Real-time Convolution of Two Unknown Signals for Use in a Musical Context

Antoine Henning Bardoz
Lars Eri Myhre

Master of Science in Electronics	
Submission date:	June 2013
Supervisor:	Jan Tro, IET
Co-supervisor:	Tor A. Ramstad, IET
	Sigurd Saue, IM
	Øyvind Brandtsegg, IM

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Abstract

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Electronics and Telecommunications

Master of Science

Cross Convolution of Live Audio Signals for Musical Applications

by Antoine Henning Bardoz
Lars Eri Myhre

This thesis proposes a method for convolution of two real-time audio signals, for use in live performances or post-production. In contrast to traditional convolution techniques, which require a predefined impulse response as one of the input signals, our method allows for convolution of two continuously updated, and unknown, signals, allowing two musicians to shape each other's timbral and temporal contributions.

The aim was to create an effect that sounded like convolution, offered low output delay, as well as giving satisfying feedback to musicians. To achieve this, a hybrid of time- and frequency domain techniques has been used, offering the low output delay associated with the time domain, and the low CPU load characteristic of FFT-based frequency domain processing. To deal with the limitations inherent in convolution, namely that to perform ideal convolution of two unending signals, an infinite amount of memory and processing power are eventually required, transient detection has been applied to segment the signals in a musically relevant way. The transient-assisted segmentation also makes the effect more intuitive for users, as it increases the users' ability to interact rhythmically.

A GUI was developed, and the effect was implemented as a VST plug-in, to allow users to easily apply the effect in DAWs.

The effect was prototyped in Matlab, and later implemented in Csound and C, using the Cabbage framework for the VST.

NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

Sammendrag

Fakultet for informasjonsteknologi, matematikk og elektronikk
Institutt for elektronikk og telekommunikasjon

Master i elektronikk

Krysskonvolusjon av sanntidslydsignaler til musikalske anvendelser

by Antoine Henning Bardoz
Lars Eri Myhre

I denne oppgaven foreslås en fremgangsmåte for konvolusjon av to sanntids lydsignaler, til bruk i live-opptredener eller post-produksjon. I motsetning til tradisjonelle konvolusjonsteknikker, som krever en forhåndsdefinert impulsrespons som ett av inngangssignalene, tillater vår metode konvolusjon av to kontinuerlig oppdaterte, og ukjente, signaler, slik at to musikere kan forme hverandres klanglige og tidsmessige bidrag.

Målet var å skape en effekt som høres ut som konvolusjon, tilbyr lav utgangsforsinkelse, og gir tilfredsstillende tilbakemelding til musikere. For å oppnå dette har en kombinasjon av tids- og frekvensdomeneteknikker blitt brukt. Dette kombinerer lav CPU-belastning, takket være FFT-basert frekvensplanprosessering, med den lave forsinkelsen assosiert med tidsdomenet. For å håndtere begrensningene forbundet med konvolusjon, nemlig at for å utføre ideell konvolusjon av to uendelige signaler, kreves det etter hvert uendelig minne og prosessorkraft, har transientdeteksjon blitt brukt til å segmentere signalene på en musikalsk relevant måte. Segmentering ved hjelp av transienter gjør også effekten mer intuitiv for brukerne ved å øke deres evne til å samhandle rytmisk.

Et grafisk brukergrensesnitt ble utviklet, og effekten ble implementer som en VST plug-in, slik at brukere enkelt kan benytte effekten i DAWer.

Effekten ble prototypet i Matlab, og senere implementert i Csound og C. Cabbage-rammeverket ble benyttet for VST-implementasjonen.

Acknowledgements

We would like to extend a special thanks Sigurd Saue for giving us valuable suggestions and technical insight, without which we would truly have been lost.

In addition we would like to thank Jan Tro for keeping music alive at Gløshaugen, and making this all possible; Øyvind Brandtsegg for technical and artistic insight, as well as tips from a users' perspective; and Tor A. Ramstad for signal processing guidance.

A special thank you goes to Rory Walsh for developing Cabbage and for being extremely helpful through the forum at www.thecabbagefoundation.com. We would also like to thank the Csound community for developing Csound and for quick and crucial help through the Csound developers list.

For wasting our time with mindless babble and keeping us sane, we thank our study hall companions Thomas Christiansen, Niklas Skyberg, Bendik Paulsrud, Jørund Kaarstad Dahl and Rune Svensrud.

For their musical contributions, we thank Thomas Etholm-Kjeldsen, Jakob Eri Myhre and Olaf Mundal.

Antoine would like to thank Lars for truly giving his all during this semester, contributing heavily every step of the way, coming up with important ideas, and remaining motivated, as well as motivating, until the very last minute.

Lars would like to thank Antoine for a partnership which will not be forgotten. His knowledge in signal processing, programming and music has been infectious. His effort has been remarkable.

Contents

List of Figures

Abbreviations

ADC	**A**nalog-to-**D**igital **C**onverter
AM	**A**mplitude **M**odulation
DAW	**D**igital **A**udio **W**orkstation
DFT	**D**iscrete **F**ourier **T**ransform
DSP	**D**igital **S**ignal **P**rocessing
FFT	**F**ast **F**ourier **T**ransform
FT	**F**ourier **T**ransform
FIFO	**F**irst **I**n, **F**irst **O**ut
GUI	**G**raphical **U**ser **I**nterface
IFFT	**I**nverse **F**ast **F**ourier **T**ransform
IR	**I**mpulse **R**esponse
JND	**J**ust **N**oticeable **D**ifference
VST	**V**irtual **S**tudio **T**echnology

Symbols

L_B	Block length	samples
N	Block number in a segment	blocks
N_max	Maximum blocks allowed in a segment	blocks
L_s	segment length ($L_\mathrm{s} = NL_\mathrm{B}$)	samples

Chapter 1

Introduction

I feel the delightful, velvety texture of a flower, and discover its remarkable convolutions; and something of the miracle of Nature is revealed to me.
-Helen Keller

Since the advent of computer music in 1951 [1, p. 55], the use of computers in music has gone from being a curiosity to revolutionizing how nearly all music is being produced. Computers are used for composition, recording, synthesis, mixing and effects processing. Where analog electronic hardware used to dominate, recent advances in Digital Signal Processing (DSP) capabilities have allowed for the replacement of analog processing in most applications. The domain of Digital Audio Effects (DAFx) has grown to include huge amounts of effects, both emulating older hardware and introducing completely new concepts, as well as being academically discussed to a great degree.

At the heart of many of these audio effects, we find convolution. Convolution is a mathematical operation which produces one output signal based on two input signals. One of the input signals is commonly known as an impulse response. Convolution is extensively used in frequency selective filters and reverberation. In these applications, impulse responses are either prerecorded or mathematically derived. Most commonly, these prerecorded impulse responses are the response from some analog equipment, or from a room whose reverberation one wishes to emulate.

In recent years, convolution has been applied using sounds which are not impulse responses, such as recordings of trains or angle grinders[2]. This approach can create timbres which differ substantially from the results of impulse response convolution, but are still musically applicable. In common with traditional convolution techniques, one of the two input signals is *prerecorded*. Work has been done to allow for live convolution between two signals which both change in real-time[3]. It discusses inherent problems with live convolution and proposes that use of transient information from the input signals can alleviate these problems.

This thesis will explore ways to perform a real-time convolution between two audio signals. An algorithm which combines time- and frequency domain signal processing techniques, as well as transient detection, will be developed. The ultimate goal is to create an effect which is musically pleasing. Emphasis will be put on usability for performing musicians, so that the effect can be used in live applications.

Prototyping of the effect will be done in Matlab, but the goal for the final real-time implementation is to implement it as a plug-in[1] for Digital Audio Workstations (DAW).

1.1 Problem Description

The aim is to create a musical effect using an algorithm that can continuously, and reliably convolve two signals together while outputting sounds at a satisfying rate for performing musicians.

Due to the problem's novelty, there are few solutions to go by, and the work will therefore mainly be experimental in nature. At the outset, the following idealized goals are proposed. The effect should:

[1]A plug-in is a computer program that extends the functionality of another computer program.

- Use convolution, and sound like convolution
- Run in real time
- Be intuitively usable for musicians

Because of the properties of convolution, a perfect solution is impossible. These goals are meant as an ideal to be pursued, but never fully reached.

1.2 How to Read This Thesis

Chapter 2 (Theory) describes relevant background theory for the thesis. It also contains a mathematical proof that justifies parts of the final implementation. Chances are that the mathematical proof will be easier to follow after chapter 4 (Algorithm) is read, and while reading section 6.4.1. Chapter 3 (Development Tools) describes the development tools that have been used. Chapter 4 describes the different algorithms that are implemented. It is a pure description of the functionality of the algorithms. Justifications of the different choices that were made during the development, and a discussion on the observations that were done during and after the development, can be found in chapter 6 (Discussion). It may be beneficial for the reader to go through chapter 4 and 6 in parallel.

Chapter 6 also contains a discussion on the computational complexity and on some esthetic considerations. Chapter 5 (Results) contains plots, and details on the audible results, that are discussed in chapter 6, as well as a presentation of the GUI. The sound files are located in the digital appendix attached to the thesis. In chapter 7, some ideas for future work are suggested. The conclusion of the thesis can be found in chapter 8. The appendices are mainly Matlab, Csound and C code, with one block diagram of the transient analysis. The code is also found in the digital appendix. On page 142, there is an index of terms which might help the reader.

If it is desirable to only learn about the final algorithm, section 4.1 (Preliminary Algorithm) and section 5.1 (Discussion of Preliminary Algorithm) can be omitted. In addition, the process handling algorithms described and discussed in sections 4.4.2, 4.4.3, 6.4.2 and 6.4.3 were not used, and are not necessary to understand the final algorithm.

For readers who are just interested in using the effect, reading section 5.5 should be sufficient.

Chapter 2

Theory

2.1 Convolution

Convolution was likely introduced in the middle of the 1700's by Jean-le-Rond D'Alembert to derive Taylor's expansion theorem. It was later, in 1822, used by Jean Baptiste Joseph Fourier in his derivation of the Fourier series, an early example of its relation to the frequency domain[4]. In Digital Signal Processing, discrete convolution holds a central position because of its applications for linear time-invariant (LTI) systems. Any LTI system can be completely mathematically described by its impulse response, and convolution of a signal with this impulse response is equivalent with sending the signal through the system[5, p. 69].

In this section we define discrete convolution, and explain its relationship with the frequency domain through the convolution theorem.

2.1.1 Time Domain

Discrete convolution of two signals, $x_1(n)$ and $x_2(n)$, is defined as

$$y(n) = \sum_{k=-\infty}^{\infty} x_1(k)x_2(n-k). \tag{2.1}$$

If we define the length of $x_1(n)$ as L_{x_1}, and the length of $x_2(n)$ as L_{x_2}, the length of $y(n)$ is

$$L_y = L_{x_1} + L_{x_2} - 1. \tag{2.2}$$

2.1.2 The (Circular) Convolution Theorem

The convolution theorem can be stated as follows in the continuous time domain:

$$\mathcal{F}\{x_1(t) * x_2(t)\} = \mathcal{F}\{x_1(t)\}\mathcal{F}\{x_2(t)\} = X_1(f)X_2(f). \tag{2.3}$$

The Fourier transform of a convolution in the time domain is equivalent to point-wise multiplication in the frequency domain.[6, p. 523]

However, because of the periodicity of the DFT, one must add an additional constraint in the discrete time domain, namely that the convolution is circular.

If

$$x_1(n) \overset{\text{DFT}}{\underset{N}{\longleftrightarrow}} X_1(k)$$

and

$$x_2(n) \overset{\text{DFT}}{\underset{N}{\longleftrightarrow}} X_2(k),$$

then

$$x_1 \textcircled{\scriptsize N} x_2(n) \overset{\text{DFT}}{\underset{N}{\longleftrightarrow}} X_1(k)X_2(k), \tag{2.4}$$

where $\overset{\text{DFT}}{\underset{N}{\longleftrightarrow}}$ denotes an N-point DFT, and $\textcircled{\scriptsize N}$ denotes circular convolution. This is known as the circular convolution theorem[5, p. 476].

Circular convolution entails that once an impulse response reaches the end of a signal, it will wrap around to the beginning. A consequence is that in order to perform a convolution by way of the frequency domain, without pollution from the wrapping, one must pad the signals with at least $\min(L_{x_1}, L_{x_2}) - 1$ zeros[7].

2.2 The Fast Fourier Transform and Frequency Domain Multiplication

The Fast Fourier Transform is an efficient way of calculating DFTs. It was popularized in 1965[8]. While it is possible to create FFT algorithms for any block size, the most common algorithm is the radix-2 FFT, which is the one that was used in this thesis. A derivation of the algorithm is beyond the scope of this thesis, and this section will only deal with the computational benefits of using it for convolution.

As stated in section 2.1.2, the Fourier transformation of a time domain convolution is equivalent to a pointwise multiplication in the frequency domain. This property can be exploited to perform efficient calculations of convolutions by way of the FFT.

Time domain convolution of a signal of length n with an impulse response of length k requires $O(kn)$ multiplications and additions, while frequency domain multiplication simply requires $k + n$ complex multiplications.

The algorithm developed in this thesis assumes that both the signal and impulse response (really signal 1 and signal 2) are the same length, i.e. $k = n$, and henceforth k is replaced by n (see section 4.2).

