed in constant time. We can do approximately the same for the elements of C, using another trick. The signal x_n can be seen as the inverse Fourier transform of the model X_m. Thus, in the formula for C that has been developed in equation 1.26, we can substitute. First we rewrite both C^1 and C^2, that are parts for calculation of the real and imaginary components of the complex amplitude, together, using a complex exponential notation:

$$\begin{aligned} C_l &= \sum_{n=0}^{N-1} \left(x_n w_n e^{2\pi i \omega_l \frac{n-n_0}{N}} \right) \\ &= \sum_{n=0}^{N-1} \left(\left[\frac{1}{N} \sum_{m=0}^{N-1} \left(X_m e^{2\pi i m \frac{n-n_0}{N}} \right) \right] w_n e^{\left(2\pi i \omega_l \frac{n-n_0}{N}\right)} \right) \\ &= \frac{1}{N} \sum_{m=0}^{N-1} \left(X_m W(m+\omega_l) \right) \\ &= \frac{1}{N} \sum_{m=m_{min}}^{m_{max}} \left(X_m W(m+\omega_l) \right) \end{aligned} \tag{1.37}$$

The last step is possible since, for every frequency ω_l in the spectrum, represented by a Fourier transform of the window (since the spectrum is a linear combination of FTs of the window), only the main lobe is important. Any factors $m+\omega_l$ that fall outside the main lobe generate an insignificantly small value and can thus be ignored. This allows us to restrict the summation only to that part of the spectrum that is around the main lobe of the frequency under consideration. This main lobe width is a constant factor. Thus, calculating C does not take a complexity of $O(NK)$, K being the number of partials, but only of $O(K)$. Taking the real and imaginary parts of the last formula gives us C_l^1 and C_l^2 back, respectively.

All these results add up to a very efficient algorithm to estimate amplitudes that calculates all amplitudes at once, is capable of handling overlapping frequency responses, and has a very low time and space complexity. Setting up matrix B has a computational time complexity of $O(K)$, since only a limited set of diagonals has to be calculated, which is maximally as wide as the width of the main lobe of the DFT of the window function. Solving an equation set like that, with a band diagonal matrix, through Gaussian elimination also takes $O(K)$ as time complexity. Which leaves only X_m as nonlinear calculation cost: calculating this spectrum takes a time complexity

of $O(Nlog(N))$.

Application to a single harmonic sound

Applying this to a single harmonic sound makes the matrices Y even simpler: all frequencies are multiples of the fundamental frequencies, so all elements $Y(\omega_l - \omega_k)$ can be rewritten as $Y((l-k)\omega)$, with ω the fundamental frequency. Matrix Y^- then looks like this:

$$Y^- = \begin{bmatrix} Y(0) & Y(\omega) & Y(2\omega) & Y(3\omega) \\ Y(-\omega) & Y(0) & Y(\omega) & Y(2\omega) \\ Y(-2\omega) & Y(-\omega) & Y(0) & Y(\omega) \\ Y(-3\omega) & Y(-2\omega) & Y(-\omega) & Y(0) \end{bmatrix} \tag{1.38}$$

Multiple harmonic sounds and polyphony

When multiple harmonic sounds are played together, especially when they are polyphonic and thus very strongly related in the harmonics, the system of equations can not always be solved: harmonics of different frequencies might coincide. This leads to multiple occurrences of a single frequency component in the model. The matrix B becomes singular, the solving routine creates rows of zeroes, and the system of equations has no single solution.

These components that occur more than once must then first be recognized and filtered out by preprocessing routines, before the amplitudes are calculated. All occurences of the same frequency are summed together and the sum of the amplitudes is calculated. There is no telling afterwards which part of that amplitude then belongs to the harmonic with this or that fundamental frequency, much the same as when the only thing you know is that result of a calculation is 6, you cannot tell whether it comes from 3+3 or 2+4.

That kind of information could eventually be derived from the past, or when it is known what instruments are being played and what the spectral envelope of these instruments looks like. But that is a very difficult and far from exact exercise, since even these envelopes can vary a lot with frequency and dynamics, and accomplished players can often produce sounds with different timbres on the same instrument. It would be a nice experiment to try how accurate it can get anyway, with an instrument like a piano, which does not produce too much timbre variation.

1.3.3 Frequency Optimization

Due to the Discrete Fourier transform being applied instead of its continuous counterpart, the frequencies are rounded to the nearest frequency bin centers. Also, spectral leakage always occurs when there are sinusoids present in the signal that do not coincide with the calculated discrete frequencies - and for real world signals, this is almost always the case. Spectral leakage introduces positive amplitude values for frequency components that are, in fact, not at all present in the signal.

As discussed earlier, windowing the signal helps to reduce the leakage significantly. But, regarding the accuracy of the calculated frequencies, it does not help a bit. The frequencies that are calculated are already rounded to the nearest frequency bins, and windowing causes even these results to be 'blurred', due to the wide main lobe of the frequency response of the windowing function.

So, when trying to approximate our signal with a sinusoidal model, we cannot start with the correct frequencies either - these are unknown. The initial frequencies ω of our model $\tilde{x}_n$ therefore can only be crude estimates, and have to be optimized too.

There are several difficulties that arise. First of all, frequencies that are present in the signal may be close together, therefore having overlapping frequency responses, therefore being difficult to detect in the spectrum itself. This is most notably the case when using small windows that result in a limited frequency resolution in the spectrum. Also at low frequencies, there is a need for better resolution, since the human ear can very well distinguish between low frequencies, and any inaccuracy will be strikingly audible for the user.

Three optimization methods will be described here that are widely used for problems of this kind, and that are fit for solving problems like this: local quadratic approximation or Newton optimization, gradient descent, and model linearization, or Gauss-Newton optimization. The theoretical results have been presented in [5]. An implementation is described and discussed in the second part of this thesis.

Frequency optimization makes use of the same error function that we have defined in equation 1.17, but now not the frequencies but the amplitudes are kept constant. For an optimization, we need a first guess of the frequencies

that are present. This first guess can be derived using pitch estimators. A good overview of several techniques for pitch estimation can be found on pp. 504-520 of [16].

Initial pitch estimation

Several simple techniques exist to estimate the pitch of a single harmonic signal, but not all are reliable when lots of harmonics or noise are present. Tracking Phase Vocoders, one of the more fastest and easiest to implement, combines spectral information with phase information from consecutive frames, to calculate the value of the fundamental period. They are well explained in [24] and thoroughly analyzed in [25].

Autocorrelation or cepstrum based methods, both very similar in nature, estimate pitch by calculating an autocorrelation (in the time domain) or cepstrum (in the frequency domain) of the given signal. This detects periodicities in time or frequency space, and thus allows for detection of the fundamental period of the signal, which is the period of the fundamental frequency. A smooth and recent introduction to the cepstrum can be found in [26].

For multiple harmonic signals or polyphonic signals, there exist multi-pitch estimators which can compute a set of starting frequencies [27]. Other methods that are recently being developed make use of heavy heuristics to eliminate potential errors [28]. In other cases, a first guess can be made by the user, who can pick out the peaks he wants to start with from the spectrum (peak picking). Sometimes (a part of) the necessary information can also be extracted from a previously computed file or a music partition. For the purpose of this thesis, peak picking works just fine.

Gradient Descent

The error function $\chi(\overline{\omega}, \overline{A})$ is multidimensional. A second order Taylor expansion of this function, around a vector $\overline{\nu}$ is defined as:

$$\chi(\overline{\omega}, \overline{A}) \approx C + (\overline{\omega} + \overline{\nu})^T \nabla_\nu \chi + \frac{1}{2!}(\overline{\omega} + \overline{\nu})^T H_\nu (\overline{\omega} + \overline{\nu}) \tag{1.39}$$

where ∇_ν represents the gradient of the function χ, evaluated at the vector of frequencies ν . ν is chosen to be the estimate of the frequencies, and has to be optimized. The square of the gradient, H_ν is called the Hessian.

The standard quick-and-dirty solution to any optimization problem is gradient descent or steepest descent. Here, we optimize the values by calculating the gradient of the function, and updating the current vector in the indicated direction with a stepsize that can be chosen (the learning rate). The algorithm can require a lot of steps to complete and choosing the optimal learning rate is a difficult task which is also decisive for the speed of convergence.

Using η as the learning rate, the iteration step of the gradient descent method looks like

$$\overline{\omega}^{r+1} = \overline{\omega}^r - \eta \nabla \overline{\omega}^r \tag{1.40}$$

The gradient ∇ of a multidimensional function is defined as the vector of its first partial derivatives in every direction. So, for any frequency vector ω, the gradient of the function χ is the a vector of first derivatives. The elements of the vector corresponding to the frequencies ω_k are then:

$$\nabla_{\omega_k} = \frac{\partial \chi(\overline{\omega}, \overline{A})}{\partial \omega_k} \tag{1.41}$$

Local quadratic approximation

Newton optimization optimizes the initial guess iteratively using the gradient and the Hessian, using the following scheme:

$$\overline{\omega}^{r+1} = \overline{\omega}^r - H_{\overline{\omega}^r}^{-1} \nabla \overline{\omega}^r \tag{1.42}$$

The Hessian is the gradient of the gradient. It is a matrix with on each row the gradient of the corresponding first partial derivative of the original function. Its elements are:

$$H_{\omega_k, \omega_l} = \frac{\partial \chi(\overline{\omega}, \overline{A})}{\partial \omega_k \partial \omega_l} \tag{1.43}$$

Note that the Hessian H is symmetric, due to the second order partial derivatives of the error function being continuous (Clairaut's theorem). When the Taylor expansion is developed at the minimum of the cost function, we know that the gradient of that function must be zero:

$$\nabla_{\varpi} = \overline{0} \tag{1.44}$$

Calculating the Hessian matrix is normally a very costly computation. Quasi-Newton methods exist that approximate the Hessian in an easier and faster way, though they introduce a slower convergence rate. Conjugate gradient methods or the Levenberg-Marquardt algorithm are also possible alternatives. A good overview can be found in chapter 7 of [29].

However, as we shall see, calculating gradient and Hessian in this case can be done in a proverbial instant. Using the same trick used for amplitude estimation, we can limit the Hessian to a band diagonal form, of which each element can be computed using a constant number of additions. The same goes for the gradient, leading to both gradient and Hessian able to be calculated in $O(k)$ time, with k being the number of frequencies, given that all terms of which the elements are composed can be accessed or calculated in constant time.

