Thesis

Name :	Sukandar Kartadinata
Title :	Algorithms for Combined Sound Analysis and Synthesis and their Implementation in a Multi–Processor Environment
Location :	Center for New Music and Audio Technologies, Berkeley, USA
Examiner :	Prof. Dr. Meisel
Co-Examiner :	Prof. Lenhardt
Supervisor :	Prof. Wessel
Co-Supervisors :	Adrian Freed Marie-Dominique Baudot

Ich erkläre hiermit an Eides statt, daß ich die vorliegende Diplomarbeit selbstständig und ohne unzulässige Hilfe angefertigt habe. Die verwendeten Literaturquellen sind im Literaturverzeichnis vollständig zitiert.

Karlsruhe, den ....

Sukandar Kartadinata
sk@zkm.de

Preface

Music and technology are topics that have fascinated me since I started playing guitar and soldering electronic kits at the age of twelve. I have always tried to combine these two interests throughout my education and I am grateful that with this thesis I had the opportunity to extend my knowledge in a project that involved guitar issues as well as my favoured fields of computer science. Although playing guitar was admittedly sometimes a distraction during the research, I must emphasize the strong synergy I experienced from this musical side of the work. Despite the fact that I value a technician - musician collaboration very much, I was happy to combine my own musical and technical skills in this unique project.

Although this thesis is submitted at the German institution 'Fachhochschule Karlsruhe - Hochschule für Technik' , it is written in English since all the research was done at an American university. The 'Center for New Music and Audio Technology' (CNMAT) at the 'University of California, Berkeley' (UCB) provided an interdisciplinary environment that boosted the progress of my research. I had the opportunity to meet a lot of people with an outstanding knowledge of computer music issues, which gave much inspiration. At CNMAT, I attended a seminar about music related digital signal processing topics that broadened my horizon beyond the scope of this thesis. In addition to the research issues, I benefited a lot from the cultural experience in a foreign country.

A lot of people have supported this project in many different ways. At this point I would like to thank all of them for their contribution.

The 'Deutscher Akademischer Auslandsdienst' (DAAD) for making this experience possible financially.

Prof. Dr. Meisel from the 'Fachhochschule Karlsruhe - Hochschule für Technik' for accepting and supporting this project. Also thanks to Prof. Exner.

CNMAT for supplying the research environment and for the warm hospitality from all of its people. In particular, Prof. David Wessel for inviting and supervising me; Adrian Freed for his research support; Leslie Delehanty for administrational issues. Special thanks to Michael Goodwin for the proof reading.

Two people from the former 'Gibson Western Industrial Zone' (GWIZ) I have to mention for their continuing assistance. Marie-Dominique Baudot for DSP Q&A and positive vibes. Keith McMillen for motivation and the shared obsession for guitar technology.

Finally, thanks to the people that helped me getting over: Heike Staff, Pierre Dutilleux, Nic Collins, and Johannes Goebel.

Table of Contents

List of Figures

List of Symbols

$x(t)$	continuous time signal
t	time
$x(n)$	discrete time signal
n	time index
$X(z)$	z-transform of a discrete time signal
z^{-1}	unit delay
$\sum_{n=a}^{b}$	summation from a to b for index n
$\int_a^b x(t)dt$	integral over x(t) from a to b
$\lfloor \ldots \rfloor$	truncating a number leaving its integer part
$x(n)$	input signal
$y(n)$	output signal
$h(n)$	impulse response
$H(z)$	transfer function
$A(z)$	zeros of a rational transfer function
$B(z)$	poles of a rational transfer function
a_n, b_n	coefficients for a rational transfer function
n, m, l, k	time indices
$y_D(n)$	decimated signal
$y_E(n)$	expanded signal
M	downsampling factor
L	upsampling factor
$\hat{H}(z)$	downsampled transfer function
$X_{STFT}(e^{j\omega}, m)$	short time fourier transform
$X_{CTSTFT}(e^{j\Omega}, m)$	continuous time short time fourier transform
ω , Ω	normalized frequency
j	imaginary number $\sqrt{-1}$
$w(n)$, $w(t)$	window function
$X_{DWT}(k, m)$	discrete wavelet transform
$X_{DTWT}(k, m)$	discrete time wavelet transform

a	scale factor
k	scale level
T_{period}	length of period of the fundamental frequency
f_s	sampling frequency
g	feedback attenuation
$y_{CFLP}(n)$	comb and lowpass filtered signal
$y_{CFLPDS}(n)$	comb and lowpass filtered signal, downsampled
$y_{CFLPDSR}(n)$	y_{CFLPDS}, rectified
$y_{max}(n)$	maxima signal
K	maxima grid size
$y_{avg}(n)$	average signal
J	average summation length
$flag_{trigger}(n)$	binary trigger decision
$y_{PSA}(n)$	pitch-synchronous average signal
P	length of delay line

1 Introduction

1.1 What Sounds ?

The title of this project, despite its length, generalizes the subject matter of the thesis. In particular, it does not imply any constraints on the range of sounds that should undergo the analysis and synthesis algorithms. However, throughout the course of the research the context has been the "world of computer music" and in the case of the analysis part it became even more specific, concentrating on the sounds produced by (electric) guitars.

This specification towards music does not mean that any particular sound is "allowed" or "forbidden" for musical use. However, it changes the point of view. Consider bird sounds; they are of interest to the ornithologist for classification or physiological purposes, whereas the musician will take them as inspiration for melodies and rhythms, use them as sound effects or even compose a whole piece by processing them. Though their applications differ, both might use the same tools and methods in order to accomplish their work, such as tape-recorders, spectrograms, time-stretching[1] or, maybe most important, their ears. There is thus the possibility that the work presented here might be of some use for non-musical audio applications, *e.g.* speech recognition.

[1] Time-stretching: changing the speed of a recording without affecting its pitch

1.2 Synthesis by Analysis

1.2.1 The Classical Approach

Since the birth of electronic music in the early 1900s (followed by computer music in the past thirty years), composers, performers, and musical instrument designers have sought various models for synthesizing sounds. The most common strategy has been to consider existing sounds or the way they are produced, and then to devise a model for their generation. This can be compared to chemistry, where a thorough analysis of natural[2] material is helpful if not necessary prior to the (re)synthesis of its artificial counterpart [35].

Achieving perfect reconstruction of an audio signal in an analysis-synthesis scheme is a desirable goal in itself, and a key issue in applications that involve data compression. However, this is not a high priority in electronic music applications, where the main goal of analysis is to extract meaningful information from the signal, thus creating an abstract representation. This allows for flexibility and transformations, which is essential to music in general and to computer music in particular.

Electronic music is not just about imitating acoustic instruments. In addition to that, the aim of musicians with an interest in electronics and computers is often to explore new territories, to find novel synthetic sounds with a quality comparable to natural sounds. Defining this quality is not easy, but good indicators include richness, nuance and reactivity.

1.2.2 Bringing it on Stage

The last term in the above list relates to the behavior of an instrument in a live situation. Reactivity[3] can be defined as the ability of a system to respond to an input, which for an instrumentalist on stage must include real-time response and a consistent user interface. For almost all acoustic instruments these features are inherently defined by their

[2] Here, 'natural' is meant in a relative sense, referring to instances prior to a technological process, while 'artificial' describes the state of its result.

[3] The term reactivity is used instead of interactivity which in addition to the former also deals with the feedback between the system and the user resp. the ability of the system to support that

physical structure[4], whereas for their electronic counterparts it depends on the overall design of the system and its processing powers.

In this thesis an attempt is made to build on the conceptual or off-line analysis-synthesis approach described in the previous section by exploring ways to use real-time sound analysis as a mean to extract instantaneous information about the actual playing of an instrument. This information is then used to drive a synthesis algorithm, which can employ a completely different model for sound generation than the analysis. For the case where the analysis and synthesis models are the same, the possibility of links on a low abstraction level will be explored.

In contrast to the common sensor-based approach to music-related user interfaces, the approach proposed here leaves the instrumentalist with the physical entity of his instrument of choice. For this application, the electric guitar has the advantage that the original sound can be almost muted if the musician only wants to listen to the synthesized sound.

[4] There are a few exceptions as to real-time response, e.g. the church organ has significant delay

1.3 The Center for New Music and Audio Technologies[5]

The Center for New Music and Audio Technologies (CNMAT) was established at the University of California, Berkeley in 1987 to provide, promote, and present creative interaction between music and technology. In addition to their performance related projects, CNMAT is actively engaged in multi- and hyper-media, composition and performance software development, digital signal processing for analysis and synthesis of audio, music perception and cognition, and educational uses of computers. Its particular emphasis is new performance technology and performance-related research issues, such as real-time synthesis and control.

CNMAT maintains close affiliations with Stanford University's CCRMA[6] and France's IRCAM[7]. In collaboration with the latter it has developed a flexible analysis/re-synthesis environment based on sinusoidal modeling and additive synthesis. It has also explored musical applications of neural networks and analog VLSI sound synthesis. More recent research has focused on World Wide Web interactivity in relation to synthesis tools, including Java applications and a new file format standard for spectral descriptions.

[5] URL: http://www.cnmat.berkeley.edu

[6] CCRMA = Center for Computer Research in Music and Acoustics

[7] IRCAM = Institut de Recherche et Coordination Acoustique/Musique

2 Real-time Sound Analysis - Existing Applications and Solutions in Music

2.1 Score Following

Distinctions have often been drawn between the different environments and communities in which (electronic) music is produced, but these are not necessarily sensible given that cultural, stylistic and production related borders have become less defined. However, in order to give an overview of the current usage of real-time sound analysis it is useful to consider two major groups that have computers in their musical toolboxes.

First there is the "academic world", mostly composed of musicians that adhere to the tradition of classical western music, where in most cases one composer creates a score that is performed by one or more interpreters. This model holds true for the subset of electronic music, with the exception that the performers have become partially obsolete. The machines required to produce this music, however, are often out of the realm of the majority of composers, either because they are simply too expensive or because their technology is too recent (or not profitable enough) to make them commercially available.

This has lead to a situation where institutions provide the means of production for a rather small number of composers. These institutions are usually externally funded, either by taxes or sponsors, and often linked to a university or conservatory. This model has proven successful in many cases. For instance, it has lead to the propagation of state-of-the-art technology from computer science to music departments. It has also encouraged close collaborations between musicians and technicians, mathematicians, psychologists and others. In addition to that, an environment for production and presentation of the musical pieces is often maintained; this includes recording studios and concert series.

The most widely used application here (within relevance to the topic of this thesis) is score following, where the computer listens to the notes played, compares them to a pre-defined score and synchronizes its "own contribution", which can be another score or the result of a specific algorithm. Usually the pitches of the notes are tracked, but other parameters such as amplitude, articulation or timbre can be observed. More elaborate implementations not only look for single notes, but groups of notes; this improves the stability in the case of undetected notes, and leads to additional parameters like playing speed [12].

Score following allows for flexible timing in situations where some part of the music is played by acoustic instruments and some by a computer plus synthesizer. The advantage is thus that the performers are not bound to a fixed beat coming from the computer, rather they conduct the machine.

2.2 Guitar Synthesizers

The second group that makes extensive use of music technology are the rock, pop and jazz musicians. In contrast to the above, they have a more commercially biased relation to music. Such musicians cannot rely on the resources of institutions, but have to buy devices that are available on the market, at least in the pre-production phase. Then they have to pay a studio to produce the music in a sufficient quality to be released by a record label, and finally are at the mercy of radio stations, convert venues, record labels and consumers.

Apart from this prosaic view of rock music, it differs from its academic counterpart on a more social level in that there is generally no composer-orchestra relation. Instead the common model for creating music is the band, which is usually comprised of a small number of musicians, with guitar players having a large share. Consequently, they are one of the main target groups of the music technology industry.

With the increasing popularity of synthesized sounds in the seventies, researchers began to consider user interfaces other than the piano-style keyboard which was the prominent input device at that time[8]. These thoughts led to the development of devices (among others) known as guitar synthesizers, the main rock music application of real-time sound analysis. The basic concept is to detect the fundamental frequency of a guitar sound and use this information to control a synthesizer, or in the language of the advertisements, to "play a trumpet with a guitar". Sound example B1.1 gives an impression of this idea.

[8] 'Keyboard' is actually often used as a synonym for 'synthesizer'

2.2.1 The History of Guitar Synthesizers

Guitar synthesizers were introduced in the late seventies, before digital technology became widespread. Also MIDI[9] was yet to be invented. Therefore these devices were (often closed) systems that relied on analog technology for most of their functionality, with the main building blocks being oscillators, filters, and amplifiers. However, there is an analog standard for controlling the frequencies, gains, and bandwidths of these modules; for frequency, a change of 1 Volt in the control voltage is to correspond to an octave change[10] in the frequency of the oscillator or filter.

The module that interfaces an external signal, *i.e.* the guitar sound, to this control "language" is a pitch-to-voltage[11] converter, usually comprised of a rectifier, schmitt-trigger, pulse generator and integrator. In a way it mirrors the behavior of an oscillator in that it maps frequency to voltage with the same underlying principle of 1V/octave. In addition to that, level detection circuitry is necessary to trigger events such as starting and ending notes.

With the invention of MIDI in the early eighties this module naturally became a pitch-to-MIDI converter, mapping frequency to note numbers and pitch bend information. This can be heard in B2.1. One of the main benefits of MIDI is that it allows for much better compatibility and thus for a wider choice of sound modules. On the other hand its limited bandwidth causes performance problems, while the strict note number concept cannot properly translate some playing techniques like slides, *i.e.* continuous frequency changes.

The main reason these guitar-to-synthesizer interfaces never became prominent was probably that they limit the range of expression in the playing style by considering only frequency and amplitude. Thus, no matter how a specific note is plucked it will always result in the same sound. Of course, the generality of MIDI allows for a much wider range of sounds but only at the price of flattening the subtleties. Interestingly enough, this reflects one of the main problems in synthesizer control, which is how to trade power with nuance.

---

[9] MIDI = Musical Instruments Digital Interface, a standard for digital communication between Synthesizers, Keyboards, Sequencers, Computers and other music related gear

[10] An octave equals to a frequency ratio of 2:1. Therefore the control law is logarithmic in accordance to pitch perception.

[11] Frequency is a physical measure whereas pitch relates to the perception. Often, the two are not clearly distinguished.

Even within their limited analysis capabilities, guitar synthesizer fail to work properly due to their inability to detect the correct pitch, which results in substantial trigger delay or the starting of unwanted notes. In general musicians are willing to learn the special behavior and adapt to the oddities of an instrument (*e.g.* spending many years in mastering the violin), but with guitar synthesizers the learning effort to get the analyzer working produces only constrained results, as can be heard in B2.2 and B2.3. Other reasons for their poor acceptance are the high prices and the fact that guitar players have generally been a conservative community for their majority - note that some of the only devices still employing vacuum tubes are guitar amplifiers.

2.2.2 The Roland VG-8, a New Approach

In 1994, Roland began promoting a high-tech guitar device based on a novel concept, the VG-8 (VG = Virtual Guitar), which was supposed to appeal to the conservative guitar players mentioned above. The idea was to simulate every type of guitar-amplifier-loudspeaker configuration established on the market during the last forty years. Their analysis is more evolved than mere frequency and amplitude detection; they employ a harmonic analysis of the incoming guitar sound. For the synthesis part, physical guitar models are used which include very specialized parameters like the type of pickup, its position on the body of the guitar, the angle of the microphone to the loudspeaker, and so on. The claim was that the VG-8 could enable one guitar to imitate every vintage guitar sound plus a whole set of new ones. The system even includes a few harmonic restructuring algorithms, which produce sounds that are less guitar-like but still closely linked to the dynamics of the guitar input.

Although it was a significant achievement, the VG-8 did not completely succeed in emulating arbitrary guitar configurations and picking up personal playing styles sufficiently; for example it introduces some trigger delay much like the earlier systems. For the community of players interested in experimental use of technology, the VG-8 was somewhat disappointing since it was a closed system with no MIDI output that did not provide the subtle guitar-based synthesis that they had anticipated.

2.2.3 The Infinity Box at GWIZ

Intending to overcome the limitations of guitar synthesizers, ZETA Instruments and CNMAT initiated the Infinity project in 1993. ZETA set up a research and development lab called GWIZ (Gibson Western Industrial Zone) with the (mainly financial) help of Gibson Guitars, one of the two major electric guitar manufactures.[12]

ZETA Instruments is a small company with a focus on electric string instruments and most widely known for its electric violins, which are equipped with optional pitch-to-MIDI converters that suffer from basically the same problems mentioned above. In the late eighties, ZETA made a considerable advance in guitar controller technology with the Mirror-6. This product introduced fret-scanning, a technology which is able to deliver pitch information very quickly by sending scanning signals along the strings that are picked up by wired frets. Thus, the control circuit can determine almost immediately when and where a string has been pressed against the fingerboard. However, the musician has to switch to the ZETA guitar which, although not a bad of its kind, might not be his favorite instrument. Also, wiring the frets requires an expensive manufacturing process that results in high cost.

In the Infinity project, GWIZ approached the other main problem, that of limited control, in a manner similar to the Roland approach. Apart from extracting frequency and amplitude information, they used algorithms that derived information about the spectral nature of the guitar signal. In contrast to the VG-8, the goal was not to emulate just guitar sounds but to control the wide variety of modern synthesizers.

Four individual processors were used to create a compact and self-contained box. On the analysis side, DSPs were used for FFTs[13] to extract partials and higher level parameters like odd/even ratio and brightness. These parameters could be mapped directly to the internal synthesis engine, allowing for vocoding-like[14] effects. Other synthesis algorithms included sampling and physical modeling. Also conventional effect processing modules were implemented like reverb, echo, phaser, chorus, equalization, nonlinear distortion and phase modulation. Finally, micro controllers were dedicated to handling the fret scanning information as well as doing additional pitch detection.

Though highly self-contained the Infinity Box had a quite open architecture, a substantial advantage over the VG-8. The major achievement in this respect was the introduction of ZIPI, a new protocol for communication between electronic instruments designed to overcome the limitations of MIDI (which are too numerous to be listed here) [23, 24, 43]. However, MIDI was still on the feature list for compatibility reasons.

[12] The other being Fender Guitars

[13] FFT = Fast Fourier Transformation, explained later in this thesis

[14] A vocoder maps the spectral envelope of one signal to another. A popular application are robot-like sounds where the voice is mapped onto a rectangular wave.

Although the Infinity Box was already an impressive product, there was still room for improvements, especially concerning the analysis algorithms, which had problems detecting certain fast playing techniques. This problem is depicted in B2.2 and B2.3.

2.3 ... and in-between

Although this project was primarily hosted by CNMAT, due to its close relation to GWIZ two different environments were involved. One provided the links to academic research, while the other was focused on commercial production. The intent was to benefit from the advantages of both groups by doing the system analysis, design and prototyping at CNMAT, and using the Infinity Box at GWIZ as the implementation platform.

Unfortunately, two weeks before implementation work was to begin, Gibson discontinued the main projects at GWIZ including Infinity. As a result another implementation platform was necessary. A natural choice was the Reson8, a Multi-DSP platform developed by CNMAT about seven years ago. It could be programmed with the same development tools as the Infinity Box, thus enabling a similar implementation of the algorithms and making the shift of platforms fairly smooth. Although this shift occurred in the middle of the project's course, the Reson8 is referenced to in the following sections as if it had been the initial choice.

3 Defining the Project

3.1 Classification

The main focus of this project is signal processing in the digital domain. It is placed somewhere between computer science and electrical engineering, where the latter is the more common environment to cover DSP issues. In the context of computer science, the project addressed such additional issues as automatization, parallel processing and user interfaces.

3.2 Requirements

3.2.1 ... induced by the Infinity Box

Given the problem mentioned at the end of Sec. 2.2.3, it was clear that a primary goal should be to improve the quality of the analysis in regards to fast playing styles. In particular, this involved considering alternative ways to make fast re-pluck decisions, especially on low strings. Here the most common technique had been simple amplitude or energy measurement.

In compliance with the title of this thesis, another requirement was to establish a link between the new efforts in real-time analysis and existing synthesis algorithms. Since this work depended on the successful completion of the analysis tasks, it was optional to some extent, especially for a low abstraction level of sound.

3.2.2 CNMAT Research Goals

In addition to the result-oriented GWIZ requirements, the project was intended to satisfy CNMAT research goals that involved methods. Specifically, the problem of describing transient events like re-plucks was of interest since this had applications to the CNMAT additive analysis/re-synthesis system.

With this in mind, wavelets seemed to be promising approach since they trade time and frequency resolution in a more flexible way than the more common Fourier transform. Also, wavelets had been used for both analysis and synthesis schemes, which seemed helpful for the second part of the research.

On the basis of these requirements and the coarse initial outline of the thesis, an updated schedule for the six month project was developed.

3.3 The Schedule

1. Learning phase
   - Advanced digital signal processing topics, leading to wavelets
   - Matlab and its matrix-oriented approach
2. Database compilation
   - Recording of guitar sounds for off-line tests in Matlab
3. Wavelets
   - Prototyping with Matlab scripts
   - Application to the database sounds
   - Interpretation of results compared to conventional approaches
4. Mapping of analysis results to suitable synthesis algorithms
5. Refinement
   - Consulting of additional literature
   - Choosing the most effective approach
6. Implementation
   - I. C - achieving real-time performance with a high level language
   - II. DSP - porting to the target platform
7. Final refinements of algorithms and optimization of parameters

3.4 The Tools

3.4.1 Hardware

The main computer for the prototyping phase was the Silicon Graphics Indigo (SGI) Workstation. In addition to that, a DAT[15] recorder, mixing boards, guitar-preamplifiers

[15] DAT = Digital Audio Tape

and other sound reinforcement devices were used, mainly for the purpose of creating the database. The SGI was also used for C programming during the first half of the implementation phase.

The target environment was comprised of the Multi-DSP platform Reson8, an Apple Macintosh IIfx serving as the host, and an Ariel ProPort Stereo Analog-to-Digital / Digital-to-Analog Converter. This was supplemented by a variety of sound gear including a Yamaha TX802 FM-Synthesizer, an IVL pitchrider, a small line mixer, an amplifier and two loudspeakers. Finally the hardware included the author's guitar and pre-amplifier.

3.4.2 Software

For the software component, the main application for analyzing signals and prototyping the algorithms was Matlab 4.2 running on the SGI. It was complemented by sound-related utilities supplied with the operating system, namely the 'soundeditor' and the 'soundfiler'. For the C implementation, the standard UNIX/C environment of the SGI was used, augmented by the audio function libraries, provided with the SGI system.

The DSP code development was done in the MPW environment on the Macintosh, which also served as the platform for the implementation of the user interface. This was done in MAX 3.0, an object-oriented patching language used by many computer musicians.

3.5 Target Results

The following lists completes the project definition by summarizing the target results:

a. Algorithms for robust real-time analysis of transient sound events
   and links to synthesis
b. Implementation / code in Matlab, C and DSP assembler
c. Simple user interfaces to demonstrate functionality
   - Matlab & C : scripts
   - DSP : MAX patches
d. Documentation

4 The Foundations of the Project

4.1 Information Flow in Music

As computer science is concerned with the processing of information, it is necessary to examine how this relates to music, especially in the context of this thesis which deals with deriving information from a signal on different abstraction levels. The following section presents an interpretation of how information flow can be observed in traditional music, *i.e.* music without the use of electronics or computers. This provides a basis for discussing the influences of computers on the information flow in music.

4.1.1 The Traditional Case

The first observation in a telecommunication model of music is that there is usually a composer representing the transmitter, while the audience can be viewed as the receiver. The former has a certain musical idea that he wants to convey, but in general cannot accomplish in a direct way. Instead, he is restricted to certain channels, instruments, and tools, which in the language of communication theory are usually referred to as media.

Three main steps are necessary to describe the information flow from the original musical concept to the perceived acoustic sensation. First, the composer has to write the score, which he does by using standardized symbols that are common to all musicians, such as staffs, notes and rests. These symbols are bound to certain sounds (including silence), and it is part of the composer's task to predict or envision the sonic result based on his experience. The actual transformation is then performed in a second step by the interpreter, who reads the score and translates the symbols into physical actions performed on an instrument. Finally the instrument responds to these actions and creates a sound; this process is governed by its physical structure. The sound is then transmitted to the listener via the air, with the room acoustics having an additional influence on the result.

In addition to that, there are subtle but important feedback paths in the information flow. This occurs in the process of score writing when the composer "tests" his tunes by playing them *e.g.* on a piano. For the musician feedback means listening to the sound of his instrument while playing and adjusting it accordingly to achieve the desired result.

Furthermore, music has some sort of influence on society to which in turn the composer reacts to in his work. This proposed model is illustrated in Fig. 4.1.1.