Taking into account the zero padding mentioned in section 2.1.2, one must double the length of the signals before the transformation occurs. Still, even considering the time complexity of computing the radix-2 FFT and IFFT, both of which are $O(n \log n)$[5, p. 519-526], one ends up with a total complexity of $4n + 2n \log 2n$, which is $O(n \log n)$, a far more computationally efficient algorithm than the $O(n^2)$ time domain convolution. The trade-off is that there is an inherent delay of n samples, as the buffers must be filled before an FFT may be performed.

2.3 Theoretical Foundation for Real-Time Blockwise Convolution

Our final algorithm is based on blockwise convolution. We claim that it is mathematically equivalent with regular convolution, may be performed in real time with an output delay of no more than the block length, and that convolution of two segments may start, and give output, before the entirety of the segments are available (i.e. buffered into memory). We also claim that early input blocks may be discarded from memory before the convolution has been completed, providing that the conceptually infinite input signals are somehow divided into segments. We have developed the following mathematical proofs of these claims.

Proposition. Blockwise convolution is mathematically equivalent with convolution, and we may partition the input into any number of blocks.

Proof. We begin by proving this for N = 2. Let $L = 2l$, where $l \in \mathbb{Z}$, and let

$$x_1(n) = \begin{cases} x_{1,1}(n), & \text{if } n \in [1, \frac{L}{2}] \\ x_{1,2}(n), & \text{if } n \in [\frac{L}{2}+1, L] \\ 0, & \text{otherwise} \end{cases} \tag{2.5}$$

and

$$x_2(n) = \begin{cases} x_{2,1}(n), & \text{if } n \in [1, \frac{L}{2}] \\ x_{2,2}(n), & \text{if } n \in [\frac{L}{2}+1, L] \\ 0, & \text{otherwise} \end{cases} \tag{2.6}$$

(Note that $x_{1,1}$, $x_{1,2}$, etc. are also 0 outside of their defined range). Then,

$$\begin{aligned} y(n) &= x_1 * x_2 \\ &= \sum_{k=-\infty}^{\infty} x_1(k)x_2(n-k) \\ &= \sum_{k=1}^{L/2} x_{1,1}(k)x_2(n-k) + \sum_{k=L/2+1}^{L} x_{1,2}(k)x_2(n-k) \\ &= x_{1,1} * x_2 + x_{1,2} * x_2. \end{aligned}$$

Lemma. $f(n) * g(n) = g(n) * f(n)$. Convolution is commutative, so

$$\begin{array}{rcccc}
y(n) & = & x_2 * x_{1,1} & + & x_2 * x_{1,2} \\
 & = & \sum\limits_{k=1}^{L} x_2(k)x_{1,1}(n-k) & + & \sum\limits_{k=1}^{L} x_2(k)x_{1,2}(n-k) \\
 & = & \sum\limits_{k=1}^{L/2} x_{2,1}(k)x_{1,1}(n-k) & + & \sum\limits_{k=L/2+1}^{L} x_{2,2}(k)x_{1,1}(n-k) \\
 & + & \sum\limits_{k=1}^{L/2} x_{2,1}(k)x_{1,2}(n-k) & + & \sum\limits_{k=L/2+1}^{L} x_{2,2}(k)x_{1,2}(n-k) \\
 & = & x_{1,1} * x_{2,1} & + & x_{1,1} * x_{2,2} \\
 & + & x_{1,2} * x_{2,1} & + & x_{1,2} * x_{2,2}.
\end{array} \tag{2.7}$$

We have now shown that the input signals may be partitioned into two blocks, and convolution may be done separately for these blocks. We will now generalize this into N blocks. Let $L = Nl$, where $N, l \in \mathbb{Z}$ and let

$$x_1(n) = \begin{cases} x_{1,1}(n), & \text{if } n \in [1, \frac{1}{N}L] \\ x_{1,2}(n), & \text{if } n \in [\frac{1}{N}L+1, \frac{2}{N}L] \\ \vdots & \vdots \\ x_{1,N-1}(n), & \text{if } n \in [\frac{(N-2)}{N}L+1, \frac{(N-1)}{N}L] \\ x_{1,N}(n), & \text{if } n \in [\frac{(N-1)}{N}L+1, L] \\ 0, & \text{otherwise} \end{cases} \tag{2.8}$$

and

$$x_2(n) = \begin{cases} x_{2,1}(n), & \text{if } n \in [1, \frac{1}{N}L] \\ x_{2,2}(n), & \text{if } n \in [\frac{1}{N}L+1, \frac{2}{N}L] \\ \vdots & \vdots \\ x_{2,N-1}(n), & \text{if } n \in [\frac{(N-2)}{N}L+1, \frac{(N-1)}{N}L] \\ x_{2,N}(n), & \text{if } n \in [\frac{(N-1)}{N}L+1, L] \\ 0, & \text{otherwise} \end{cases} \tag{2.9}$$

(Again $x_{1,1}$, $x_{1,2}$, etc. are also 0 outside of their defined range). We may now partition the convolution into

$$\begin{aligned} y(n) &= \sum_{k=1}^{L/N} x_{1,1}(k)x_2(n-k) + \cdots + \sum_{k=\frac{(N-1)}{N}L+1}^{L} x_{1,N}(k)x_2(n-k) \\ &= x_{1,1} * x_2 + \cdots + x_{1,N} * x_2. \end{aligned}$$

Applying the same commutativity logic used in the N = 2 example, we get

$$\begin{aligned} y(n) &= \sum_{k=1}^{L/N} x_{2,1}(k)x_{1,1}(n-k) + \cdots + \sum_{k=\frac{(N-1)}{N}L+1}^{L} x_{2,N}(k)x_{1,1}(n-k) \\ &\quad \vdots \qquad \ddots \qquad \vdots \\ &+ \sum_{k=1}^{L/N} x_{2,1}(k)x_{1,N}(n-k) + \cdots + \sum_{k=\frac{(N-1)}{N}L+1}^{L} x_{2,N}(k)x_{1,N}(n-k) \\ &= x_{2,1} * x_{1,1} + \cdots + x_{2,N} * x_{1,1} \\ &\quad \vdots \qquad \ddots \qquad \vdots \\ &+ x_{2,1} * x_{1,N} + \cdots + x_{2,N} * x_{1,N}, \end{aligned} \tag{2.10}$$

Q.E.D.

Proposition. Blockwise convolution can: (1.) Be performed in real time, with an output delay of no more than the block size, and provide output before the entire signals are available, and (2.) discard early blocks before the entire convolution has been finished, provided that the signals are finite in length.

Proof. Let x_1 and x_2 be defined as in eq. (2.9).

We will now show that there may be output after only L/N samples have entered the system. Consider

$$x_{1,i}(n) = \begin{cases} \text{values}, & \text{if } n \in [\frac{(i-1)}{N}L+1, \frac{i}{N}L] \\ 0, & \text{otherwise} \end{cases} \tag{2.11}$$

and

$$x_{2,j}(n) = \begin{cases} \text{values}, & \text{if } n \in [\frac{(j-1)}{N}L + 1, \frac{j}{N}L] \\ 0, & \text{otherwise.} \end{cases} \tag{2.12}$$

We wish to find the start- and end points of each convolution result. The result of a convolution has values when

$$(x_{1,i} * x_{2,j})(n) = \begin{cases} \text{values}, & \text{if } n \in [\frac{(i+j-2)}{N}L + 2, \frac{i+j}{N}L] \\ 0, & \text{otherwise.} \end{cases} \tag{2.13}$$

For simplicity, we define the start- and end points of eq. (2.13) as

$$S_{i,j} = S_{j,i} = \frac{(i+j-2)}{N}L + 2 \tag{2.14}$$

and

$$E_{i,j} = E_{j,i} = \frac{i+j}{N}L, \tag{2.15}$$

respectively. This denotes that no samples from $x_{1,i} * x_{2,j}$ are needed before $S_{i,j}$ or after $E_{i,j}$. Note that both eq. (2.14) and (2.15) are strictly growing. We also define output time

$$T_k = \frac{k}{N}L + 1, \tag{2.16}$$

which denotes the time when output block k must be ready.

(1.) For $n = T_1$, we only have a contribution from the first block, $x_{1,1} * x_{2,1}$, since $S_{1,2}, S_{2,1} > T_1$. $x_{1,1}$ and $x_{2,1}$ have fully entered the system when $n = T_1$, and we may output the first L/N samples at this time. The same goes for the second output block, at $n = T_2$, where we can see that $S_{2,3}, S_{3,2} > T_2$. In general we have $S_{k+1,1}, S_{1,k+1} > T_k$, and we therefore do not need contributions from future blocks when $n = T_k$. We have shown that for every output block, we only need blocks that have already been buffered by the time output must be produced. (2.) We have $T_k > E_{i,j}$ when $k > i + j$. If the signals were infinite in length, blocks would have to be kept in memory forever, as $E_{1,\infty}$ never occurs. However, both signals have $N < \infty$ blocks, so at time T_{N+1}, we no longer have any contribution from blocks $x_{1,1}$ and $x_{2,1}$, since $T_{N+1} > E_{1,N}$ and they may be discarded. In general $x_{1,k}$ and $x_{2,k}$ may be discarded at $n = T_{N+k}$.

Q.E.D.

2.4 Transients and Transient Detection

Transients are short intervals of audio signals where the signal evolves quickly and in an unpredictable or nontrivial manner. Percussive sounds from drums or from claps are examples of signals with transients. Transients are also associated with the excitation of strings on string instruments. When a string is plucked, a transient will dominate the signal for a short time interval before the resonant frequency of the string and the body of the instrument takes over. A transient usually lasts for 50 ms [9].

Several transient detection methods exist, as it is used in a wide range of applications, among them note transcription, time-stretching of audio signals, pitch-shifting of audio signals and audio coding. The methods have to take into account that it is not necessarily straightforward to decide whether a portion of a signal is a transient or not. Transients can for instance be classified as weak or strong, depending on the strength of the envelope of the signal. They can also be classified as slow or fast depending on the rate of change of the envelope. The methods also have to decide on a minimum duration between successive transients. The methods used for transient detection do not vary only because of different definitions on what should be regarded as a transient, but also because of the fact that in some applications one deals with pre-recorded signals and in other applications the method is to function in real-time.

One way to do transient detection is to compare the energy of new samples with some threshold which is based on the energy of previous samples. A transient is occurring if an incoming sample has a higher energy than the threshold. With this method one would get an adaptive threshold which is important because musical signals often has a large dynamic range.

2.5 Latency Tolerance for Musicans

When playing an acoustical instrument, there will be some latency associated with the time it takes for the sound waves to travel from the instrument to the ear. If the distance between the ear and the instrument is one meter, this time will roughly be 3 ms if the speed of sound is 340 m/s. This is obviously low enough for musicians to handle, proven by the fact that people have been playing acoustic instruments for a long time, and is thus rarely considered a problem. When using a computer to process the sound from an instrument, the latency will necessarily be larger because it takes time for a signal to be converted from analog to digital and for the computer to do the actual processing. It is therefore, when designing a digital effect, important to keep the latency within the limits of what can be considered tolerable for musicians. If the latency associated with playing an instrument is to high, it would weaken the performers ability to interact rhythmically with other musicians. The just noticeable difference (JND) is the time where a performer just notices a difference when comparing a delayed source with a source without delay. It was found to be between 20 ms and 30 ms in [10][11].

Chapter 3

Development Tools

In this chapter the tools used to develop and explore the algorithms will be described.

3.1 Matlab

Matlab is a high-level programming environment in which signal processing application development can be done quickly compared to development in lower-level languages such as C or C++. As opposed to programs written in C or C++, which are compiled, Matlab programs are interpreted. Thus, programs written in Matlab are easier to run, but often run less efficiently. Matlab has a large library of built-in functions such as an FFT, time-domain convolution, and filter design algorithms, available through Matlab tool boxes. This can simplify and speed up development in a lot of situations. In addition to quick development, Matlab provides the ability of quick and informative analysis of what the developed programs actually do, thanks to its extensive and easy to use plotting capabilities. A lot of the the signal processing courses at NTNU use Matlab as their main tool, and consequently many students and professors are familiar with it. It was therefore chosen to prototype the effect in Matlab. For more information on Matlab, see [12].

3.2 Csound

Csound is a free open-source audio programming environment. Initially developed by Barry Vercoe since 1985[13, p. *xxix*], Csound is continuously beeing extended. It includes a large library of signal processing modules, called *opcodes*, which are usually written in C or C++. An opcode is a basic Csound module that generates or modifies signals. The opcodes can be connected together to form sound effects and virtual instruments that can function in real-time. It is also possible to write new opcodes whenever the existing opcodes are not sufficient. Because of the novelty of the signal processing tasks faced in the live convolution effect, the tools available in Csound were not sufficient for an intuitive implementation. It was deemed necessary to implemented an opcode using C. The final real-time implementation was implemented in Csound using this self made opcode. For more information on Csound, see [13] and [14].