Model Linearization

Another possibility, Gauss-Newton optimization, is actually a modified Newton optimization algorithm, but then without the second order derivatives. It uses a first order Tailor expansion as starting point. D'haes shows in [5] that the workout of this model actually comes down to the same gradient and matrix as Newton optimization, only the elements of the main diagonal of the matrix are different. No significant difference in convergence speed was noted though.

For the sake of brevity, we will concentrate on the Newton optimization method. All methods use the same gradient, and Newton optimization requires a Hessian. The difference between Gauss-Newton and Newton seems to be insignificant. At the moment it is not worth the trouble of further mathematically working out the model linearization theory here, since an

investigation of whether the Gauss-Newton method is any useful in this context, needs to happen by a thorough set of practical experiments first.

The mathematical workout

Starting with the gradient, we need to take the partial derivatives with respect to each of the frequency components. This is largely analogous to taking the partial derivatives with respect to the amplitudes, as seen in equations 1.18 and further. For the model, we now use the complex exponential notation from equation 1.15, instead of equation 1.16 that was used in the amplitude estimation calculations.

$$\frac{\partial \chi(\overline{A};\overline{\omega})}{\partial \omega_l} = \frac{\partial}{\partial \omega_l} \sum_{n=0}^{N-1} \left(x_n - w_n \frac{1}{2} \sum_{k=0}^{K-1} \left(A_k e^{2\pi i \omega_k \frac{n-n_0}{N}} + A_k^* e^{-2\pi i \omega_k \frac{n-n_0}{N}} \right) \right)^2 \tag{1.45}$$

Applying the rules of differentiation for powers and exponentials, concatenated using the chain rule, we arrive at the following:

$$\begin{aligned} \frac{\partial \chi(\overline{A};\overline{\omega})}{\partial \omega_l} &= 2 \sum_{n=0}^{N-1} \left(x_n - w_n \frac{1}{2} \sum_{k=0}^{K-1} \left(A_k e^{2\pi i \omega_k \frac{n-n_0}{N}} + A_k^* e^{-2\pi i \omega_k \frac{n-n_0}{N}} \right) \right) \\ &\quad \left(-w_n \frac{1}{2} A_l e^{2\pi i \omega_l \frac{n-n_0}{N}} 2\pi i \frac{n-n_0}{N} - w_n \frac{1}{2} A_l^* e^{-2\pi i \omega_l \frac{n-n_0}{N}} \left(-2\pi i \frac{n-n_0}{N} \right) \right) \end{aligned} \tag{1.46}$$

The first part of the multiplication is still the original signal minus the model. In practice, this corresponds to the error on the model, in the form of a residue signal. The residue signal r_n, like any other signal, has a Fourier transform R_m. So, the first part of the multiplication can just as well be written as the inverse Fourier transform of the residue spectrum. At the same time working out some constants and signs, the following equation holds:

$$\frac{\partial\chi(\overline{A};\overline{\omega})}{\partial\omega_l} = \sum_{n=0}^{N-1}\left[\frac{1}{N}\sum_{m=0}^{N-1}\left(R_m e^{2\pi i m\frac{n-n_0}{N}}\right)\right] \left(-w_n A_l e^{2\pi i\omega_l\frac{n-n_0}{N}}2\pi i\frac{n-n_0}{N} + w_n A_l^* e^{-2\pi i\omega_l\frac{n-n_0}{N}}2\pi i\frac{n-n_0}{N}\right) \tag{1.47}$$

The indices m and n are different indices but are iterating over the same amount of values. We can reorder summations and work out the multiplication of the exponentials, yielding the following result:

$$\frac{\partial\chi(\overline{A};\overline{\omega})}{\partial\omega_l} = \frac{1}{N}\sum_{m=0}^{N-1}R_m\left[-A_l\sum_{n=0}^{N-1}\left(w_n 2\pi i\frac{n-n_0}{N}e^{2\pi i(m+\omega_l)\frac{n-n_0}{N}}\right) +A_l^*\sum_{n=0}^{N-1}\left(w_n 2\pi i\frac{n-n_0}{N}e^{2\pi i(m-\omega_l)\frac{n-n_0}{N}}\right)\right] \tag{1.48}$$

Looking in detail to the two terms inside the inner summation, observe the striking similarity to the window frequency response, presented in equation 1.31. In fact, we can extract the first derivative of that function. Denoted as W', it is defined as:

$$W'(m) = \sum_{n=0}^{N-1}\left(-2\pi i\frac{n-n_0}{N}w_n e^{-2\pi i m\frac{n-n_0}{N}}\right) \tag{1.49}$$

This can be implemented, just like w_n and W_m earlier, using a lookup table, thus making the calculations in which these elements are needed run in constant time. The gradient of the error function then, in turn, now looks like

$$\frac{\partial\chi(\overline{A};\overline{\omega})}{\partial\omega_l} = \frac{1}{N}\sum_{m=0}^{N-1}R_m\left(A_l^*W'(m-\omega_l) - A_l W'(m+\omega_l)\right) \tag{1.50}$$

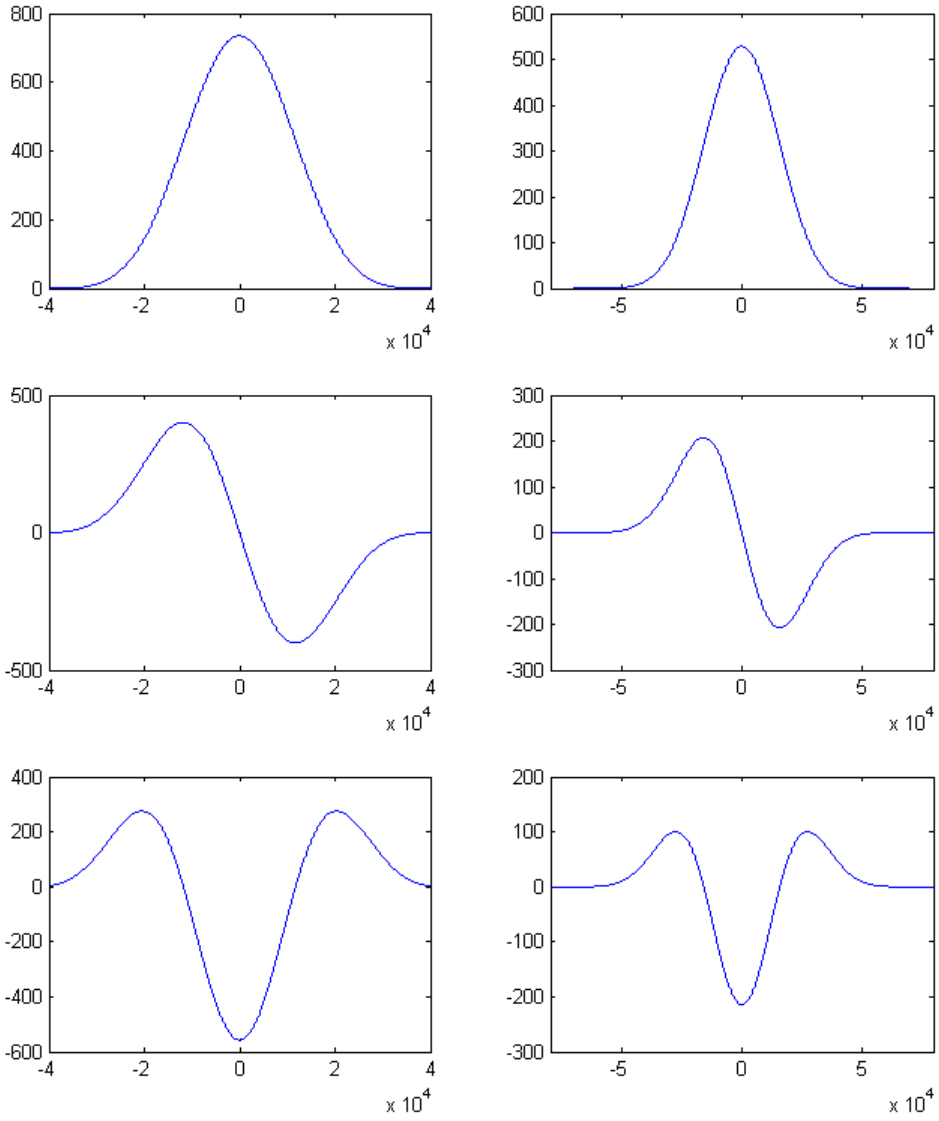

Figure 1.17: on the left, top-down: functions W, W' and W"; on the right, top-down: functions Y, Y' and Y", all oversampled with a factor 10000

To construct the elements of the Hessian, we need to take second order partial derivatives:

$$\frac{\partial\chi(\overline{A};\overline{\omega})}{\partial\omega_p\partial\omega_l} = \frac{\partial}{\partial\omega_p}\frac{\partial}{\partial\omega_l}\sum_{n=0}^{N-1}\left(x_n - w_n\frac{1}{2}\sum_{k=0}^{K-1}\left(A_k e^{2\pi i\omega_k\frac{n-n_0}{N}} + A_k^* e^{-2\pi i\omega_k\frac{n-n_0}{N}}\right)\right)^2 \tag{1.51}$$

The order in which the partial derivatives are taken does not matter, since the second order partial derivatives are continuous. Working out the innermost of the partial derivatives, we get:

$$\begin{aligned}\frac{\partial\chi(\overline{A};\overline{\omega})}{\partial\omega_p\partial\omega_l} &= \frac{\partial}{\partial\omega_p}2\sum_{n=0}^{N-1}\left(x_n - w_n\frac{1}{2}\sum_{k=0}^{K-1}\left(A_k e^{2\pi i\omega_k\frac{n-n_0}{N}} + A_k^* e^{-2\pi i\omega_k\frac{n-n_0}{N}}\right)\right)\\ &\quad\left(-w_n\frac{1}{2}A_l e^{2\pi i\omega_l\frac{n-n_0}{N}}2\pi i\frac{n-n_0}{N} - w_n\frac{1}{2}A_l^* e^{-2\pi i\omega_l\frac{n-n_0}{N}}\left(-2\pi i\frac{n-n_0}{N}\right)\right)\end{aligned} \tag{1.52}$$