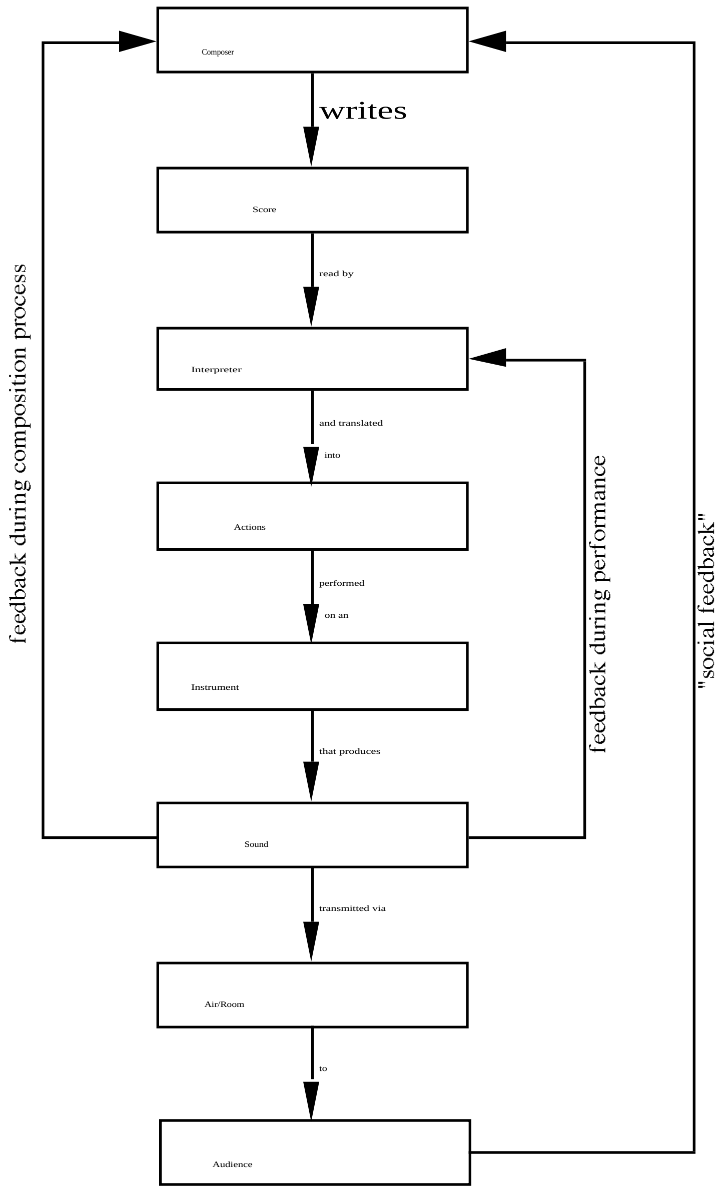


Figure 4.1.1: Information flow in music

4.1.2. Influences of Information Technology

Perhaps the most important reason that information technology has influenced so many aspects of life is the fact that information, which is inherent to every process, can be treated in a uniform way. In contrast, processing the other two entities, matter and energy, requires specific technology. This is why a computer can be used for word processing as well as for controlling a nuclear power plant.

In the context of the previous section, a computer can replace almost every part in the link between the source/composer and the target/audience. Scores can be written on a computer much like a letter or source code, using the so-called sequencer; electronic instruments can be designed to translate a physical action into sound by interfacing sensors to a synthesis engine; and room acoustics can be simulated with artificial reverberators. However, there is currently no computerized way to bypass the last element of the chain, *i.e.* the ear of the listener, which must be stimulated by an air pressure wave.[16] One conclusion is that electronic music always depends on loudspeakers to convert electric currents into air movements.

Apart from these replacements, it is important to mention some consequences that are more complex than just having new sounds or more efficient score editing. One is the striking fact that instrumentalists are no longer a necessary part of music production. An electronic score does not need to be printed, read, and interpreted. Rather it can be transferred directly to the synthesis algorithm. Another quality of information is the possibility to store, copy and transport it, which with regard to musical applications laid the foundation of the record industry; currently it enables the distribution of musical data on the internet. Computers also changed the way music was composed, *e.g.* by introducing algorithmic composition where the composer does not specify the score note by note, but instead by using more abstract structures, like scales or other patterns.

4.1.3 Sensors vs. Sound Analysis

In the introduction it was mentioned that the real-time sound analysis approach of this thesis can be viewed as an alternative to the more common sensorial concept for creating musical electronic instruments. This sensor-based approach is described next.

[16] A (rare) exception are cochlear implants

The most well-known sensor instrument is the keyboard[17], which resembles the traditional "user interface" of the piano. However, in the electronic case a key is nothing more than two switches where the time lag between opening the upper and closing the lower one is usually used as a loudness control. Other sensory input comes from pressure applied to the keys or potentiometers build into foot pedals and hand wheels.

More exotic sensor instruments have been developed for players who wanted to move beyond the limited concept of a keyboard.[18] Some approaches mimicked the user interface of other classes of instruments; *e.g.* wind instrument controllers were developed that had keys for bores and measured strength of breath. Other designs overcame the constraints of classical instruments and tried to make better use of the general physical capabilities of the human body. Also, the use of space gained importance and localization techniques were used to interface dance to music, for example, thereby reversing the usual causalities.

In the context of the aforementioned information flow in music, the difference between the sensor-based and the sound analysis approaches becomes more defined. While sensors derive measures (directly) from the movements/actions of the performers, real-time sound analysis obtains meaningful parameters by examining the result of such actions on a physical entity, as shown in Fig. 4.1.2. An advantage of this approach is that it leaves the player with the instrument one has acquired a large set of skills for and feels most comfortable with. However, as indicated in the discussion about guitar synthesizers in Sec. 2.2.1 this does not necessarily mean that all of these skills will be mapped into the synthesized sound.

[17] Not to be confused with the computer keyboard

[18] Here, STEIM (Studio voor Electro Instrumentale Muziek, located in Amsterdam) has to be mentioned as their work has been dedicated to this task for more than 25 years.
URL: http://www.dds.nl/~steim/

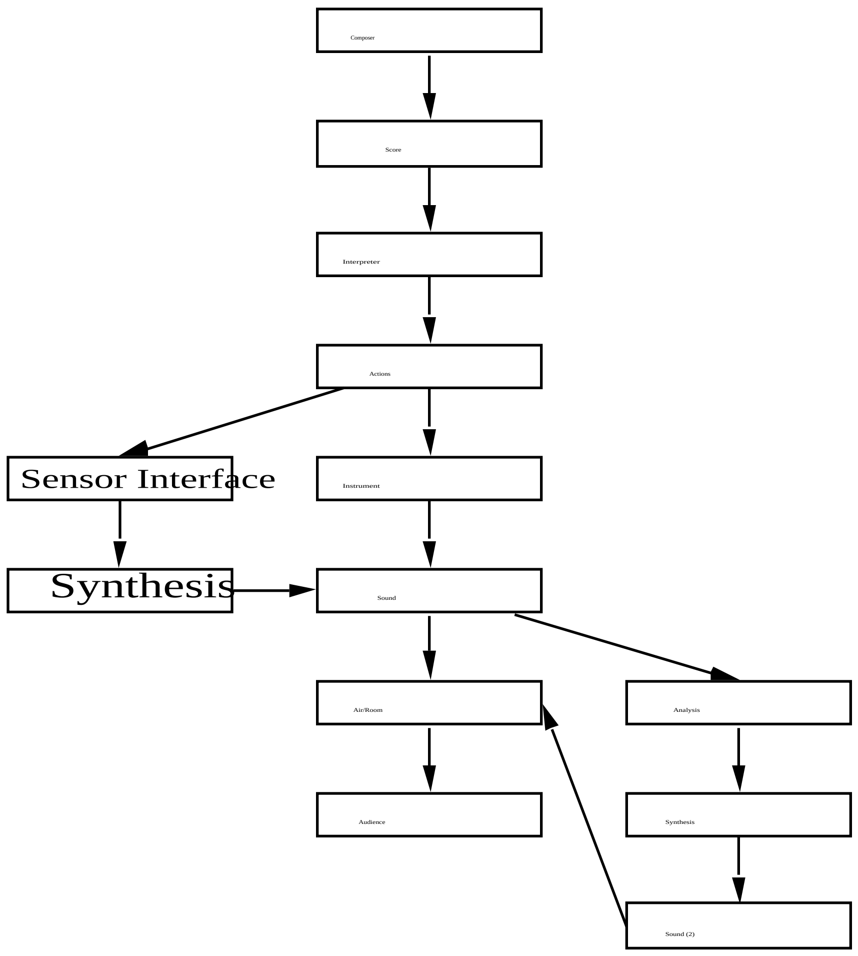


Figure 4.1.2: Sensors (left) vs. real-time sound analysis (right)

4.1.4 Benefits of Abstraction

A main goal of analysis is to extract meaningful information from the signal and thereby make an abstraction allowing for more flexibility and transformations. The following example illustrates this.

A popular technique in music technology is the sampling of sounds, which is much like recording to a tape but has the advantage of random access. However, it only allows for very simple transformations like playing the sound at different speeds (resulting in different pitches), cutting unwanted parts, or playing it backwards. In other words, with this flat[19] model the user only has access to the time domain. One possible abstraction from this is to move to the frequency domain using FFTs followed by the extraction of partials, *i.e.* the sinusoidal components of the sound. Each of these sinusoids can then be treated separately allowing for effects like morphing from one sound to another or changing the length of a sound independently from its pitch.

A side-effect of the sinusoidal transformation is a significant data reduction in that it is not necessary to store or transmit the complete signal itself but only the frequencies and amplitudes of sinusoids, which can be done at a much lower sampling rate.

Apart from these practical benefits, splitting a sound into its components also results in an increased knowledge about its structure. For example, it was shown that a prominent feature of a trumpet sound is that the fundamental frequency has a much slower attack than the first overtone, which in simple pitch detectors could result in octave errors, *i.e.* detecting the right note but the wrong register.

[19] In the context of the thesis 'flat' refers to unstructered data. Information might be present but is not (yet) retrieved.

4.2 The Constraints of Real-time Sound Analysis

4.2.1 Technical Limits

For one, real-time sound analysis is constrained by the capabilities of the system used for the implementation. Apart from the obvious dependency on the overall processing power, which can always be outgrown by complex analysis methods, latency is an important issue. Many operating systems of general purpose computer systems have a minimum block size for audio I/O operations as block transfers are efficient since they avoid excessive context switching. However, this strategy introduces a minimum input-output delay because the input buffer must be filled first; then the data is processed and finally the output buffer is emptied. For most operating systems, this delay is too long to be ignored, which is one reason why DSPs are still favorable in some applications.[20]

4.2.2 Psychoacoustics

Another source of delay is much more difficult to handle because it is inherent to the process. It is obvious that information cannot be retrieved from the audio signal without waiting for some data to arrive. So the main question is if we can acquire enough information before the auditory system detects a delay. This is not easy to answer and depends on the specific parameter. For example, the human ear is very sensitive to the time lag between onsets of separate acoustic events. Pitch perception, on the other hand, is much less critical and takes at least 4 cycles of the fundamental frequency [36], which can be as much as 100ms for the low E of a bass guitar. A general strategy for this scenario could be to trigger an unpitched sound first, and then incorporate a pitched sound after the fundamental frequency is determined.

[20] However, there are systems that have low latency like the SGI and the BeBox.

4.3 Digital Signal Processing

This section provides a review of the necessary background for understanding wavelets. For a more complete introduction the reader is referred to the literature [14, 30, 34].

The following subsections describe the basic concepts of discrete-time signals and LTI systems before moving on to multirate systems and short-time fourier transformation, which are a prerequisite to the treatment of wavelets [42, 44]. Finally, Matlab and its matrix-oriented concept will be discussed.

4.3.1 Discrete-time Signals

Digital signal processing is concerned with the notion of discrete-time signals, *i.e.* signals that represent an original, continuous-time signal at specific, usually equidistant, instants of time, thus discarding all the information in-between. This "sampling" process is governed by the Nyquist theorem, which says that the sampling rate f_s must be at least twice the maximum frequency f_{max} that is to be represented in the digital domain. Furthermore, the signal must be band-limited to f_{max} <u>before</u> it is sampled to avoid aliasing of higher frequency components. Likewise, it has to be band-limited again, after being converted back to the analog domain.

It is often convenient to work with transformed versions of signals such as the z-transform and the Fourier transform. The z-transform of a sequence *x(n)* is defined as

$$X(z) = \sum_{n=-\infty}^{\infty} x(n) z^{-n} \qquad (4.3.1)$$

assuming that the summation converges. It is the discrete-time equivalent of the Laplace-transform, which is used in the continuous-time case to describe systems in a simpler way than in the time domain.

The Fourier transform can be viewed as a harmonic analysis of a time domain signal that describes it as a sum of sinusoids. It is defined as

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x(n) e^{-j\omega n} \qquad (4.3.2)$$

with the inverse transform being

$$x(n) = \frac{1}{2\pi}\int_0^{2\pi} X(e^{j\omega})e^{j\omega n}d\omega \tag{4.3.3}$$

In addition to being discretized in time, signals are usually quantized, *i.e.* made discrete in amplitude, when converted from analog to digital. This introduces a non-linearity to the processing that results in a quantization error or quantization noise, which decreases as the resolution grows. For (musical) audio signals, 16 to 20 bits resolution is usually considered to be sufficient for representing the original signal, and with such a resolution the whole system is assumed to be linear.

4.3.2 LTI Systems

Beyond converting signals from one domain to the other, it is generally of interest to process them in some way, which brings up the notion of discrete-time systems that operate on an input sequence $x(n)$ to produce an output sequence $y(n)$. Two properties are important with respect to these systems: linearity and time invariance.

Linearity means that if the input sequences $x_0(n)$ and $x_1(n)$ result in output sequences $y_0(n)$ and $y_1(n)$, then the response to the input $a_0x_0(n)+a_1x_1(n)$ must be equal to $a_0y_0(n)+a_1y_1(n)$ for all a_0 and a_1 and every possible $x_0(n)$ and $x_1(n)$. Time invariance means that if the input sequence $x(n)$ results in the output sequence $y(n)$, then the response to the shifted version $x(n - N)$ is $y(n-N)$ for all integers N and all input sequences $x(n)$.

A system with both properties is called an LTI system, abbreviating the two expressions. It can be completely characterized by its impulse response sequence $h(n)$ which is the output in response to the unit-pulse input

$$\delta(n) = \begin{cases} 1, & n = 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.3.4}$$

The input-output relation is given by

$$y(n) = \sum_{m=-\infty}^{\infty} h(m)x(n-m) \tag{4.3.5}$$

which is called the convolution summation. Applying the z-transform, this can be expressed as

$$Y(z) = H(z)X(z) \tag{4.3.6}$$

where *H(z)* is called the transfer function of the LTI system. In this thesis all transfer functions are rational, *i.e.* of the form

$$H(z) = \frac{A(z)}{B(z)} \tag{4.3.7}$$

with

$$A(z) = \sum_{n=0}^{N} a_n z^{-n}\ ,\ B(z) = \sum_{n=0}^{N} b_n z^{-n} \tag{4.3.8}$$

If the fraction in (4.3.7) is irreducible, the zeros of *A(z)* and *B(z)* are said to be the zeros and poles, respectively, of *H(z)*. *N* is called the order of the system assuming that at least one of a_n or b_n is non-zero.

In the special case where b_n in (4.3.8) is non-zero for only one value of *n*, the system is labeled FIR, *i.e.* finite impulse response; otherwise *H(z)* corresponds to an infinite impulse response (IIR) system. Both terms are often used in connection with digital filters, the most common LTI systems. Designing these filters to meet certain specifications is a whole art in itself, and will not be addressed here. There are a variety of programs that facilitate this task; for this thesis the signal processing toolbox of Matlab was used.

4.3.3 Multirate Systems

Multirate systems are important in this context as they are one of the foundations of wavelets. Furthermore, two of their earliest applications were in professional digital music systems, where oversampling, a related technique, was used to improve sound quality, and sampling rate conversion was needed to convert digital audio from *e.g.* CD players (44.1kHz) to DAT recorders (usually operated at 48kHz). In other applications the use of multiple sampling rates brings advantages such as reduced computational complexity, transmission rate, and storage requirements.

The most basic operations in multirate digital signal processing are decimation and interpolation. In order to describe these, two new building blocks are introduced, the decimator and the expander.

The M-fold decimator takes an input sequence *x(n)* and produces the output sequence

$$y_D(n) = x(Mn) \tag{4.3.9}$$

where *M* is an integer. The decimation retains only those samples of *x(n)* which occur at times equal to multiples of *M*. The process is often referred to as downsampling.

In turn, the output sequence of the L-fold expander is defined as

$$y_E(n) = \begin{cases} x(n/L), & \text{if } n \text{ is an integer multiple of } L \\ 0, & \text{otherwise} \end{cases} \tag{4.3.10}$$

where *L* is an integer. Thus, the expander inserts *L-1* zeros between two adjacent samples of *x(n)*. This process is also known as upsampling.

In the frequency domain up- and downsampling corresponds to a scaling of the spectrum as is shown next for the case of the expander. From (4.3.1) we have

$$Y_E(z) = \sum_{n=-\infty}^{\infty} y_E(n) z^{-n} \tag{4.3.11a}$$

As $y_E(n)$ equals zero if *n* is not a multiple of *L*, this can be written as

$$Y_E(z) = \sum_{k=-\infty}^{\infty} y_E(kL) z^{-kL} \tag{4.3.11b}$$

and with (4.3.9) we arrive at

$$\begin{aligned} Y_E(z) &= \sum_{k=-\infty}^{\infty} x(k) z^{-kL} \\ &= X(z^L). \end{aligned} \tag{4.3.11c}$$

Likewise $Y_E(e^{j\omega}) = X(e^{j\omega L})$, which means that $Y_E(e^{j\omega})$ is an *L*-fold compressed version of $X(e^{j\omega})$ as demonstrated in Figs. 4.3.1a&b.

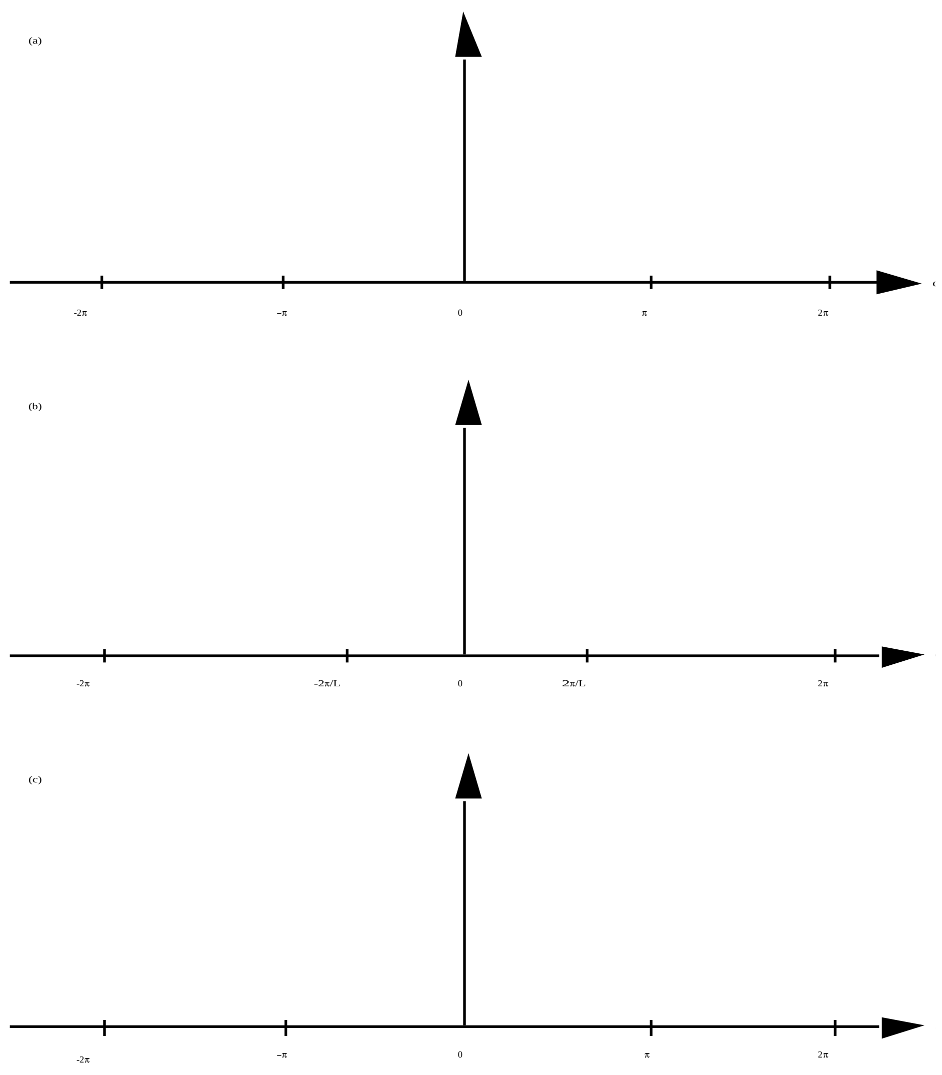


Figure 4.3.1: Effect of expander and decimator in the frequency domain. Fourier transforms of (a) the original signal, (b) the expanded signal (*L = 3*) and (c) the decimated signal (*M = 2*)

Thus, expanding in time results in compression in frequency.

Fig. 4.3.1c shows that for decimation the effect is somewhat reversed and the spectrum is expanded, which in this case creates overlapping images. This is known as aliasing and should be met with proper band-limiting (as mentioned in Sec. 4.3.1 for the analog domain).

4.3.4 From STFTs ...

The short-time Fourier transform (STFT) differs from the standard Fourier transform (4.3.2) in that it does not look at the signal for an infinite amount of time, which is impractical in real-time analysis, to say the least. Instead, it observes windowed portions of the sound and subsequently shifts the window ahead as new data becomes available. It is defined as

$$X_{STFT}(e^{j\omega},m)=\sum_{n=-\infty}^{\infty}x(n)w(n-m)e^{-j\omega n} \qquad (4.3.12a)$$

where *w(n)* is a window function which is only non-zero for a finite number *k* of *n*. Thus the summation reduces to

$$X_{STFT}(e^{j\omega},m)=\sum_{n=m-k+1}^{m}x(n)w(n-m)e^{-j\omega n}.[21] \qquad (4.3.12b)$$

This procedure involves a tradeoff between time and frequency resolution. For time resolution, it is quite obvious that by making the window smaller we gain more precise information about how the analysis results are associated to a specific instant in time. However, as the window size *k* becomes smaller, frequency domain resolution decreases because X_{STFT} is only non-zero for *k* values of *n* from 0 Hz to f_s and for frequencies beyond this range the spectrum is periodic, *i.e.* the values are repeated, and do not provide any additional information. In turn, for larger *k* the grid on the frequency axis becomes denser, so it can be determined more precisely what frequencies are contained in the signal. But in doing this, timing information is lost and if the signal's behavior changes within the frame of the window it is not clear exactly when the changes occurred.

Wavelets are a possible solution for this dilemma, but before explaining their theory, two more steps concerning the STFT are necessary. First, it is helpful to interpret the STFT as a bank of bandpass filters with identical bandwidths and center frequencies distributed on a uniform grid. Multiplying (4.3.12a) with $e^{j\omega m}$ inside the summation and with $e^{-j\omega m}$ outside we arrive at

[21] It is more standard to reference to the start of the window, i.e. doing the summation as $\sum_{n=m}^{m+k-1}$ ...

In the real-time context of this application, however, it seemed more adequate to let *m* mark the moment where a computation is initiated and use the samples preceeding this moment.

$$X_{STFT}(e^{j\omega},m) = e^{-j\omega m}\sum_{n=-\infty}^{\infty} x(n)w(n-m)e^{j\omega(m-n)} \tag{4.3.13}$$

This can be viewed as a convolution of *x(n)* with the sequence $w(-n)e^{j\omega n}$, followed by a modulation with the sequence $e^{-j\omega n}$. Accordingly, for a particular frequency ω_0 and time m, $X_{STFT}(e^{j\omega_0},m)$ can be obtained as the modulated output at time *m* of a filter with impulse response $w(-n)e^{j\omega_0 n}$.

As a second step towards wavelets it can be argued (like in [42, 44]) that wavelets are more easily introduced for the continuous-time case and therefore the continuous version of the STFT is given next as

$$X_{CTSTFT}(j\Omega,\tau) = \int_{-\infty}^{\infty} x(t)w(t-\tau)e^{-j\Omega t}dt \tag{4.3.14}$$

for which the filter bank interpretation is

$$X_{CTSTFT}(j\Omega_k,\tau) = e^{-j\Omega_k\tau}\int_{-\infty}^{\infty} x(t)h_k(\tau-t)dt \tag{4.3.15}$$

with

$$h_k(t) = w(-t)e^{j\Omega_k t} \tag{4.3.16}$$

4.3.5 ... to Wavelets

Wavelets are a relatively new topic in signal processing, yet their mathematic background is too sophisticated to be covered here even remotely. Thus the discussion will concentrate on their ability to trade time and frequency resolution in a more flexible way than the STFT described in the previous section.

In the STFT the window function *w(n)* has a fixed length of non-zero values. This means that for a high frequency signal, many of its cycles are captured while for a signal with a low frequency only a few cycles are within the window. In other words there is much better resolution for high frequencies than for low ones. This relates to the fact that the bandpass filters of (4.3.16) have identical bandwidths rather than bandwidth increasing with center-frequency. Another way of looking at this resolution issue is that it is desirable to achieve good frequency resolution for steady state signals while still capturing transient events precisely.

One way to solve this problem is to look at the signal at different scales at the same time; this is exactly what can be accomplished with the wavelet transform (WT). The aforementioned STFT bandpass filters have equal bandwidths because they are derived from a single filter by modulation. With wavelets this filter changes as a function of both time and frequency, or according to the standard terminology, time and scale. Their impulse responses are defined as

$$h_k(t) = a^{-k/2} h(a^k t) \tag{4.3.17}$$

where *h(t)* is the filter prototype, *k* the (integer) scale parameter and *a* the scale factor with *a>1.* a^{-k} is also called the dilation factor. The corresponding transfer functions are as

$$H_k(j\Omega) = a^{k/2} H(ja^k \Omega) \tag{4.3.18}$$

Given the convolution integral

$$y(\tau) = \int_{-\infty}^{\infty} x(t) h_k(\tau - t) dt$$

this yields

$$y(\tau) = a^{-k/2} \int_{-\infty}^{\infty} x(t) h(a^{-m}(\tau - t)) dt \tag{4.3.19}$$

Since the bandwidth of $H_m(j\Omega)$ becomes smaller for large *m*, we can sample its output at a correspondingly smaller rate, as implied in the section on multirate systems. Thus, if the filter prototype $H(j\Omega)$ is sampled at intervals *T* we consequently sample $H_m(j\Omega)$ with period $a^m T$, which leads to the discrete wavelet transform (DWT) [22]

$$X_{DWT}(k,m) = a^{-k/2} \int_{-\infty}^{\infty} x(t) h(mT - a^{-k} t) dt \tag{4.3.20}$$

[22] Here 'discrete' relates to the scale parameter; time is continuous. There is also a generalized wavelet transform where both parameters are continuous, which is of less interest here.

Within the framework of filter banks this can be regarded as splitting the original into a set of filtered and downsampled signals that represent it at different scales. It is possible to retrieve the original signal from these signals by an inverse wavelet transform, which is omitted here for simplicity.