3.3 Cabbage

One of the goals for this thesis was to have the final real-time implementation as a plug-in for DAWs. Plug-ins are programs that enhance or extends the functionality of existing software. For DAWs, many formats exist, such as VST (Virtual Studio Technology), AU (Audio Unit) and LADSPA (Linux Audio Developers Simple Application Programming Interface), each supported by different DAWs. For this thesis, the VST format was chosen, because of it's large range of compatible DAWs, and because both Mac and PC have DAWs which support VSTs. The final real-time version of the effect in this thesis is available as a VST for both Mac and PC. Both versions were made with the help of *Cabbage* which is an audio plug-in framework for Csound made by Rory Walsh. Cabbage makes it possible to easily develop a GUI (Graphical User Interface) which can be connected to parameters in Csound code, and then export the code and its associated GUI to the VST format. For more info on Cabbage, see [15] and [16].

Chapter 4

Algorithm

This chapter describes the final algorithm, as well as the algorithms developed on the way to the final algorithm, in detail. Section 4.1 describes an algorithm that was developed early in the process to gain insight in real-time convolution in general and to identify future problems that might be encountered. Section 4.2 describes an algorithm that is based on a Csound opcode, written by Istvan Varga[17], which provides low latency frequency domain convolution. We extend it by allowing it to convolve two live signals. In section 4.3 we further develop this algorithm so that it may use information about transients in the input signals to vary parameters used in the algorithm. Section 4.4 describes three transient handling methods. These extend the algorithm to allow several processes running in parallel. They differ in the way they handle the parallel processes. The process handling used in the final implementation is described in section 4.4.1.

4.1 Preliminary Algorithm

This section describes the inner workings of the preliminary live convolution algorithm. The implementation was done in Matlab and can be found in appendix B.1.

4.1.1 Short Description

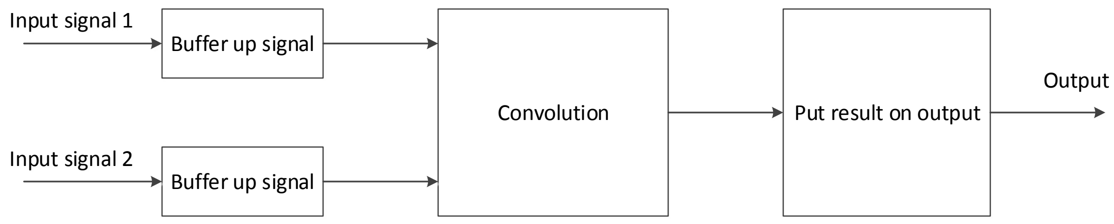


FIGURE 4.1: Block diagram of the preliminary algorithm.

Fig. 4.1 shows an overview of the preliminary algorithm. The input signals are first buffered up in blocks. The blocks can have any size, and block sizes do not have to be the same for the two input signals. After the blocks are filled with samples, the blocks are passed on to the part of the algorithm where the actual convolution is computed. The convolution result is then passed on to a part that puts the result on the output. Because of the unequal block size, the way the convolution result is put on the output is not necessarily trivial, and can be done in several ways, more on this in section 4.1.4.

4.1.2 Buffer Up Signals

Because the algorithm is to function in real-time, the input signals are buffered up in blocks. This allows for more efficient processing than sample-by-sample input. If the block sizes are the same, it is straightforward to take in samples from the input signals. One takes in the same amount of samples from each input signal and then puts the samples in two separate blocks. The next time one takes in samples, the samples are taken in starting from the sample after the one that was taken in last the previous time. This will be at the same index in both of the input signals if the block sizes are the same.

If the block sizes are not the same for the two input signals, it is not immediately intuitive how the samples should be taken in. This algorithm has two different modes that take in samples in two different ways if the block sizes differ between the two input signals. The two modes are called `SkipOnSmall` and `OverlapOnLarge`, and are illustrated in fig. 4.2 and 4.3, respectively.

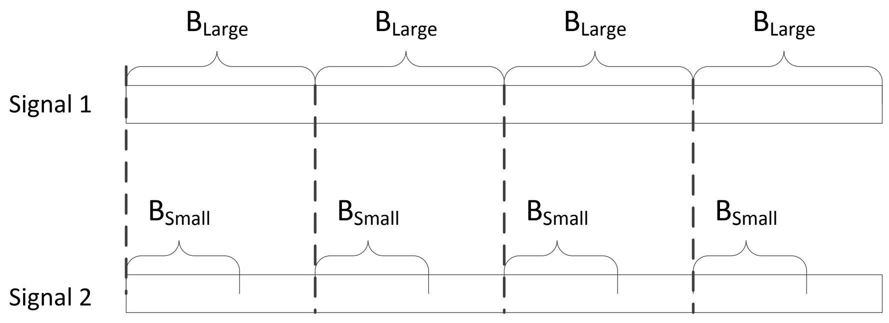


FIGURE 4.2: The `SkipOnSmall` mode. Note that samples are skipped on the signal with the smallest buffer.

In the `SkipOnSmall` mode the largest block size determines which samples should be taken out. Each time blocks are to be filled up, the blocks starts where the large block ended the previous time. This causes the algorithm to skip samples on the input signal with the smallest block size.

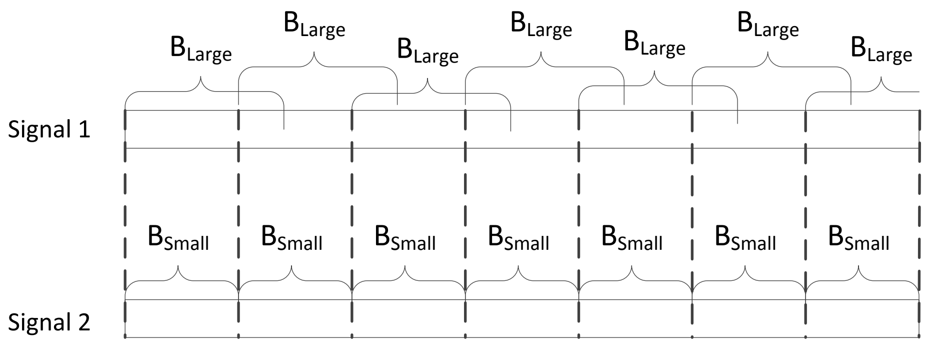


FIGURE 4.3: The `OverlapOnLarge` mode. Note that on the signal with the longest buffer, some of the samples are used more than once.

In the `OverlapOnLarge` mode it is the smallest block size that determines which samples should be taken in. Each time blocks are to be filled up, the blocks start where the smallest block ended the previous time. A consequence of doing it this way is that some samples from the signal with the largest block size will be used more than once.

4.1.3 Convolution Computation

The computation of the convolution sum is done in the time domain. This part of the algorithm takes in two blocks. If the length of the blocks are L_{B1} and L_{B2}, the result will be a vector with length $L_{B1} + L_{B2} - 1$.

4.1.4 Put Convolution Result on Output

The preliminary algorithm provides different modes for putting the result of the convolution of two blocks on the output. All the modes involve some overlap between successive convolution results, since the output blocks are longer than the input. The overlapping samples are added together.

The mode `overAdd_small` has overlap equal to the smallest block. `overAdd_large` has overlap equal to the largest block. This is illustrated in fig. 4.4 and 4.5 respectively.

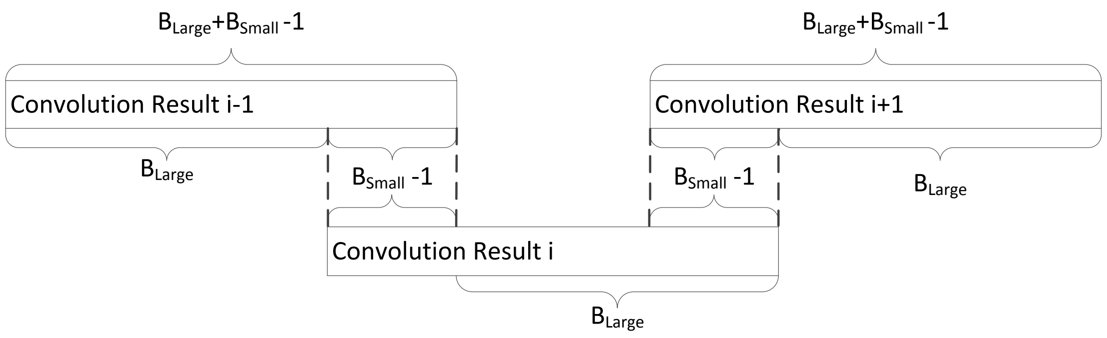


FIGURE 4.4: The `overAdd_small` mode.

The algorithm has additional modes that provide fading in and fading out of the overlapping areas. The modes `expFade` and `expFade2` fade the convolution results in and out exponentially, as illustrated in fig. 4.6 and 4.7, respectively. The mode `linFade` fades the convolution results in and out linearly as illustrated in 4.8. The rate of change of the fading functions are adjustable.

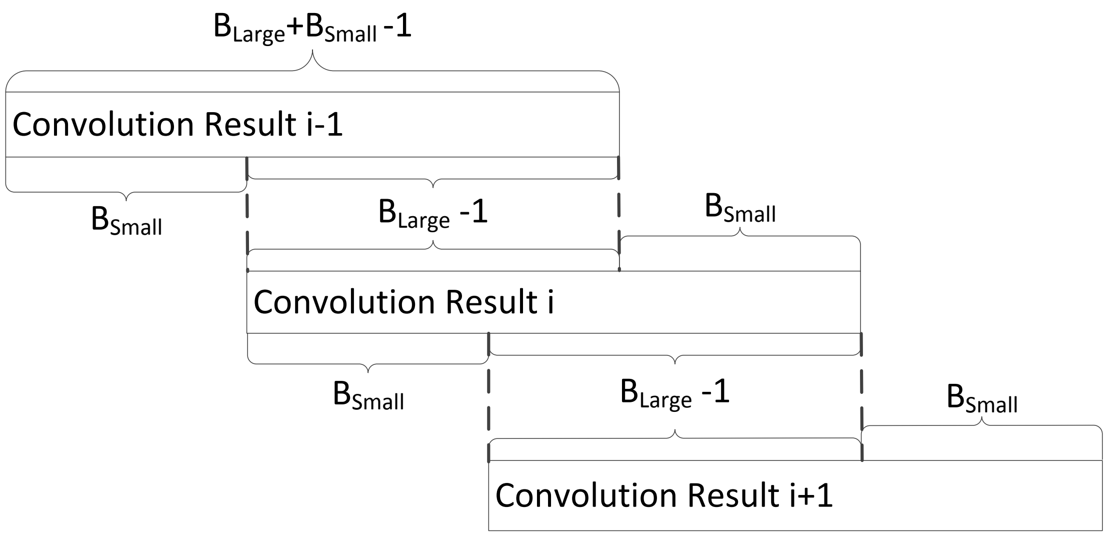


FIGURE 4.5: The `overAdd_large` mode.

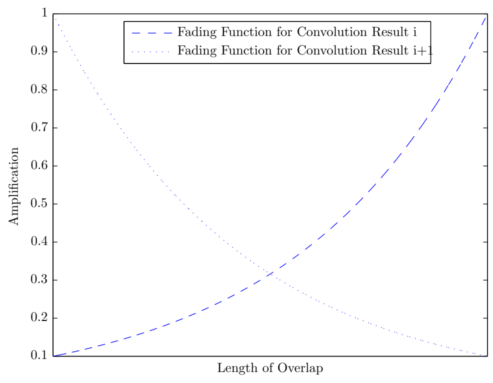


FIGURE 4.6: The `expFade` mode.

4.2 Algorithm Version 1

This section describes the first stage of the final algorithm. It is based on Istvan Varga's opcode `ftconv`. The opcode is modified to support two live audio signals, as opposed to one prerecorded impulse response and one live audio signal. A block diagram is given in fig. 4.9. The implementation was done in Matlab, and can be found in Appendix B.2.

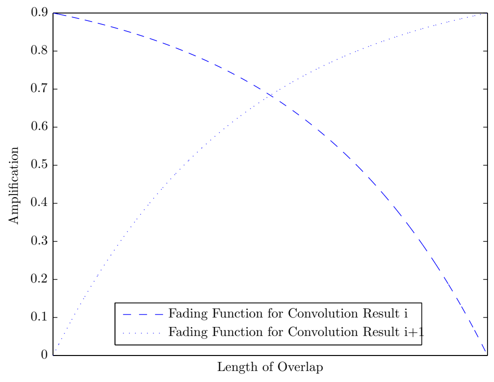


FIGURE 4.7: The `expFade2` mode.

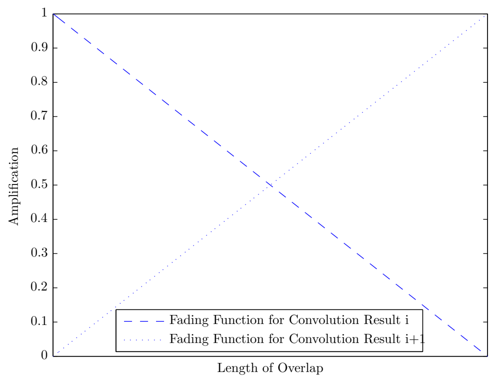


FIGURE 4.8: The `linFade` mode.

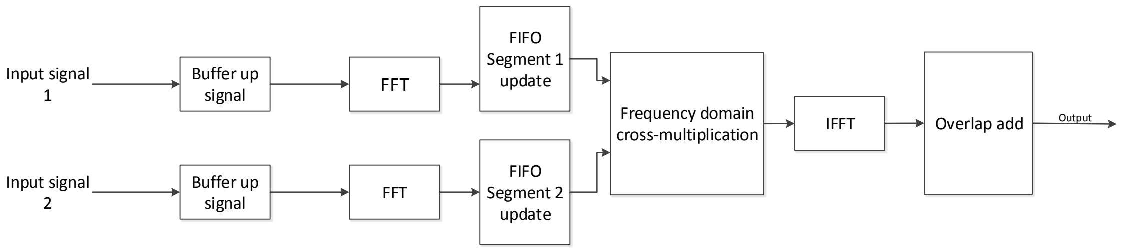


FIGURE 4.9: Block diagram of algorithm version 1.