This is a partial derivative of a product of functions. Taking a derivative of a product of functions yields a sum of 2 terms in the following way: $(f\cdot g)'(x) = f(x)g'(x) + f'(x)g(x)$. We work out the 2 terms separately, for the sake of clarity. The first term equals

$$\begin{aligned}&2\sum_{n=0}^{N-1}\left(x_n - w_n\frac{1}{2}\sum_{k=0}^{K-1}\left(A_k e^{2\pi i\omega_k\frac{n-n_0}{N}} + A_k^* e^{-2\pi i\omega_k\frac{n-n_0}{N}}\right)\right)\\ &\frac{\partial}{\partial\omega_p}\left(-w_n\frac{1}{2}A_l e^{2\pi i\omega_l\frac{n-n_0}{N}}2\pi i\frac{n-n_0}{N} - w_n\frac{1}{2}A_l^* e^{-2\pi i\omega_l\frac{n-n_0}{N}}\left(-2\pi i\frac{n-n_0}{N}\right)\right)\end{aligned} \tag{1.53}$$

Taking the partial derivative with respect to ω_p of a summation of functions of ω_l only has a result different than 0 when p equals l. At the same time, just as in equation 1.47 of the gradient derivation, we can use the residue spectrum R_m to express the difference between the signal and the model. So we can rewrite the result as:

$$\sum_{n=0}^{N-1}\left[\frac{1}{N}\sum_{m=0}^{N-1}\left(R_m e^{2\pi i m\frac{n-n_0}{N}}\right)\right]$$
$$\delta_{lp}\left(-w_n A_p e^{2\pi i\omega_p\frac{n-n_0}{N}}\left(2\pi i\frac{n-n_0}{N}\right)^2 - w_n A_p^* e^{-2\pi i\omega_p\frac{n-n_0}{N}}\left(-2\pi i\frac{n-n_0}{N}\right)^2\right) \tag{1.54}$$

Reordering terms and merging the exponential functions then brings us to:

$$-\delta_{lp}\frac{1}{N}\sum_{m=0}^{N-1}R_m\left[A_p\sum_{n=0}^{N-1}\left(w_n\left(2\pi i\frac{n-n_0}{N}\right)^2 e^{2\pi i(m+\omega_p)\frac{n-n_0}{N}}\right)\right.$$
$$\left.+A_p^*\sum_{n=0}^{N-1}\left(w_n\left(2\pi i\frac{n-n_0}{N}\right)^2 e^{2\pi i(m-\omega_p)\frac{n-n_0}{N}}\right)\right] \tag{1.55}$$

Remind that after equation 1.48 it was possible to extract the first derivative of the Fourier transform of the window function from the result. Here we can do the same for the second derivative of the Fourier transform of the window function. Analogous to the first derivative counterpart in equation 1.49, it is defined as:

$$W''(m) = \sum_{n=0}^{N-1}\left(\left(-2\pi i\frac{n-n_0}{N}\right)^2 w_n e^{-2\pi i m\frac{n-n_0}{N}}\right) \tag{1.56}$$

and thus the first term of each Hessian element becomes:

$$-\delta_{lp}\frac{1}{N}\sum_{m=0}^{N-1}R_m\left(A_p W''(m+\omega_p) + A_p^* W''(m-\omega_p)\right) \tag{1.57}$$

In practice, δ_{lp} makes that this term only contributes to the elements of the Hessian on the main diagonal. Also, W'' can be implemented using a lookup table. Given R_m is also precalculated, this makes that this contribution to the Hessian only needs constant time.

The second term of each Hessian element is:

$$\sum_{n=0}^{N-1}\left(-w_n\frac{1}{2}A_l e^{2\pi i\omega_l\frac{n-n_0}{N}}2\pi i\frac{n-n_0}{N}-w_n\frac{1}{2}A_l^* e^{-2\pi i\omega_l\frac{n-n_0}{N}}\left(-2\pi i\frac{n-n_0}{N}\right)\right)$$
$$\frac{\partial}{\partial\omega_p}2\left(x_n-w_n\frac{1}{2}\sum_{k=0}^{K-1}\left(A_k e^{2\pi i\omega_k\frac{n-n_0}{N}}+A_k^* e^{-2\pi i\omega_k\frac{n-n_0}{N}}\right)\right) \tag{1.58}$$

The second derivative with respect to ω_p only has nonzero results here when k equals p. We can do away with the sum over k and use the chain rule to work out the partial derivative, which results in

$$\sum_{n=0}^{N-1}\left(-w_n\frac{1}{2}A_l e^{2\pi i\omega_l\frac{n-n_0}{N}}2\pi i\frac{n-n_0}{N}-w_n\frac{1}{2}A_l^* e^{-2\pi i\omega_l\frac{n-n_0}{N}}\left(-2\pi i\frac{n-n_0}{N}\right)\right)$$
$$\left(-w_n A_p e^{2\pi i\omega_p\frac{n-n_0}{N}}2\pi i\frac{n-n_0}{N}-w_n A_p^* e^{-2\pi i\omega_p\frac{n-n_0}{N}}\left(-2\pi i\frac{n-n_0}{N}\right)\right) \tag{1.59}$$

Here, too, it is possible to merge the exponential functions, resulting in 4 terms inside the summation over n. There is clearly a distinguishable structure to be noted in each of these terms:

$$\begin{aligned}
&\frac{1}{2}\sum_{n=0}^{N-1}A_l A_p w_n^2\left(2\pi i\frac{n-n_0}{N}\right)^2 e^{2\pi i(\omega_l+\omega_p)\frac{n-n_0}{N}}\\
-&\frac{1}{2}\sum_{n=0}^{N-1}A_l A_p^* w_n^2\left(2\pi i\frac{n-n_0}{N}\right)^2 e^{2\pi i(\omega_l-\omega_p)\frac{n-n_0}{N}}\\
-&\frac{1}{2}\sum_{n=0}^{N-1}A_l^* A_p w_n^2\left(2\pi i\frac{n-n_0}{N}\right)^2 e^{-2\pi i(\omega_l-\omega_p)\frac{n-n_0}{N}}\\
+&\frac{1}{2}\sum_{n=0}^{N-1}A_l^* A_p^* w_n^2\left(2\pi i\frac{n-n_0}{N}\right)^2 e^{-2\pi i(\omega_l+\omega_p)\frac{n-n_0}{N}}
\end{aligned} \tag{1.60}$$

The structure that we can observe is the second derivative of the Fourier transform of the square window. The Fourier transform of the square window was given in equation 1.32, and its second derivative is:

$$Y''(m) = \sum_{n=0}^{N-1} \left(\left(-2\pi i \frac{n-n_0}{N} \right)^2 w_n^2 e^{-2\pi i m \frac{n-n_0}{N}} \right) \tag{1.61}$$

As a result, the second term of the Hessian becomes

$$\begin{aligned} &\frac{1}{2} \left[A_l A_p Y''(\omega_l + \omega_p) - A_l A_p^* Y''(\omega_l - \omega_p) \right. \\ &\left. -A_l^* A_p Y''(-\omega_l + \omega_p) + A_l^* A_p^* Y''(-\omega_l - \omega_p) \right] \end{aligned} \tag{1.62}$$

This term of each Hessian element only produces values around the main diagonal, since Y'' is also bandlimited. Just as matrix B in the amplitude calculation, this Hessian is band diagonal, too. Solving the system of equations formed by the gradient and the Hessian, we can obtain a new, better estimate of the frequencies of our model, using equation 1.42.

Multiple harmonic sounds and polyphony

When the input signal consists of a harmonic sound, we only have to optimize the fundamental frequency, since we know that the harmonics are integer multiples of that frequency and will get optimized correctly together with the fundamental. The sinusoidal model can be split up into a summation of a summation: one over the fundamentals, and one over each's harmonics. Hessian and gradient become significantly smaller, but calculation of each element of them requires an extra summation over all the harmonics of the specific frequency.

D'haes notes in [5] that the Hessian for multiple harmonic sounds is not band diagonal any more, but the fact that it is usually very small - only a few sound sources present - compensates for that. Without going through the trouble of making another long mathematical derivation, we can also note that the cross products of all harmonics with eachother that appear in the Hessian when using a harmonic model, can be optimized and linearized.

Chapter 2

Implementation of a Sinusoidal Modeler

2.1 Useful Technologies

2.1.1 The VST framework

VST stands for Virtual Studio Technology, and is a proprietary technology from the Germany-based company Steinberg Media Technologies GmbH [2]. It is a plugin framework: it enables a programmer to write audio processing plugins (like filters, analyzers, etc.) that can be loaded into a host program and then be used by that host program. Popular hosts that support VST plugins are Cubase, Wavelab, Nuendo (all by Steinberg), Ableton Live, Bidule (by Plogue), ... Recently, also some important open source players have included support for the framework: Audacity, CLAM, RoseGarden, Ardour, ...

The VST SDK (Software Development Kit) [30] is a cross-platform set of classes, written in C++ - albeit with a lot of C-style programming constructs, like use of `malloc()`, `free()` and structs. The SDK is downloadable free of charge from the Steinberg website. The license makes integration into open source software difficult. It does not allow redistribution of the SDK and as such it is not compatible with most open source licenses. All users willing to integrate the VST SDK in a software product must manually register at the Steinberg website and download the framework for themselves. The reason for this is that Steinberg wishes to assure that is only one version of the VST framework around, and that forks become impossible. This has been and still is a topic of major discussion among VST plugin developers, certainly because of the lack of documentation, the frequent changes in the interface,

and the messy coding style of the current framework.

The core of the framework is very simple: it provides a developer with the possibility to implement and/or extend the standard plugin class `AudioEffectX`, using inheritance. The main task is to implement a function `virtual void processReplacing (float** inputs, float** outputs, VstInt32 sampleFrames)` , that accepts an input buffer and an output buffer of audio samples (possibly multichannel). The developer, in the body of the `processReplacing()` function, can read the input buffer, do whatever he wants to do with the audio data, and write the results back to the output buffer.