In the discrete time (DTWT) case these (sub-) signals are defined as

$$y_k(m) = \sum_{l=-\infty}^{\infty} x(l) h_k(2^{k+1} m - l), \quad k = 0, 1, \ldots, K-2 \tag{4.3.21a}$$

and

$$y_{K-1}(m) = \sum_{l=-\infty}^{\infty} x(l) h_{K-1}(2^{K-1} m - l) \tag{4.3.21b}$$

To illustrate the difference between the STFT and the DWT/DTWT two figures are presented. Fig. 4.3.2 compares the frequency responses of the bandpass filter interpretation, while Fig. 4.3.3 shows the time-frequency grids. In Fig. 4.3.3b it can be seen that parameters are computed more often at high frequencies than for low frequencies. The logarithmic frequency spacing shown in Figs 4.3.2b and 4.3.3b suits acoustic applications since the human auditory system exhibits a similar response characteristic.

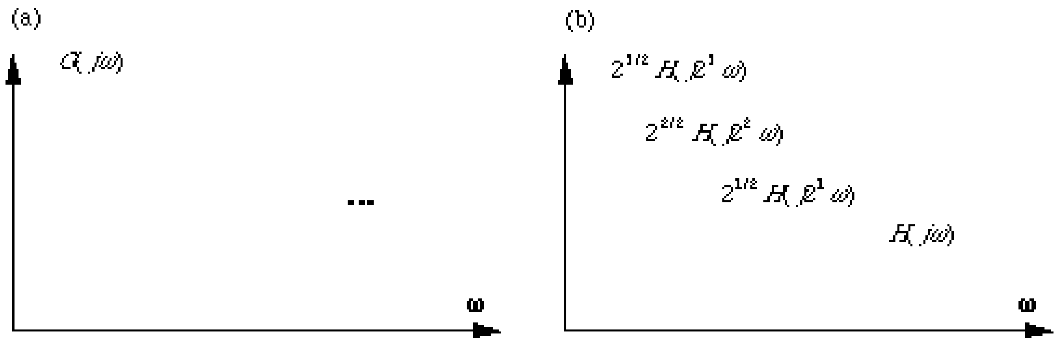


Figure 4.3.2: Bandpass filter representation of (a) STFT and (b) DWT. In (a) a single filter response is modulated, *i.e.* frequency-shifted to equispaced center frequencies. In (b) the filter bandwidths and amplitudes are changed for logarithmically increasing center frequencies (*a* = 2).

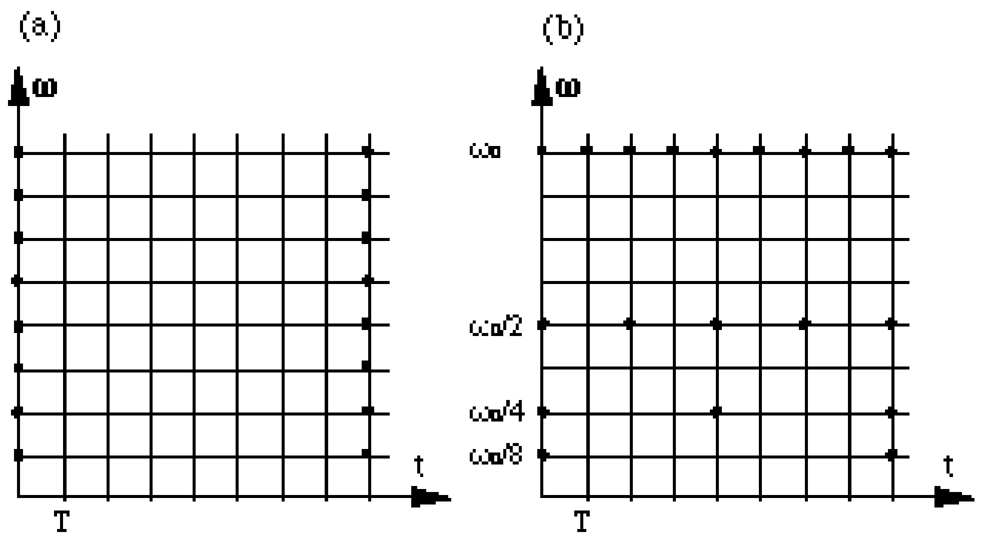


Figure 4.3.3: Time-Frequency grid for (a) STFT and (b) DWT. In (a) a parameter set is computed with a dense linear frequency resolution at equispaced times and a slow rate. In (b) parameters are computed with a logarithmic frequency resolution and rates depending on frequency.

4.3.6 Thinking in Matlab

Matlab is highly matrix-oriented, which makes it ideal for linear algebra. With respect to signal processing, this has a strong impact on the way the algorithms have to be formulated. Especially when an operation must be performed for each element of a large data set such as discrete-time musical sounds, using loops is not a good idea as they slow Matlab down. Consider the following example.

A decimation by an integer factor of `d` is usually performed by taking every `dth` sample while ignoring the rest of them. Implemented with a `for` instruction this would look like this:

```
for n = 1:length(sound)/d
      deci_sound(n) = sound((n-1)*d+1);
end
```

For large soundfiles this takes a considerable amount of time. By using the Matlab matrix manipulation constructs, the same task can be done with the single assignment

```
deci_sound = sound(1:d:length(sound));
```

The execution time is often improved by at least one order of magnitude, confirming that alternatives to loops are worth investigating.

4.4 Models of Sound

4.4.1 The Main Models

The variety of algorithms to create sound is immense and thus only the most common ones will be mentioned here in order to give some background for the discussion of how analysis and synthesis algorithms might be linked.

The oldest technology is often called Subtractive Synthesis because it uses oscillators with simple waveform and changes their color by "subtracting", *i.e.* attenuating, certain parts of the spectrum with filters. It is closely linked to the paradigm of voltage control of parameters in the analog domain, and thus uses a variety of control modules, like *e.g.* envelope generators to shape the filter response over time.

Its counterpart in some sense is the Additive Synthesis already mentioned in previous sections, that creates sound by adding a large number of simple sinusoids to form a complex sound. While much more flexible and powerful, its main problems are the processing power (resp. analog component count) required to handle the numerous components, and the difficulty of creating an interface to control the many parameters.

The prominent synthesis today is the Sample Player that plays back digital recordings of natural sounds. It is often combined with Subtractive Synthesis, replacing the latter's simple oscillators with complex waveforms. The terminology is sometimes confusing in this respect; other names for the same basic technology are Wavetable Synthesis or Linear Arithmetic Synthesis. While sampling is able to reproduce particular sounds of *e.g.* a violin almost perfectly, it cannot compete with the degree of nuance that is achieved with an acoustic instrument. Even compared to other synthesis approaches it falls short due to the static nature of a sampled sound. The problem addressed here is that of flexibility and limited control, especially concerning the temporal evolution of a sound.

In the last few years considerable progress has been made in the quest for synthesized sounds with more natural quality. One very promising approach is Physical Modeling, which differs from the others because it does not model the sound but rather the way it is produced. Thus a (complex) description of the physical structure of an instrument and its interaction with the performer is the main foundation for the algorithms.

Consequently the sounds produced by these models are fairly close approximations of acoustic instrument sounds. Current research is working on abstractions of these models to achieve a greater variety and more original sounds [5].

4.4.2 Analysis / Synthesis Links

The discussion in the previous section resembles the one about sensors and real-time sound analysis in Sec 4.1.3; in both cases a distinction is made between working on the level of sound and the level of physical action. This brings us to the question of which types of analysis can be linked with which kind of synthesis.

The analysis that seems to fit best to Physical Modeling is a sensor-based one, although it might be possible to derive physical parameters from the sound. However, not much research has been undertaken in this direction yet.

A very close link can be found for Additive Synthesis, where the parameters for all the single sinusoids generally come from a spectral analysis of natural sounds. This analysis is usually based on Fourier transformations followed by peak picking and tracking to determine the partials [7].

The model behind Subtractive Synthesis is that of an excitation followed by a resonance which is often used to describe voice as well as many classical instruments. For voice, the glottal tract creates an oscillation which is filtered by various parts of the throat, mouth and nose. For a violin, the excitation is represented by the bow-string interaction while its body creates the resonance. As for analysis, the impulse response of the filter can be measured by a technique called linear predictive coding (LPC).

Finally for the Sample Player there is no corresponding analysis to be found as sampling is simply the recording of a sound where no abstraction is made. In other words, sampling is a completely flat model.

4.4.3 Mapping on Different Abstraction Levels

In the introduction it was mentioned that one focus of the research should be the mapping of analysis and synthesis algorithms on different levels of abstraction. In this consideration the lowest level is the signal itself, a flat representation of the sound event. Through analysis information is retrieved from the signal to yield a gradual abstraction.

For the example of Additive Synthesis, the first step of analysis is the determination of the partials, as mentioned in the previous section. In contrast to the single stream of samples of the original signal they are represented by a set of frequencies and amplitudes varying over time at a much lower sampling rate. In synthesis this can be used directly to re-synthesize the sound, possibly after some transformation like time-stretching or morphing. These partials carry much more information than the original signal, but even more meaningful parameters can be extracted. For example, by examining the distribution and levels of the partials parameters like 'brightness', 'nasality' or 'spectral flux' can be obtained. These describe the sound more intuitively. Even higher levels of abstraction might consider the formal structure of groups of events and arrive at parameters like 'density' or 'variance', and might even describe a whole piece as 'sad' or 'in the style of Stockhausen'.

Mapping the parameters from one analysis-synthesis approach to another is not problematic for a high level of abstraction. For instance, a measure for brightness obtained under the described paradigm of Additive Synthesis is easily mapped to Subtractive Synthesis, where a different mechanism is employed to translate this to the lower level description. In this case the brightness measure would not be used to weigh the single partials but to change the cutoff frequency of a filter. Besides, it is left to the composer to make completely different mappings, e.g. 'pitch bend' to 'inharmonicity'.

On a low level of abstraction this mapping is not that easy, if possible at all. In the given example, there is hardly a way to use the set of partials directly in a subtractive synthesis scheme. Thus, some degree of similarity is needed for the analysis and synthesis models to allow for such a mapping.

4.5 The Transient Guitar Signal Database

4.5.1 The Selection of Relevant Sounds

Although the main focus of this work is on fast re-plucks, it was necessary to compile a more complete set of different guitar sounds in order to evaluate and compare the results of the analysis when applied to other playing styles.

The selection is focused on transient sound events as opposed to steady state sounds, since most of the articulation of a guitar player lies in the attack of a note and in the subtle variations of left hand fingering and right hand plucking, where timing plays an important role. After a note has been plucked, the only possible manual control is vibrato, which was of no interest here.

The following sound prototypes were recorded:

a. Attack :	a new note is plucked with the right hand
b. Re-pluck :	the string is still vibrating, when a new note is plucked with the same frequency on the same string
c. Hammer-on :	a new note is triggered by a hard fingering (hammering) of a left hand finger
d. Pull-off :	a new note is triggered by plucking the string with a left hand finger while removing it from its former position
e. Damping :	the string's vibration is stopped either by the left or right hand
f. Glissando :	a left hand finger is moved along a string across the fingerboard without losing contact to it, thereby changing the pitch of the note
g. "Noise" :	usually unwanted sounds, like string squeaking when sliding a left hand finger along a string, to test against false triggering

All of these sounds were recorded with three different notes, *i.e.* frequencies, a very low one, a very high one and one in the middle:

note	fret	string	frequency [Hz]
F	1.	E	87.3
h	9.	d	246.9
$g^{\#}$''	16.	e'	830.6

Another dimension was added by recording some of the sounds at different degrees of loudness, not by simply lowering the input gain of the recorder but by actually playing the notes with differing strengths.

More variety was derived by playing the notes in many different manners, especially for the attacks and re-plucks. This included up- vs. down-strokes, different distances between pluck position and bridge[23], right vs. left hand damping and more advanced playing styles, like muted notes (the palm of the right hand partially damps the strings close to the bridge before plucking a note, which creates a shorter and less bright sound) and harmonics (the left hand slightly touches the string at a position where it is divided into an integer number of equal length segments, then the string is plucked).

[23] The bridge is the termination of the string on the body of the guitar, usually on the right hand side.

4.5.2 The Recording

For recording all these guitar sounds there were two options. The Silicon Graphics Indigo had build-in analog-to-digital converters facilitating direct recording from a microphone or line input. The other option was recording to a DAT recorder and transferring the sounds to the SGI via the build-in tape drive. Although the latter option was more work it was chosen because of the superior quality of the DAT due to its better analog-to-digital converters.

The sounds were recorded as flat as possible, *i.e.* directly from the pickups of the guitar without any equalization, using only a high quality preamplifier connected to the DAT recorder with balanced lines. The sampling frequency was set to 32kHz, which limited the supported bandwidth to about 15kHz; this sampling rate was chosen to reduce the memory requirements and the processor load.

4.5.3 Transfer to Matlab

After recording the sounds to DAT, they had to be transferred into a format that could be read by Matlab. This was accomplished in basically three steps. The first one was a change of media, *i.e.* from DAT to harddisk. As the sounds were stored on the tape without an SGI-compatible file system, of course, they had to be recorded with the application 'soundeditor' by starting the tape, listening to it and enabling the transfer process manually whenever a sound in question appeared. The whole tape could have been copied into one large soundfile and then chopped up into the desired parts, but memory space limitations prevented this approach.

The next step was to truncate the (still quite long) sound samples, leaving the essential parts of the signal for the analysis, *e.g.* the first few hundred milliseconds of an attack or a series of fast re-plucks. For some sounds the length was further trimmed to a power of two to comply with FFT-based algorithms. The 'soundeditor' proved useful here, as well as another application called 'MXV' which had integrated analysis features.

Yet another procedure was necessary after the truncation, since sounds could only be recorded into the 16bit linear AIFF[24] format while Matlab was only capable of loading µlaw-encoded sound files without a header. The format conversion program 'soundfiler' was used for this step.

[24] AIFF = Audio Interchange File Format

4.6 A First Look at the Sounds

4.6.1 The Time Domain

Two simple time domain plots of database examples introduce some of the basic problems. Figs. 4.6.1 and 4.6.2 both show re-plucks, of notes g$^{\#}$" and F played at quite a fast rate of about 10Hz.

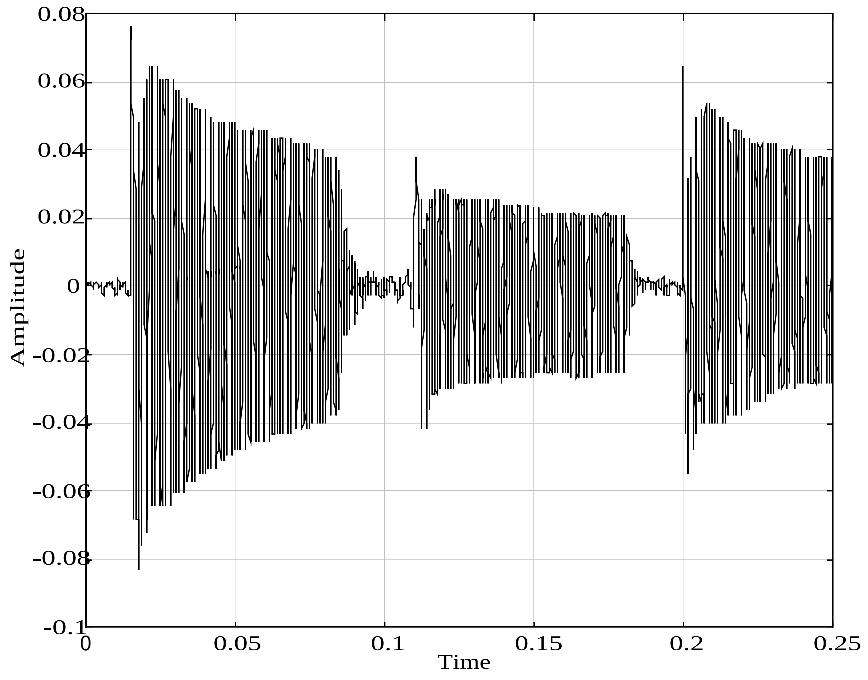
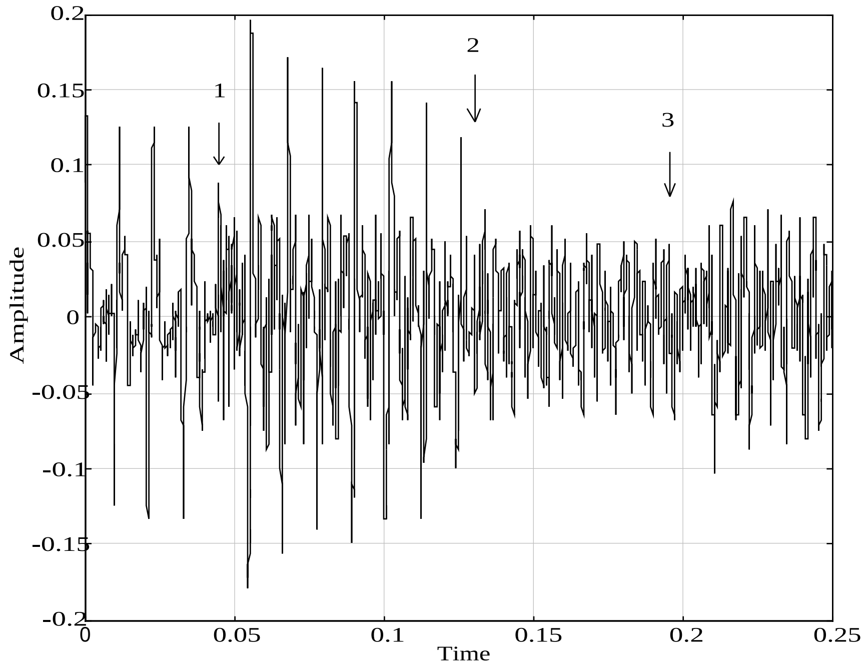


Figure 4.6.1: 3 re-plucks on g#" (f_S = 32kHz; re-pluck rate = 10Hz)



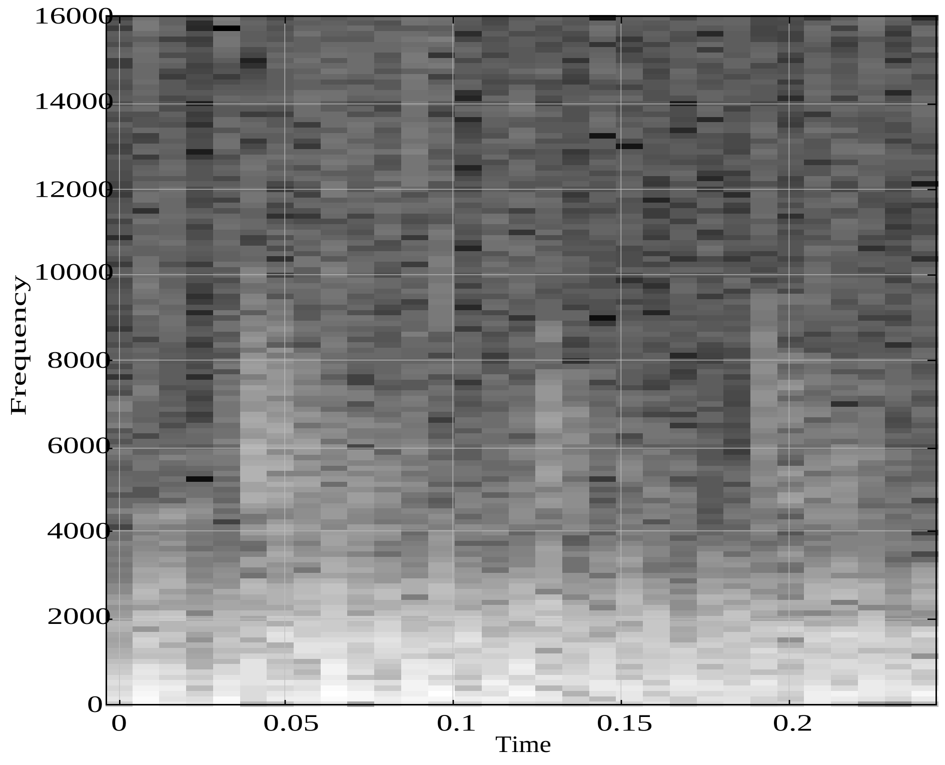
.

Figure 4.6.2: 3 re-plucks on F (f_S = 32kHz; re-pluck rate = 13Hz)

In Fig. 4.6.1 it can be clearly seen that each new onset is preceded by the decay of the previous note; detection is easily done by examining the signal's amplitude or energy. While this approach works well with high frequencies, it deteriorates when applied to re-plucks played on the lower strings. Due to the higher mass of these strings, it takes much longer for low notes to decay, *i.e.* lose their energy, as can be observed in Fig. 4.6.2. Here re-plucks can be heard at 0.045s, 0.13s and 0.195s as indicated by the arrows. While at positions 1 and 3 there is at least a significant rise in amplitude (though no preceding decay), arrow 2 marks an instance where a re-pluck actually causes a drop in energy. Yet, when listening to this sample the event can be clearly heard in sound example B3.1.

4.6.2 The Frequency Domain

Looking at Fig. 4.6.3, the frequency domain representation of the signal in Fig. 4.6.2, it can be seen that the re-plucks are marked by a rise in high frequency components, with the best contrast between 4kHz and 7kHz. This is true even for the re-pluck at position 2, though less apparent when compared to the other two instances. At low volumes, however, this feature becomes less defined and cannot be used for making trigger decisions. This stems from the fact that the real difference between re-pluck sounds and those of vibrating strings is not their spectral envelope but the ratio of harmonic to inharmonic components. It emphasizes the need for a more elaborate scheme than Fourier transformations.



.

Figure 4.6.3: Spectrogram of the re-pluck shown in Fig. 4.6.2 (Block size = 256; no overlap; light regions correspond to high energy)

5 Designing the System

5.1 Exploring Wavelets

5.1.1 Prototyping Wavelets in Matlab

It has been shown in the literature [6] that the basic DTWT algorithm defined by (4.3.21a/b) can be easily implemented by a tree like structure that uses only two filters and subsequent downsampling. This is shown in Fig. 5.1.1.

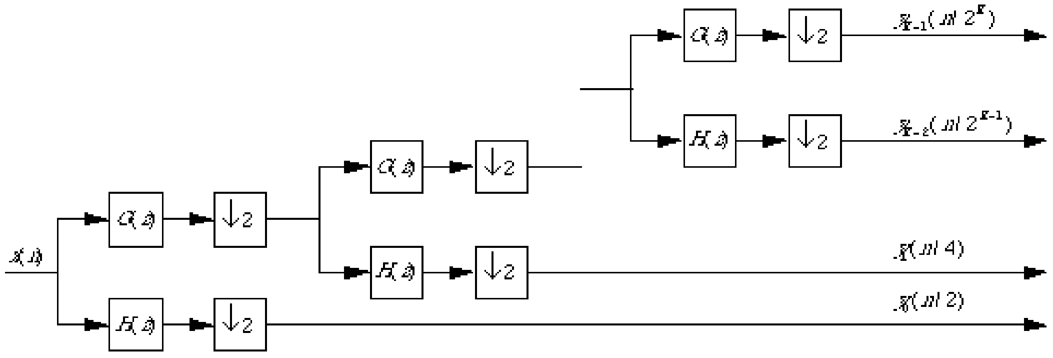


Figure 5.1.1: Tree-structured analysis filter bank

In the lowest branch *H(z)* is a highpass filter that delivers the upper half of the frequency spectrum. Because this reduces the bandwidth by a factor of two, the output can be downsampled accordingly without losing information. Likewise *G(z)* is the equivalent lowpass filter providing the spectral counterpart. However, this signal is not part of the final wavelet coefficients. Rather it is filtered and downsampled again using the same transfer functions *H(z)* and *G(z)* which gives us the first and the second quarter of the spectrum at a quarter of the sampling rate. This process can be repeated until only one sample is left per filter output. It essentially implements the bandpass filters depicted in Fig. 4.3.2b. Note that this results in a dataset that has the same size as the input sequence.

It should be noted that other schemes are possible for splitting the branches of the tree, depending on which parts of the spectrum are of particular interest. This approach is known as the wavelet packet transform; there are algorithms to find the best tree structure for a specific data set. This "best basis search" is of importance in data compression applications, a major field for wavelet usage. In such cases, the inverse transformation is also of great interest since it is important to reconstruct the original signal as accurately as possible; issues like energy conservation and orthogonality are well-known problems in this endeavor. If the filterbank tree is split at every branch, the resultant decomposition of the signal roughly corresponds to an N-point STFT as it entails N "leaves" covering the complete frequency range and downsampled to one Nth of the sampling rate [20].

The basic DTWT can be implemented with Matlab's `filter` command, a down-sampling operation, and a simple iteration. The core of the program is given here; the complete code can be found in A1.1

```
1   for n = levels-1:-1:L
2      tmp = filter(H,1,buffer);
3      out(2^n+1:2^(n+1)) = tmp(1:2:length(tmp)-1);
4      tmp = filter(G,1,buffer);
5      buffer = tmp(1:2:length(tmp)-1);
6   end
7   out(1:2^L) = buffer;
```

In lines `2` and `4`, the signal buffer is filtered by the highpass and lowpass filters respectively, while in lines `3` and `5` the downsampling is performed. The result of the highpass filtering is saved to the output data structure while the lowpass filtered signal becomes the new input signal for the next filter stage (line `5`). The variable `L` in line `1` specifies the coarsest level of the decomposition .

A major question here is the specification of the wavelet filters; this is widely discussed in the literature [6]. This project relied on established filters, like those developed by Daubechies and Morlet. The coefficients were obtained from public domain wavelet resources, which also served for verifying my implementation.[25]

5.1.2 Running the Wavelets on the Database

Applying the described algorithm to the re-pluck of Fig. 4.6.1 with 12 point Daubechies filters yields the representation shown in Fig. 5.1.2, where all the $y_k(n)$ of Fig. 5.1.1 are concatenated as indicated in the figure.