4.2.1 Short Description

The main idea of Istvan Varga's `ftconv` is to perform blockwise frequency domain multiplication with a prerecorded impulse response (IR), allowing for efficient low latency convolution. The IR is divided into blocks of size 2^n, and a live audio input signal is then buffered up into blocks of the same length as the IR blocks, and multiplied with the IR in the frequency domain as shown in fig. 4.10. This results in an output delay of 2^n samples, instead of a delay equal to the length of the IR. See section 2.3 for a theoretical justification of this method.

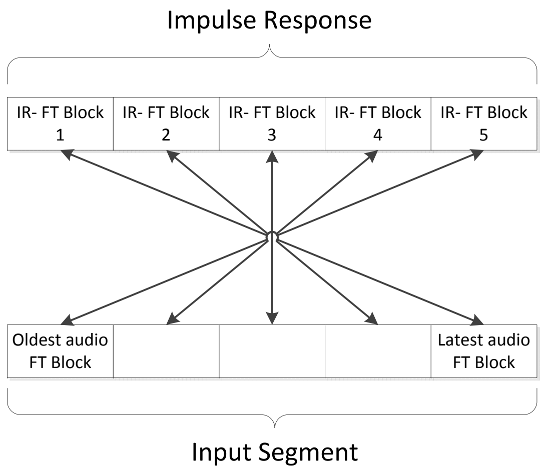


FIGURE 4.10: Illustration of `ftconv`, example with 5-block impulse response. The arrows represent multiplication.

4.2.2 Buffer Partitioning

Both input signals are buffered into a pair of blocks, each of length L_B and padded with L_B zeros. The blocks are then Fourier transformed. Henceforth these transformed blocks are referred to as *FT blocks* (Fourier Transformed blocks). The FT blocks are then put into their respective *segments*. The two input signals each have one segment associated with them, referred to as *segment 1* and *segment 2* when necessary, or *the segments* when referred to jointly. The segments contain N FT blocks each.

The FT blocks are always handled as pairs, and therefore when it is stated that a pair of blocks is added to or thrown from the segments, it always implies the blocks that were buffered up at the same time.

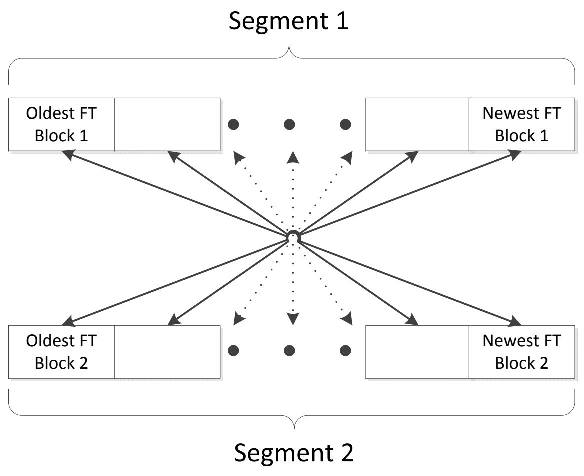


FIGURE 4.11: Illustration of frequency domain cross-multiplication with n blocks. The arrows represent multiplication.

4.2.3 Cross Convolution of a Segment

We perform *cross convolution* as a blockwise frequency domain multiplication of two segments. The newest FT block of signal 1 is multiplied with the oldest FT block of signal 2. The second newest FT block of signal 1 is multiplied with the second oldest FT block of signal 2, and so forth. See fig. 4.11, where the arrows represent a multiplication. The results of each multiplication are then summed. A cross convolution is computed once every time a new pair of input buffers have been filled. It can be mathematically expressed, in the digital frequency domain, as

$$Y_T(k) = \sum_{i=T-N}^{T} X_{1,i}(k) X_{2,N-i}(k), \tag{4.1}$$

where T is the block number of the output ($T = 1$ would denote the first output block), and $X_{m,i}$ denotes FT block i from segment m. An IFFT is performed on Y_T, and it is sent to the output buffer.

4.2.4 Output Buffer

As mentioned in section 4.2.2, the output blocks are about twice as long as the input blocks, because of zero-padding. The output blocks have convolution tails on both ends. When inserting the blocks into the output buffer, the following overlap add method is used:

$$O_T(n) = y_T(n) + y_{T-1}(n + L_{\text{B}}), n \in (0, L_{\text{B}} - 1). \tag{4.2}$$

Following this step, the output is sent to the DAC, and the processing is complete.

4.3 Algorithm Version 2 (Transient Detection)

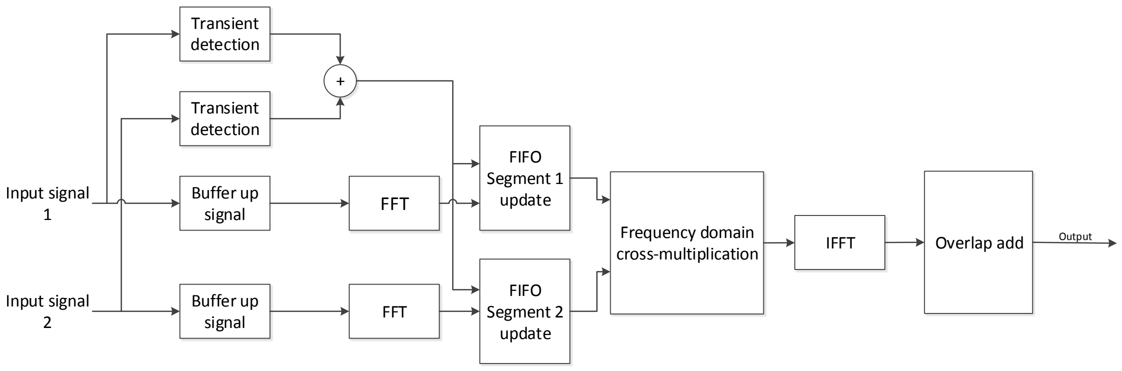


FIGURE 4.12: Block diagram of algorithm version 2.

Algorithm version 2 is an extension of algorithm version 1 described in section 4.2. Version 2 is extended in that it uses transient information from the input signals to adjust the segment lengths. The implementation was done in Matlab, and can be found in appendix B.3.

When a transient occurs in one of the input signals, all the FT blocks previously contained in the segments are thrown away, keeping only the new pair of FT blocks. Thus, when a transient occurs, the output is a result of a convolution between only the latest block pair. The next time a pair of blocks is buffered up, it is put into the segments as in version 1. Algorithm version 1 has a constant segment length of

N blocks, and throws away the oldest FT block pair in the segments each time a new pair is put in. In version two, the oldest FT block pair is thrown away only if the segments are full, i.e. if the amount of blocks in the segments is greater than a user specified maximum we henceforth refer to as N_{max}. The *Transient Detection* blocks and the *FIFO Segment update* blocks in fig. 4.12 are where the extensions to version 1 happen. When the transient detection blocks detect a transient, a signal is sent to the FIFO segment updates. A flow chart describing the inner workings of the FIFO segment update blocks is shown in fig. 4.13.

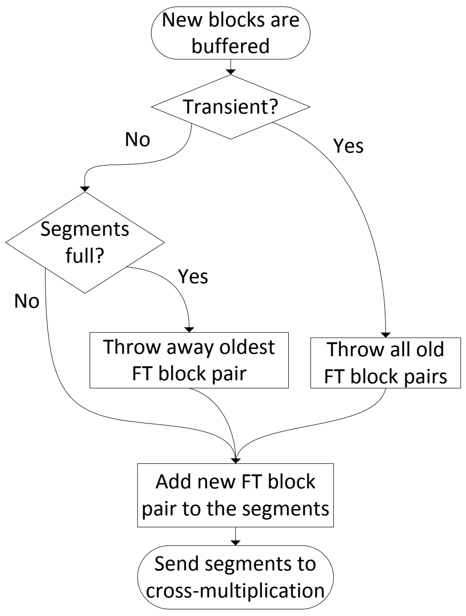

New blocks are buffered
Transient?
No
Yes
Segments full?
Yes
No
Throw away oldest FT block pair
Throw all old FT block pairs
Add new FT block pair to the segments
Send segments to cross-multiplication

FIGURE 4.13: Flow chart of the inner workings in the *FIFO Segment update* blocks of version 2, shown in fig. 4.12.

The transient detection blocks detect transients as defined in 2.4. The methods used in the Matlab and Csound implementation differ. In the final implementation (Csound), a transient detection algorithm written by Øyvind Brandtsegg was used. Since this is not the main focus of this algorithm, see Appendix C for details. The transient detection algorithm implemented in Matlab is in listing B.8.

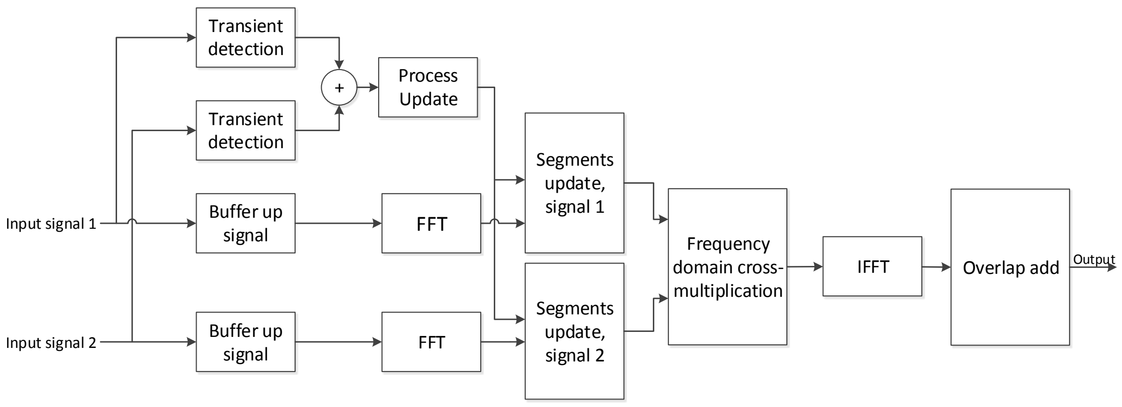


FIGURE 4.14: Block diagram of algorithm version 3.

4.4 Algorithm Version 3 (Parallel Processes)

These versions are extensions of algorithm version 2, described in section 4.3. In this section, different ways to handle the FT blocks, which are discarded after a transient detection, are explored. As opposed to algorithm version 2, the FT blocks contained in a segment before a transient occurs are not thrown away immediately once a transient is detected. Their respective segments are kept in a parallel process to contribute to output blocks following a transient. The three algorithms described in this section operate differently in the way these processes receive and throw away FT blocks. All extensions in this section are in the *process update* and *segments update* blocks in fig. 4.14. All of the following versions have some key features in common, namely what will be referred to as the *active process* and *semi-active processes*. The active process handles the segment pair that is receiving blocks from the input. The semi-active processes contain segment pairs that no longer receive input, but still contribute to the output signal.

What all these processes have in common is that they contain two segments, one for each signal. The segments are cross-multiplied as in fig. 4.11, separately for each process, then the results are added together and normalized. An IFFT is then performed, and the block is sent to output, as seen in fig. 4.18.

4.4.1 Alternative 1: `ThrowAll` (Used in Final Implementation)

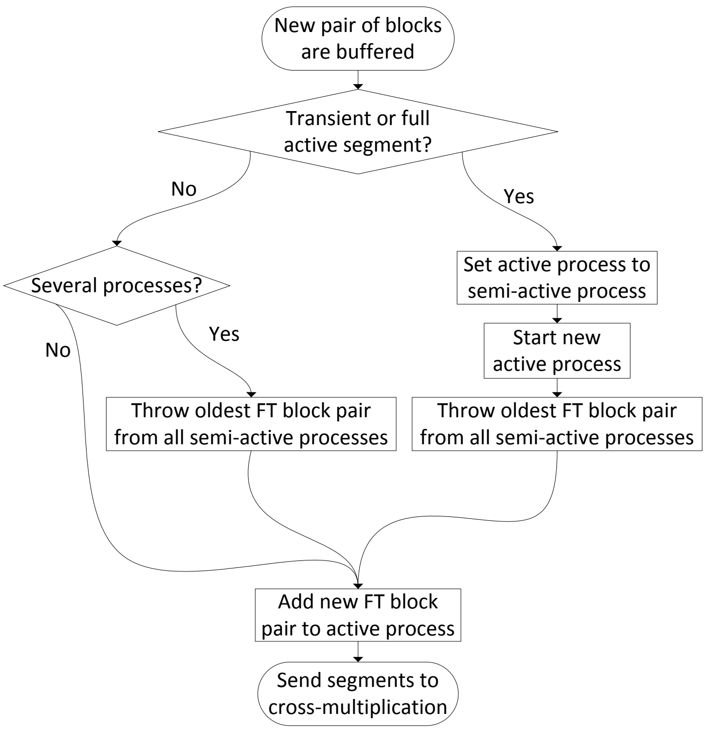

New pair of blocks are buffered
Transient or full active segment?
No
Yes
Several processes?
Set active process to semi-active process
No
Yes
Start new active process
Throw oldest FT block pair from all semi-active processes
Throw oldest FT block pair from all semi-active processes
Add new FT block pair to active process
Send segments to cross-multiplication

FIGURE 4.15: Flow chart of the inner workings in the *process update* and *segments update* blocks in fig.4.14 for `ThrowAll`.