This gets compiled into a DLL (Dynamic Linked Library). When copied to some specific folder, the host scans the folder for valid plugin files, and then can load the plugin and use its `processReplacing()` function. In the host program, the user is enabled to link the plugin to an audio stream (possibly a real-time one). The audio stream gets divided into different buffers, that are iteratively fed to the plugin, and the host then copies the processed data from the plugin output buffer back into the audio stream. Callbacks are used for communication between host and plugins: the constructor of the `AudioEffectX` class looks like `AudioEffectX(audioMasterCallback audioMaster, VstInt32 numPrograms, VstInt32 numParams)`.

Next to audio handling, VST provides functions for receiving and sending MIDI [4] events, setting the properties of the plugin, requesting information about the host, saving and loading plugin settings, etc. The host must of course support all these functions, and this is certainly not always the case. The VST framework gets updated sometimes, unfortunately breaking the interface of previous versions, and thus backward compatibility. It can be important for plugins to be written in the same VST version as the host supports. Very often one finds out by trial and error: a plugin has been developed using host X, but uses functions that seem not yet to be supported by host Y.

With this range of functionality, it is for example possible to make virtual instruments, often abbreviated as VSTi: a plugin that accepts MIDI messages and outputs audio buffers only. When the input audio is processed and outputted as audio, one often speaks of an effect, or VSTfx. Plugins that analyze incoming audio and output MIDI are often called triggers. The input and output buffers are arrays of 32-bit floats, regardless the capabilities of the host. From version 2.4 on, support for 64-bit systems is provided, in the form of a special function for handling double precision audio buffers, and

the introduction of the `VstInt32` datatype that assures 32-bit width of an integer (previously `long`).

This change of datatype, and some other changes, make that any code written on top of the 2.3 version of the SDK, needs rewriting in order to compile with version 2.4. From version 2.4 forth, the interface of the framework is properly separated from the actual code. To make an eventual upgrade from version 2.3 to 2.4, the latter version still includes deprecated functions from the former, which can be activated by setting a compile time flag.

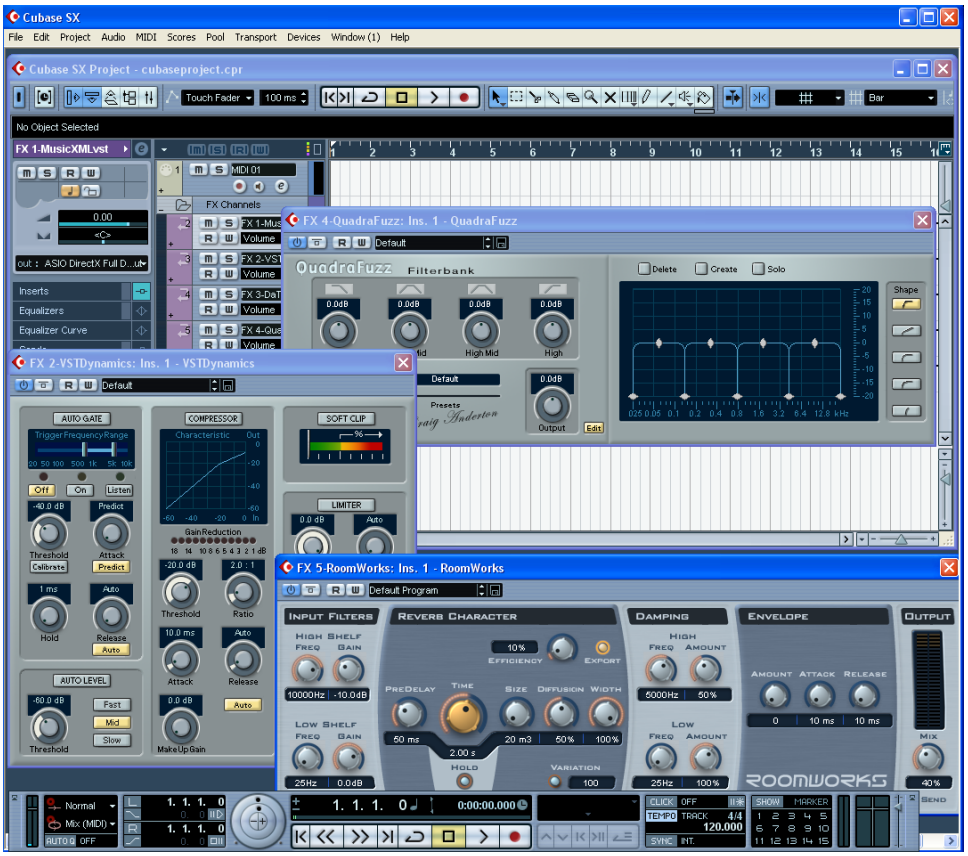

Figure 2.1: A Cubase SX3 session with some VST plugins loaded

The greatest strength of the VST framework is that it allows the user to code anything he wants in a VST plugin - it needn't even be audio related. Full control is handed over to the plugin, that can make any calls to the operating system that it wants. In theory, it is very well possible to implement a card game in a VST plugin - the sky is the proverbial limit. It is

possible to attach an own GUI to the plugin so that it interfaces with the user graphically. If no GUI is attached, most hosts create a generic GUI for the plugin, allowing to set the different kinds of parameters.

This absolute freedom forms at the same time the greatest risk. Though no examples are known (yet), it is theoretically possible to write a VST plugin that installs a trojan or computer virus on the computer. The plugin is granted complete control over the system, and any code can be run in the `processReplacing()` function. For reasons of performance, no host implements a sandbox for plugins. End-users do therefore good only to accept and use plugins that were obtained from a reliable source. Currently, a good deal of the audio plugin developing world and general resources on VST can for example be found on the website of KVR audio [31].

VST is not the only framework around for audio processing plugins. Popular alternatives are AU (Audio Units) by Apple Computer [32], RTAS (Real Time Audio Suite) by Digidesign, a subdivision of Avid [33], for the Windows platform DirectX [34], or for the Linux platform LADSPA (Linux Audio Developer's Simple Plugin API) [35]. Wrapper technologies exist to integrate these frameworks and enable a VST plugin to be used in RTAS or AU environments [36]. Ports to other languages than C++ also exist for VST, most notably to Delphi [37] and Java [38].

2.1.2 VSTGUI

As mentioned earlier, a GUI (Graphical User Interface) can be attached to a VST plugin. VSTGUI [39] was developed to facilitate making plugins for VST plugins. It is a set of platform independent classes that provide basic GUI components and the necessary coding for interacting with them - mouse clicking, typing etc. The VSTGUI project was released open source by Steinberg in september 2003. Since then it is a community project managed by Steinberg but contributed to by many users, and hosted on SourceForge. VSTGUI suffers from the same major drawback as the VST SDK: it is a pain to upgrade from older to newer versions due to interface changes.

The VST SDK distributed by Steinberg comes with an old version of VSTGUI, but more recent versions can be obtained through [39]. An own GUI to a plugin can be made using multiple inheritance: `class MyGuiEditor: public AEffGUIEditor, public CControlListener { }` . In the constructor of the plugin then, its GUI is initialized:

`editor = new MyGuiEditor(this);` . If no own GUI is provided, the host of the plugin creates a default GUI of its own, to allow plugin parameter setting. Using VSTGUI, much more visual appeal can be conjured up, though.

The constructor of the GUI class is in charge of initializing the GUI and drawing all components on the screen. VSTGUI provides standard controls to draw knobs, sliders, handles, buttons, switches, textboxes etc. At the moment of writing, the latest version of the development branch has some basic support for tabs and scrollbars. The plugin programmer just has to pick a few of the offered components (or write his/her own), position them on the screen as wanted, and connect them to their corresponding plugin parameters.

The possibility to provide own graphics can lead to some very spectacular GUI designs. A knob can be drawn using virtually any normal bitmap. Lots of synthesizer plugins have GUIs that look just like an analog synthesizer - it requires a little drawing talent to fit the right knob and slider bitmaps onto the right background. When compiled for Windows operating systems, VSTGUI translates its drawing functions to GDI+ [40] commands, on Unix based systems Motif [41] is used. Both are relatively low level APIs that enable using graphics on the computer display.

Each VSTGUI component implements a method `draw(CDrawContext *pContext)` so that it can be drawn on the screen. Each component can be `setDirty()` in order to signal that the specific component needs to be redrawn. This way, only the changed components are redrawn, which saves a lot of precious time. Each component can, but not necessarily has to, implement a function that can handle mouse events: `mouse(CDrawContext *pContext, CPoint& where, long buttons)`, where it should be noted that the last parameter is currently only present in VSTGUI version 3.5. When the mouse is clicked on that component, this function can consist of routines to change some GUI parameters.

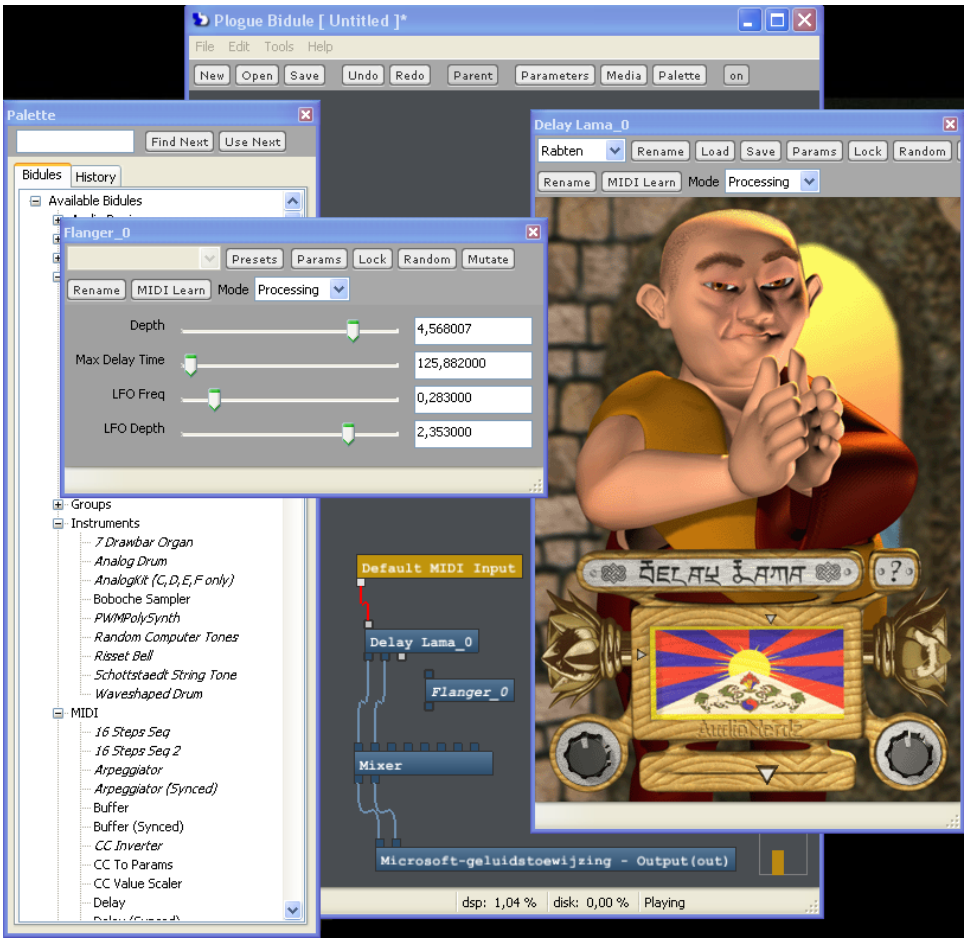

Figure 2.2: A Bidule session showing a plugin with the default host GUI, and one that has its 3D GUI [42]