[25] WaveLab : http://playfair.stanford.edu/~wavelab/

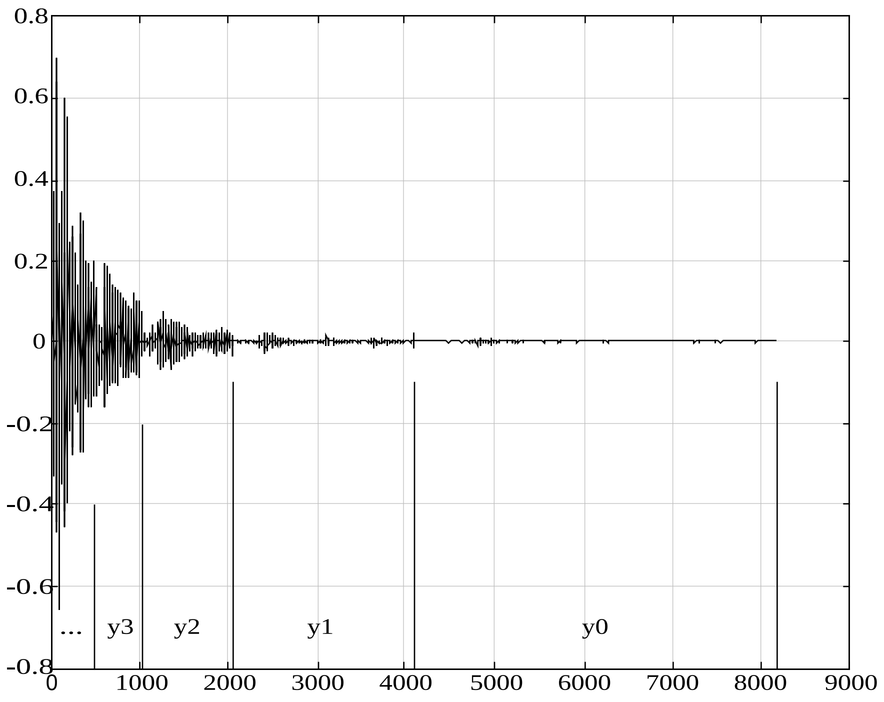


Figure 5.1.2: Flat representation of DTWT

This shows that the discrete-time wavelet transform (DTWT), like the STFT, yields a dataset the same size as the input vector. A better visualization, however, is achieved by stretching the $y_k(n)$ and aligning them in time so that the transients become more visible. This is shown in Fig. 5.1.3 for $k = 0$ *to* 7, where it can be seen that the temporal resolution degrades with the scale level, as mentioned in the previous sections. The corresponding code is given in A1.2.

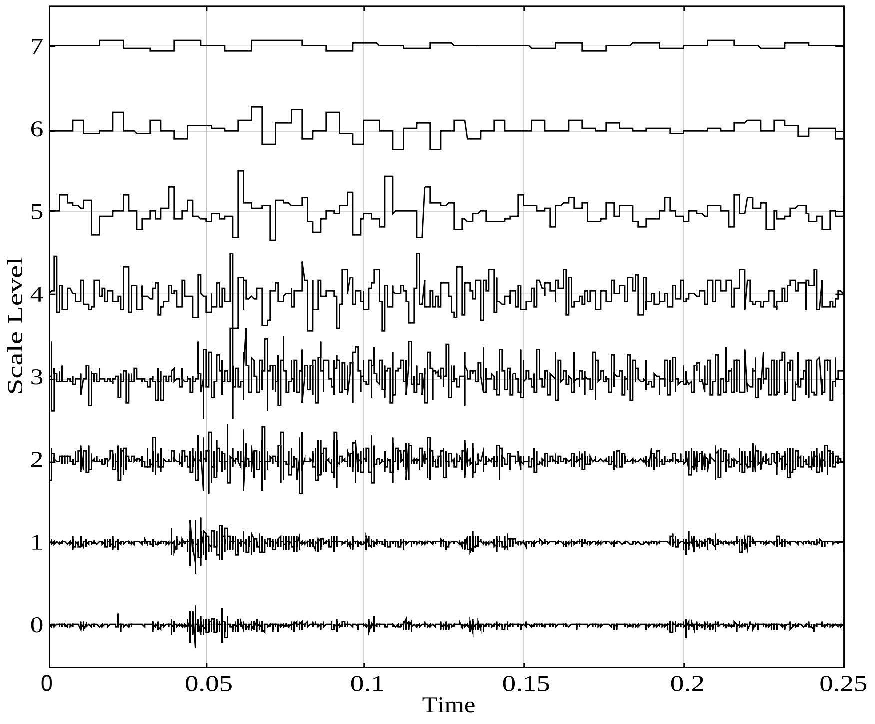


Figure 5.1.3: Multiscale representation of DTWT

The first re-pluck at 0.045s exhibits a distinct rise in amplitude across all scales, where due to the growing length of the filter response the onset is more and more delayed for higher scale levels. For example, in level 5 the peak occurs at 0.06s introducing a 15ms delay. For the other two instances the event is less apparent at all scales. Here, the best contrast between periodic phases and transients is achieved on scale level 1, *i.e.* $y_1(k)$. This is further clarified by extracting this signal as in Fig. 5.1.4.

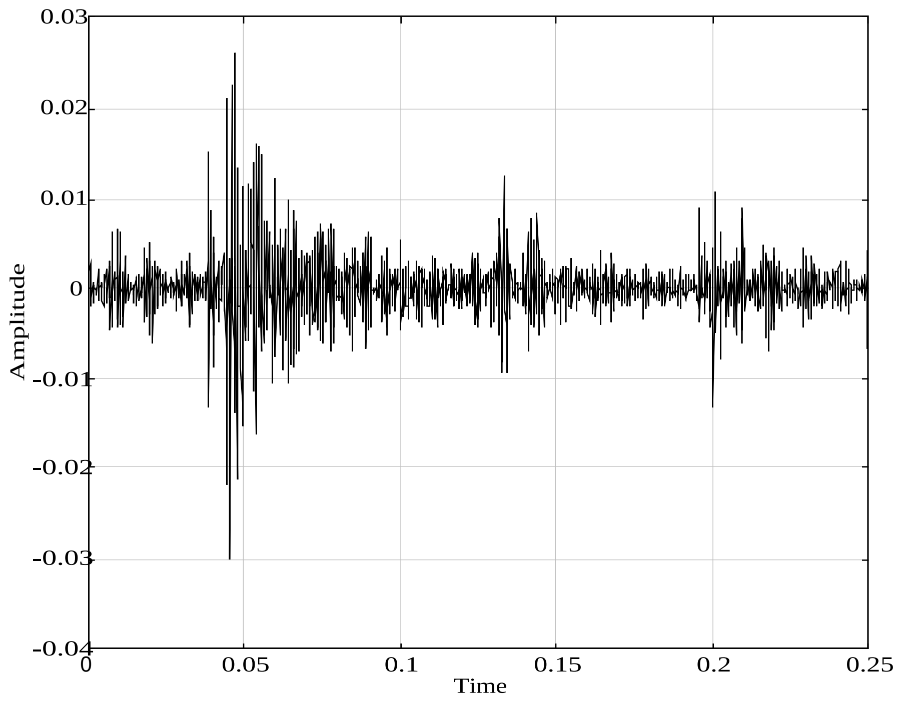


Figure 5.1.4: y_1 of the DTWT

The peaks correspond to the rise in high frequencies observed in the STFT in Fig. 4.6.3 and explained in Sec. 4.6.2. The main advantage of the DTWT is that the re-plucks can be detected much faster because the delay does not depend on a fixed block size but only on the filter impulse response which in this case is rather short.

Like the STFT, however, the DTWT approach suffered from the same problem of not being able to detect softly played notes as can be observed in Fig. 5.1.5, where the processed signal shows no improvement compared to the original signal.

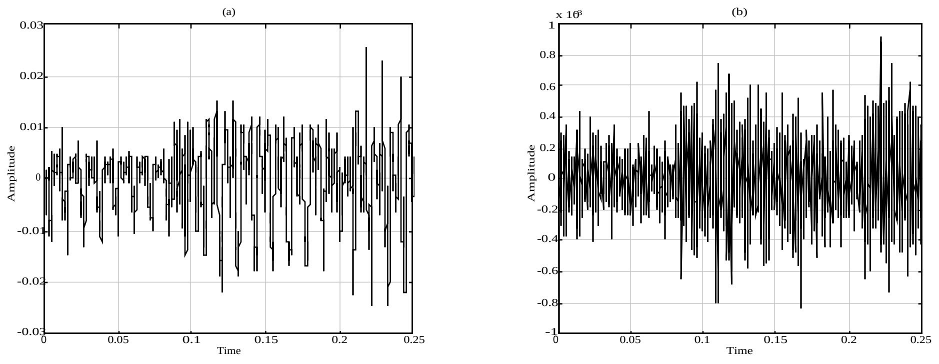


Figure 5.1.5: DTWT results for softly played re-plucks:
(a) original (b) scale level 1 of the DTWT

After all, using just y_1 of the DTWT simply corresponds to using a filter with transfer function $G(z)H(z^2)$ to extract frequencies between 4kHz and 8kHz. This can also be accomplished with an STFT, but with a fixed block size. This shortcoming prompted a consideration of more advanced wavelet designs.

In [19] and [20] wavelet packet transforms are used for transient signal classification. A complete tree decomposition is done as mentioned in 5.1.1. This is followed by the computation of the energy of each "leaf", which yields a specific spectral energy signature for each sound with additional information compared to the STFT or basic DTWT.

In [8] it is shown that a unit-pulse embedded in an aperiodic signal can be detected by computing the wavelet transform on a very dense scale grid. This cannot be done with the efficient scheme of the basic DTWT described above, and thus takes a considerable amount of time to compute.

Because of the computational complexity of these approaches they are not useful in the context of this thesis because they do not address real-time issues. Other algorithms require large data sets to deliver precise results, so signals are mostly processed off-line or with delay times too long to be neglected.

However, the biggest problem is, that the majority of the wavelet transformations simply provide alternative time-frequency descriptions, and do not consider the special characteristics of transient events embedded in a mostly periodic signal. As indicated in Sec. 4.6.2 this combination of sounds, but also musical sounds in general, can be described by a model of deterministic plus stochastic components as in [39] and [38], where 'deterministic' relates to harmonic or periodic sounds, and noise-like events are represented by stochastic models.

The remaining problem of how to tell harmonic from inharmonic sounds is addressed by the pitch-synchronous wavelet transform proposed by Gianpaolo Evangelista. This is discussed in the next section.

5.1.3 The Pitch-Synchronous Approach

Pitch-synchronous schemes have often been used to describe speech and musical sounds [17, 22, 25]. These differ from standard block-based approaches in that they do not analyze a fixed amount of samples but a varying number that depends on the pitch of the signal. This idea was combined with the theory of wavelets by Gianpaolo Evangelista in [9] and [10].

The basic idea is to oversample the wavelet filter prototypes by a factor representing the pitch of the relevant signal, which creates spectral copies of the response as depicted in Fig. 4.3.1a&b. Starting from a lowpass filter, the oversampling yields spectral peaks that are centered around the fundamental frequency and its integer multiples, *i.e.* the harmonics of the signal. Likewise, oversampling a highpass filter produces peaks between the harmonic partials, thus amplifying the inharmonic components.

Iterating the filtering process at the lowpass outputs of the filterbank tree as described in Sec. 5.1.1 has a new meaning in this context. Rather than increasing the frequency resolution for low frequencies with each stage, the peaks of filter frequency responses move closer to the harmonic partials, which are finally covered by the last oversampled lowpass filter. This is further clarified in Fig. 5.1.6, where the filter prototypes are compared to their oversampled versions, which due to their shape are often referred to as comb filters.

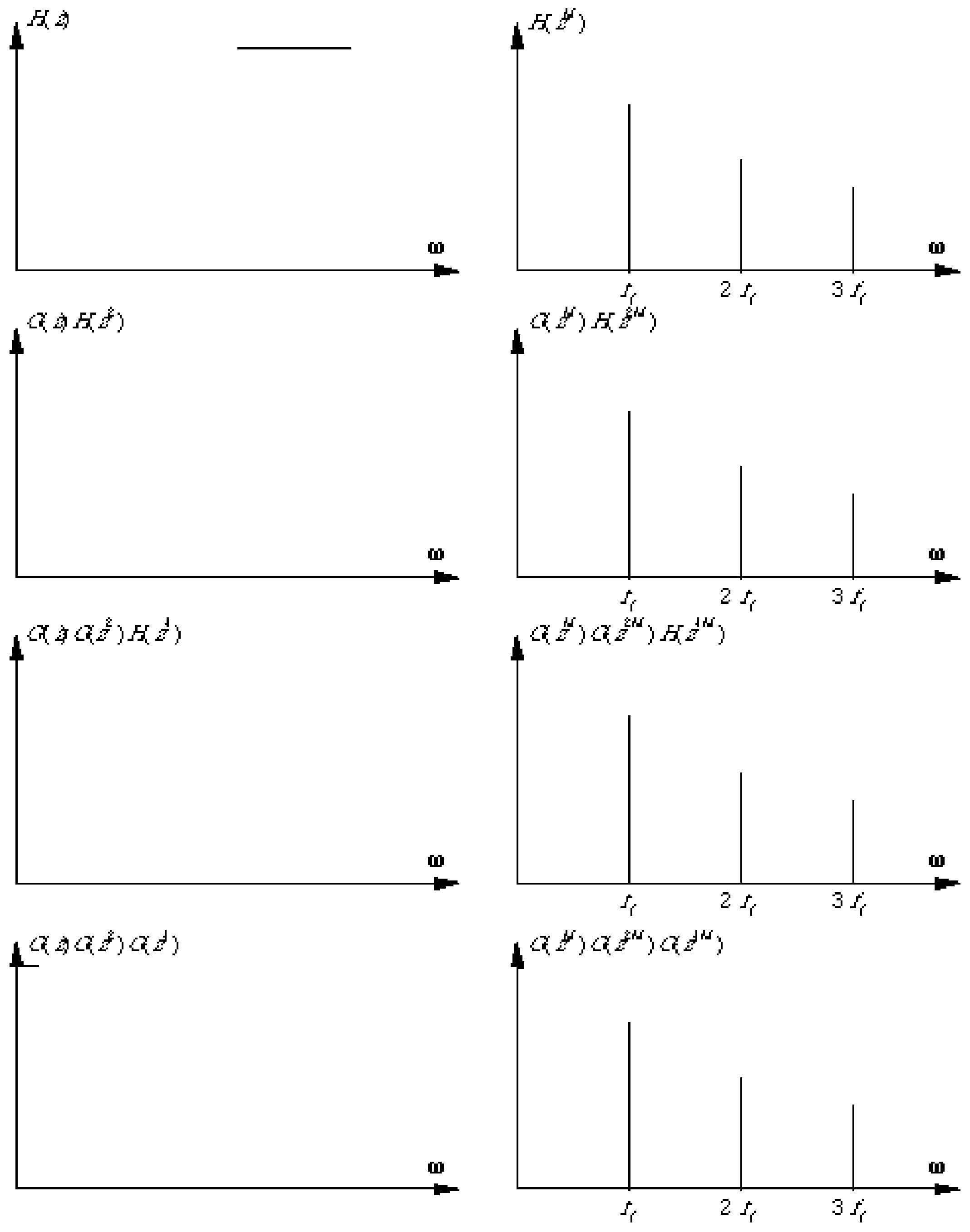


Figure 5.1.6: Compared filter responses of prototypes and their oversampled versions (f_f = fundamental frequency of analyzed sound).

This scheme not only divides the pitched and unpitched components of a signal, but also obtains information about the quality of inharmonicity. The output of the first highpass filter contains the signal components that are farthest from the harmonics, and thus represents the noise part of the sound. Subsequent filter stages, which are progressively closer to the partials, yield components that represent more long-term fluctuations like vibrato, *i.e.* small and slow variations of the basic frequency.

Evangelista remarks that the functionality of the algorithm heavily depends on precise pitch information; otherwise the outputs of the filters deliver misleading results. The need for a pitch in advance to the wavelet transformation also limits the usefulness for real-time operation. Furthermore, as the filter responses are substantially long due to their oversampling, latency is a problem in this approach.

However, the pitch-synchronous scheme motivated the further exploration of comb filters. Indeed, a simplification of Evangelista's algorithm provided useful results in real-time, although this solution had little do with wavelets anymore.

5.2 Focusing on Comb Filters

5.2.1 The Mathematics

The first simplification in order to achieve real-time response was to use the shortest filter possible as a prototype. For a highpass this is achieved by an FIR filter that subtracts each sample from its predecessor, defined as

$$y(n) = x(n) - x(n-1) \tag{5.2.1}$$

which corresponds to the following transfer function

$$H(z) = 1 - z^{-1} \tag{5.2.2}$$

whose frequency response is shown in Fig. 5.2.1a. In the wavelet terminology this filter is called the 'Haar Wavelet'. In Matlab it can be easily implemented with

```
H = [1 -1];               % highpass filter
freqz(H,1);               % plot its frequency response
out = filter(H,1,in);     % filter a signal
```

Upsampling this filter by a factor of M using (4.3.11) we obtain

$$\hat{H}(z) = H(z^M) = 1 - z^{-M} \tag{5.2.3}$$

with

$$M = \lfloor f_s \, T_{period} \rfloor$$

This has the frequency response shown in Fig. 5.2.1b. The plot clearly displays the equidistant notches that essentially cancel all harmonic components if *M* is tuned to the length of the period of the fundamental frequency.

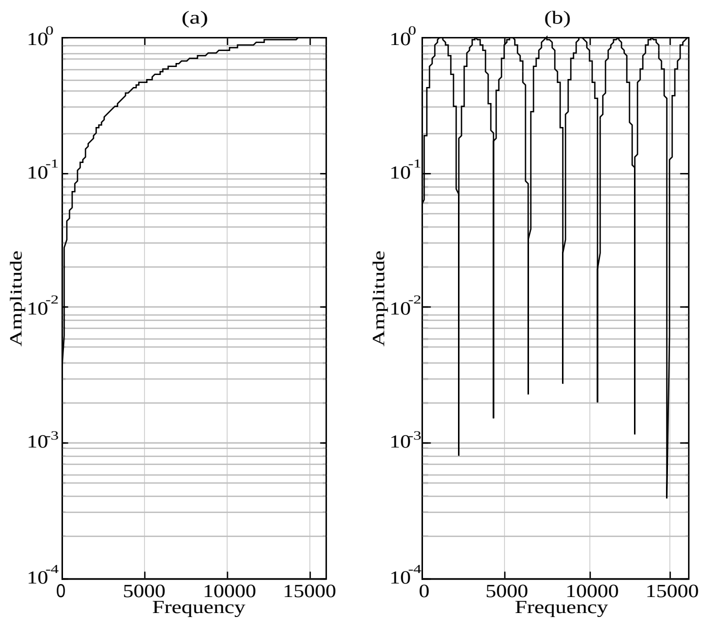


Figure 5.2.1: Frequency response of (a) *H(z)* and (b) *H(z^M)* for *M=15*

In Matlab `H` is stretched by inserting *M-1* zeros:

```
    Hov = [1 zeros(1,M-1) -1];
```

The lowpass filter corresponding to (5.2.1) is defined as the addition of two adjacent samples:

$$y(n) = x(n) + x(n-1) \tag{5.2.4}$$

The lowpass filtering, as well as all of the subsequent downsampling and filtering, was not used because the main interest was the "really noisy" signal, *i.e.* the inharmonic components, not those in the vicinity of the partials. This was the second simplification which basically eliminated all wavelet paradigms.

5.2.2 Towards Realization

With a single one-pole comb filter left, implementation reduced to a simple convolution (defined by (4.3.5)) of the input signal with the FIR filter `Hov` defined above. This filter could be as long as 800 samples[26] which would normally require quite a lot of processing power if computed at every sampling interval. However, in this case a much simpler scheme could be used since almost all of the coefficients are zero. Performing the inverse z-transform on (5.2.3) yields

$$y(n) = x(n) - x(n-M) \tag{5.2.5}$$

This can be easily implemented by buffering the input signal and subtracting each new sample from the one that occurred *M* samples before. This yields a time-domain interpretation of the comb filter: it measures the difference between one period and its predecessor. For completely periodic signals this measure is zero while for transient or aperiodic events it reaches a maximum. For slow fluctuations in frequency and amplitude the output is accordingly smaller. Fig. 5.2.2 illustrates the scheme using a delay line; the code can be found in A1.3.

[26] For a fundamental frequency of 40Hz at a sampling rate of 32kHz.

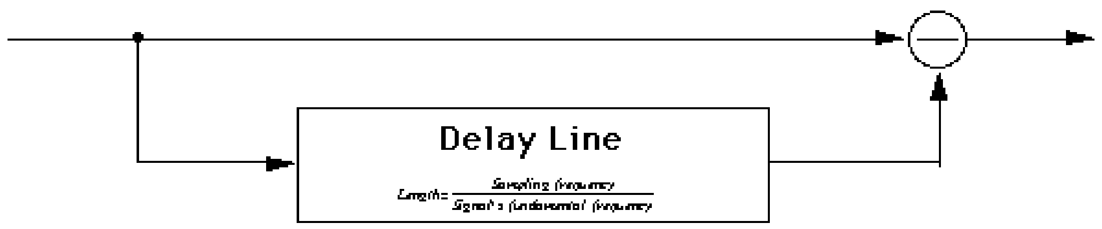

Delay Line
Length = Sampling frequency / Signal's fundamental frequency

Figure 5.2.2: Delay line implementation of the comb filter

Two problems have to be mentioned at this point. First, it follows from (5.2.3) that M can only be an integer, which means that delay lengths can only be obtained for a limited set of fundamental frequencies. In practice, this does not affect the functionality as the resolution is adequate for low frequencies. For higher frequencies, this is more of a problem, but re-pluck detection is easier in that case anyway as mentioned in Sec. 4.6.1. However, for more critical applications there are approaches in the literature such as [18] that deal with fractional delay lengths. Some of these schemes were implemented but did not show significant improvements.

The second problem of a pitch-synchronous comb filter is, that pitch information is an absolute necessity for its functionality. This, however, will not be discussed further since the focus of the project was on the improved detection of re-plucks where a pitch datum is available in advance. For separating the inharmonic part of an initial attack from its harmonic components, this comb-filtering scheme would not be useful. One way around this limitation is the fret-scanning technology mentioned in Sec. 2.2.3, where a pitch is known in advance of the attack.

5.2.3 Analyzing the Database

The comb-filtering algorithm was applied the database described in Sec. 4.5. As an example, the re-pluck of Fig. 4.6.2 is used once more to allow comparison of the result to the one obtained from the DTWT. The comb-filtered signal is displayed in Fig. 5.2.3.

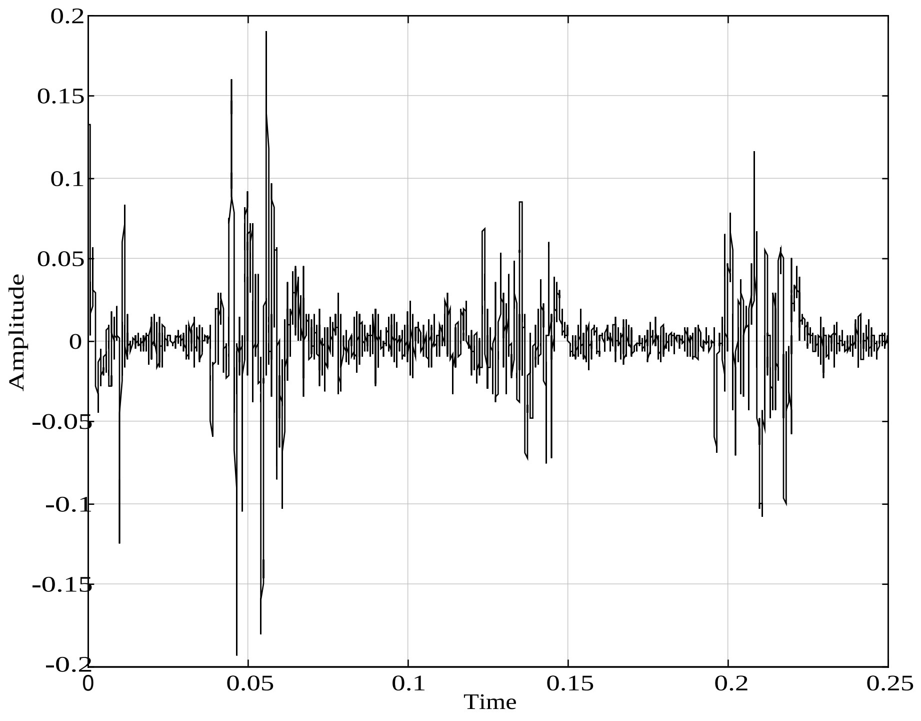


Figure 5.2.3: Comb filtered re-pluck of Fig. 4.6.2

The spectrogram of this signal is given in Fig. 5.2.4. This is similar to the one for the original re-pluck in Fig. 4.6.3, but clarifies the improved contrast between periodic and transient phases achieved by the pitch-synchronous comb filter.

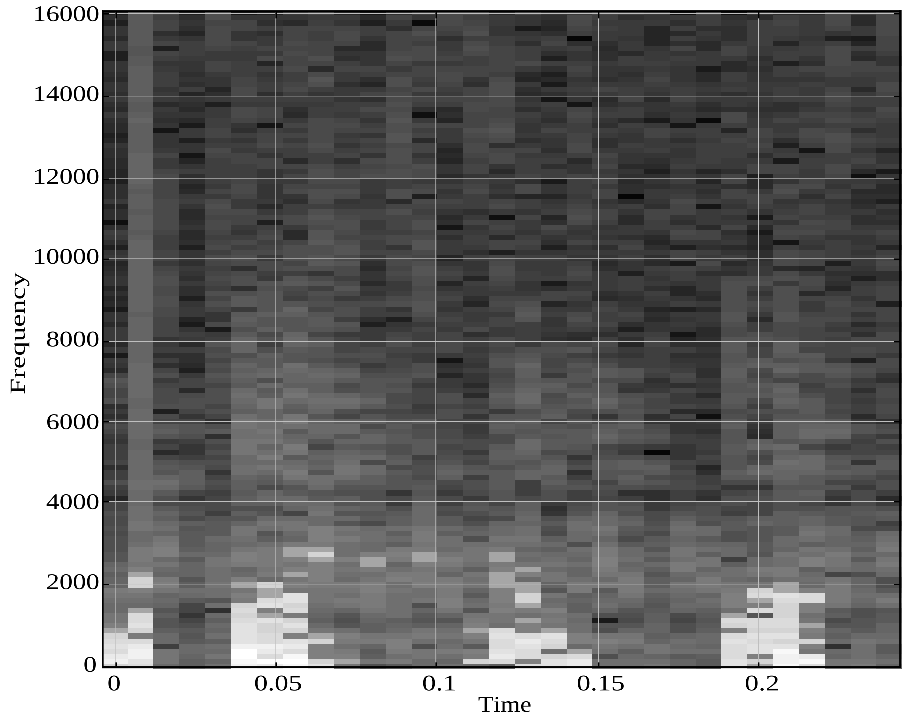


Figure 5.2.4: Spectrogram of the comb filtered re-pluck of Fig. 5.2.3 (the peak around 0.01s is caused by boundary errors, not by a re-pluck)

The contrast is not much improved for the high frequencies that were of particular interest in the STFT and DTWT. Instead, the low frequency regions promise a better detection rate.