A flow chart of this version's process handling is shown in figure 4.15. This version, which is the version used in the final product, treats each part of the signal between two transient as what we call a *convolution event*. We define convolution events as the convolution of segments between two transients. They are processed separately, without directly affecting, or being directly affected by, surrounding convolution events. We further discuss convolution events in 6.4.1.

This final algorithm was implemented both in Matlab (appendix B.4) and in Csound with an opcode written in C (appendix A).

Each time a transient occurs, the active process is turned into a semi-active process. A new active process is then created, which starts taking in new FT blocks from the input.

The way processes are handled in this version can be seen in fig. 4.15. The main idea is that the oldest FT block pair from all semi-active processes are thrown in each iteration, while the active process keeps receiving FT block pairs from the input, and does not throw away old blocks. If the number of FT block pairs in the active process reaches N_{max}, it is treated as if a transient is detected, and the process is set to be semi-active. If neither a transient is detected, nor the active segment becomes full, the oldest FT block pairs in each semi-active process are thrown, and the newest FT block pairs from the signals are appended to the segments in the active process.

4.4.2 Alternative 2: `ThrowLast`

A flow chart of this version's process handling is shown in fig. 4.16. This version was implemented in Matlab, see appendix B.5.

As in `ThrowAll`, `ThrowLast` starts a new active process whenever a transient is detected and sets the previous active process to semi-active. However, as opposed to `ThrowAll`, `ThrowLast` only throws out the oldest FT block pair in the oldest semi-active process. The other semi-active processes remain constant until they become the oldest one. When the oldest semi-active process is empty, the second oldest process is set to be the oldest one, and will thus be the process from which FT block pairs are thrown out in the next iteration. If no transients occur and no new processes are started, one can end up with a case where all semi-active processes have empty segments, and the only process running is the active one. If the active process is the only one running, the algorithm checks if the segments associated with this process are full, i.e. they contain N_{max} FT block pairs. If they are full, the oldest block pair is thrown out. If the segments are not full, no blocks are thrown out.

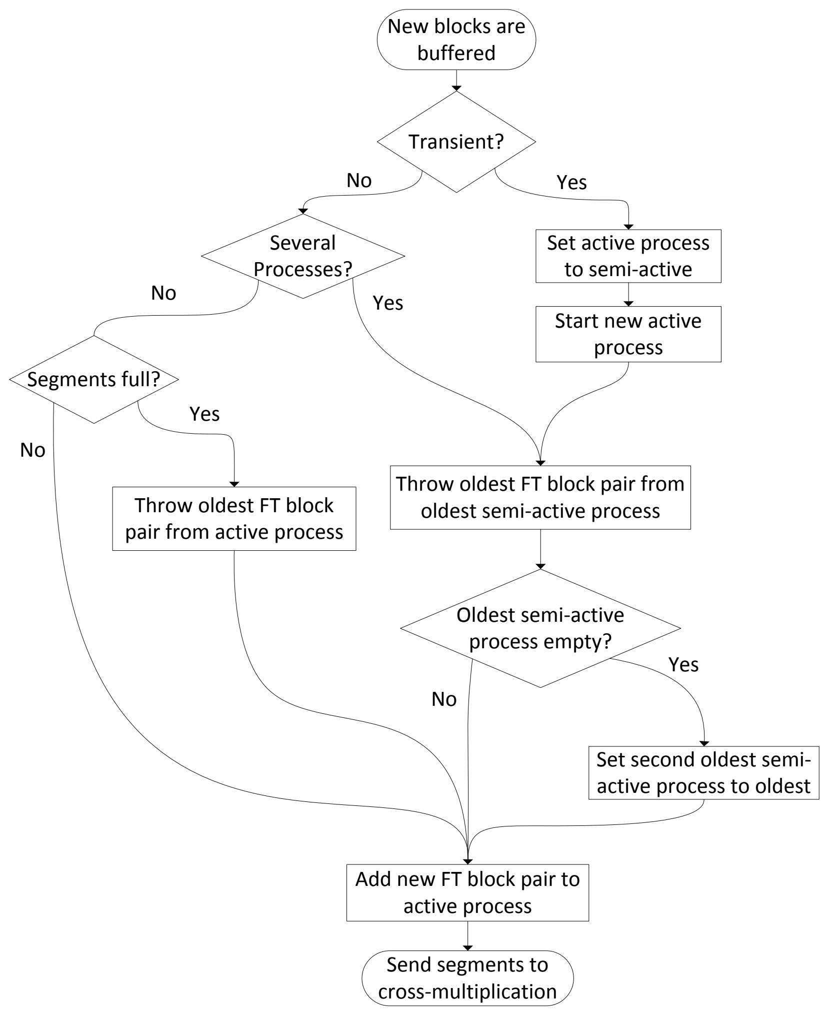

New blocks are buffered
Transient?
No
Yes
Several Processes?
Set active process to semi-active
No
Yes
Start new active process
Segments full?
Yes
No
Throw oldest FT block pair from active process
Throw oldest FT block pair from oldest semi-active process
Oldest semi-active process empty?
Yes
No
Set second oldest semi-active process to oldest
Add new FT block pair to active process
Send segments to cross-multiplication

FIGURE 4.16: Flow chart of the inner workings in the *process update* and *segments update* blocks in fig. 4.14 for `ThrowLast`.

4.4.3 Alternative 3: `TwoProc`

A flow chart of this version's process handling is shown in fig. 4.17. This version was implemented in Matlab, see appendix B.6.

This version has a maximum of two processes running in parallel. When a transient is detected on one of the input signals, all the FT blocks in the active process are appended to the semi-active process, and the newest FT block pair is put into the active process. An FT block pair is thrown out of the semi-active process if the sum of the number of FT block pairs contained in the active and semi-active process is equal to $N_{\max}$. If the semi-active process is empty, an FT block pair is thrown out of the active process once it reaches $N_{\max}$ FT block pairs.

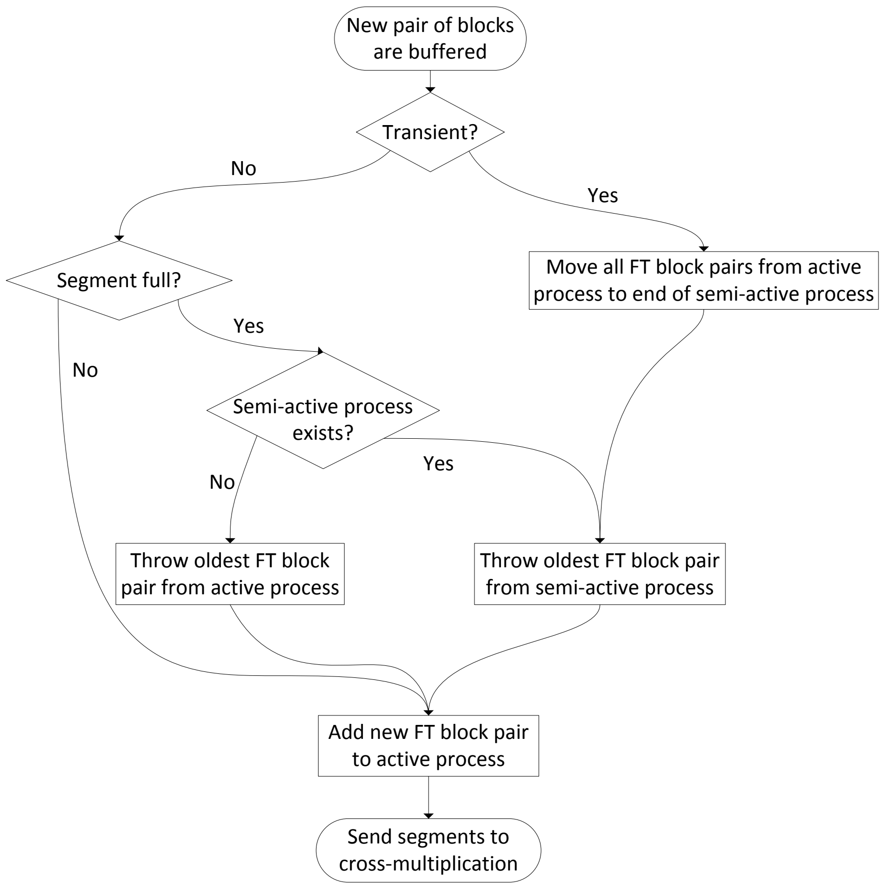

New pair of blocks are buffered
Transient?
No
Yes
Segment full?
Move all FT block pairs from active process to end of semi-active process
Yes
No
Semi-active process exists?
No
Yes
Throw oldest FT block pair from active process
Throw oldest FT block pair from semi-active process
Add new FT block pair to active process
Send segments to cross-multiplication

FIGURE 4.17: Flow chart of the inner workings in the *process update* and *segments update* blocks in fig. 4.14 for `TwoProc`.

4.4.4 Normalization

There is no obviously correct way to normalize the blocks of the different processes. What could be considered an optimal normalization depends on which criteria one optimizes for. We opted to normalize with a stable output amplitude in mind. Our normalization scheme is illustrated in fig. 4.18.

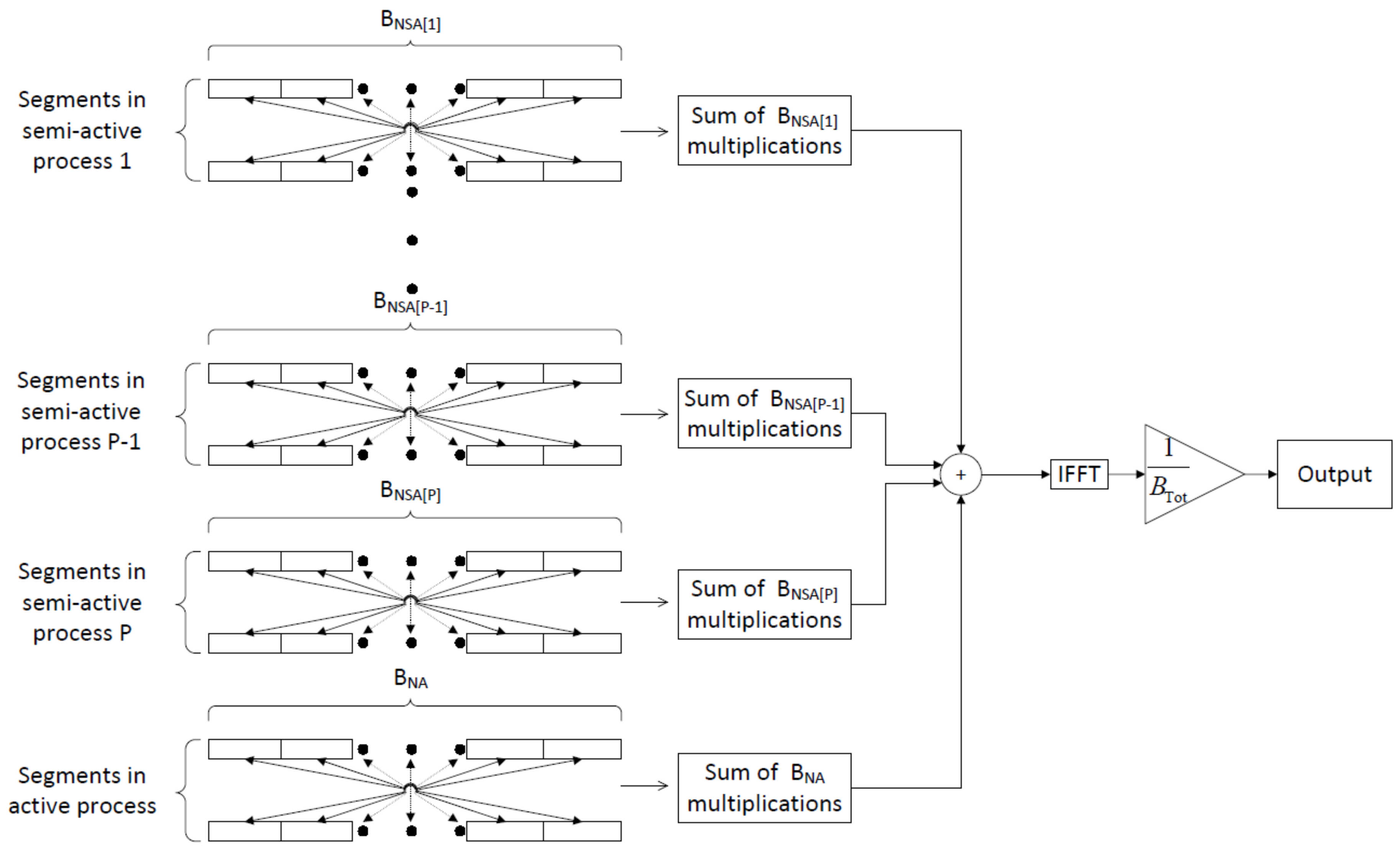

$B_{NSA[1]}$
Segments in semi-active process 1
Sum of $B_{NSA[1]}$ multiplications
$B_{NSA[P-1]}$
Segments in semi-active process P-1
Sum of $B_{NSA[P-1]}$ multiplications
$B_{NSA[P]}$
Segments in semi-active process P
Sum of $B_{NSA[P]}$ multiplications
B_{NA}
Segments in active process
Sum of B_{NA} multiplications
+
IFFT
$\frac{1}{B_{Tot}}$
Output

FIGURE 4.18: Generation of output with parallel processes. The active process and P semi-active processes contribute to the output. B_{NA} is the number of blocks in the active process. $B_{\mathrm{NSA}}[\mathrm{P}]$ is the number of blocks in semi-active process P.