When the user changes something in the GUI, it might be necessary for the GUI to relay some of the changes to the plugin itself, and possibly redraw itself on the screen. This is done by the function `valueChanged (CDrawContext* pContext, CControl* pControl)` in the GUI main class. Note here, too, that the first parameter has been removed in the latest development versions of VSTGUI. The `valueChanged()` function consists of a huge `case` construct, iterating over all GUI components. When the component that has changed is encountered, this is the place where the GUI forwards the parameter change to the plugin itself, and eventually sets the component dirty, forcing a redraw.

On certain times, the host redraws the plugin. The GUI is somewhat constructed as a tree of components - a background with controls on top of that

- and the `draw()` function just walks the tree. Whenever a component that has been set dirty is encountered, the `draw()` function of that component is called and the component redraws itself on the screen. Other components are left alone and are not redrawn.

Extending VSTGUI with own controls is easy. Every control, and thus your own too, needs to extend the `CControl` class and (re-)implement the constructor, the `draw()` and `mouse()` functions. The constructor can of course be overloaded if it needs any more parameters. The recent VSTGUI development versions also require a copy constructor in the form of a function `virtual CPlot* newCopy ()`. For the rest, a control class can be equipped with as much members and methods as required. When finished, it can be used as any other control in the plugin GUI.

Since the GUI just needs to be constructed from the plugin constructor, any GUI framework can be used to develop GUIs for VST plugins. The VSTGUI framework is just one of the numerous graphics frameworks that are around. VSTGUI is just a small set of a very basic GUI classes, but it is fast and platform independent. Users wishing a larger set of functionality to begin with, might consider using the also platform independent Qt [43] or even a 3D framework like DirectX [34] or OpenGL [44]. A very funny freeware plugin using the latter is the Delay Lama plugin by AudioNerdz [42].

2.1.3 SSE

SSE stands for Streaming SIMD Extensions, SIMD being an abbreviation of Single Instruction, Multiple Data. It was introduced in 1999 in the Pentium III line of microprocessors by Intel [45], as an extension on MMX technology and as response to competitor AMD's 3DNow! technology. It is a large instruction set that can be used on some special purpose 128-bit registers, allowing for parallellization of calculations on arrays of data. SSE is mostly used in environments where performance matters, or where large amounts of calculation time are involved - multimedia applications being the standard example. SSE has seen many extensions itself: SSE2 and SSE3 were introduced in the Pentium IV processor, SSSE3 is an extension for the Core processor, and SSE4 is expected to be available in 2008.

SSE is specific to fairly recent processors. Due to its popularity, AMD was obliged to include SSE in their processors too, and it was from 2001 available on the Athlon XP processor. Using SSE in multimedia applications

also limits the usage of the application to recent processors. Therefore most applications implement multiple versions of routines in which SSE could be used: one with and one without SSE, and perhaps some more for different versions of processors that support later additions to SSE.

SSE can be used using inline assembly mnemonics, but a far more interesting approach is the use of intrinsics. Intrinsics can be seen as 'wrappers' around one or more assembly commands. Assembly language is substituted for the intrinsics at compile time. The advantages of intrinsics are a better readability of the code, and better optimization possibilities for the compiler (especially in register allocation and instruction scheduling). The enhanced readability also reduces the chances on mistakes and errors, often a major problem when coding in an assembly language.

To use SSE intrinsics in a Windows based environment, one has to include the header `<xmmintrin.h>` in the source code. SSE2 requires the header `<emmintrin.h>` and MMX, the predecessor of SSE, the header `<mmintrin.h>`. I will demonstrate the use of SSE intrinsics using this simple example: the calculation of a dot product of 2 arrays of single precision (32-bit) floating point numbers. Encoding this in normal C or C++ would look something along the lines of:

```
for(int i=0; i<N; ++i)
  c[i] = a[i]*b[i];
```

In order to use the arrays as inputs to SSE registers, they need to be 16bit aligned in the memory. A requirement that is absolutely necessary to avoid random code crashes, but that is almost always forgotten - a lot of the documentation on SSE fails to mention it. On a Windows platform, the arrays thus need to be declared as `__declspec(align(16)) float a[N]` . Then, it is possible to cast the array to the datatype `__m128` , which is used by the SSE intrinsics. SSE code for the array multiplication would look like:

```
__m128 *a_sse = (__m128*) a;
__m128 *b_sse = (__m128*) b;
__m128 *c_sse = (__m128*) c;
for(int i=0; i<(N/4); ++i)
  c_sse[i] = _mm_mul_ps(a_sse[i], b_sse[i]);
```

By parallellizing the multiplication using SSE, we actually perform 4 floating point multiplications at the same time, thus significantly improving performance. If the arrays are not really small, then the few extra instructions for conversion and eventual boundary checks are well worth the trouble.

Multiplication is of course not the only intrinsic around. Several logical and arithmetic intrinsics are provided to perform on all 4 floats at once, or on only 1 of the floats. Load, set and store operations for the registers, comparison operators, rounding and casting functions, shuffle and swap functions are all available. SSE2 expands this functionality with the support of double precision floating point numbers: 2 of those fit in a 128-bit register, and 2 operations on doubles can be done at the same time.

When it comes down to programming with SSE the way that benefits most to Intel processors, the information provided by Intel itself is of course the best. Have a special look at their Optimization Reference Manual [46], available online and updated regularly, for information on the best practices and habits regarding SSE and other Intel technologies. Using SSE remains one of the most popular ways to speed up even well-established algorithms like FFT [47].

2.1.4 Matlab Mex Functions

Matlab [48] is one of the most popular environments for numerical computing, built around an own language (M-language) and containing a myriad of functionality. It is especially useful for linear algebra - and thus matrix and vector processing - and contains numerous possibilities for creating plots. It is a proprietary product by a company called The MathWorks, and is in widespread use among scientists.

Not all functionality that one wants in Matlab can be provided, though. It is possible to write your own functions in the M-language, but a commercially more interesting method is using Mex functions. This allows Fortran or C to be compiled as some kind of dynamic linked library, a Mex file, which can be used by Matlab like a VST plugin DLL can be used by a VST host. The major advantage is that C++ code is independent from Matlab and can thus easily be reused in other projects, and that the routines are compiled natively for the system and can thus be made as efficient as one can get them.

Suppose we have a function, written in C, that we wish to make available in Matlab. After including the `<mex.h>` library, the only thing that needs to be done is wrap the function in a so-called Mex function, that looks like `void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])` . The parameters of the function represent from left to right: the number of left hand side parameters, the left hand side parameters, the number of right hand side parameters, and the right hand side parameters. This is because a function in Matlab does not only accept multiple arguments, but can also output multiple results.

An `mxArray` is used by Matlab to store all data. The input of `mexFunction` is a `const` array of pointers to `mxArrays`. To use the data in C, they need to be extracted to the correct datatype. The `<mex.h>` header provides some functions to extract data from mxArrays to pointers of arrays (`mxGetPr(prhs[i])`), scalar values (`mxGetScalar(prhs[i])`), strings... The whole list of functions for data extraction can be found in the Matlab help files. It is possible to do range and data type checks on the input.

When in the `mexFunction` body the inputs are translated to their correct C counterparts, the only thing left to do is to call all C functions you want to use. Afterwards you convert the outputs back to `mxArrays`, using similar functions provided in the `<mex.h>` header like the data extraction functions.

In order to have `<mex.h>` included and `mexFunction` compiled only when used in Matlab, it can be conveniently wrapped in a preprocessor directive `#IF MEX` . The M-language `mex -setup` command will configure Matlab to select a C or Fortran compiler to use - it will even search for compilers on your system and offer a choice. After that, a simple `mex -DMEX myfile.c` command will compile the C function into a Matlab Mex library. The C function can now be called inside Matlab as if it were a normal Matlab function. The name of the function is the filename of the C source file.

The `mex` command will run the selected compiler on the C source file, and afterwards provide the correct linking with Matlab functions defined in the `<mex.h>` header. Calls to `printf()` for example are rerouted to the Matlab command prompt. When using Mex functions, special care must be taken that the arguments have the correct data types. The standard Matlab datatype is the 64-bit `double` but when the C code uses SSE1, only the 32-bit `float` types can be used. Passing an array of floats to a Mex function requires that the array in Matlab be explicitly created as an array of 32-bit floats.