Furthermore, although Fig. 5.2.3 does not show a substantial improvement when compared to Fig. 5.1.4, it was much more effective for softly played notes as can be seen in Fig. 5.2.5.

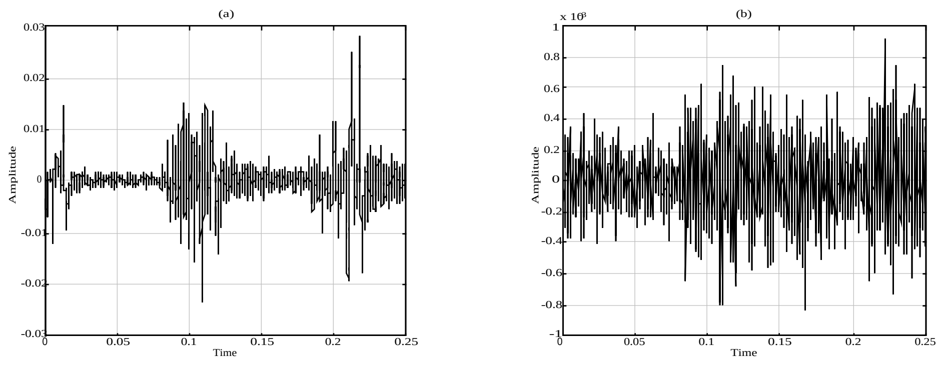


Figure 5.2.5: Comparing the analysis results for softly played re-plucks: (a) from the comb filter (b) from the DTWT (= Fig. 5.1.5b)

Finally, it should be noted that the sound examples B3.2 and B4.1b (the latter as a preview of the results from the real-time implementation) show that the output of the pitch-synchronous comb filter is not merely a useful signal for making onset decisions but actually delivers the very sound of the plucking itself.

5.2.4 Side Effects of Comb Filters

Two side effects of applying the algorithm to various sounds of the database are caused by using the wrong delay length. For instance, dividing it by two results in a dilation of the notches. This eliminates the even harmonics of the sound as is shown in Fig. 5.2.6. This procedure does not leave a percussive pluck noise but a harmonic sound with a somewhat nasal quality.

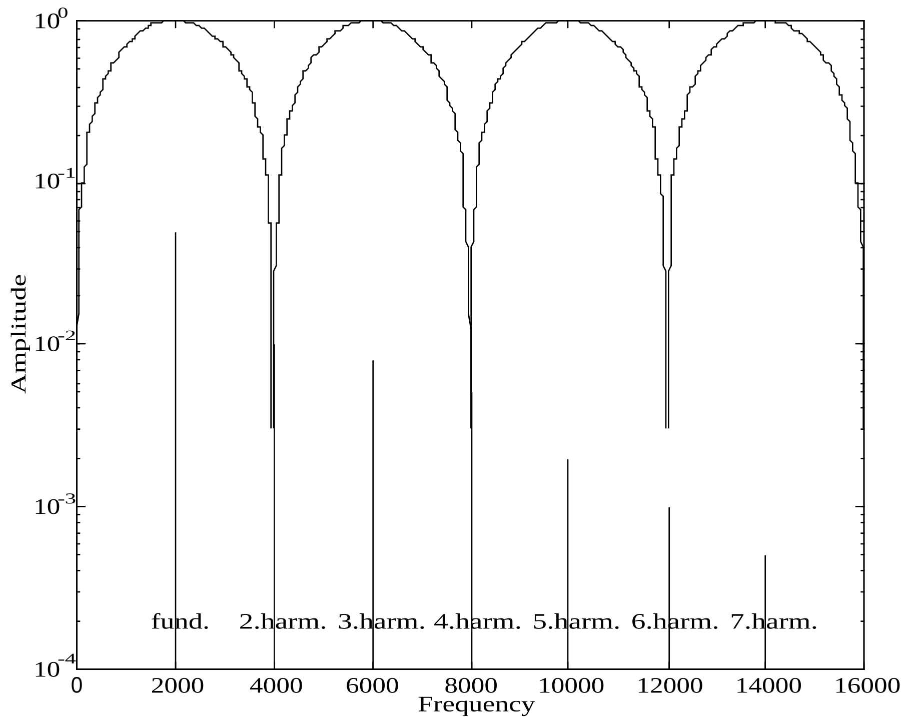


Figure 5.2.6: Tuning the comb filter to the even harmonics

Another modification of the comb filter is to use the lowpass (5.2.4) instead of the highpass filter (5.2.1) prototype while maintaining the shortened delay length. This shifts the above frequency response such that the odd harmonics are canceled, including the fundamental. The resulting effect resembles an octave transposition of the original sound.

5.3 The Synthesis Part

5.3.1 General Observations

The output of the pitch-synchronous comb filter provides a useful trigger decision, which was the main requirement of the analysis. In terms of synthesis links, the derived trigger events can be mapped to any arbitrary synthesis algorithm. A standard use could be to start a new note, without any further specifications coming from this part of the analysis. The main achievement in this respect is the trigger decision quality and the precise timing or improved responsiveness as can be heard in B4.2.

It was mentioned above that the comb filter does not deliver just trigger information but actually the sound of the finger plucking the string, without the sound of the (vibrating) string itself . Building on the work of J.O. Smith, who had used percussive samples to drive his physical modeling synthesis algorithms, this 'pluck noise' was used to establish an analysis-synthesis link on a very low level of abstraction. Strictly speaking, this signal does not carry much information apart from the event timing. Instead, it has been shaped by information from an outside analysis, *i.e.* the pitch detector. Still, it is useful as an input signal for a number of synthesis algorithms that allow for an external excitation.

Apart from the previously mentioned work by Smith, which is an extension of the Karplus Strong Synthesis, a resonance synthesis implemented at CNMAT was of interest since it involved the Reson8 implementation platform. The main idea is to use banks of filters with a high resonance [1] based on the earlier research by [2, 31, 37]. When excited by an impulse, these filters start oscillations that decay at a specified rate, and create a complex sound when mixed together.[27] In principle, the excitation source can be any desired sound; however, if a sustained pitched sound is used its frequencies create strange modulation products if they are different from the frequencies of the filters. In other words, the result in this case resembles the excitation, and is not a strong function of the synthesis settings. While an interesting effect on its own, the primary idea is to excite the filters by short percussive sounds with no clearly defined pitch, like drum sounds, synthesized noise bursts, or the sound of a string pluck. Then, the output is shaped by the excitation mainly during the attack of the sound, while its sustained part is determined by the synthesis filter parameters.

These considerations also hold true for the excited Karplus Strong Synthesis, which was used in favor of the Resonance Synthesis for reasons explained in the next section.

[27] This should not be mixed up with the excitation-resonance concept of Subtractive Synthesis where the resonator is usually a few filters whose main purpose is not being sound sources. Rather they are used to shape the signals coming from oscillators. However, the different concepts of synthesis cannot be strictly differentiated.

5.3.2 The Karplus Strong Synthesis - a Simple Physical Modeling Algorithm

The Karplus Strong Synthesis (KSS) was introduced by Alex Strong and Kevin Karplus in 1978 and since then has been extended by the authors [16] as well as many other researchers [13, 15, 40]. The main idea is to use a delay line with the output fed back to its input after it has been attenuated and maybe filtered. This is depicted in Fig. 5.3.1. To trigger a sound, the delay line is filled with random values that create a noise burst during the first cycles. Due to the feedback scheme, this becomes a periodic signal with a fundamental frequency corresponding to the delay length. This is the reverse process of the pitch-synchronous comb filter, and is thus a logical synthesis approach. Also, the characteristic sounds produced by the KSS resemble plucked strings.

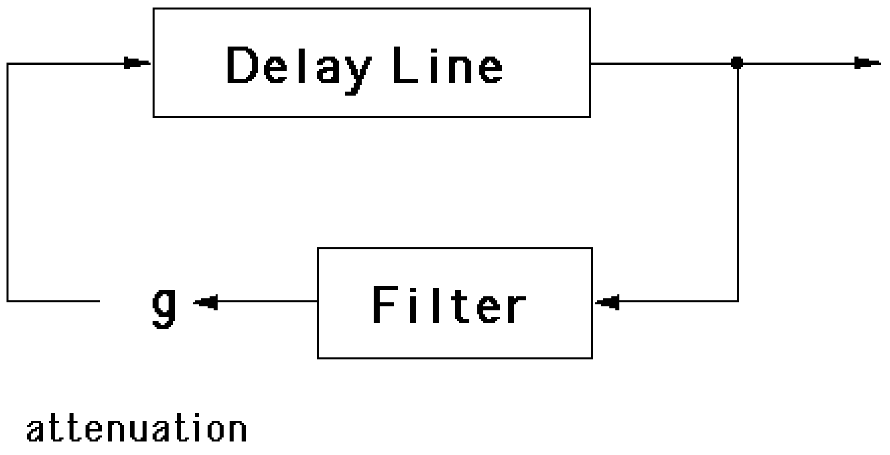


Figure 5.3.1: Delay line implementation of the basic Karplus Strong Synthesis

The key factors that influence the sonic result of the KSS are the attenuation and the filter in the feedback path, which determine the decay rate of the amplitude and the brightness of the sound. Furthermore the "color" of the random noise burst characterizes the attack of the signal. Its pitch, as mentioned, corresponds to the delay length and schemes have been developed to create frequencies that are not constrained by an integer number of samples; this mirrors the problem of fractional delay lines mentioned in Sec. 5.2.2.

The main extension to this algorithm used for this project is the one described in [40]. Here, instead of filling the delay line buffer with random numbers, an external excitation is introduced to trigger the sound. This is shown in Fig. 5.3.2. It allows for a greater flexibility in shaping the attack, for example by using sampled percussion sounds. For this project the output of the pitch-synchronous comb filter, *i.e.* the plucking sound, is fed into this externally excited Karplus Strong Synthesis (XKSS). This establishes a close link between the way the guitar string is plucked and the sonic result of the synthesis.

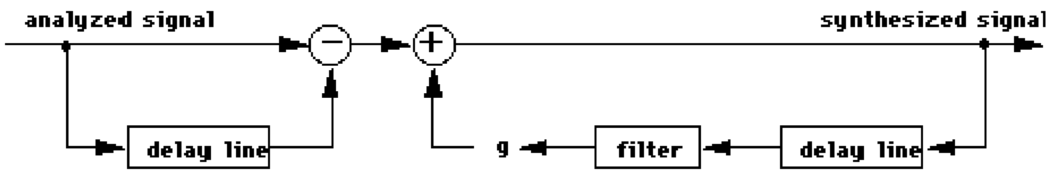


Figure 5.3.2: The KSS excited by the comb filtered re-pluck

Like its analysis counterpart the XKSS can be interpreted as a filter:

$$H(z) = \frac{1}{1 + gH_f(z)z^{-M}} \tag{5.3.1}$$

with

$$M = \lfloor f_s T_{period} \rfloor$$

and $H_f(z)$ the transfer function of the filter in the feedback pass, which was omitted in this project for simplicity. The Matlab code is given in A1.9.

5.4 Refining the Design

5.4.1 Additional Processing

As can be seen from the spectrogram of the comb filtered re-pluck in Fig. 5.2.4 the best contrast between the harmonic steady-state phases and the inharmonic transients is found in the frequency range from 0Hz to 1.5kHz. This motivates the use of a lowpass filter to improve the detection rate; specifically a 4th-order Chebyshev type II filter with a flat passband response is used. The cutoff frequency is set to 1.5kHz, the stopband rejection to 30dB. As this filter limits the bandwidth by a factor of 10.7, the filter output is downsampled subsequently by 8 (to be on the safe side) for data reduction.[28] Furthermore, the signal is rectified; positive and negative parts of the signal are not

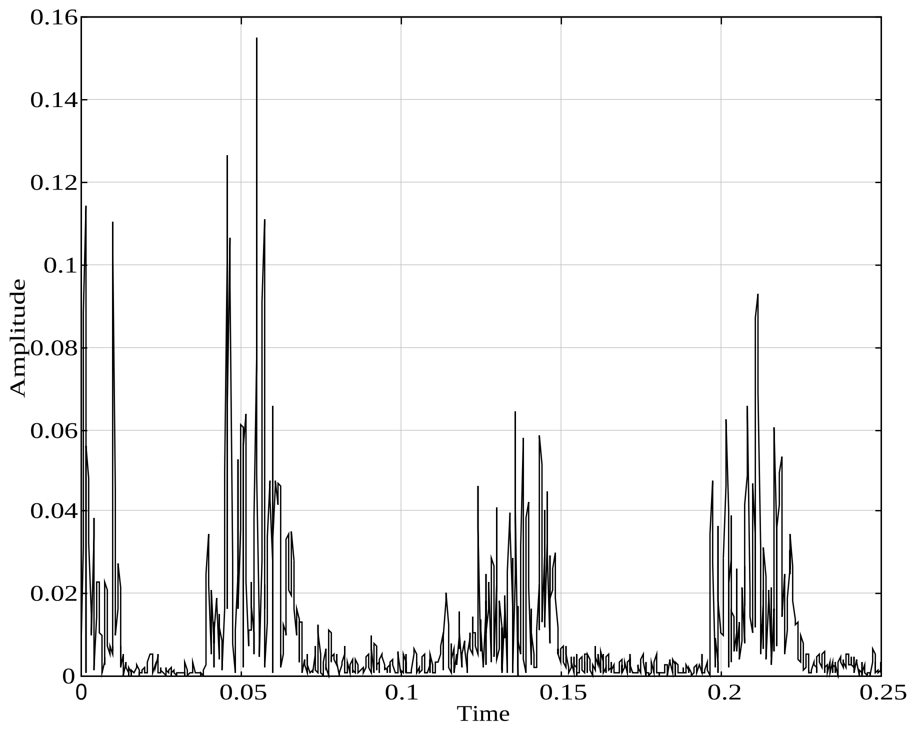
[28] $(f_s/2)/1.5\text{kHz} = 10.7$, with f_s=32kHz

treated separately. The corresponding routines can be found in A1.4; their result is shown in Fig. 5.4.1 with the formulas defined as

$$y_{CFLP}(n) = \sum_{m=-\infty}^{\infty} h_{ChebyII}(m) y_{CF}(n-m), \text{ with (4.3.5)} \tag{5.4.1}$$

$$y_{CFLPDS}(l) = y_{CFLP}(8l), \text{ with (4.3.9)} \tag{5.4.2}$$

$$y_{CFLPDSR}(l) = |y_{CFLPDS}(l)| \tag{5.4.3}$$



.

Figure 5.4.1: Comb filtered re-pluck as in Fig. 5.2.3 subsequently lowpass filtered, downsampled (f_s = 4kHz) and rectified

The sonic result is presented in B3.3, however, without the rectification stage.

A filter was later added to block the DC[29] component of the signal as it could be problematic for the analysis in certain cases. DC is often introduced by piezo-electric pickups which were used in this project for recording the guitar sounds. Usually it is blocked in the analog domain, however, sometimes the Analog-to-Digital converters themselves have a slight DC offset. The filter is designed as a highpass with the cutoff frequency well below the audible range.

[29] DC = direct current

5.4.2 Energy Measurements

In Fig. 5.4.1, the onsets of the re-plucks are generally visible as sudden rises in energy. However, it is not possible to derive trigger events by using absolute thresholds as this strategy misses events with lower amplitude or causes false triggering when the threshold level is set too high or too low, respectively. This calls for a relative scheme where each new sample is compared to its immediate predecessors. The following solution was found to work best.

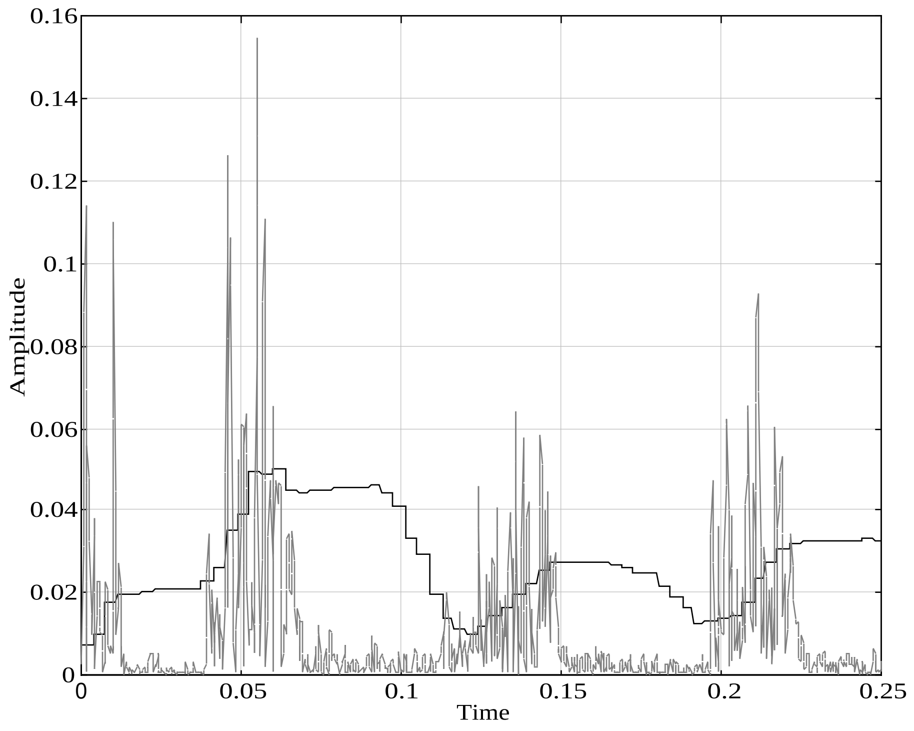
First, maxima are evaluated on a specified grid size *K* of about 5-20 samples.

$$y_{\max}(k) = max(y_{CFLPDSR}(l))\ \Big|_{l=K(k-1)+1}^{Kk} \qquad (5.4.4)$$

Then the arithmetic mean is calculated for the last *J* maxima (again about 5-20), subsequently weighted by a factor and compared to the most recent sample for a trigger decision.

$$\begin{aligned} y_{avg}(j) &= c_{threshold}\frac{1}{J}\sum_{k=J(j-1)+1}^{Jj} y_{\max}(k) \\ flag_{trigger}(l) &= (y_{CFLPDSR}(l) > y_{avg}(\left\lfloor \frac{l}{JK} \right\rfloor)) \end{aligned} \qquad (5.4.5)$$

The result is illustrated in Fig. 5.4.2. The code is given in A1.5.



.

Figure 5.4.2: Filtered re-pluck as in Fig. 5.4.1 vs. averaged signal (K=15; J=15; $c_{threshold}$=1) providing an adaptive threshold level

In addition to detecting the onset of a re-pluck, it is also necessary to find its end, *i.e.* the moment when the string is vibrating again. One approach is to use further filtered versions of $y_{CFLPDSR}$ to find the end by looking for drops of energy in the comb-filtered signal. A more favorable approach involves a pitch-synchronous energy measure. Instead of taking the average of a fixed number of rectified samples, it operates on a whole period of the signal, thus resembling the described strategy for the comb filter. In contrast to the first energy measure it uses the original signal instead of the comb filtered one:

$$y_{PSA}(n) = \frac{1}{P} \sum_{p=n-P+1}^{n} |x(p)| \qquad (5.4.6)$$

with

$$P = \lfloor f_s T_{period} \rfloor$$

The result of this process is depicted in Fig. 5.4.3, where it can be observed that the end of the re-plucks are marked by a distinct energy peak, even for the middle re-pluck where the overall level drops. In practice this scheme produces more reliable results than the first method. The corresponding is given in A1.6.

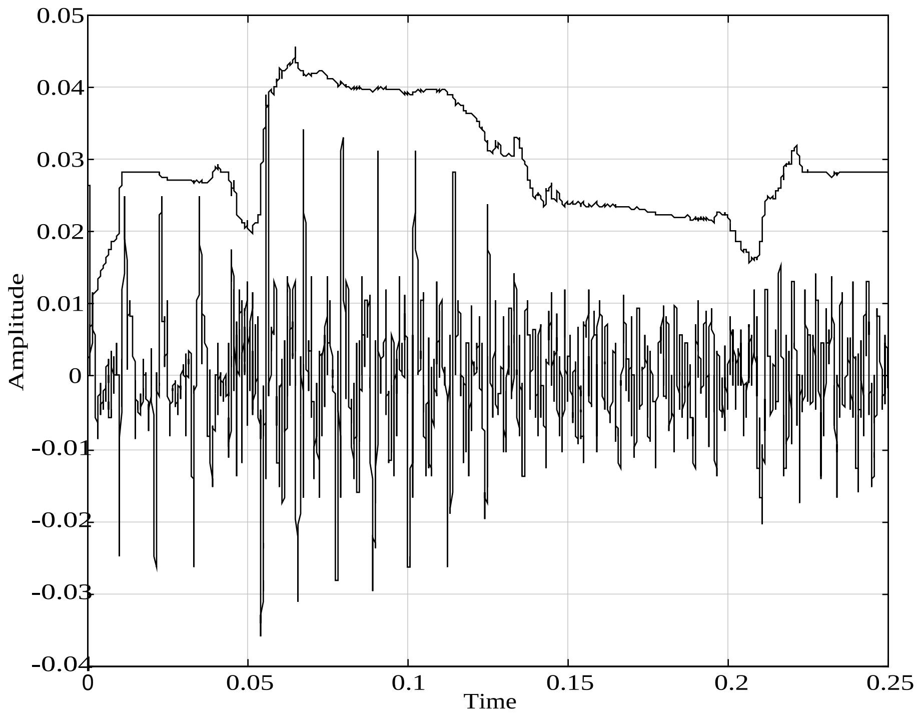


Figure 5.4.3: Pitch-synchronous average compared to the original re-pluck

5.4.3 The Trigger Decisions

The trigger decision for the onset of the re-plucks can be easily done by comparing $y_{CFLPDSR}$ to the adaptive threshold level obtained by the combined maximum/average scheme of Sec. 5.4.2. Detecting the peaks of the pitch-synchronous average requires a more elaborate approach described next.

The proposed scheme combines minima and relative maxima searches with hold times and an absolute threshold parameter. This is depicted with the following example written in pseudo code. The Matlab implementation is given in A1.7.

```
repeat
   minimum = min(signal_level, minimum);
   relative_maximum = signal_level - minimum;
until relative_maxima > threshold;

maximum_hold_time = maximum_hold_time_default;

repeat
   decrement(maximum_hold_time);
   if new_maximum_found(signal_level, maximum)
      maximum_hold_time = maximum_hold_time_default;
   endif
until maximum_hold_time = 0;
```

Finally, the resulting trigger signal is shown in Fig. 5.4.4 as well as all the signals discussed in the previous sections. This figure was created with the "main.m" program presented in A1.8. The trigger signal has also been recorded (B3.4)

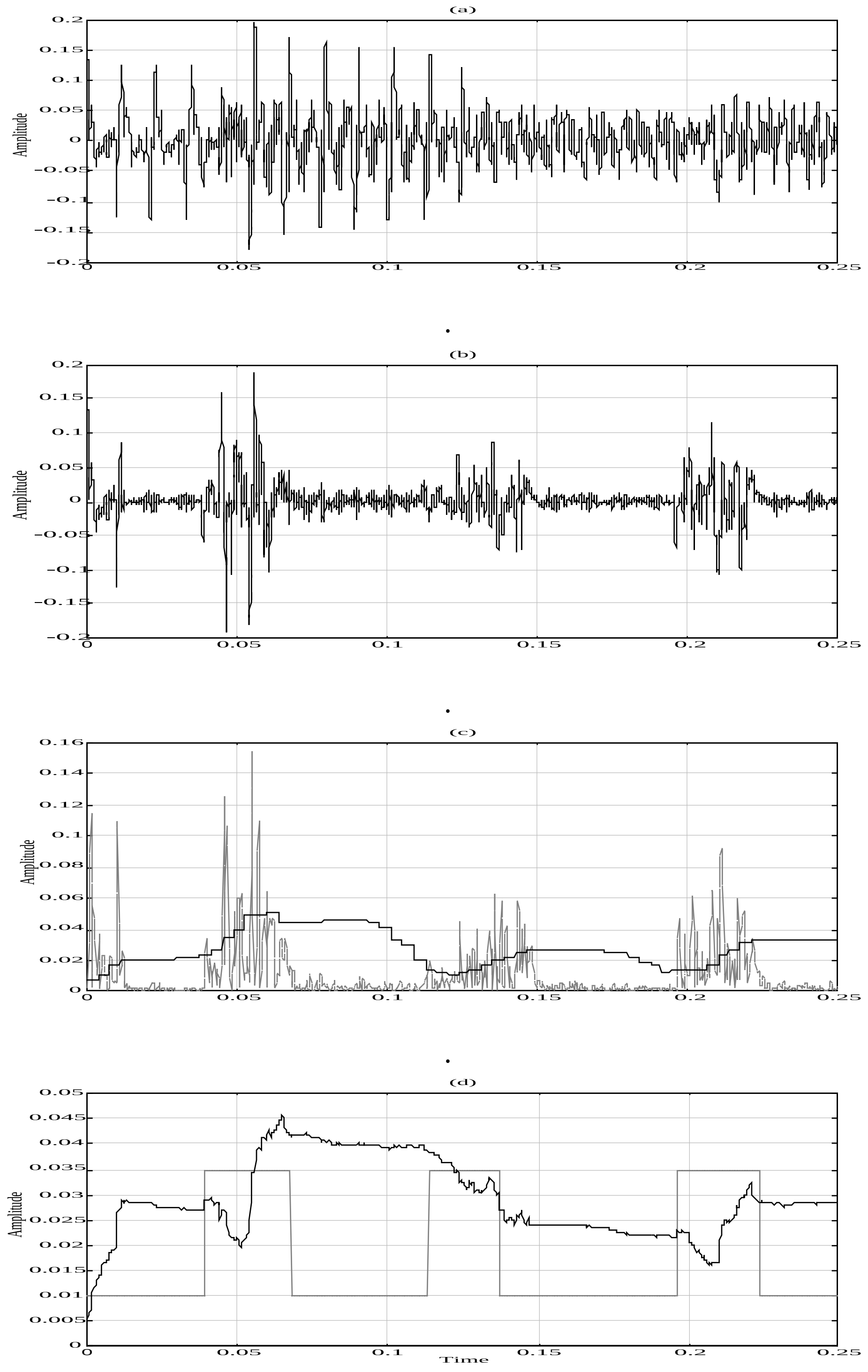


Figure 5.4.4: Comparison of all the discussed signals (a) the original re-pluck signal, (b) the pitch-synchronous comb filtered signal, (c) b. further lowpass filtered and downsampled & adaptive threshold (max-mean average) and (d) the pitch–synchronous average & trigger decision

5.4.4 A State Machine to Define the Phases of a Guitar Sound

Apart from the aforementioned onsets and endings of the re-pluck it has to be taken into account that in a real-time implementation, one cannot assume a vibrating string as the initial state, but silence. Thus, it is necessary to define a set of signal states, including some that describe the attack of a guitar signal. For a more complete description states for the initial unpitched parts of the attack are included as shown in Fig. 5.4.5; this provides for further distinction than 'silent' and 'oscillating'. First, the process waits for a threshold to be exceeded, then looks for a maximum value within a specified hold time, and finally waits for pitch information from an external pitch detector. Another state was later added in order for one period to pass before validating the comb filter output.