With this method, one normalizes by the total number of blocks being processed, which is

$$B_{\text{Tot}} = B_{\text{NA}} + \sum_{i=1}^{P-1} B_{\text{NSA[i]}}, \tag{4.3}$$

where P is the total number of processes, B_{NA} is the number of block pairs in the active process, and $B_{\text{NSA[i]}}$ is the number of block pairs in semi-active process i. This means that the amplitude stabilizes quickly, even as the number of blocks grows.

This method was only implemented for `ThrowAll`, as all the other versions are only implemented in Matlab, and the scaling of the output is done automatically by Matlab's built in function `soundsc()`.

Chapter 5

Results

This chapter presents results relevant for the discussion in chapter 6. All the sound files mentioned here can be found in the digital appendix delivered with this thesis. The files are organized in folders with the same names as the headlines in this chapters.

All input signals used to generate these audio files can be found in the folder *Test input signals*.

5.1 Preliminary Algorithm

Sound files from this version (found in the *PreliminaryAlgorithm* folder in the digital appendix):

- 440SinesAsInput_Buffer100.wav
- 440SinesAsInput_Buffer300.wav
- 440SinesAsInput_Buffer350.wav
- 440SinesAsInput_Buffer500.wav
- 440SinesAsInput_Buffer550.wav

- 440SinesAsInput_B_1_2000_B_2_100_expFade2.wav
- 440SinesAsInput_B_1_2000_B_2_100_NoFade.wav
- 440SinesAsInput_B_1_2000_B_2_150_expFade2.wav

All of these files were generated with 440 Hz sines on both inputs.

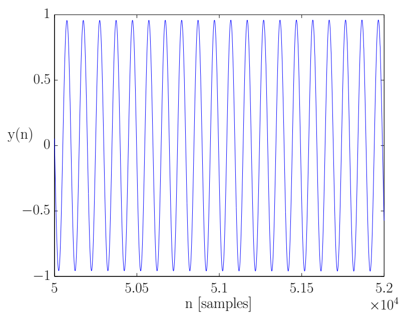
440SinesAsInput_BufferX.wav were generated with buffer sizes of X samples on both inputs. No fading functions were used.

440SinesAsInput_B_1_2000_B_2_100_expFade2.wav was generated with buffer sizes of 2000 and 100 samples for the two input signals, using the `OverlapOnLarge` method and the `expFade2` fading function.

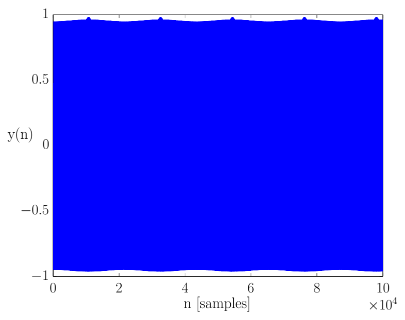
440SinesAsInput_B_1_2000_B_2_100_NoFade.wav was generated with buffer sizes of 2000 and 100 samples for the two input signals, using the `OverlapOnLarge` method without any fading function.

440SinesAsInput_B_1_2000_B_2_150_expFade2.wav was generated with buffer sizes of 2000 and 150 samples for the two input signals, using the `OverlapOnLarge` method and the `expFade2` fading function.

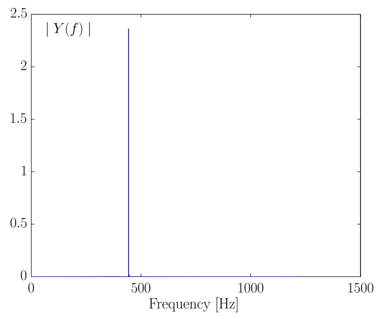
The (A) and (B) figures in fig. 5.1, 5.2 and 5.3 are all time-domain plots of their respective soundfiles. The (A) figures span over a short interval to show the waveform properly. The (B) figures span over longer intervals, and they are all included to show the low frequency amplitude modulation (AM) seen in the envelope of the signal, but which is not clearly visible in the (A) figures. All the (C) figures show the frequency content of the sound files.



(A)

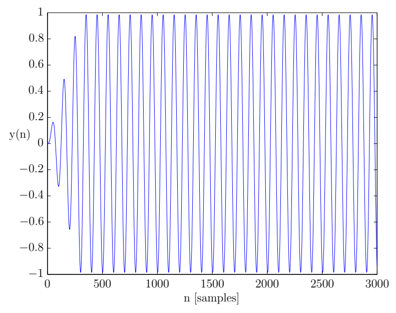


(B)

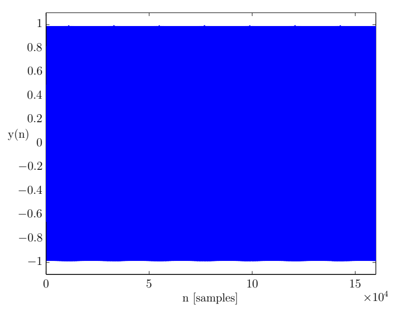


(C)

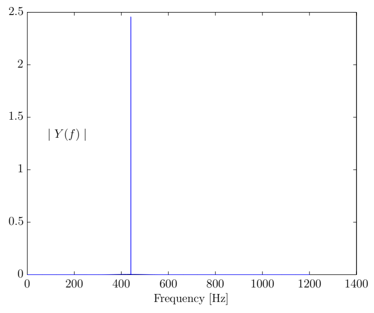
Figure 5.1: Plots from the preliminary algorithm, with 440 Hz sines as input and a buffer size of 100 samples. (A) shows a short time interval of the soundfile. The output is clearly a sine. (B) shows a long time interval of the soundfile. The low frequency AM can be seen in the envelope of the signal. The AM has a low amplitude and does not produce noticeable sidelobes. (C) shows the frequency content of the soundfile. The energy is situated at 440 Hz.



(A)



(B)



(C)

Figure 5.2: Plots from the preliminary algorithm, with 440 Hz sines as input and a buffer size of 300 samples. The low frequency AM shown in (B) is even smaller than in Fig 5.1b.

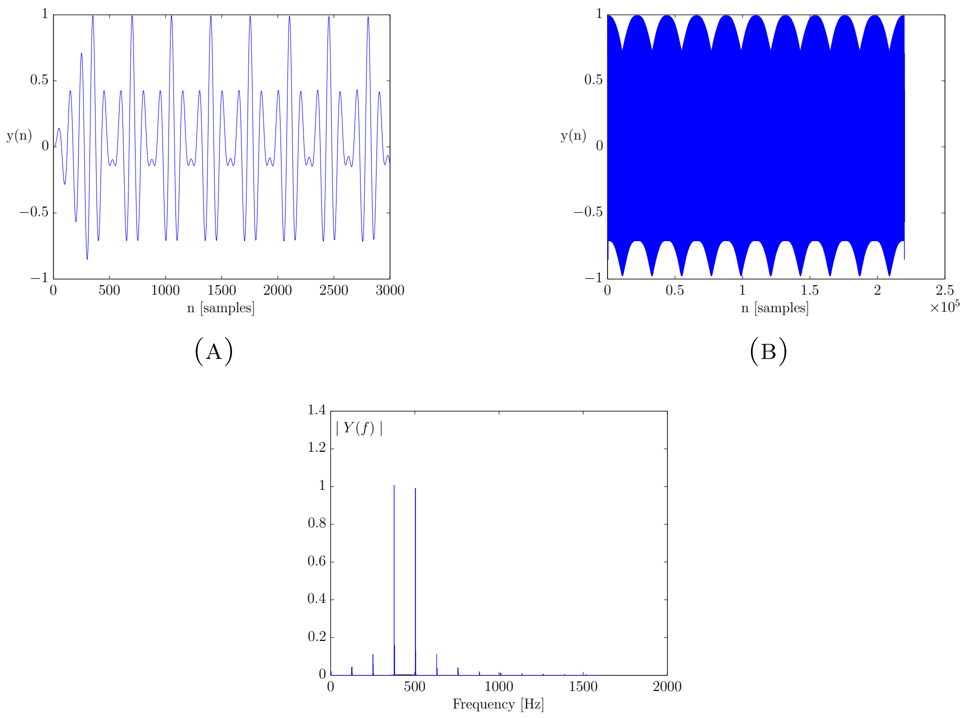


FIGURE 5.3: Plots from the preliminary algorithm, with 440 Hz sines as input and a buffer size of 350 samples. The output in (A) is clearly not a sine. There is significant AM, as can be seen in (B) . The frequency plot in (C) shows that the energy is situated not only at 440 Hz.

5.2 Algorithm Version 1

Sound files from this version (found in the *Version1Results* folder in the digital appendix):

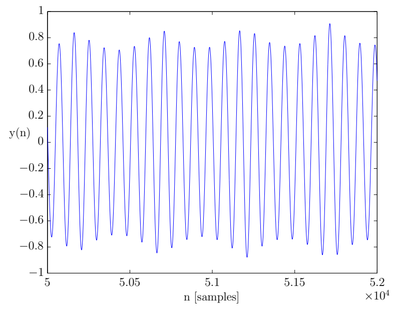
- 500HzSineInput_BlockSize512_BlockNum100.wav
- disharmonyFromDelayedChange.wav
- indistinctTransientsSynthDrumloop2.wav

All sound files were generated with $L_\mathrm{B} = 512$ samples, and segment length $N = 100$ blocks.

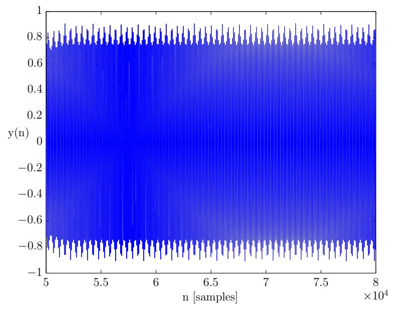
500HzSineInput_BlockSize512_BlockNum100.wav has two equal sines on the inputs. Relevant plots are in fig. 5.4.

disharmonyFromDelayedChange.wav has *synth.wav* on both inputs. Relevant plots are in fig. 5.5.

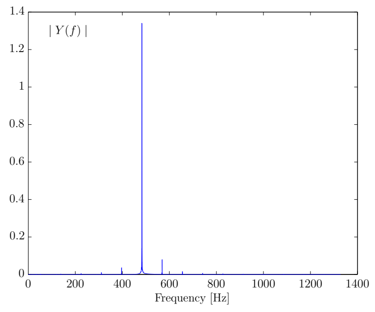
indistinctTransientsSynthDrumloop2.wav has *synth.wav* on one input, and *drumloop2.wav* on the other. Relevant plots are in fig. 5.6.



(A)



(B)



(C)

FIGURE 5.4: Plots from Algorithm Version 1, with 500 Hz sines on both input channels. Block size of 512 samples, 100 block segments. The AM is less prominent than in 5.3, but still creates some sidelobes.

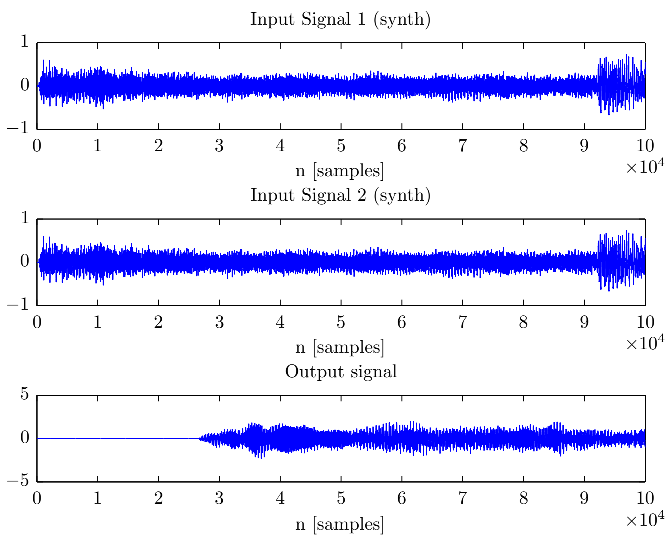


FIGURE 5.5: Plot of first 100000 samples of input and output of Algorithm Version 1, with synth.wav on both input channels. Slow rise of initial transient. Output is delayed by $L_s/2$ samples. A block size of 512 samples was used. The segment size was 100 blocks.