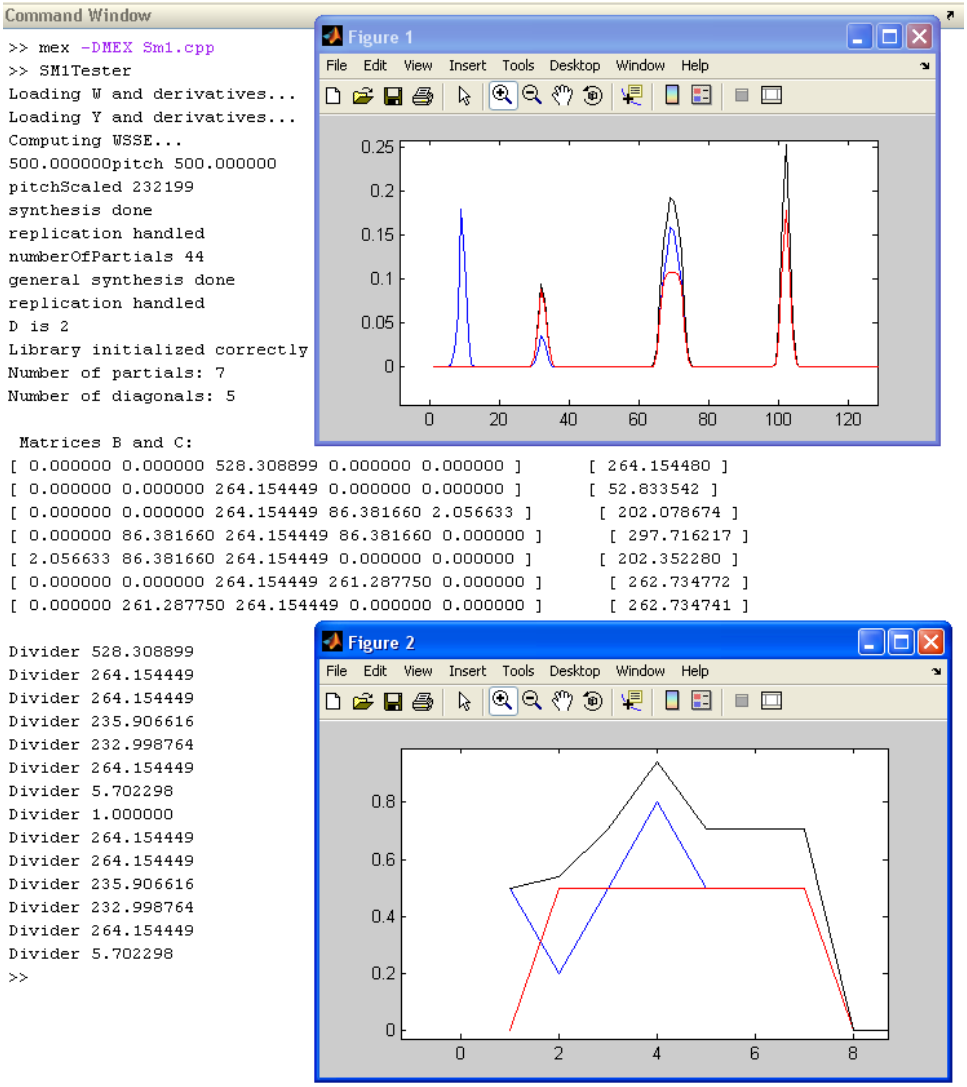

Figure 2.3: Using Mex functions in Matlab to test parts of the sinusoidal modelling implementation

The requirement for SSE to use 16-bit aligned arrays of data is not a problem for Matlab arrays. The fact that during implementation no random crashes occurred when using single precision floating point mxArrays in Mex files, strongly suggests that Matlab internally also uses 16-bit aligned arrays, possibly using SSE too. Since Matlab is closed source, this is however not verifiable.

2.2 Implementation

After this technology overview, it is probably already clear what the goal of an implementation is. We can implement the sinusoidal modelling routines in C and test them thoroughly in Matlab. Once they work fine, they can be seamlessly integrated into a VST plugin, and a GUI can be attached so that it is directly usable in practice. I will not go into detail about how to code all the theory in C - it is, after all, not much more than filling in the elements of the vectors and matrices and running a Gaussian elimination solver routine on them. But some of the implementation details are very much worth a closer look: some general notes that have to be taken into account when programming audio software, and some implementation constructs that save speed and space.

2.2.1 Audio Processing Difficulties

Latency

When used in real-time applications, an audio plugin causes a delay that is at minimum the size of the buffers that it is being given, called latency. Before the host program can send audio data to the output (which can be speakers or another plugin or piece of software), it must first fill up an input buffer, then let the plugin do its work and only after that it can start sending the first of the samples of what is now the output buffer to the output. Thus, for real-time applications, it is desirable to make the size of the input and output buffers as small as possible.

Latency is a drawback in real-time performance, because the delay, if large enough, could be heard as an echo-like effect. There are very different figures around on what delays are hearable for humans. Experiments show that the human ear tends to eliminate delays up to 40-50 ms (the so called

Haas limit) [49], other measurements indicate that users are very quickly fatigued when hearing delays of 30 ms and more [50]. Recent experiments show that musicians playing in ensemble are able to automatically compensate for small delays introduced by, for instance, large distance between the performers. Even more than that: an imposed delay of 11.5 ms seems to lead to the best synchronization [51] .

Discrete Fourier transforms are calculated on a buffer of samples. The length of the buffer is a measure for the spectral accuracy that can be obtained. Suppose the sample rate is 44100 Hz, a normal cd sampling rate. When choosing a buffer size of 2048 samples, which is extremely large, the DFT frequencies will be interspaced by 44100/2048, which is approximately 21.5, Hz. This also leads to a delay of 2048/44100 seconds, which is approximately 46.5 milliseconds. That is about the maximum that is bearable, if sticking to the Haas limit.

Since the human ear can distinguish frequencies starting from about 20 Hz, it is relatively important that a good spectral resolution is obtained. Using buffers of 1024 samples, the lowest distinguishable frequency in the FFT is only approximately 43 Hz. Though the delay will be only little more than 23 ms, the loss of accuracy in low frequencies is hardly tolerable for musical applications.

Using the methods in this thesis, a loss in frequency resolution also results in broader spectral peaks in the absolute sense: the width of the window FFT is a constant number of frequency bins. Even if those bins are more widely interspaced, the number of necessary bins remains the same. For applications like the one developed here, that try to obtain an exact spectrum of the sound wave, a larger buffer size is better. For reasons of real-time usage, we cannot ignore the Haas limit, and a buffer size of 2048 at a sample rate of 44100 Hz seems the best compromise.

Thread Scheduling

The smaller the buffer sizes are, the more times the plugin must call its `processReplacing()` routine to process the buffer. If the `processReplacing()` routine takes a lot of time, this can eventually lead to the plugin being fed data buffers more quickly than it can handle them. The resulting audio stream will suffer from data loss: gaps and clicks. Audio data are in general very large: one second of cd quality audio contains

44100 samples. If each of these samples would have to be processed separately, a process routine would have to be called every $2.268 * 10^{-5}$ seconds ,which is not attainable, so buffers are used - most often sized as a power of 2.

The thread scheduling system of the operating system can start to interfere with the process calls. To assure fluent continuation of the audio, the audio process must be active enough times every second to be able to handle all buffers. Smaller buffers mean more processing calls that are needed, resulting in a need for tighter scheduling. Most thread schedulers switch threads at rates from 250 Hz (server) to 1000 Hz (desktop), so when another thread running in the operating system gets processor time, some input buffers might already have been missed (and consequently dropped) when the audio processing thread gets back to run on the processor.

On systems that are heavily loaded, a large buffer size is needed so as to process more samples in just one process call, and relax the scheduling constraints. Audio systems that work with very small buffers need a system with a low load and an excellent response time in order to avoid buffers getting dropped.

Denormalized floating point numbers

A reoccurring question on many multimedia newsgroups and mailing lists on the internet is: "There is hardly any input signal present, yet I experience an immense CPU load. What is the cause?" The cause is an effect that is sometimes known as the Denormal Bug.

Floating point units of processors are very well capable of doing floating point operations very quickly, except when a denormalized floating point number is involved. Calculating with denormalized floats can be up to 30 times slower than calculations without them, depending on the processor. A good description of the problem, its cause and some possible workarounds can be found in [52].

There are 2 workarounds for this problem. A first one is curing: to try to detect denormalized floating point numbers, and just replace them by 0 - since they are very small anyway. This implies however that all data must be tested. Though the test is very simple, it could be a burden if a lot of data or a lot of intermediate results are involved. Using SSE, there is a simple workaround: setting the 15th bit of the control

register, we can make SSE registers automatically flush denormalized results to 0, thus eliminating the problem. The SSE intrinsic to be used is `_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);` .

A second method is prevention: to try and avoid having any denormalized floating point numbers as result. If a calculation would yield a very low value, the result can be automatically set to 0. Whether this is possible, depends on the the calculations involved. The most popular way is probably testing operands on size, using constants for very large and very small values. For example, when the nominator of a fraction is a great deal smaller than the denominator, the result will probably be so small that without too much trouble it can be rounded to 0 before even making the division.

memory allocation

There is just one rule for memory allocation in real-time processing routines: don't. A call to `malloc()` or a `new` operator relay their request for a chunk of memory to the operating system, which can take an unspecified amount of time to actually process the request. Response time of the operating system can vary from just a few milliseconds to several seconds, depending on system load, structure of the main memory, etc.

Since dependency on the memory allocation facilities of the operating system completely wrecks run-time stability, it is common in every real-time signal processing application to allocate all possible memory that should be needed during processing, before the processing starts - for a VST plugin, the constructor or a special initialization function are ideal places. The processing routine itself then just has to use the reserved space.

2.2.2 Oversampled Lookup Tables

In the first section of this thesis, it became clear that for amplitude calculation and frequency estimation, several functions can be provided that can be precomputed. All are based upon the window function, for which a Blackman-Harris function was chosen, as defined in equation 1.9 and which will be denoted as w further on. Its Fourier transform, W, was defined in equation 1.31. Of importance are also its first and second derivatives W' and W'', defined in equations 1.49 and 1.56 respectively. The Fourier transform of the squared window function was defined in equation 1.32 and will be de-

noted Y. Its second derivative Y'' is needed too, as defined in equation 1.61.

All of these functions are either even or uneven, thus requiring that only half of it is precomputed. The main problem is that the functions are continuous: they can have any real argument. Precomputation of those functions over the whole range of floating point numbers is not feasible, instead we can use without too many problems a small approximation: the functions are clearly all very smooth, continuous, and have a very limited range, so an oversampling factor of say, 10000, is more than enough to obtain a sufficient degree of precision.

Each element of W consists of a summation over the elements of w, multiplied by an complex exponential factor $e^{-2\pi i m \frac{n-n_0}{N}}$. Since the window function w is real and symmetric, its Fourier transform is also real and symmetric. Using Eulers formula as seen in equation 1.3, we can only retain the real part of the complex exponential, which is $cos(-2\pi m \frac{n-n_0}{N})$. Also, since the cosine is an even function, we can omit the $-$ sign inside.