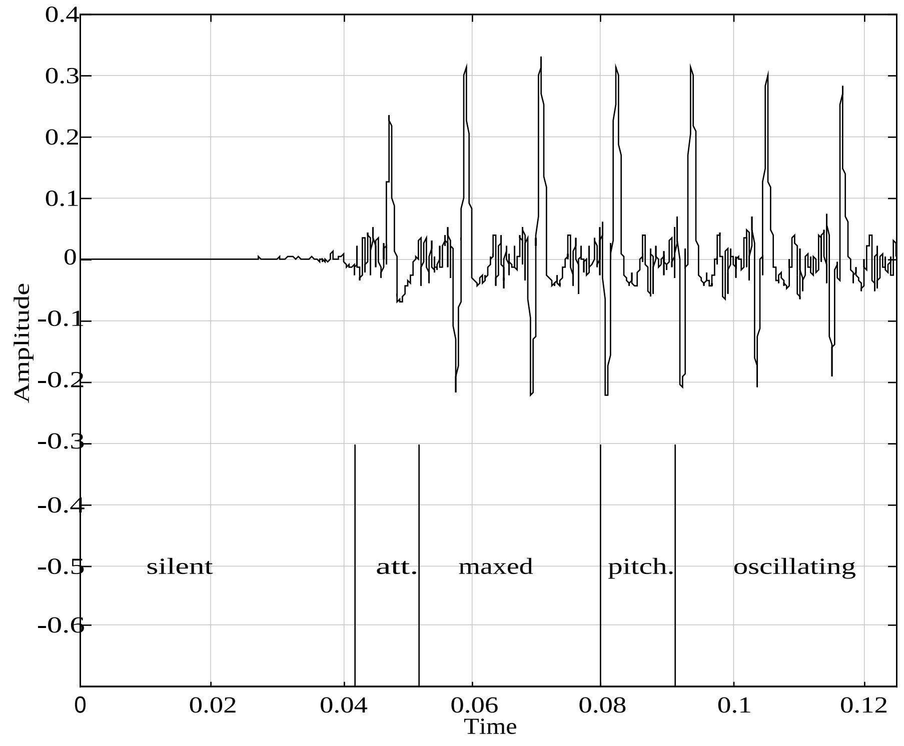
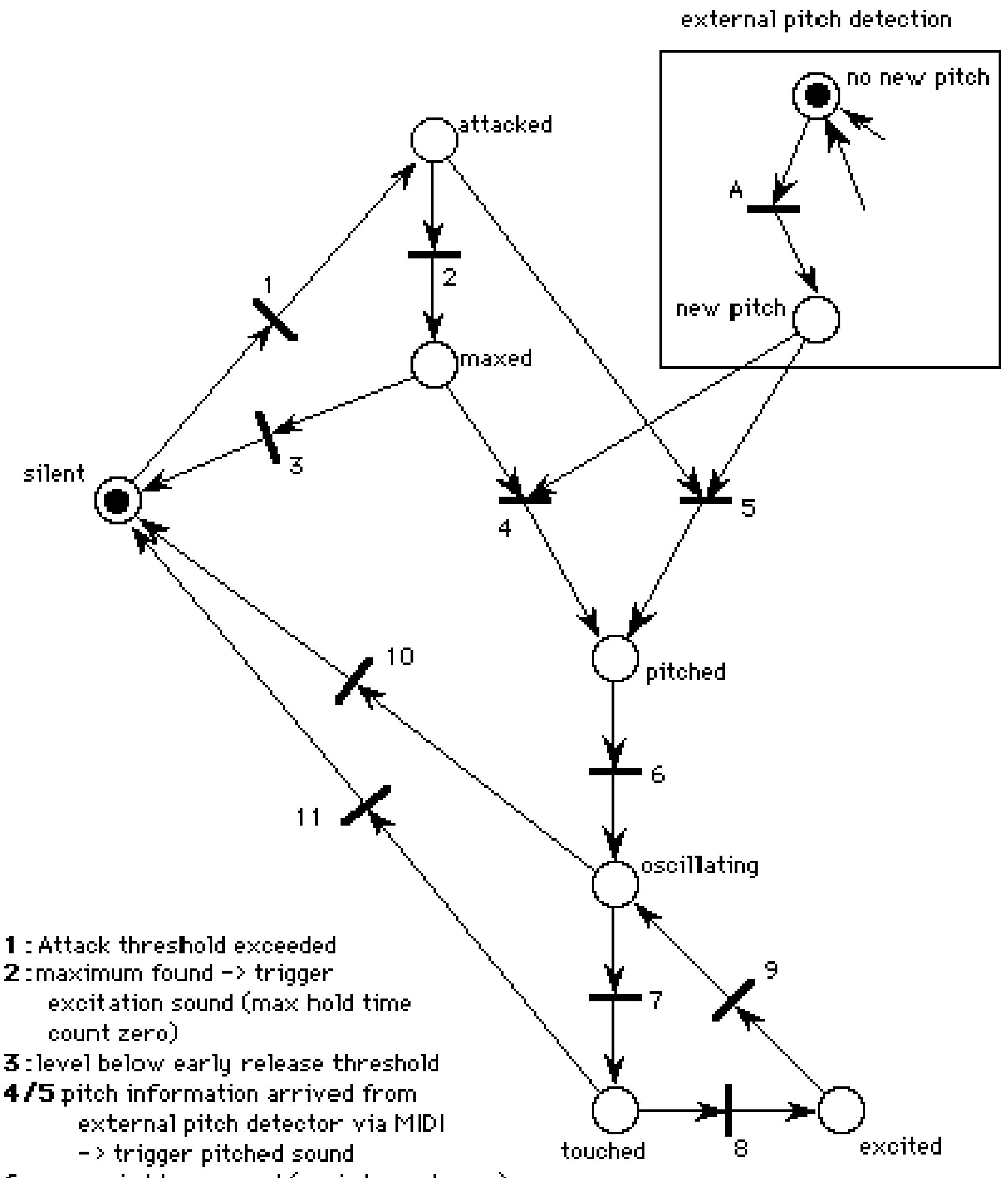


Figure 5.4.5: The phases of a guitar attack

With this scheme it is possible to trigger a transient unpitched sound as soon as the 'maxed' state had been reached, which made sense for reasons mentioned in Sec. 4.2.2. Then in the 'pitched' state the actual sustained sound could be incorporated. An additional benefit of this is that if the player makes a percussive noise on the guitar, like hitting the bridge as in flamenco, this will be mapped to a corresponding percussive signal from the synthesizer.

The complete state machine is described by the petri-net shown in Fig. 5.4.6. It includes the external process of pitch detection. A commercial pitch-to-MIDI converter is used and interfaced to the process via MIDI. Note that a typical series of re-plucks would cycle through the states 7, 8, and 9. The 'excited' state is included to avoid immediate re-triggering after leaving the 'touched' state, which occurred sometimes with very loud notes. A few milliseconds of hold time are enough to circumvent this difficulty, though.



1 : Attack threshold exceeded
2 :maximum found -> trigger
excitation sound (max hold time
count zero)
3 :level below early release threshold
4/5 pitch information arrived from
external pitch detector via MIDI
-> trigger pitched sound
6 :one period has passed (period count zero)
lock filter (comb, PSA)
7 :string touched (rise in comb filter output
= inharmonic energy)
8 :peak in PSA found (max hold time count zero)
9 :excited hold time count zero
10 :energy level below release threshold
-> release sound
11 :energy level below release threshold
-> damp sound

A :new pitch detected

Figure 5.4.6: The state machine for processing transient guitar events

5.4.5 A New Feature

Apart from the improved triggering and the low level mapping to synthesis, the algorithm provides an additional feature. As described in Secs. 5.2.1 and 5.4.2, there are two processed signals that mark the onset and the ending of a re-pluck. These can be used to derive a parameter that represents its length, *i.e.* the time lag between touching the string and releasing it for vibration.

A guitar player does not generally think about how long it takes to pluck a string. However, this pluck-length parameter is closely related to the articulation and on a subtle level it is influenced by striking the string more gently or aggressively. This parameter can be mapped to any desired high level synthesis parameter, where influencing the attack time of a sound would be a logical choice.

Experiments showed that this new parameter ranges from a few ten milliseconds to several hundred; at a certain point it actually makes more sense to split the re-pluck into two events, a damping and a new attack. It was not easy to perform this subdivision precisely as it depends on the playing style. Consequently, the 'maximum pluck length' setting is left to the preference of the user.

5.4.6 The Resulting Main Building Blocks

The diagram of the main building blocks in Fig. 5.4.7 summarizes all the described functionality. First, the guitar signal is fed into the **pre-processor** which comprises all the filters and energy measurements. These are sent to the **state machine**, which arrives at a trigger decision for loading the random noise into the **synthesizer**. At this stage there is the choice of using the comb filtered output instead to drive the XKSS directly, thereby bypassing the event level.

Two external devices should be mentioned. A pitch-to-MIDI converter is necessary to provide all of the internal modules with pitch information. Also, an optional external sound generator for percussive sounds provides a third alternative for the input to the XKSS; this can be heard in B4.4.

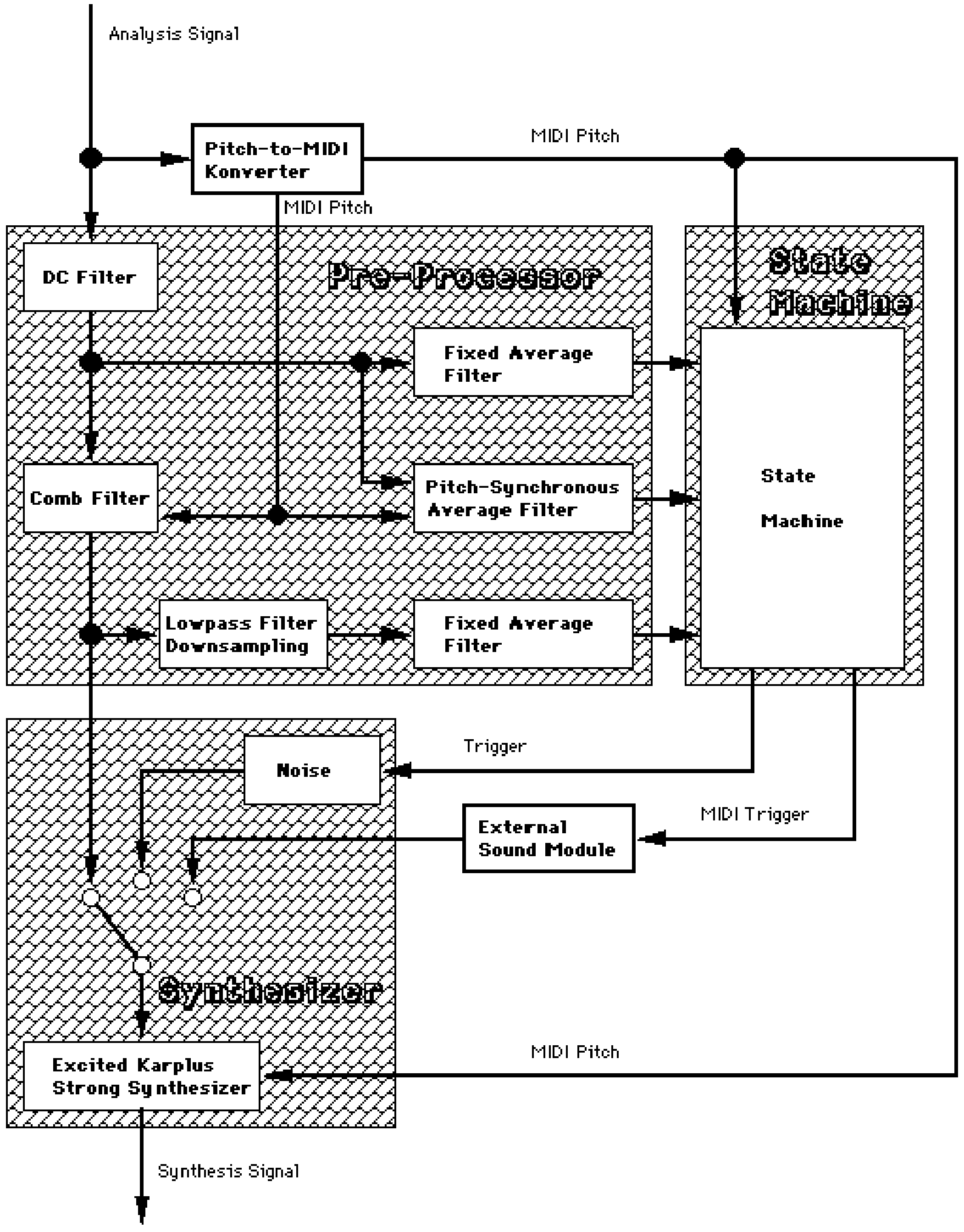


Figure 5.4.7: The main building blocks. Note that from left to right we move from the sound to the event level

6 Implementation

6.1 Moving Towards Real-time Behaviour with C

Following the schedule, the first task of the implementation part of the project was the conversion to C. After the algorithms had been successfully optimized in Matlab to work properly for the database sound examples, this step was necessary to verify the strategy for a large set of different re-plucks. With the good audio facilities of the Silicon Graphics Indigo it was a logical step to move on to a real-time implementation, without having to deal with too much implementation detail of a DSP-based system at this point. This decision was encouraged by the fact that with the integrated audio libraries it was possible to get the basic comb filter delay line working in very short time. Nevertheless, the C implementation was abandoned after a week, mainly because after a few discussions it was decided that too much time would be lost towards the completion of the project.

6.2 The Environment

6.2.1 The Reson8 [1, 3]

The Reson8 is a stack of eight processor boards interconnected with flat cable busses. Each board contains a Motorola DSP56001 [26] clocked at 27 MHz, 1k x 24-bit words of fast dual-ported static RAM, a DB15 connector for serial I/O, and glue logic for an 8-bit bussed connection to a controlling host processor.

One of the eight processor is customized to be a master "move engine" as shown in Fig. 6.2.1. The move engine has an additional 32k x 24-bit words of fast static RAM and is the only processor that has access to all the external memory of the remaining seven processors. It can be used to route signals between the processors and to monitor and control the activities of each processor. In this case it was also the only processor connected to the Analog-to-Digital / Digital-to-Analog Converter via the SSI port.

Each processor is booted and controlled over the 8-bit host bus. A commercial NuBus card was slightly modified to interface the Reson8 to the host processor, which was chosen to be the Apple Macintosh II.

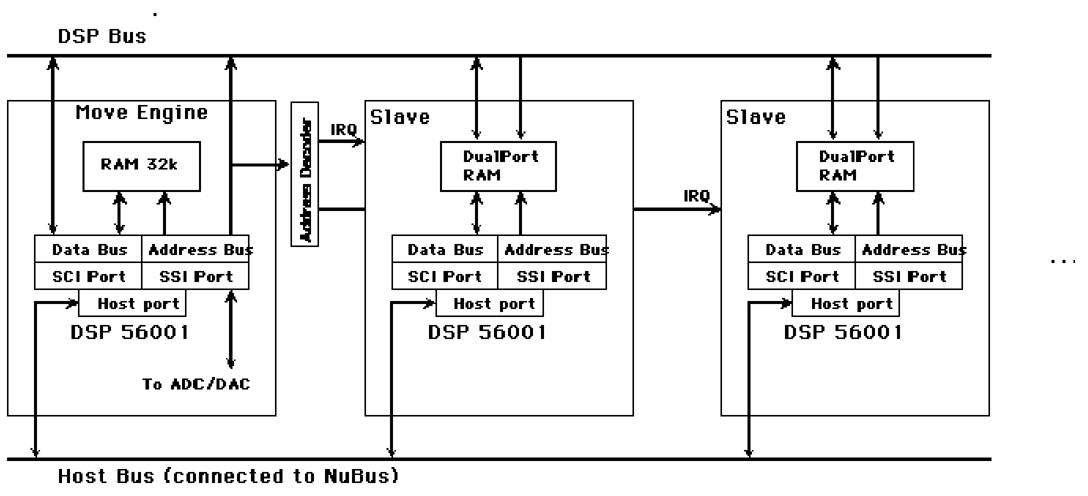


Figure 6.2.1: Reson8 overall architecture

6.2.2 DSP Code Development

The software environment for developing DSP code was based on the standard tools provided by Motorola [27-29], comprised of a macro cross assembler, a linker and a simple non-symbolic simulator. They were embedded in MPW 3.2, the Macintosh Programmer's Workshop, the standard development platform provided by Apple. It can be used for a variety of programming languages that are added to the environment as modules. In this case it served as a shell for the DSP assembler tools as well as a signal processing package that was used to a small extent to verify filters. MPW also provided an integrated editor, however, the simulator was launched as an extra application.

Debugging DSP assembly code is sometimes difficult as algorithms that prove correct in single-step mode, do not always work under real-time constraints especially when interrupts disturb the normal program flow. Register use has to be observed precisely, as for efficiency reasons register contents are not always pushed on a stack. Moving data around from DSP to DSP adds to the confusion. Also, disabling interrupts is no solution as the algorithms rely on the data from the SSI and host port which can only be provided by using interrupts.

When a program was erroneous, the following strategy was used. The first debugging technique was to single step the program in the simulator to check for obvious mistakes.

This simulator was also used to prove arithmetic related code fragments. For all other problems, registers or memory locations that were related to the problem were copied into display variables (see Sec. 6.2.5) and observed from MAX. Finally, breakpoint flags were introduced at adequate points in the code that stopped the execution of the program. Then the variables were checked from MAX, and program flow resumed by clearing the flag.

6.2.3 MAX[30] [32, 33]

MAX is a very popular application in the computer music community that grew out of the need for a real-time programming language which should be easy to use but yet powerful. MAX is an object-oriented environment built around the idea of scheduled messages. At its core it is a real-time scheduler upon which a graphical programming language has been build that lets the user specify the flow of messages with virtual patch cords and broadcast message transmitters and receivers. MAX possesses a hierarchical abstraction mechanism allowing to encapsulate a given patch into a new object with its own inlets and outlets for handling message traffic.

MAX exists in two versions. The one used in this project works on the control level, *i.e.* it cannot be used to write patches that model sound processing. This is possible with the 'sound-version' of MAX which runs on a special signal processing workstation developed at IRCAM [21]. This workstation is controlled by a NeXT computer; the 'MIDI-version' only runs on the Apple Macintosh.

MAX has a built-in and extensible help mechanism. An object can simply be interrogated with an option-click and a functioning program demonstrating the object appears in a new window. An example is given in Fig. 6.2.2 with the help window for the `makenote` object. This also shows some of MAX' specialization towards musical application in using the `noteout` object which provides I/O functionality to the MIDI interface.

[30] MAX is named after Max Mathews, a pioneer of computer music

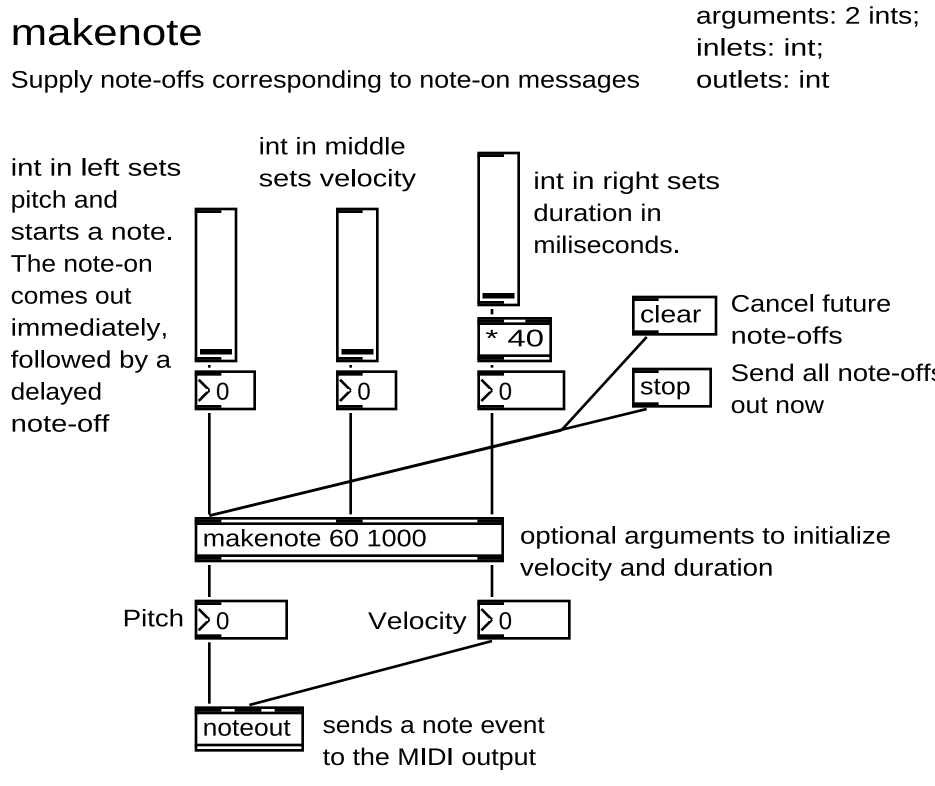

makenote
Supply note-offs corresponding to note-on messages
arguments: 2 ints;
inlets: int;
outlets: int
int in left sets pitch and starts a note. The note-on comes out immediately, followed by a delayed note-off
int in middle sets velocity
int in right sets duration in miliseconds.
* 40
clear
Cancel future note-offs
stop
Send all note-offs out now
0
makenote 60 1000
optional arguments to initialize velocity and duration
Pitch
Velocity
noteout
sends a note event to the MIDI output

Figure 6.2.2: Example MAX patch (help file for `makenote`)

Although MAX comes with a vast amount of objects it is possible to further extend it by adding external code resources. This is done in C, where libraries and headers are provided for. The programming model employs objects, classes and methods, much like other object-oriented programming languages. More recently, links to C++ have been implemented. An example for these externals are the MAX-DSP objects that have been developed in conjunction with the Reson8. They are covered by the next section.

6.2.4 The MAX DSP Objects [11]

To enable communication between MAX and the Reson8[31] a set of objects were designed to facilitate DSP control and development. This allows to upload code from the host into the DSP with the **`DSP load`** object and start its execution with **`DSP go`** . Then, with **`DSP peek`** one can observe memory locations and change them with **`DSP poke`**, both without interrupting the program flow. A range can be specified to map to the DSP's 24 bit integer representation. Furthermore, it is possible to transfer whole

[31] Links to other platforms with the Motorola DSP56001 are possible

blocks of data rather than single parameters. This improves the transfer rate significantly, and even allows for basic oscilloscope implementations as shown in the MAX patch B3.2.

On the assembler side macros are provided to link memory locations to a symbol as shown in the following code example where the same location is linked to a "peek symbol" with the `display2` command and to a "poke symbol" with the `control2` command.

```
control2    DelayLen,1,0,$1fff,0,$1fff
display2    DelayLen,1,0,$1fff,0,$1fff
```

where `DelayLen` has been assigned a memory location and set to a default with

```
DelayLenD   EQU     388          ; 388 = 32000/82.5 default (low E)

            org     x:9          ; place DelayLen at X memory
DelayLen    dc      DelayLenD    ;  location 9
```

The second parameter of `display2` and `control2` tells the parser that `DelayLen` is a scalar while the remaining four specify a direct mapping between integers in MAX and 24 bit integers in the DSP for a range of 0 to $1fff (= 8191). The corresponding MAX patch is shown in Fig. 6.2.3.

In the assembling process the macros create entries in the object file that is subsequently linked, parsed and split into four different files. Three are uploaded into the DSP's program, X data, and Y data memory, while the remaining one contains the symbols with their associated memory location and is referenced by the MAX DSP objects.