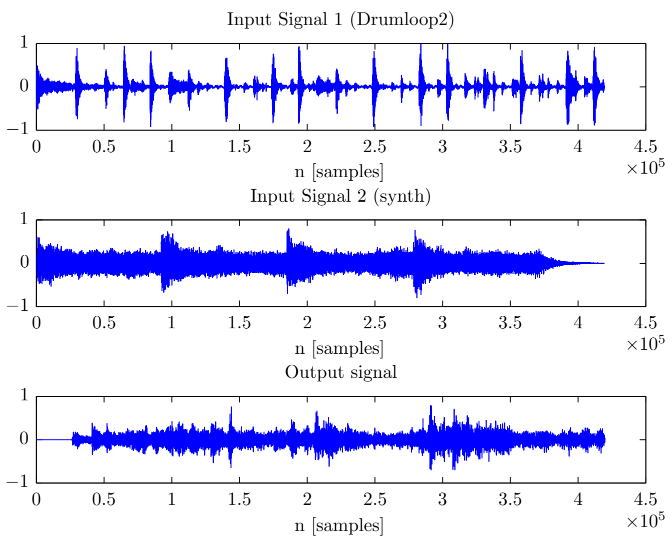


FIGURE 5.6: Plot of input and output of Algorithm Version 1, with drumloop2.wav and synth.wav as input. Transients are very indistinct on output. Output is delayed by $L_s/2$ samples. The Block size was 512 samples. The segment size was 100 blocks.

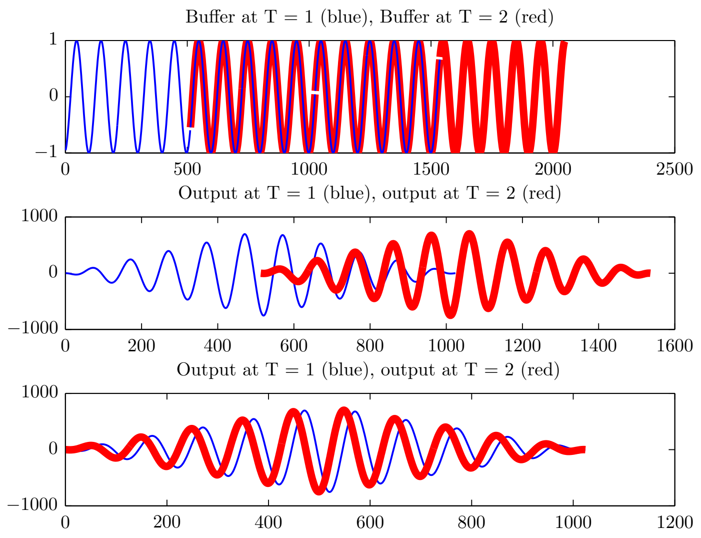


FIGURE 5.7: Plot of input and output of Algorithm Version 1, with two equal 440 Hz sines on the inputs. As can be seen, to following output blocks are out of phase, even though the input signals are in phase. The block size was 512 samples. The segment size was 3 blocks.

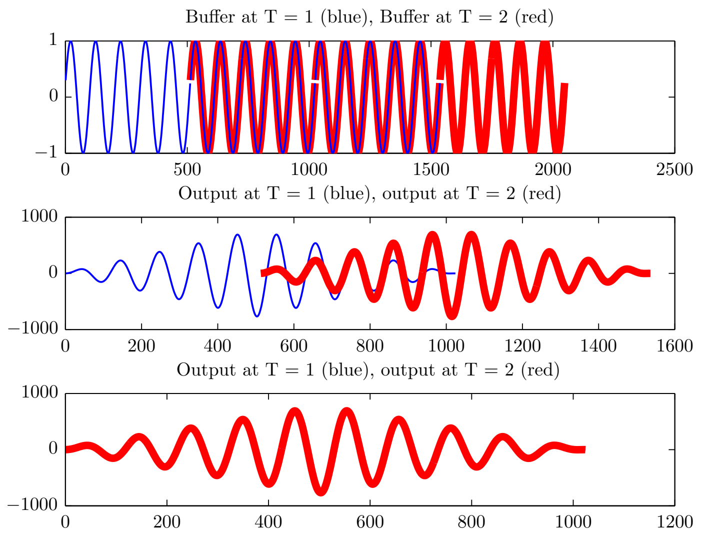


FIGURE 5.8: Plot of input and output of Algorithm Version 1, with two equal 430.7 Hz sines on the inputs. As can be seen, to following output blocks are in phase, because a 430.7 Hz sine has a period of 512/5 samples with $F_s = 44100$ Hz. The block size was 512 samples. The segment size was 3 blocks.

5.3 Algorithm Version 2

Sound files from this version (found in the *Version2Results* folder in the digital appendix):

- drumloop2_synth_version2_transDet.wav
- Git1Akk_Syn1Akk_ver2.wav.wav

All sound files were generated with $L_{\mathrm{B}} = 512$ samples, and maximum segment length $N_{\mathrm{max}} = 100$ blocks.

drumloop2_synth_version2_transDet.wav has *synth.wav* on one input, and *drumloop2.wav* on the other. Relevant plots are in fig. 5.9 and 5.10.

Git1Akk_Syn1Akk_ver2.wav has *Gitar1akkord.wav* on one input, and *Synth1akkord.wav* on the other. Relevant plots are in fig. 5.17 and 5.12.

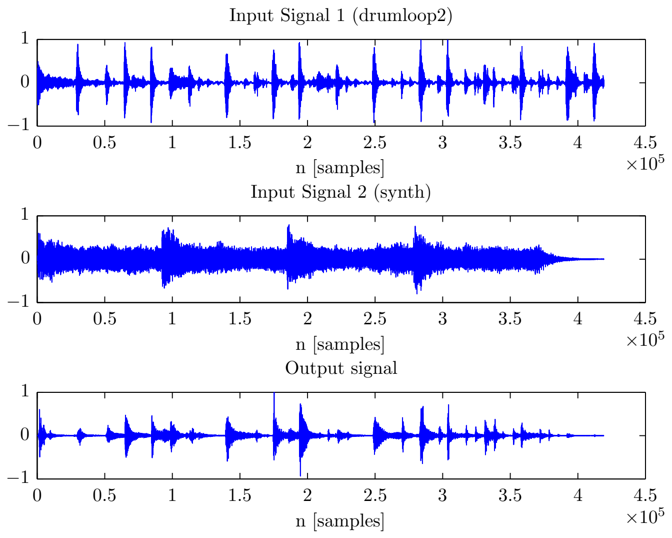


FIGURE 5.9: Plot of input and output of Algorithm Version 2, with drumloop2.wav and synth.wav as input. Transients are much more distinct on output, compared to fig. 5.6. Output is **no longer** delayed by $L_{\mathrm{S}}/2$ samples. A Block size of 512 samples was used. The segment size was 100 blocks.

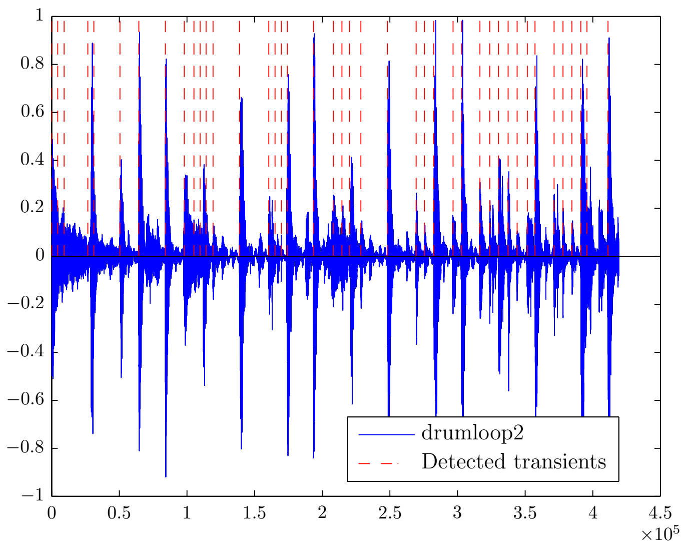


FIGURE 5.10: Plot of drumloop2.wav, with transients detected used to generate the output in fig. 5.9.

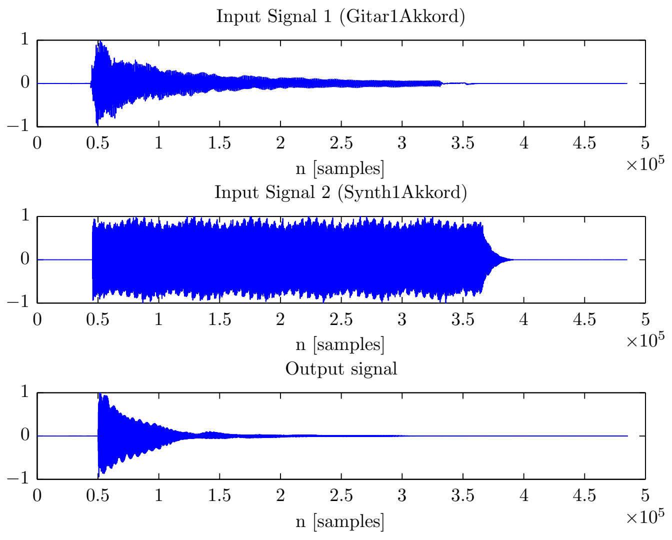


FIGURE 5.11: Plot of input and output of Algorithm Version 2, with Gitar1Akkord.wav and Synth1Akkord.wav as input. Output becomes disharmonic once the segments are full, that is $512 * 100 = 51200$ samples after the transient. 5.6. Output is **no longer** delayed by $L_S/2$ samples. Block size of 512 samples, 100 blocks segments.

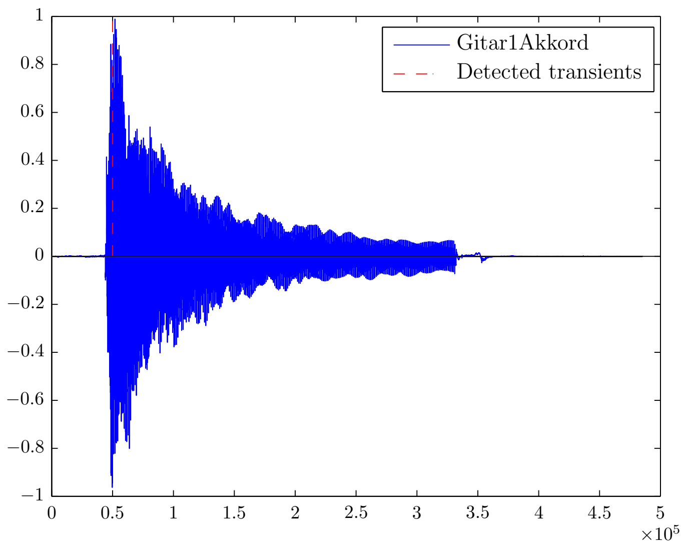


FIGURE 5.12: Plot of Gitar1Akkord.wav, with transient detected used to generate the output in fig. 5.17.

5.4 Algorithm Version 3

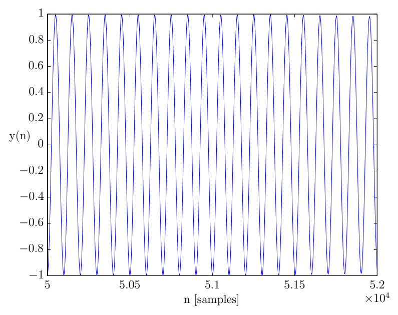
5.4.1 `ThrowAll` (Final Algorithm)

Sound files from this version (found in the *Final Version* folder in the digital appendix):

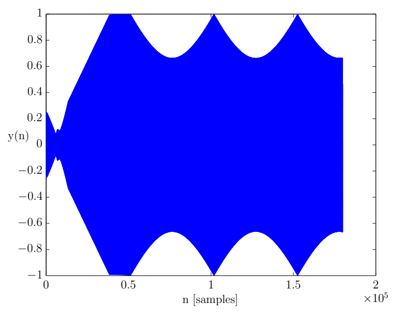
- 440HzSine_FinalVersion_512_BS_100B.wav
- 440HzSine_FinalVersion_512_BS_50B.wav
- drumloop2_synth_FinalVersion_1Process.wav
- drumloop2_synth_FinalVersion_10Process.wav
- gitar1akkord_synth1akkord_FinalVersion.wav

All sound files were generated with $L_B = 512$ samples, and maximum segment length $N_{\max} = 100$ blocks, except for *440HzSine_FinalVersion_512_BS_50B.wav*, which was generated with 50 block segments.

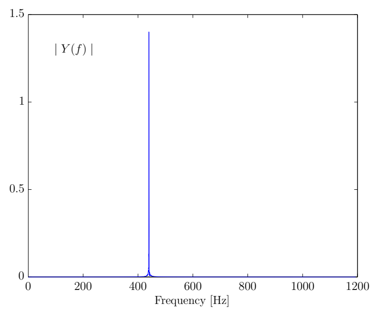
All sound files were generated with maxNumProc = 10, except for *drumloop2_synth_FinalVersion_1Process.wav*, which was generated with maxNumProc = 1.