The entire computation of W' can then be done in a single routine. The window is supposed to be stored in an array of 2048 samples wide, equally spread among 1 period of the window - that is between $-\pi$ and π, since it is a sum of cosines. When calculating the Fourier transform, we apply the same trick: $\frac{n-n_0}{N}$ and $nStep$ are calculated such that n iterates over 2048 samples equally spaced between almost -0.5 and 0.5. The multiplication with 2π in the cosine then yields a range between $-\pi$ and π.

Since the Fourier transform of the window is also bandlimited, the values outside the main lobe are insignificant. The width of the main lobe is no more than 8 frequency bins as can be calculated from figure 1.13, and since W is a symmetric function too, W has only significant values between 0 and 4. To obtain the oversampling, we calculate the value of W not for integer samples between 0 and 4, but for floating point samples between 0.0000 and 3.9999 by increments of 0.0001, so we can store only the main lobe of W in an array of 40000 floating point numbers. The result can be clearly seen in figure 1.17.

```
float nStep = 1.0f / ((float)N);
for (int m=0; m < 40000; ++m) {
    float n = -((((float)N)-1)/2)/N;
    for (j=0; j<2048 ; ++j) {
        W[m] +=  w[j]*((float) cos(2*pi*m*n/10000));
        n += nStep;
    }
}
```

The derivatives of the window can also be calculated in a similar way - but make sure to oversample the derivative of W, and not to take the derivative of the oversampled W. Both are very different: the oversampling factor 10000 does not need to be taken into account in the derivation. The derivative of W is also tricky to use because of its antisymmetry. It is just fine to store only half of the function, but one has to keep in mind that when using it, a negative argument also yields multiplying by -1.

As can be seen from equation 1.49, W' is the same as W, apart from the multiplication with an imaginary line $-2\pi in$. Working out the complex exponential just like before using equation 1.3, and noting that the derivative of a real function must also be real, we arrive at the following equality:

$$\Re\left[-2\pi i n w_n \left(cos(-2\pi mn) + isin(-2\pi mn)\right)\right] = 2\pi n w_n sin(-2\pi mn) \quad (2.1)$$

which can be coded exactly like the previous code listing, but then with the calculation

```
Wd[m] += 2*pi*n * w[j]*((float) sin(-2*pi*m*n/10000));
```

For the second derivative, we can follow the same logic. The equality

$$\Re\left[4\pi^2 i^2 n^2 w_n \left(cos(-2\pi mn) + isin(-2\pi mn)\right)\right] = -4\pi^2 n^2 w_n cos(2\pi mn) \quad (2.2)$$

leads to the calculation

```
Wdd[m] += -4*pi*pi*n*n * w[j]*((float) cos(2*pi*m*n/10000));
```

For Y, the Fourier transform of the squared window and its derivatives, the code is analogous. As can be calculated from figure 1.15, the width of the main lobe of Y and its derivatives is 16 frequency bins. Exploiting symmetry or antisymmetry of the function here leads us to the requirement that, when using an oversampling factor of 10000, an array of 80000 elements is needed to store the complete oversampled main lobes. The resulting functions can be seen in figure 1.17.

2.2.3 Shifted Matrix Storage

The matrices used in the amplitude calculation and frequency optimization routines are band diagonal. It is known beforehand that the elements in the upper right and lower left corners are all zero. Imagine storing the matrix in memory, instead of as a square, just as a thick diagonal line. That way we can get rid of the zeroes. The main diagonal of the matrix translates to the middle column of the shifted matrix. The diagonals underneath the main diagonal translate to columns on the left side in the shifted matrix, and diagonals above the main diagonal translate to columns on the right side of the shifted matrix. It could be seen as just turning the normal matrix by 45 degrees to the right, and cutting away all zeroes.

The first thing we need to know then is how wide this shifted matrix needs to be, that is, how many diagonals are there in the band diagonal matrix. This value can be calculated from the model. In the matrices, we need the values of the functions Y or Y''. When the argument to these functions is inside their main lobe, this yields a non-zero value. This is the case when the argument, of the form $\omega_k \pm \omega_l$, is small enough - no larger than the width of the main lobe, which is 8 frequency bins to the positive side and 8 frequency bins to the negative side.

We can calculate how many diagonal bands the matrices will have, starting from the model. Iterating over all frequencies, we can count how many other frequencies are within a distance of 8 frequency bins of the current frequency. Those frequencies will have a difference that falls inside the main lobe. If frequencies ω_a and ω_b lie close together, then the diagonal matrices will have 3 diagonals: the leftmost one for the values $\omega_a - \omega_b$, the middle one for the values 0, the rightmost one for the values $\omega_b - \omega_a$. All yield non-zero values.

In general, for N frequencies lying close enough together, the band diagonal matrix will contain a square non-zero submatrix on the main diagonal of width and height N. The maximum number of diagonals is then proportional to the number of elements of the largest frequency cluster that we can find. Clusters of N frequencies give rise to $2N - 1$ diagonals, including the main diagonal. This number of diagonals of the normal matrix forms the width of the shifted matrix. The height of this matrix stays the same, and equals the number of frequencies that are under consideration.

When accessing the elements of the matrix, the coordinates of the elements of the old diagonal matrix must be translated to coordinates of the new shifted matrix. Suppose that we access an element on row k, then this row k stays the same in the shifted matrix. Column l in the old matrix translates to column $l + D - k$, D being half the number of diagonals (rounded up when column enumeration starts from 1, rounded down when it starts from 0). An element on the main diagonal has $l = k$, and thus will be positioned in column D, the middle column of the shifted matrix.

If a matrix is symmetric, the element on position k, l has the same value as the element on position l, k, and thus it needs to be computed only once and filled in in the 2 positions. The symmetry of arguments does not hold any more in the shifted matrix. An element $k, l + D - k$ in a shifted matrix has its symmetric counterpart in the element $l, k + D - l$. In its own coordinates, say m, n, an element on that position has a symmetric element on position $n + m - D, 2D - n$.

Special care must be taken when accessing elements of a matrix when it is stored in a shifted matrix form: if the accessed element is zero valued and is not stored, one shouldn't try to retrieve it from the shifted matrix - which gives öut of boundërrors. As with every data structure, the necessary boundary checks have to be implemented to assure that nothing of the sort happens.

Chapter 3

Conclusions and Future

3.1 On the theorical level

Sinusoidal modelling is a very powerful technique for audio analysis, concerned with finding the real frequencies and their corresponding amplitudes in a signal. The inherently large complexity of the methods used has been fundamentally reduced by the optimizations developed by D'Haes [5]. Incorporating the window function into the modelling allows for a significant speedup that can even run in real-time. The presented sinusoidal model allows for any set of frequencies and amplitudes, but eventually features of harmonic sounds and polyphonic audio can be incorporated relatively easy.

3.1.1 Windowing

The techniques described here are very powerful, but nevertheless they can fail. A lot depends on the initial estimate of the sinusoidal model. If the initial model estimate differs too much from the real signal, convergence to the optimal model will fail because no optimization is possible. The elements of the model must all be within the main lobe of one of the real frequencies, if gradient and Hessian calculation are to work. A window with a very wide main lobe will allow for more relaxation on the initial model.

The other way round, when the signal contains frequencies that are so close to eachother that their frequency responses almost overlap, the spectral peaks of the responses are not distinguishable any more and the optimizer cannot distinguish the 2 separate frequencies. The frequencies that are present in the signal, must thus be spaced enough from eachother so that each frequency still corresponds to a local maximum in the spectrum. This

is also due to the window: a window with a very narrow frequency response will perform much better in this respect.

In every respect, we are thus prisoner of the chosen window function. It might be interesting to investigate whether this could be exploited in any way: start the optimization using a window with a wide main lobe, and after a few iterations, switch to a window with a small main lobe. The window could also be adapted to the pitch that is most likely present in the signal. Instead of changing the window, changing the frame size might also be interesting. These so-called multiresolutional approaches often require heavy calculations, but are frequently used in audio encoding ([53], [54]).

3.1.2 The Model

The sinusoidal model as has been used here, assumes that amplitudes are independent of frequencies. The frequencies have to be optimized, the amplitudes only have to be calculated. Alternatively doing both can, in a worst case scenario, converge very slowly.

But actually, the amplitudes could be seen as a function of the frequency. In the Fourier domain, each frequency is linked to exactly one amplitude. When adapting the model in such way that the amplitudes are considered as dependent on the frequency (take equation 1.10 and replace a_k by a_{ω_k}), it might be possible to devise an optimization method in which both amplitudes and frequencies can be optimized in a single step. The partial derivatives of the error function, to begin with, would look entirely different.

The model as set up here works for any series of frequencies, but does not exploit eventual internal relations (yet). For instance, Coinciding frequencies have to be filtered out first by preprocessing to avoid crashes in the optimization routines. Investigating the special cases that arise when working with harmonic sounds and polyphonic music, might lead to the development of special cases of the model that are better suited for working with sounds that are comprised of different interrelated components.

3.2 On the practical level

When implementing audio routines for real-time usage, speed and efficiency is all that matters. Developing audio applications require from the programmer that some specific concerns are kept in mind that one often finds out the hard way. Mathematically complicated and thus error-prone algorithms like the ones here can be implemented in C, tested in Matlab using Mex functions, and afterwards seamlessly incorporated into other productions - for example VST plugins.

3.2.1 Deployment

The implementations made here only form a framework that can be used by projects in other contexts. Just analyzing incoming audio is not really useful if one is not going to do anything with the analysis. A sinusoidal model is much more interesting to use than a normal signal spectrum, because the model contains the real frequencies and amplitudes, and a spectrum is only a leaky approximation of those.

To remain in the sound analysis and synthesis world, a possible usage could be multipitch tracking and/or correcting [27]. The model allows for independent manipulation of frequencies and their amplitudes, so one can also easily resynthesize a changed frequency. When using harmonic sounds, a whole set of frequencies might be adapted at once: one fundamental frequency and its harmonics, without interfering with other frequencies independent from that sound. Polyphonic audio is here a special challenge too, because of the great amount of overlap in present frequencies.