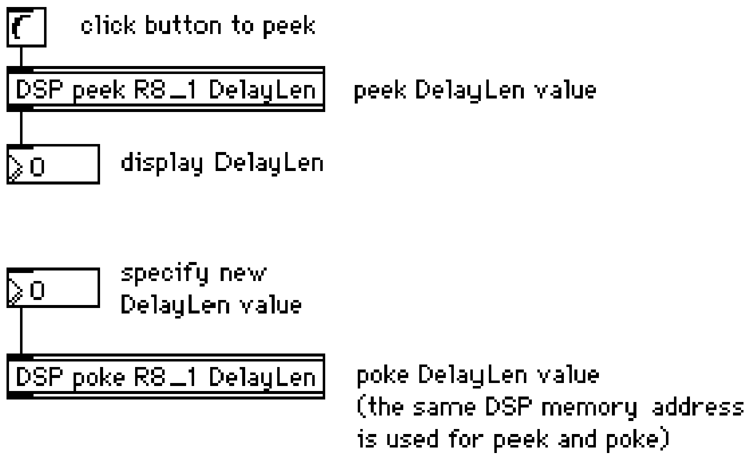

click button to peek
DSP peek R8_1 DelayLen
peek DelayLen value
0
display DelayLen
0
specify new DelayLen value
DSP poke R8_1 DelayLen
poke DelayLen value (the same DSP memory address is used for peek and poke)

Figure 6.2.3: MAX patch to display and control DelayLen

6.2.5 An Overview of the Setup

With Fig. 6.2.4 a diagram is presented to supplement the internal scheme of Fig. 5.4.7 showing the external connections. It includes the additional sound devices mentioned in Sec. 3.4.1.

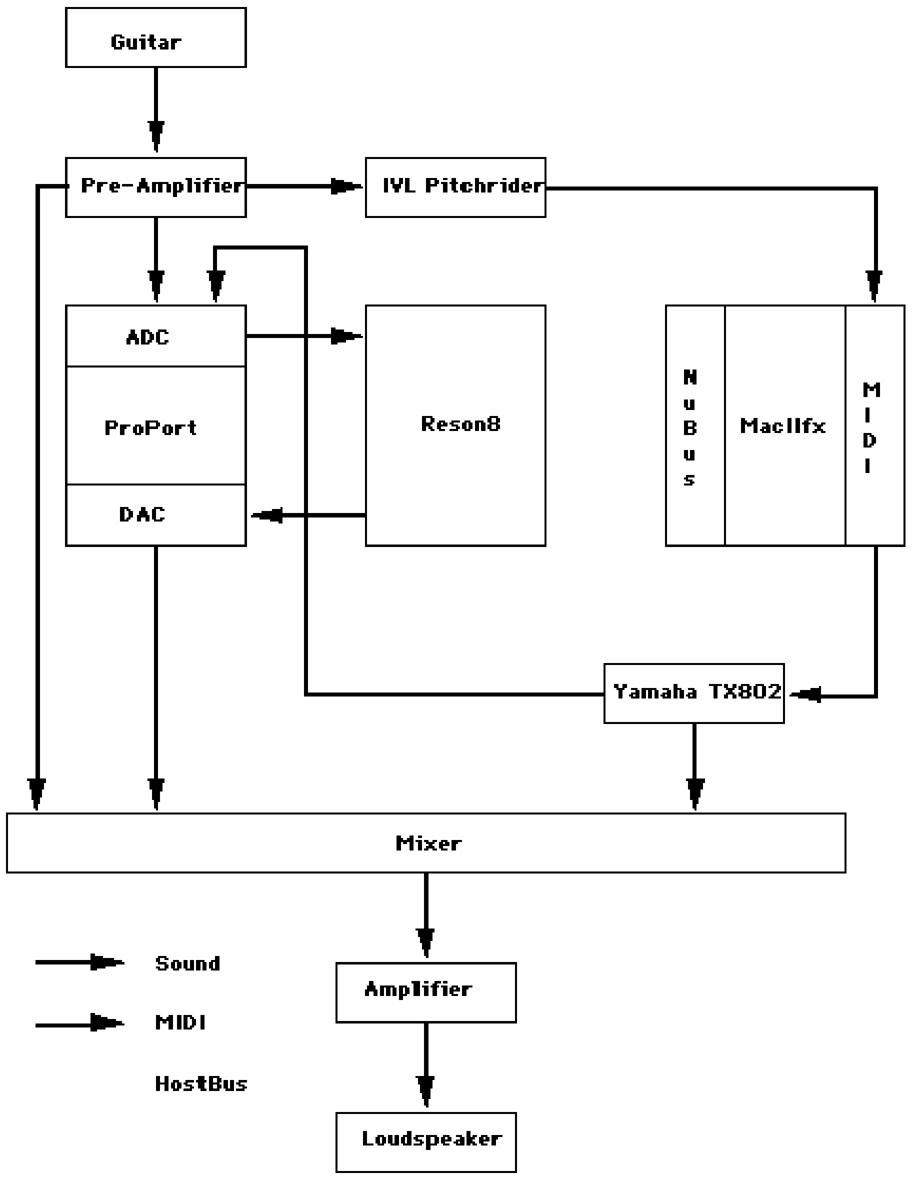


Figure 6.2.4: Hardware setup of the implementation environment

6.3 Multi DSP Issues

6.3.1 Distribution of the Functional Blocks

Initially, there was no determination as to how many processors should be used for the implementation. If one DSP was enough to realize the algorithms, there would be no reason to use the other ones. However, as it turned out, the code grew to an extent where the internal program memory could not accommodate them any longer. Limited processing power became a problem, too.

The question of how to distribute the code among how many DSPs was answered by the logical choice of assigning each main functional block to one processor. Furthermore, it was decided that the pre-processor should be located in the move engine since it was the only processor with enough external memory to host the various sample buffers. Accordingly DSP1 was assigned to the state machine, which did not need external memory apart from the Dual-Port-RAM for communication with the move engine. DSP2 hosted the synthesizer algorithms, which had to live with the restrictions of 1k external RAM. This mainly limited the lowest possible frequency to about 32Hz[32].

Finally, the host computer served as an interface to MIDI via MAX, while also providing a user interface, which included signal monitoring and parameter control. Moreover, some of the arithmetic functionality was implemented on the host, *e.g.* operations that require a lot of instruction cycles on DSP, like divisions.

6.3.2 Inter-DSP Communication

As indicated in Sec. 6.2.1 the DSPs of the Reson8 are interconnected with Dual-Ported-RAM, a type of memory that has dual access capabilities in providing two address and data busses. This can be compared to the notion of local (protected) and shared memory which is an issue in topics like operating systems and multi-tasking; there we deal with multiple processes instead of processors. Both cases share the problem of access control, where care must be taken to avoid simultaneous writing to the same memory location.

However, it is important to point out that for the Reson8 the local memory is physically local whereas in a single-processor multi-process system the operating

[32] 1k / 32kHz = 31.25 ms

system must protect the memory space defined as local. However, it is much more difficult to negotiate the shared memory in a DSP system as there is often no OS functionality. For this application the problem was mainly met by synchronizing the slave DSPs to the master DSP with interrupts. This is shown in the following code examples.

```
CheckDSPs
1    move   y:error,b
2    move   y:Mbox1,a          ; check if mailbox has been cleared
3    tst    a
4    jeq    _OK1                    ; if not write to error
5    move   #>DSP1frameerror,x0
6    or     x0,b
_OK1
7    move   b,y:error


Talk2DSPs
8    move   x:Rsignal,x0
9    move   x0,y:Rsignal1
10   move   y:RsignalCFLP,x0
11   move   x0,y:RsignalCFLP1
12   move   y:AvgC,x0
13   move   x0,y:AvgC1
14   move   y:AvgO,x0
15   move   x0,y:AvgO1
16   move   y:EO,x0
17   move   x0,y:EO1
18   move   y:status1,x0


IntDSPs
19   move   #>1,a              ; write to mailbox -> create
                               ; external interrupt for DSPs
20   move   a,y:Mbox1
```

In lines `8` to `18` the current signal values of the Pre-Processor are written to the Dual-Port RAM in order to provide the State Machine in DSP1 with the necessary data to make its decisions. Then, in line `19/20` the master DSP writes an (arbitrary) value to the memory location $8fff which is the top address of the memory space defined for communication with DSP1. A write access to this address is decoded by the hardware to create an external interrupt at the /IRQB pin of DSP1 which responds as in the following code excerpt.

```
interrupt_vectors
1    org   p:irqb
2    jsr   new_data

     ...

new_data
3    movep y:Mbox,y:test4    ; read to clear interrupt

     ...

ClearFlag
4    clr   a
5    move  a,y:Mbox          ; tell move engine that we're done
6    rti                     ; return from interrupt routine
```

Line `1 & 2` define where to jump upon an external interrupt by placing a 'jump subroutine' command at the program memory location that is executed by the DSP if such an interrupt occurs. In this example, the program counter is changed to `new_data`, where the data memory location $ffff is read to clear the interrupt request. Note that this address points to the same physical address as location $8fff for the move engine, because memory mapping differs for the slave and master DSPs as shown in Fig. 6.3.1. At the end of the interrupt routine, this memory location is cleared to tell the move engine that the computation has been completed; this gives the flag dual functionality.

However, if it has not been cleared in time, the master DSP will report a synchronization error as shown in lines `1` to `7` of the first code example. There, an error code is written to a memory location that has been defined as a display variable and thus can be observed from the host. It is remarked that suspending further operations until the flag gets cleared does not make sense in this real-time application as we would inevitably lose samples. This would result in audible clicks of the audio output or wrong results of the analysis. In applications that do not require real-time behaviour this is not critical as the user only has to wait a few more milliseconds before the result is presented on the screen. In this project, however, synchronization errors are crucial.

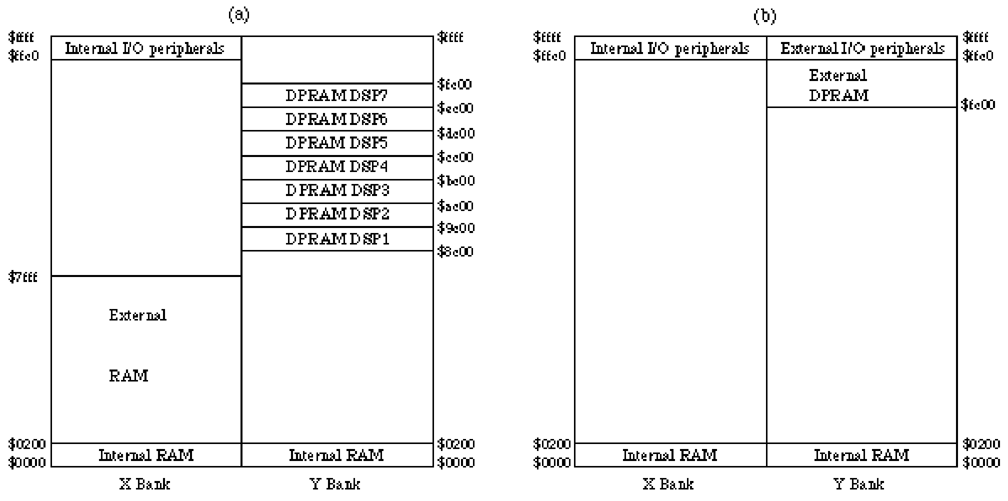


Figure 6.3.1: Memory mapping of (a) move engine and (b) slave DSPs

6.3.3 DSP-Host Communication

Communication with the host is accomplished via the Macintosh NuBus interface and the host port of the DSPs, where sending a command from MAX causes a 'host command interrupt' in the addressed DSP. This usually initiates a fast interrupt routine, *i.e.* one that does not require a jump to a subroutine but gets completed within a single instructions [26]. This is enough to transfer the new datum to the selected memory location:

```
org    p:hostiv1          ; host interrupt 1
movep  x:M_HRX,r0         ; start address into r0
org    p:hostiv3          ; host interrupt 3
movep  x:M_HRX,x:(r0)+    ; value into memory location indexed
                          ;  by r0. Autoincrement to speed up
                          ;  block transfers.
```

With this schemes, parameter updates are completed very quickly.

For the opposite direction, data coming from the DSP has to be polled by the host. Although DSP to host interrupts are supported by the hardware, there is no corresponding functionality in the software implementation. This somehow limits the responsiveness on the host side as the jitter of MAX's scheduler is up to 5 milliseconds.

6.4 Optimization

6.4.1 Efficiency Issues

Although in general efficiency should not be the main factor of software development anymore, this is not entirely true for DSP applications. Basically there are two problems to consider. One is the limited amount of program memory space, which in the case of the DSP56001 is not more than 512 words. Of course, it is possible to add external program memory to the processor. However, this cannot be done not without a speed penalty as the program memory access would have to share the address and data busses with the X and Y memory. This is due to the design limitations of the DSP56001 as separate memory interfaces would require at least another 80 additional pins[33].

The other problem is limited processing power. Although affordable DSPs as well as general purpose processors can perform up to a few 10 million instructions per second this is still not fast enough to be wasted with un-optimized code in certain applications. A simple example is given to illustrate this fact:

The DSP56001 used in the Reson-8 is clocked at 27MHz delivering 13.5 MIPS. Dividing by a sampling rate of 32kHz yields approximately 420 instructions per sampling frame which reduces to 210 instructions for stereo signals. A standard application is a stereo 31-band graphic equalizer using a 4-pole filter for each band which sums up to 124 poles per channel. If it is possible to implement one pole in one instruction we are well within our constraints, however, with two instructions per pole this cannot be achieved. In fact, it is possible to realize one pole with just one instruction, but only if the special commands of the DSP56001 are used, most of which were developed to speed up the implementation of filters and fast fourier transformations.

6.4.2 Parallel Processes

The most important code optimization technique is the use of combined multiplication and accumulation, which is the core operation of filter implementations as well as many others, such as fast Fourier transformations. The `mac` instruction realizes this computation and can be executed in a single clock cycle by most of the modern DSPs and RISC processors.

[33] 2 * (16 address lines + 24 data lines), not counting control signals

Further optimization is achieved by using the parallel move capabilities of the DSP56001. While performing the multiply-add instruction in the arithmetic logic unit (ALU), it is possible to load two new operands for the next operation. In the case of a filter these could be the next data sample and the related parameter. Moreover, the pointers to the sample memory and the parameter set can be updated at the same time.

Thus the following commands

```
  DSP assembler           C

  mpy  x0,y0,b            accu += sample * parameter;
  add  b,a
  move x:(r0),x0          sample = sample_buffer[sample_ptr];
  move y:(r4),y0          parameter = parameter_buffer[param_ptr];
  move                    r0,b   sample_ptr += sample_inc;
  move n0,x0
  add  x0,b
  move b,r0
  move                    r4,b   param_ptr += param_inc;
  move n4,x4
  add  x4,b
  move b,r4
```

can be comprised in this single command:

```
    mac   x0,y0,a     x:(r0)+n0,x0      y:(r4)+n4,y0
```

with no equivalence in standard C. However, this only works if there are enough address registers available; then it is not necessary to load and write back the pointers. In addition, this command relies on fast memory, *i.e.* expensive static RAM, which results in a higher system cost if very long sample buffers are involved.

6.4.3 Other Code Optimization Techniques

Some of the other optimization techniques do not exploit the optimized architecture of the DSP56001, but are used to work around some of its limitations. *E.g.* updating an address register by a direct move instruction cannot be completed in a single instruction cycle, which means that one can not rely on a correct pointer in the next instruction, but has to wait for an additional cycle. However, instead of inserting a `nop` instruction, it is more efficient to do something else in the meantime, like in the following example:

```
move  x:ptr1,r0         ; load pointer into address register
nop                     ; no operation to wait for the update
move  x:(r0),x0         ; use the updated address register to
                        ;  load an indexed memory location
move  x:ptr2,r2         ; dito, for another address register
nop
move  x:(r2),y0
```

can be replaced by

```
move  x:ptr1,r0
move  x:ptr2,r2
move  x:(r0),x0
move  x:(r2),y0
```

saving two instructions cycles.

Another techniques is taking advantage of the fact that the DSP56001 is able to perform memory to memory moves for data located in the peripheral memory space, i.e the top 64 memory locations, namely \$ffc0 - \$ffff. For the Reson8 the Dual-Ported-RAM is located around this area viewed from the slave DSPs.[34] Thus, with the `movep` instruction it is possible to copy the contents of a shared variable directly to the protected address space in one instruction cycle.

A very common technique to speed up execution is block processing. Instead of processing samples one by one, a specific number of samples is processed at a time, reducing the need for register updates. This is exemplified in Sec. 6.5.1 with the implementation of the comb filter. However, block processing is closely linked to the problem of latency mentioned in Sec. 4.2.1. Therefore a small block size of 8 is used.

Instruction cycles can be saved in loop constructs by placing memory updates at the beginning of the loop, even before the first result of the loop operation is computed. Synchronization is achieved outside the loop by an additional memory read (line `1`) of the previous (meaningless) sample.

[34] Actually it is aliased across the whole memory space, but for the mentioned efficiency reasons it is addressed as peripheral data

```
1  move  x:-(r6),a          ; to sync with first a,x:(r6)+ get a
                            ;  value out of buffer and write it
                            ;  back in first cycle
2  do    #block,_end
3  mpy   x1,y0,a   a,x:(r6)+ ; multiply ... , save
                            ;  the output from last cycle
4  sub   x0,a      x:(r6),x0 ; get new sample into x0
5  add   x0,a               ;  then add it to accu leave it there,
                            ;  and subtract it in next cycle
6  move  a,y0               ; update
_end

7  move  a,x:(r6)+          ; save last output to input buffer
```

Usually the parallel move command in line `3` (`a,x:(r6)+`) would be placed after line `6` to save the new value in the accumulator to the output buffer. However, an instruction can be saved by performing the update at the beginning (!) of the computation in a parallel move. This parallel operation cannot be done in line `6` because this line already contains a move command that uses the accumulator.

After the loop the last output sample has to be saved with an additional move command, but this is done outside the loop and thus only performed once.

6.5 Examples of Essential Code

Three examples are given which are essential for the functionality of the algorithms. The complete code can be found in B2.1 (Pre-Processor), B2.2 (State Machine) and B2.3 (Synthesizer).

6.5.1 The Comb Filter Delay Line Implementation

```
CombFilter
1  move  x:LINptr,r6           ; get pointer to input buffer
2  move  m7,m6                 ; r6 and r5 operate on same buffer as
3  move  m7,m5                 ;  r7, i.e. the input buffer
4  move  x:DelayLen,n6         ; get delay length (=length of period)
5  move  y:CFptr,r2            ; get pointer, modifier and increment
6  move  #(CFbuffersize-1),m2  ;  of comb filter buffer
7  move  #block,n2             ;  (=output buffer)
8  lua   (r6)-n6,r5            ; use delay length to offset r5
9  lua   (r2)+n2,r2            ; update CFptr before it gets modified
10 move  x:(r6)+,x0            ; get first new sample (interweaved)
11 move  r2,y:CFptr            ;  (cont. of line 9)
12 move  x:(r5)+,a             ; get first delayed sample

13 do    #block,_end
14 sub   x0,a    x:(r6)+,x0    ; subtract new sample from delayed
                               ;  sample, get next new sample
```

```
15 asr   a       x:(r5)+,y0    ; avoid clipping, get next delayed
                               ;  sample (detour via y0)
16 tfr   y0,a    a,x:(r2)+     ; move delayed sample into 'a', save
                               ;  previous 'a' to comb filter buffer
_end
```

This code fragment, which implements the core algorithm of the thesis, consists of two parts. In line `1` to `12` the registers are initialized, before the actual delay line algorithm is executed in lines `13` to `16`. In the following paragraphs, a few considerations concerning the DSP56001 specific implementation are presented.

The above program excerpt shows how much overhead is often needed to set up the registers, which depends somewhat on the size of the application. In small programs the address registers can sometimes be dedicated to a single task, so there is no need for loading and saving pointers. In this example the ratio between initialization and execution is at a disproportion of 4:1. This gave rise to block processing as mentioned in Sec. 6.4.3, which in this case changed the ratio to about 1:2[35] and the total instruction count from

$15 * 8 = \underline{120}$[36]

to

$12 + 4 * 8 = \underline{44}$.

In line `15` and `16`, the next delayed sample is loaded into the accumulator in a detour via the `y0` register as this is the only way to implement the algorithm in three instruction cycles. Moving it directly into the accumulator in line `16` like with

```
15   asr   a
16   tfr   x:(r5)+,a         a,x:(r2)+
```

is not possible, as parallel moves are only allowed for different types of memory space, *i.e.* moving one value to/from X memory, the other to/from Y memory. Likewise the following approach does not work, as the accumulator has to be saved to the comb filter buffer first, before loading a new value.

```
15   asr   a                 x:(r5)+,a
16   move  a,x:(r2)+
```

[35] A block size of 8 is assumed for the calculation

[36] 15 instructions instead of 16 as the `do` instruction would be obsolete if no block processing was used

The only other option is introducing an additional instruction, slowing down the execution:

```
15   asr   a
16   move  a,x:(r2)+
17   move  x:(r5)+,a
```

Finally, a few comments on the initialization part are presented. Line `4` and `8` show how the control variable `x:DelayLen` is used to offset the pointer to the delayed sample memory location, in relation to the pointer to the most recent sample. This is accomplished by using the offset register `n6` and performing the register update, *i.e.* a subtraction, in the Address Generation Unit of the DSP56001 with the `lua` command.

With line `9` and `11`, the pointer to the comb filter buffer, `y:CFptr`, is advanced and updated before it is used the first time for indexing samples in line `16`. This is done to ensure that all following routines that work on the comb filtered sound can rely on `y:CFptr` to point to the beginning of the recent block. Lines `9` to `11` are also an example of the interweaving of instructions mentioned in Sec. 6.4.3.

The result of the implementation can be heard in sound example B4.1.

6.5.2 State Machine

The main code for the state machine in DSP1 is basically programmed like the `case` construct of C. Depending on the current status, a specific routine is executed that checks certain conditions like flags and counters or compares energy levels supplied by the pre-processor to thresholds.

The first code example shows how the status variable is used as an offset to advance the program counter to the jump instruction that specifies the entry address of the related code. It also exemplifies how a program memory location can be loaded into an address register rather than the usual data memory pointers.

```
CaseOfStatus
     tfr   y1,a  #>base,r1   ; y1 is reserved for the status
     asl   a                 ; status is multiplied by 2 because the
     move  a,n1              ;  jmp instruction takes 2 program
     nop                     ;  memory locations
     jmp   (r1+n1)           ; m1 assumed to be set to linear
                             ;  base address mode
     jmp   WaitForSignal
     jmp   WaitForMax
     jmp   WaitForPitch
     jmp   WaitForPeriod
     jmp   WaitForTouch
```

```
        jmp   WaitForLeave
        jmp   HoldExcited
        jmp   Hold
        jmp   EndTrigger
```

The second is an excerpt from one of the routines that check certain conditions in the output of the pre-processor. Here, it is the routine that decides whether a string has been touched, by comparing the comb filtered signal to the adaptive threshold level introduced in Sec. 5.4.2

```
    if (signal / average > threshold) change status to 'touched'
```

However, as divisions require too many instruction cycles in the DSP56001, this was changed to

```
    if (signal * (1/threshold) > average) change status to 'touched'
```

Now, the DSP has to execute a simple multiplication, while the division `(1/threshold)` can be performed at the host, which updates the reciprocal instead of the threshold itself. This has the additional advantage that the threshold value is always below 1, which makes DSP arithmetic easier.

```
WaitForTouch
    ...
1    movep y:RsignalCFLP,x0      ; get filtered signal
2    move  x:TouchThreshold,y0   ; get (reciprocal) touch threshold
3    mpy   x0,y0,a               ; scale signal
4    movep y:AvgC,x0             ; get average
5    cmp   x0,a                  ; compare signal to average
6    jlt   EndTrigger            ; if a<x0 (scaled signal<average)
                                 ;  jump (don't trigger)
7    move  #>touched,y1          ; status = touched
    ...
8    jmp   EndTrigger            ; jump to end of state machine
```

Lines `1`, `2` & `4` load the relevant variables; the signal is scaled in line `3`. The comparison in line `5` sets the negative bit in the status register if the scaled signal is below threshold. This causes a break to the end of the status machine by line `6` and the previous status is retained. In the case that the scaled signal is above threshold, the status is changed to 'touched' in line `7` before the jump to `EndTrigger` is executed

6.5.3 Moving Sum Average

The computation of the pitch-synchronous average mentioned in Sec. 5.4.2 asks for the summation of P samples, with P representing the length of a period at the specified sampling rate (5.4.6). For a fundamental frequency of *e.g.* 80Hz at f_s = 32kHz, 400 additions are required each sampling interval causing a heavy drain on the processing power. This calls for a more efficient scheme.

As no coefficients are involved in (5.4.6), it is possible to implement the summation by adding the latest sample (line `10`), while subtracting the one that occurred P samples before (`16`); thus only operations are needed. This algorithm is often labeled 'moving sum average' (MSA) [41] which in this case became a pitch-synchronous MSA, *i.e.* PSMSA.

One problem that has to be taken care of is proper synchronization to changing lengths of period (checked in `2-5`). In this implementation it is met by setting the sum to zero (`8`) and a counter to the new length (`7`), then decrementing the latter at each sampling interval (`18-20`) and not subtracting from the accumulator until the counter is zero (checked in `12-14`).

```
    move   x:(r5)+,a     ; get new sample, advance pointer
    move   y:(r1),x0     ; get current summation length
    move   x:(r3),b      ; get possibly updated length
    cmp    x0,b          ; check if summation length has changed
    jeq    _noupdate     ; jump if they're equal
    move   b,y:(r1)      ;  otherwise update energy summation length
    move   b,y:(r2)      ;  and energy summation counter
    move   #>$0,x1       ;  and clear the sum
_noupdate
    move   y:(r1),n5     ; get offset for old sample
    add    x1,a          ; add latest sum to new sample
    lua    (r5)-n5,r6    ; update pointer to old sample value
    move   y:(r2),b      ; get energy summation counter
    tst    b             ; jump to nosub if the summation
    jgt    _nosub        ;  counter is >0
                         ;  otherwise move on and subtract the old
                         ;  sample
    move   x:(r6),b      ; get old sample value
    sub    b,a           ; subtract it from energy
    jmp    _nodec
_nosub
    move   #>$1,x0       ; decrement the counter
    sub    x0,b
    move   b,y:(r2)      ; update energy summation counter
_nodec
```

This scheme also motivated the additional 'pitched' state mentioned in Sec. 5.4.3.

A more advanced solution might use the difference between the old and the new summation length to achieve faster synchronization by stopping addition or subtraction for a certain time shorter than the whole period.

6.6 The User Interface

In most computer science application designing a user interface means to provide the user with an interactive computer display where he can type commands or use the mouse to command the program. In a hardware implementation one would equip the box with a set of buttons and dials.

In contrast, this should be avoided in this project. Here the main aspect of interfacing the user, *i.e.* the guitar player, to the computer is to maximize the mapping between his playing and an adequate response of the synthesizer. He should be required to operate any controls as few as possible to be able to concentrate on the guitar itself.

However, some knobs are actually necessary in order to adjust the parameters of the system or call up different setups that are optimized for specific playing styles, like strumming chords vs. playing melody lines. In addition, a graphic interface to display parameters and signal waveforms is needed for development. This is one of the reasons why the MAX-DSP link had been developed at CNMAT and why it was chosen for this project. The MAX patches that constitute the UI functionality can be found in the appendix B3.1-4. One example that shows how the current state of the state machine is queried and used for triggering MIDI events is presented in Fig. 6.6.1

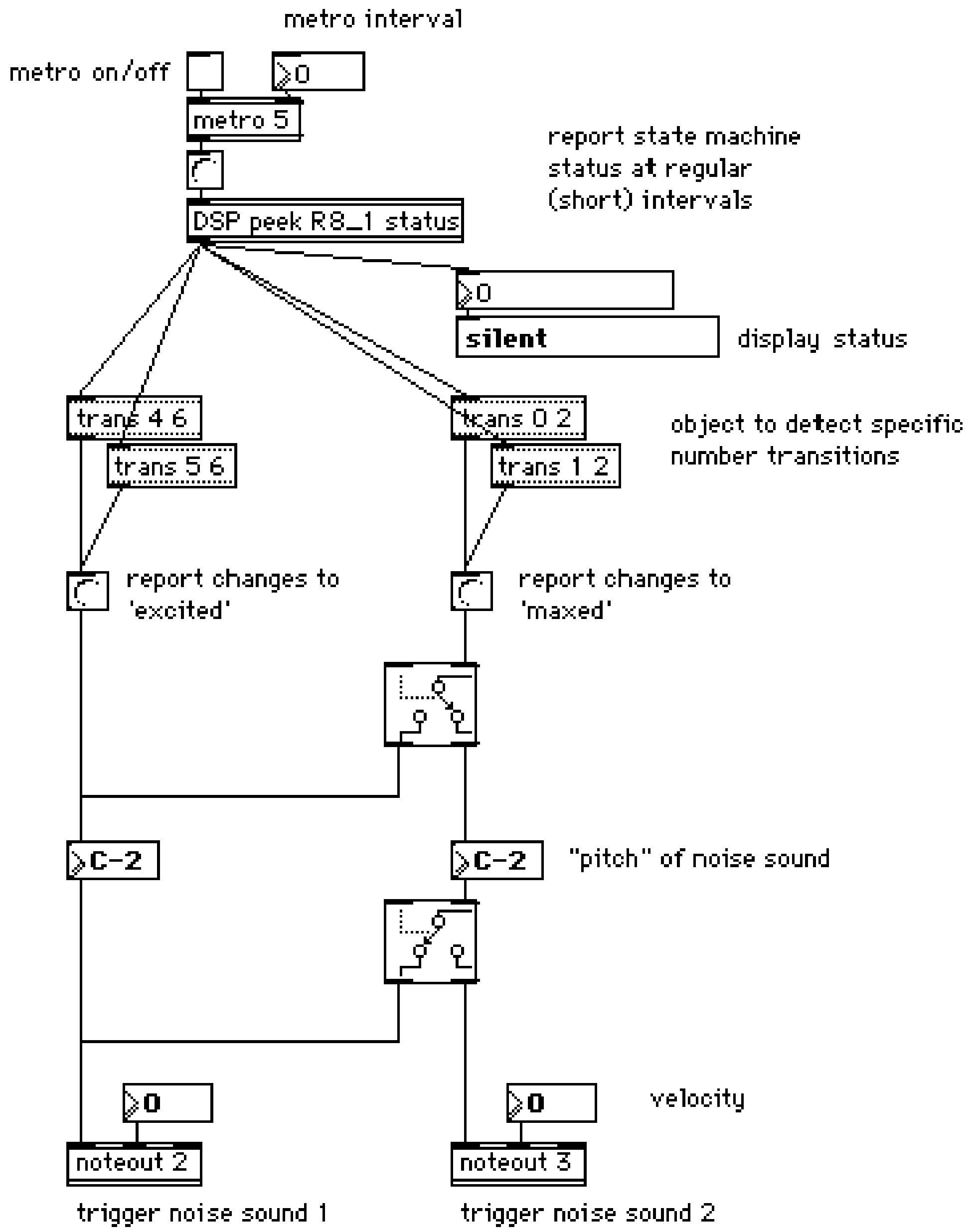


Figure 6.6.1: Querying the state machine from MAX

6.7 Impromptu Modifications of the Synthesis

Listening to the results of the project (sound examples B4.2-4), one can hear a significant improvement in the detection of fast re-plucks. However, one fact remains slightly disturbing, which is that the output resembles too much the input, *i.e.* it sounds like a guitar. This, of course, is caused by the choice for the Karplus Strong Synthesis which is well known for producing guitar-like sounds (Sec. 5.3.2). While this is not a real problem from a purely scientific point of view, it is of course very questionable from a more musical one. Here, the aim is to achieve some kind of transformation as mentioned in Sec. 1.2.1.

Out of this need, extensions to the basic KSS algorithm were taken into consideration. In particular, the approach of P. Cook was evaluated, who had implemented physical models of wind instruments on the NeXT's build-in DSP56001 [4]. Although physical modeling can reach complexity very fast, some of its basic principles are quite simple. As with the KSS's string model, the pitch is often determined by the length of a delay line, and filters are employed for shaping the spectral characteristics. In addition to that, non-linearities can be found in these models that are used to describe *e.g.* the transfer function between the air pressure in the tube and the pressure coming from the musician's mouth.

In the most simple algorithm one such non-linearity is simply placed in the feedback path of the delay line. This was implemented with a lookup table that mapped input to output values according to a function that could be drawn in a MAX 'multislider' object. The relevant code can be found at the end of B2.3; the MAX patch is shown in Fig. 6.7.1. In addition, experiments with polynomials were considered, as proposed in [5]. In the sound example B4.5 one can hear the effect of a squaring function which resembles the attack of a trombone. However, this example also shows that non-linearities tend to be unstable and that the algorithm is still far away from the sonic quality of a trombone. This is where physical modeling starts to grow more complex.

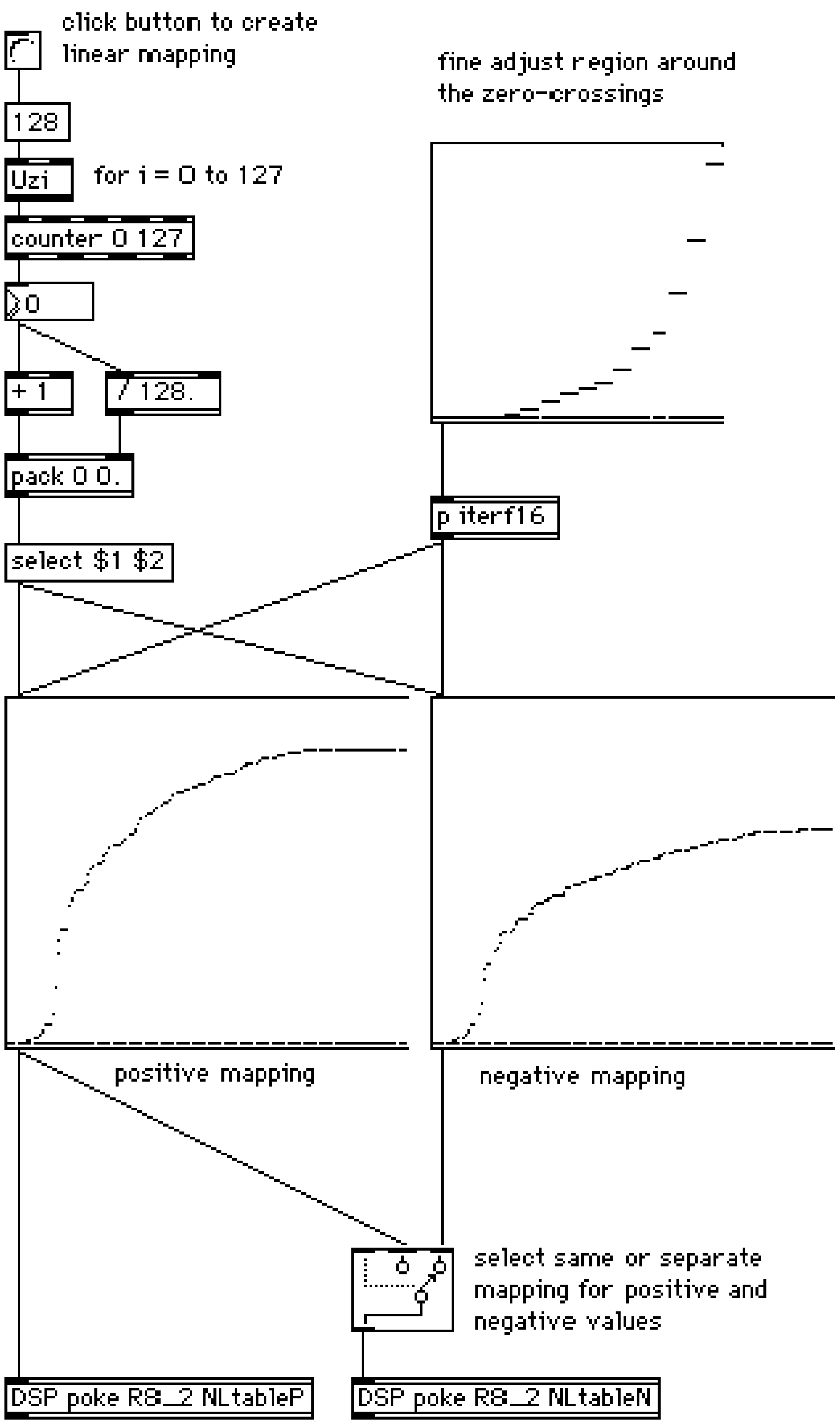


Figure 6.7.1: Editing non-linear tables in MAX

7 Results

Comparing the results of the project to the requirements of Sec. 3 it can be said that the target results have been met in essence, although not always with the suggested methods. In particular the effort taken towards the understanding of advanced digital signal processing did not end up in a fruitful application of wavelets. However, it inspired the use of a pitch-synchronous scheme and the related comb filters, which showed significant improvement in detecting fast re-plucks.

In relation to the combination of analysis and synthesis a link was established on the sound level by using a processed input signal to excite a Karplus Strong Synthesis algorithm. On a higher level a parameter was derived that represents the length of a pluck action, which is closely related to the articulation of guitar playing.

Towards implementation Matlab simplified prototyping very much. The intermediary C implementation was skipped to keep up with the schedule, which proved to be a good decision as the DSP implementation turned out to be surprisingly smooth. The combination of Matlab for prototyping and DSP assembler for implementation was found to make good sense for developing real-time applications.

The realization also benefited from the use of MAX as a graphical front end. It was very helpful for the visualization of parameters and signals and shortened development time significantly by providing the graphic constructs and a real-time environment on the event level.

Although the Reson-8 proved to be a viable implementation platform and a good replacement for the Infinity Box, it was a big disappointment that Gibson Guitar discontinued the development of this device.

8 Future Perspectives

8.1 Algorithms

Concerning the analysis algorithms and synthesis links, it is believed that more research is necessary in regard to real-time applications. While many papers can be found that deal with sound analysis in general and transient detection and classification in particular, only a few care about a real-time implementation and almost none approaches the specific requirements of low-latency musical instruments. Even among the computer music community much of the analysis is done off-line (with exceptions like score-following (Sec. 2.1)).

Guitar sound signals in particular carry much of their information in the rather short period of time placed around the attacks. While it was possible to derive a new parameter that represents the length of a pluck, it is the opinion of the author that more information can be retrieved from these events. It is agreed upon among guitar players that the subtle timing of left and right hand action is of great significance for shaping the guitar sound. In the frequency domain it might be interesting to map the spectral envelope of a pluck sound to the synthesized or sampled excitation.

In regard to Sec. 4.2.2 more elaborate schemes have to be developed that do not wait until precise information has been gathered before setting things in action. Rather they should use what is available at any moment to start loosely specified events that are updated incrementally.

Finally, in a similar way non-linear functions have enriched the sonic quality of physical models, they might be useful in sound analysis. This field is currently under research.[37]

[37] URL: http://physics.www.media.mit.edu/~metois/research.html

8.2 Implementation Platforms

Talking about implementation platforms for real-time sound analysis and synthesis many people feel that DSPs have become obsolete or will be so in the near future.[38] In terms of sheer processing power this is true in that modern General Purpose CPUs (GPCPU) lead the race for more multiply-accumulate operations per seconds, which used to be a strong domain of DSPs. In addition, GPCPUs have better stack support, branch prediction, wider address space and on-chip cache among many advanced features. However, they achieve this enhanced performance with large die sizes and high clock rates that cause a high power dissipation. Also, this enlarged functionality comes at a much higher price.

Maybe more important though is the environment in which the processors are embedded. GPCPUs are used in personal computers and workstations that support a wide range of applications. This is managed by a complex operating system that arbitrates and schedules all the system resources. Due to the generality and distribution of PCs a lot of tools are available to maximize the productivity. Code can be written on a very high level to make maintenance and portability easy while still being efficient enough for most applications due to the high standard of compilers. However, a big drawback in terms of real-time response is that all this high level functionality has a significant influence on the system's latency as mentioned in Sec. 4.2.1.

In contrast, DSPs are much more dedicated to special tasks, *i.e.* those related to signal processing. They are mostly used in embedded systems where usually no sophisticated (if at all) operating system is worth to be developed. DSPs are still programmed in Assembler (although C compilers are available and preferred by some), which is where they gain a performance advantage over GPCPUs.

As mentioned in Sec. 7 the combination of high level tools like Matlab with DSP assembler for real-time implementation was considered a good solution. In addition, coding could be simplified by using highly optimized modules/macros from a higher level language, instead of a more general compiler, because most of the DSP applications are based on the same basic filter and delay line structures.

On a more practical level embedded systems are necessary for guitar players who cannot carry big workstations on stage that cost a fortune. They need a small box that is optimized for a specific task, easy to setup, reliable and affordable. Here DSPs are still the best solution.[39]

---

[38] URL: http://cnmat.CNMAT.Berkeley.edu/~adrian/engines.html

[39] And ASICs for larger quantities

Bibliography

[1] Barrière, J.-B., Freed, A., Baisnée, P.-F. and Baudot, M.-D. :
A digital signal multiprocessor and its musical applications
Proceedings of the International Computer Music Conference (ICMC), San Francisco, CA, USA, 1989, pp. -

[2] Barrière, J.-B., Potard, Y. and Baisnée, P.-F. :
Models of continuity between synthesis and processing for the elaboration and control of timbre structures
Proceedings of the International Computer Music Conference (ICMC), Vancouver, CA, 1985, pp. 158-162

[3] Baudot, M.-D. and Freed, A. :
Reson8 Programmer's Manual
CNMAT, Berkeley, 1991

[4] Cook, P.R. :
TBone: An interactive waveguide brass instrument synthesis workbench for the NeXT machine
Proceedings of the International Computer Music Conference (ICMC), Montreal, CA, 1991, pp. 297-299

[5] Cook, P.R. :
A meta-wind-instrument physical model, and a meta-controller for real time performance control
Proceedings of the International Computer Music Conference (ICMC), San Jose, CA, USA, 1992, pp. 273-276

[6] Daubechies, I. :
Ten lectures on wavelets
Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1992

[7] Depalle, P., Garcia, G. and Rodet, X. :
Tracking of partials for additive sound synthesis using hidden Markov models
ICASSP-93. 1993 IEEE International Conference on Acoustics, Speech, and Signal Processing (Cat. No.92CH3252-4) Proceedings of ICASSP '93, Minneapolis, MN, USA, 1993, pp. 225-8 vol.1

[8] Dutilleux, P. :
Vers la Machine à Sculpter le Son
Modification en Temps Réel des Caractéristiques Fréquentielles et Temporelles des Sons
Docteur de l'Université d'Aix-Marseille II
Université d'Aix-Marseille II, Institut de Mécanique de Marseille, 1991, N° 2079187

[9] Evangelista, G. :
Pitch-synchronous wavelet representations of speech and music signals
IEEE Transactions on Signal Processing, Vol. 41, No. 12 1993, pp. 3313-30

[10] Evangelista, G. :
Comb and multiplexed wavelet transforms and their applications to signal processing
IEEE Transactions on Signal Processing, Vol. 42, No. 2 1994, pp. 292-303

[11] Freed, A., Lee, M., Baudot, M.-D. and Gordon, K. :
MaxDsp 56k DSP development tools for Motorola DSP56001 development on the Macintosh Computer - User Manual
CNMAT, 1990

[12] Giraudon, P. :
A score follower for MAX-ISPW
Report
Zentrum für Kunst und Medientechnologie (ZKM), 1994

[13] Jaffe, D.A. and Smith, J.O. :
Extensions of the Karplus-Strong plucked-string algorithm
Computer Music Journal, Vol. 7, No. 2 1983, pp. 56-69

[14] Kammeyer, K.D. and Kroschel, K. :
Digitale Signalverarbeitung
Teubner, Stuttgart, 1992

[15] Karjalainen, M. and Laine, U.K. :
A model for real-time sound synthesis of guitar on a floating-point signal processor
ICASSP 91: 1991 International Conference on Acoustics, Speech and Signal Processing (Cat. No.91CH2977-7), Toronto, Ont., Canada, 1991, pp. 3653-6 vol.5

[16] Karplus, K. and Strong, A. :
Digital synthesis of plucked-string and drum timbres
Computer Music Journal, Vol. 7, No. 2 1983, pp. 43-55

[17] Keeler, J.S. :
Piecewise-periodic analysis of almost-periodic sounds and musical transients
IEEE Transactions on Audio and Electroacoustics, Vol. 20, No. 5 1972, pp. 338-44

[18] Laakso, T.I., Valimaki, V., Karjalainen, M. and Laine, U.K. :
Splitting the unit delay [FIR/all pass filters design]
IEEE Signal Processing Magazine, Vol. 13, No. 1 1996, pp. 30-60

[19] Learned, R.E., Karl, W.C. and Willsky, A.S. :
Wavelet packet based transient signal classification
Proceedings of the IEEE-SP International Symposium Time-Frequency and Time-Scale Analysis (Cat.No.92TH0478-8), Victoria, BC, Canada, 1992, pp. 109-12

[20] Learned, R.E. and Willsky, A.S. :
A wavelet packet approach to transient signal classification
Applied and Computational Harmonic Analysis, Vol. 2, No. 3 1995, pp. 265-78

[21] Lindemann, E., François, D., Starkier, M. and Smith, B. :
The architecture of the IRCAM musical workstation
Computer Music Journal, Vol. 15, No. 3 1991, pp. 41-50

[22] Mathews, M.V., Miller, J.E. and David, E.E.J. :
Pitch-synchronous analysis of voiced sounds
Journal of the Acoustical Society of America, Vol. 33, No. 2 1961, pp. 179-186

[23] McMillen, K. :
ZIPI: origins and motivations
Computer Music Journal, Vol. 18, No. 4 1994, pp. 47-51

[24] McMillen, K., Wessel, D. and Wright, M. :
The ZIPI Music Parameter Description Language
Computer Music Journal, Vol. 18, No. 4 1994, pp. 52-73

[25] Medan, Y. and Yair, E. :
Pitch synchronous spectral analysis scheme for voiced speech
IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 37, No. 9 1989, pp. 1321-1328

[26] Motorola :
DSP56000/DSP56001 Digital Signal Processor
User's Manual
Motorola, 1990

[27] Motorola :
Motorola DSP56000
Macro Assembler Reference Manual
Motorola, 1990

[28] Motorola :
Motorola DSP56000/1 Digital Signal Processor
Simulator Reference Manual
Motorola, 1990

[29] Motorola :
Real time digital signal processing applications with Motrola's DSP56000 family
Prentice Hall, Englewood Cliffs, New Jersey 07632, 1990

[30] Oppenheim, A.V. and Schafer, R.W. :
Digital Signal Processing
Prentice-Hall, New Jersey, 1975

[31] Potard, Y., Baisnée, P.-F. and Barrière, J.-B. :
Experimenting with models of resonance produced by a new technique for the analysis of impulsive sounds
Proceedings of the International Computer Music Conference (ICMC), Den Haag, NL, 1986, pp. 269-274

[32] Puckette, M. :
The Patcher
Proceedings of the International Computer Music Conference (ICMC), Köln, Germany, 1988, pp. -

[33] Puckette, M. and Zicarelli, D. :
MAX - An Interactive Graphic Programming Environment
Opcode Systems, Menlo Park, CA, USA, 1990

[34] Rabiner, L.R. and Gold, B. :
Theory and application of digital signal processing
Prentice-Hall, New Jersey, 1975

[35] Risset, J.-C. :
Timbre analysis by synthesis: representations, imitations, and variants for musical composition
from: Representations on musical signals
edited by: De Poli, G., Piccialli, A. and Roads, C.
MIT Press, Cambridge, MA, USA, 1991

[36] Robinson, K. and Patterson, R.D. :
The Duration Required To Identify the Instrument, the Octave, or the Pitch Chroma of a Musical Note
Music Perception, Vol. No. 1995, pp. 1-15

[37] Rodet, X., Potard, Y. and Barriere, J.-B. :
The CHANT project: from the synthesis of the singing voice to synthesis in general
Computer Music Journal, Vol. 8, No. 3 1984, pp. 15-31

[38] Serra, X. and Smith, J. :
Spectral modeling synthesis: a sound analysis/synthesis system based on a deterministic plus stochastic decomposition
Computer Music Journal, Vol. 14, No. 4 1990, pp. 12-24

[39] Serra, X. and Smith, J.O. :
A system for sound analysis/transformation/synthesis based on a deterministic plus stochastic decomposition
Signal Processing V. Theories and Applications. Proceedings of EUSIPCO-90, Fifth European Signal Processing Conference, Barcelona, Spain, 1990, pp. 1347-50 vol.2

[40] Sullivan, C.R. :
Extending the Karplus-Strong algorithm to synthesize electric guitar timbres with distortion and feedback
Computer Music Journal, Vol. 14, No. 3 1990, pp. 26-37

[41] Unser, M.A. and Aldroubi, A. :
Fast algorithms for running wavelet analyses
Wavelet Applications in Signal and Image Processing II, San Diego, CA, USA, 1994, pp. 308-19

[42] Vaidyanathan, P.P. :
Multirate systems and filter banks
Prentice-Hall, New Jersey, 1993

[43] Wright, M. :
A comparison of MIDI and ZIPI
Computer Music Journal, Vol. 18, No. 4 1994, pp. 86-91

[44] Zelniker, G. and Tayor, F.J. :
Advanced digital signal processing. Theory and applications
Marcel Dekker, New York, NY, USA, 1994

B The Tape

An audio tape accompanies the thesis to illustrate the results of the project as well as the problems that gave rise to it. Throughout B2 and B4 we can hear the original signal on the left channel, the processed or synthesized signal on the right one.

B1 is an excerpt from a recording by a guitar player who has been associated with guitar synthesizers from their very first days - Adrian Belew. It features a lot of synthesized sounds, where none of them has been played from a keyboard.

B2 depicts the main functionality of a standard guitar synthesizer as well as one of its problems, *i.e.* detection problems for fast and softly played notes. The observant listener will also recognize a short delay between the guitar and the synthesizer sound which is in the range of 20 to 30 milliseconds.

B3 shows examples from the prototyping under Matlab that were kept very short to speed up computation. They are played at both the original and a reduced speed to make the events easier to observe. For the same reason they are repeated three times. While the comb filtered re-plucks have a distinct sonic quality, the trigger signal is just a series of clicks where each marks an onset or an ending of a re-pluck.

B4 presents the final results of the real-time implementation on the Reson8. The sound of the finger nail plucking the string can be clearly heard in B4.1b/c. The other examples show the much improved decision quality for the triggering of sounds as well as the guitar like sound of the Karplus Strong Synthesis. In fact, it is hard to tell for the first seconds of B4.3 which signal is the original one, and the listener is encouraged to play with the 'Balance' knob to find out the difference. In B4.4 the XKSS has been excited by a snare sound with a lot of high frequency energy which produces a totally different sonic result, resembling more a cembalo if anything. B4.5 has been added rather for amusement than scientific or musical enlightenment. However, it hints at the possibilities of analysis - synthesis mappings on a low abstraction level.

B1	0:10	**Introduction**
B1.1	0:39	Adrian Belew - excerpt from 'Beach Creatures Dancing like Cranes' from the CD 'Desire Caught by the Tail' (1986)
B2	2:00	**The IVL 'Pitchrider' - a common guitar synthesizer**
B2.1	2:18	IVL following a slow melody
B2.2	2:34	Dropouts at fast re-plucks on a low frequency string
B2.3	3:04	Dropouts at fast+soft re-plucks on a high frequency string
B3	3:33	**Sound Examples from Matlab (each repeated 3 times)**
B3.1a	3:43	Re-plucks
B3.1b	3:58	Re-plucks at half speed
B3.2a	4:16	Comb filtered re-plucks
B3.2b	4:31	Comb filtered re-plucks at half speed
B3.3a	4:45	Comb and lowpass filtered re-plucks
B3.3b	4:58	Comb and lowpass filtered re-plucks at half speed
B3.4a	5:13	Trigger signal
B3.4b	5:24	Trigger signal at a fourth of the speed
B4	5:41	**Reson8 Real-time Implementation**
B4.1a	5:55	Original re-plucks
B4.1b	6:13	Comb filtered re-plucks
B4.1c	6:33	Original and comb filtered re-plucks
B4.2	6:58	Triggered percussion
B4.3	7:36	XKSS excited by comb filtered re-plucks
B4.4	8:12	XKSS excited by triggered percussion
B4.5	9:16	Guitar controlling trombone-like sound
	10:14	End of tape