On a more ambitious path, it might be possible to attach a pitch tracker to a learning algorithm, thereby learning a sinusoidal model to detect different instruments or behave like different instruments ([55], [56]). If the model can be trained to take specific information on the acoustic properties of the present instruments into account, pitch tracking and polyphonic audio handling become much easier - it is also the way humans distinguish different instruments playing together: each instrument has other harmonical 'model' characteristics. This might also free up the path to advanced sound source separation [11].

The quasi-exact pitch tracking capabilities of a sinusoidal model can be used to make advances in score following, performance error analysis, or comparing an audio recording to a musical partition - so called score-performance matching ([57], [58]). Applications in automatic accompaniment are also possible ([59], [60]).

Sinusoidal modelling can be seen as a form of feature extraction from an audio source. The frequencies and amplitudes of the signal are features, much more accurate and compact than a spectrum. This allows much better for all kinds of analysis of musical collections, in the fields of music theory, structure, etc. The extracted features can be stored and thus form a description of the audio, which can be used later in data mining applications ([61], [55]).

3.2.2 A broader perspective

The model and algorithms described here were developed with specifically sound applications in mind. But they are so general that they could be used in any environment where signal processing is used. It might be worthwile to search for other uses of sinusoidal modelling and try a valorization in other domains.

List of Figures

Bibliography

[1] J. Ganseman and W. D'haes, "Score-performance matching in practice: Problems encountered and solutions proposed," 2006, presented at RMA Research Students' Conference 2006.

[2] Steinberg Media Technologies GmbH, "VST (Virtual Studio Technology)." [Online]. Available: http://www.steinberg.net/325_1.html

[3] Recordare LLC, "MusicXML." [Online]. Available: http://www.musicxml.org/xml.html

[4] MIDI Manufacturers Association, *Complete MIDI 1.0 Detailed Specification*, MIDI Manufacturers Association Std., Rev. 96.1, november 2001.

[5] W. D'haes, "Automatic estimation of control parameters for musical synthesis algorithms," Ph.D. dissertation, University of Antwerp, june 2004.

[6] J. L. Flanagan, "Parametric coding of speech spectra," in *Journal of the Acoustical Society of America*, vol. 68, no. 2, august 1980, pp. 412–419.

[7] R. J. McAulay and T. F. Quatieri, "Speech analysis/synthesis based on a sinusoidal representation," in *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 34, no. 4, august 1986, pp. 744–754.

[8] G. Bailly, E. Bernard, and P. Coisnon, "Sinusoidal modelling," 1998.

[9] A. Syrdal, Y. Stylianou, L. Garrison, A. Conkie, and J. Schroeter, "TD-PSOLA versus harmonic plus noise model in diphone based speech synthesis," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 1, may 1998, pp. 273–276.

[10] J. Paulus and A. Klapuri, "Measuring the similarity of rhytmic patterns," in *Proceedings of the 3rd International Conference on Music Information Retrieval*, october 2002, pp. 150–156.

[11] T. Virtanen and A. Klapuri, "Separation of harmonic sound sources using sinusoidal modeling," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 2, may 2000, pp. 765–768.

[12] H. Ye and S. Young, "High quality voice morphing," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, may 2004, pp. 9–12.

[13] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," in *Mathematics of Computation*, vol. 19, april 1965, pp. 297–301.

[14] I. J. Good, "The interaction algorithm and practical fourier analysis," in *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 20, no. 2, 1958, pp. 361–372.

[15] S. G. Johnson and M. Frigo, "A modified split-radix fft with fewer arithmetic operations," in *Mathematics of Computation*, vol. 55, no. 1, january 2007, pp. 111–119.

[16] C. Roads, *The computer music tutorial*. MIT Press, 1996.

[17] W. H. Press, W. T. Vetterling, S. A. Teukolsky, and B. P. Flannery, *Numerical Recipes in C++: the art of scientific computing*, 2nd ed. Cambridge University Press, 2002.

[18] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd ed. Prentice Hall, 2002.

[19] H. A. Gaberson, "A comprehensive windows tutorial," *Sound and Vibration*, pp. 14–23, march 2006.

[20] F. J. Harris, "On the use of windows for harmonic analysis with the discrete fourier transform," in *Proceedings of the IEEE*, vol. 66, no. 1, january 1978, p. 5183.

[21] A. H. Nuttall, "Some windows with very good sidelobe behavior," in *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 1, february 1981, p. 8491.

[22] H. Taube, "Automatic tonal analysis: Toward the implementation of a music theory workbench," *Computer Music Journal*, vol. 23, no. 4, pp. 18–32, 1999.

[23] D. R. Hofstadter, *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books, 1979.

[24] M. R. Portnoff, "Implementation of the digital phase vocoder using the fast fourier transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, no. 3, pp. 243– 248, june 1976.

[25] M. S. Puckette and J. C. Brown, "Accuracy of frequency estimates using the phase vocoder," *IEEE Transactions on Speech and Audio Processing*, vol. 6, no. 2, pp. 166–176, march 1998.

[26] A. V. Oppenheim and R. W. Schafer, "From frequency to quefrency: A history of the cepstrum," *IEEE Signal Processing Magazine*, vol. 21, no. 5, pp. 95–106, september 2004.

[27] T. Tolonen and M. Karjalainen, "A computationally efficient multipitch analysis model," *IEEE Transactions on Speech and Audio Processing*, vol. 8, no. 6, pp. 708–716, november 2000.

[28] C. Yeh, A. Röbel, and X. Rodet, "Multiple fundamental frequency estimation of polyphonic music signals," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 3, march 2005, pp. 225–228.

[29] C. Bishop, *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[30] Steinberg Media Technologies GmbH, "VST Software Development Kit version 2.4 rev. 2," november 2006. [Online]. Available: http://www.steinberg.de/324_1.html

[31] Muse Research, Inc., "KVR Audio Plugin Resources." [Online]. Available: http://www.kvraudio.com

[32] Apple Computer, Inc., "Audio Units." [Online]. Available: http://developer.apple.com/audio/audiounits.html

[33] Avid Technology, Inc., "Real Time Audio Suite." [Online]. Available: http://www.digidesign.com/

[34] Microsoft Corporation, "DirectX." [Online]. Available: http://msdn.microsoft.com/directx/

[35] R. Furse, "LADSPA." [Online]. Available: http://www.ladpsa.org

[36] FXpansion, "VST to AU and VST to RTAS adapters." [Online]. Available: http://www.fxpansion.com/index.php?page=31

[37] F. Vanmol, "Cubase VST SDK for Delphi v2.4.2.1." [Online]. Available: http://www.axiworld.be/vst.html

[38] D. Martin, "jVSTwrapper." [Online]. Available: http://jvstwrapper.sourceforge.net/

[39] Steinberg Media Technologies GmbH, "VSTGUI: Graphical User Interface Framework for VST plugins, version 3.5," february 2007. [Online]. Available: http://vstgui.sourceforge.net/

[40] Microsoft Corporation, "GDI+." [Online]. Available: http://msdn2.microsoft.com/en-us/library/ms533798.aspx

[41] The Open Group, "Motif 2.1." [Online]. Available: http://www.opengroup.org/motif/

[42] AudioNerdz, "Delay Lama," may 2002. [Online]. Available: http://www.audionerdz.com

[43] TrollTech, "Qt: Cross-Platform Rich Client Development Framework." [Online]. Available: http://trolltech.com/products/qt

[44] OpenGL Working Group, "OpenGL version 2.1," august 2006. [Online]. Available: http://www.opengl.org/

[45] S. Thakkar and T. Huff, "The Internet Streaming SIMD Extensions," *Intel Technology Journal*, vol. Q2, pp. 1–8, may 1999.

[46] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual," may 2007. [Online]. Available: http://developer.intel.com/products/processor/manuals/index.htm

[47] F. Franchetti and M. Püschel, "SIMD Vectorization of non-two-powered sized FFTs," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 2, april 2007, pp. 17–20.

[48] The MathWorks, "Matlab 7 for Windows," 2006. [Online]. Available: http://www.mathworks.com/products/matlab/

[49] H. Haas, "The influence of a single echo on the audibility of speech," *Journal of the Audio Engineering Society*, vol. 20, no. 2, pp. 146–159, march 1972.

[50] J. R. Ashley, "Echoes, reverberation, speech intelligibility and musical performance," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 6, april 1981, pp. 770–772.

[51] C. Chafe, M. Gurevich, G. Leslie, and S. Tyan, "Effect of time delay on ensemble accuracy," in *Proceedings of the International Symposium on Musical Acoustics*, april 2004.

[52] L. de Soras, "Denormal numbers in floating point signal processing applications," april 2004. [Online]. Available: http://ldesoras.free.fr/

[53] S. N. Levine, T. S. Verma, and J. O. Smith III, "Multiresolution sinusoidal modeling for wideband audio with modifications," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, may 1998.

[54] K.-H. Kim and I.-H. Hwang, "A multi-resolution sinusoidal model using adaptive analysis frame," in *Proceedings of the 12th European Signal Processing Conference (EUSIPCO 2004)*, september 2004, pp. 2267–2270.

[55] P. Herrera-Boyer, G. Peeters, and S. Dubnov, "Automatic classification of musical instrument sounds," *Journal of New Music Research*, vol. 32, no. 1, pp. 3–21, 2003.

[56] P. Herrera, A. Yeterian, and F. Gouyon, "Automatic classification of drum sounds: a comparison of feature selection methods and classification techniques," in *International Conference on Music and Artificial Intelligence*, september 2002.

[57] H. Heijink, L. Windsor, and P. Desain, "Data processing in music performance research: Using structural information to improve score-performance matching," *Behavior Research Methods, Instruments & Computers*, vol. 32, no. 4, pp. 546–554, august 2000.

[58] H. Heijink, P. Desain, H. Honing, and L. Windsor, "Make me a match: an evaluation of different approaches to score-performance matching," *Computer Music Journal*, vol. 24, no. 1, pp. 43–56, april 2000.

[59] R. B. Dannenberg, "An on-line algorithm for real-time accompaniment," in *Proceedings of the International Computer Music Conference*, 1984, pp. 193–198.

[60] M. Puckette and C. Lippe, "Score following in practice," in *Proceedings of the International Computer Music Conference*, 1992, pp. 182–185.

[61] A. Lerch, G. Eisenberg, and K. Tanghe, "Feapi: A low level feature extraction plugin api," Proceedings of the 8th International Conference on Digital Audio Effects, Madrid, Spain, September 2005.