(A)

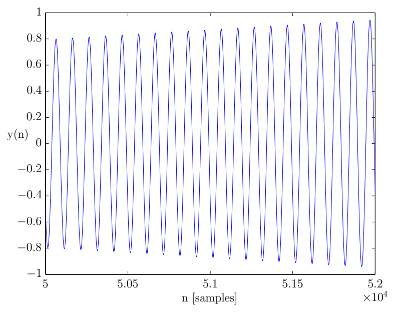


(B)

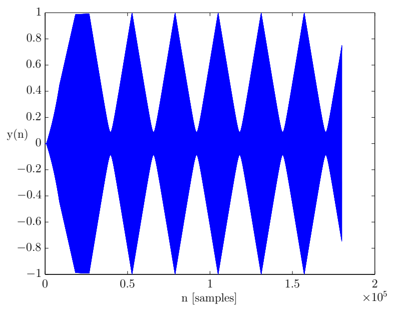


(C)

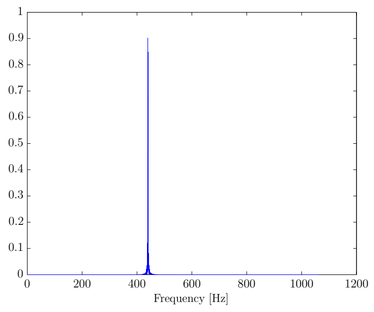
FIGURE 5.13: Plots from Algorithm Version 3 `ThrowAll`, with 440 Hz sines on both input channels. The are no longer any sidelobes, but there is an AM with period L_s. This is, however **much** less disturbing than a period of L_B. The block size was 512 samples. The segment size was 100 blocks.



(A)



(B)



(C)

Figure 5.14: Plots from Algorithm Version 3 `ThrowAll`, with 440 Hz sines on both input channels. The segment has half the length compared to 5.13, and the period of the AM is therefore half as long. There are still no sidelobes. The block size was 512 samples. The segment size was 50 blocks.

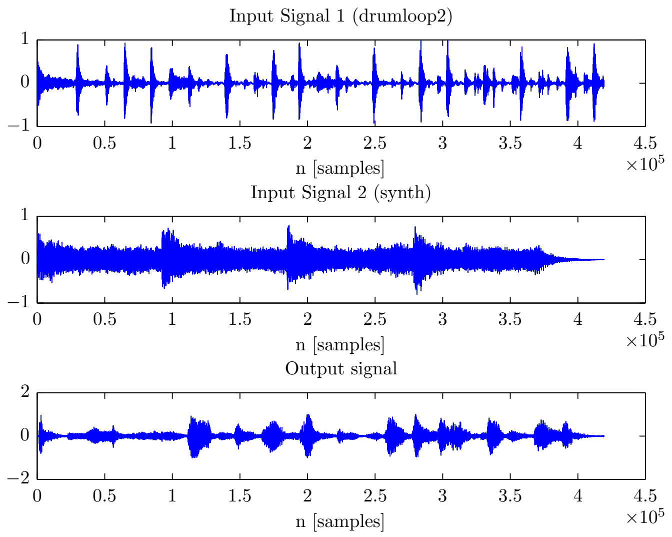


Figure 5.15: Plot of input and output of Algorithm Version 3 `ThrowAll`, with drumloop2.wav and synth.wav as input, with maxNumProc 10. Transients are a bit less distinct on output, compared to fig. 5.9. The block size was 512 samples. The segment size was 100 blocks.

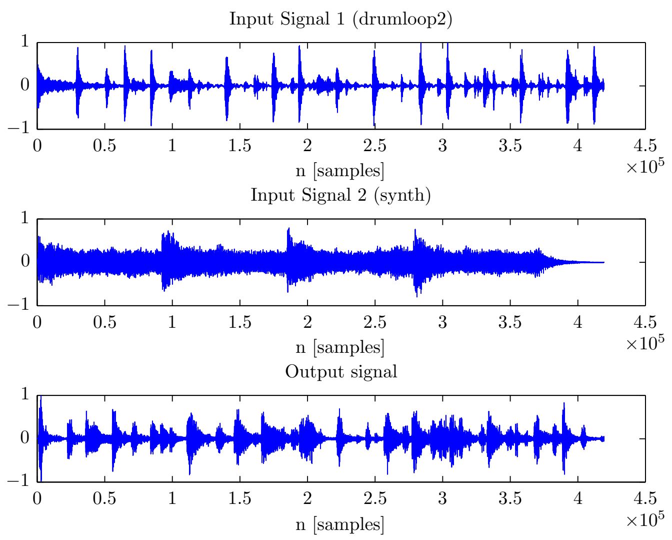


FIGURE 5.16: Plot of input and output of Algorithm Version 3 `ThrowAll`, with drumloop2.wav and synth.wav as input, with maxNumProc 1. Transients are more distinct than with 10 processes, as in fig. 5.15. The block size was 512 samples. The segment size was 100 blocks.

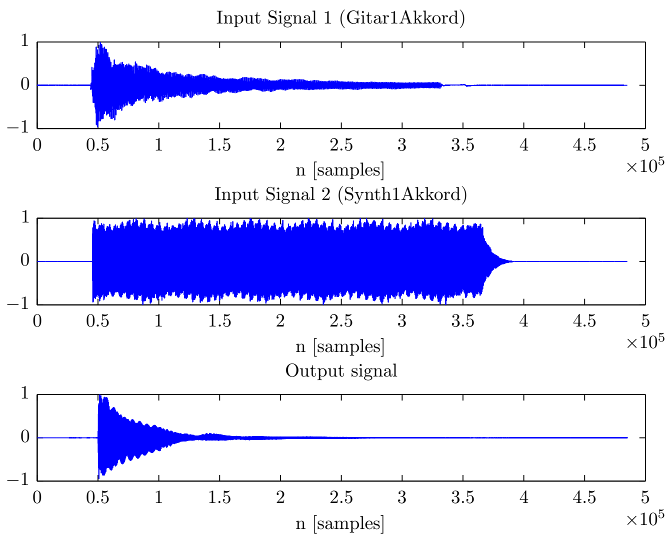


FIGURE 5.17: Plot of input and output of Algorithm Version 3 `ThrowAll`, with Gitar1Akkord.wav and Synth1Akkord.wav as input. Output no longer becomes disharmonic. The block size was 512 samples. The segment size was 100 blocks.

5.4.2 ThrowLast

Sound files from this version (found in the *ThrowLast* folder in the digital appendix):

- ThrowLast256+SingleSine172Hz.wav
- ThrowLast512+SingleSine86Hz.wav
- ThrowLastUnwantedPeriodicityBlocksize256Input440Hz.wav
- ThrowLastUnwantedPeriodicityBlocksize512Input440Hz.wav
- ThrowLastUnwantedPeriodicityBlocksize512Input500Hz.wav
- ThrowLastUnwantedPeriodicityBlocksize512Inputsynth.wav

Fig. 5.18 and fig. 5.19 illustrate the weakness of the `ThrowLast` algorithm, i.e. the unwanted periodicity. The audio files were generated with a 440 Hz sine and *drumloop2.wav* as input signals. N_{max} was set to 200. A transient detetion was applied to the drumloop to ensure that initialization of new processes occured, which is required for the unwanted periodicity to arise. The transient detection was applied using the Matlab script `TransDet2`, which can be found in appendix B.3.

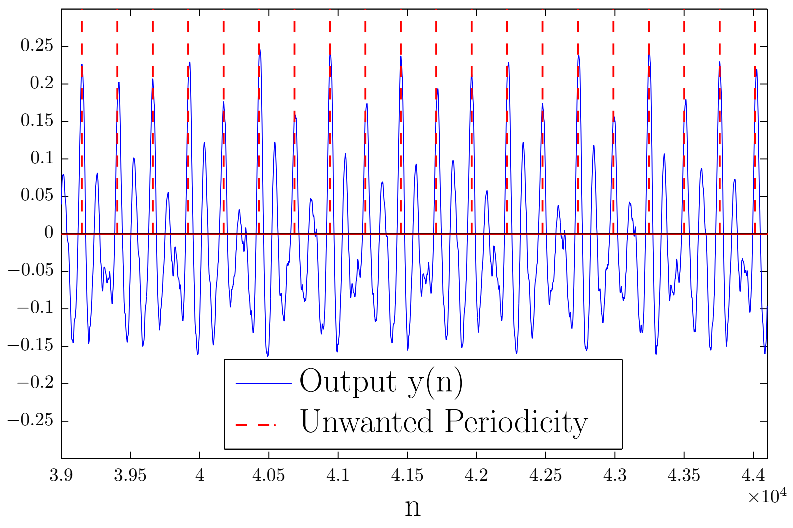


FIGURE 5.18: Excerpt from the audio file *ThrowLastUnwantedPeriodicityBlock-size256Input440Hz.wav*, showing the unwanted periodicity when a block size of 256 samples is used.

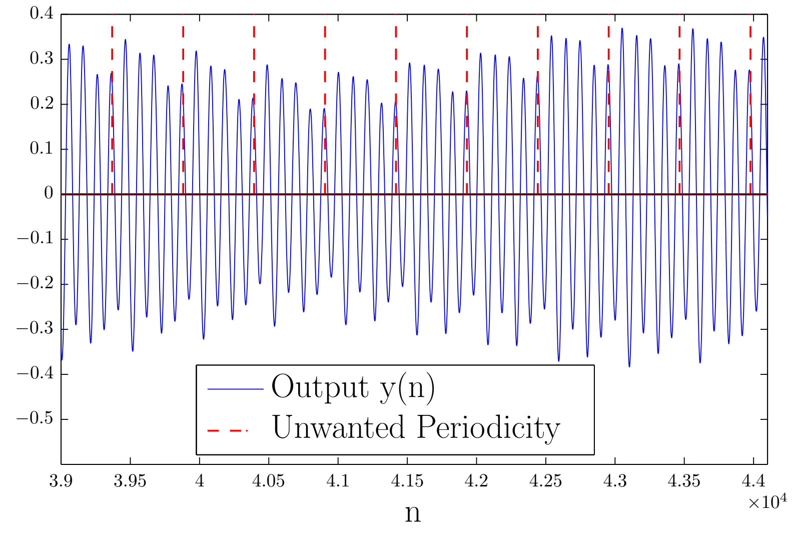


FIGURE 5.19: Excerpt from the audio file *ThrowLastUnwantedPeriodicityBlock-size512Input440Hz.wav.wav*, showing the unwanted periodicity when a block size of 512 samples is used.

5.4.3 `TwoProc`

Sound files from this version (found in the *TwoProc* folder in the digital appendix):

- TwoProcSynthDrumloopMNB50.wav

- TwoProcSynthDrumloopMNB100.wav
- TwoProcSynthDrumloopMNB200.wav

All examples were generated with $L_\mathrm{B} = 512$, using input signals *synth.wav* and *drumloop2.wav*.

TwoProcSynthDrumloopMNB50.wav has $N_\mathrm{max} = 50$.

TwoProcSynthDrumloopMNB100.wav has $N_\mathrm{max} = 100$.

TwoProcSynthDrumloopMNB200.wav has $N_\mathrm{max} = 200$.

5.5 Graphical User Interface

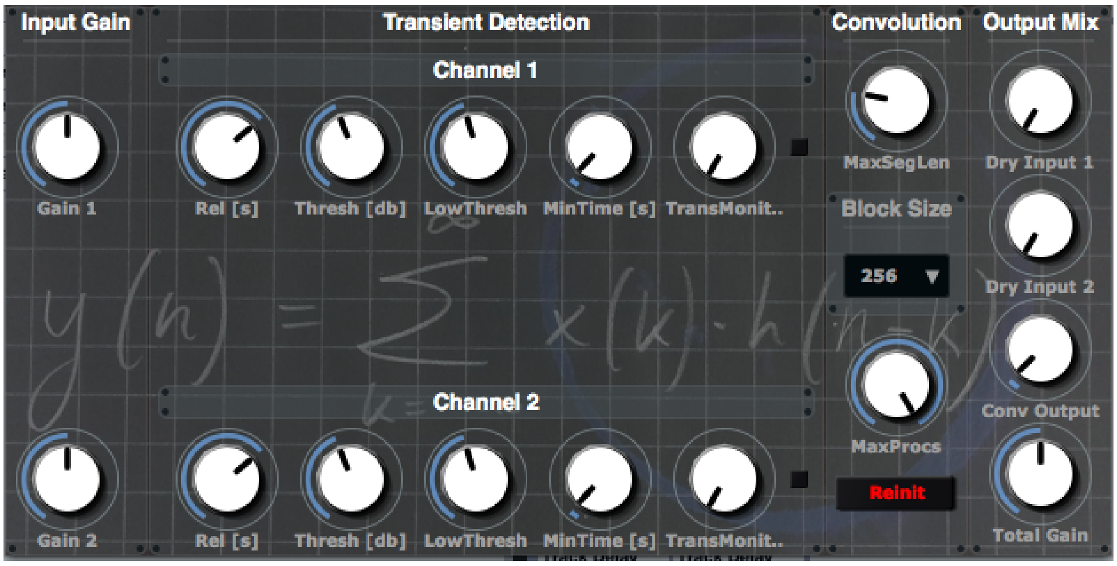


FIGURE 5.20: Graphical User Interface of VST plug-in.

For the VST implementation, a GUI was implemented, as seen in fig. 5.20. When using the VST in a DAW, the VST has to be put on an auxillary channel in the DAW. The two input signals has to be sent to this auxillary channel. One of the input signals has to panned all the way to the left and the other signal has to be panned all the way to the right. An inherent limitation in the current VST-standard forced us to handle the input signals this way.

In the next sections, the parameters available to users will be described.

5.5.1 Gain Knobs

Gain 1/2 sets the input gain of the signals.

Dry Input 1/2 sets the output gain for the unprocessed input signals. Leave these at 0 if only the convolution result should be heard.

Conv Output sets the output gain for the convolution result.

Total Gain sets the output gain for the final mix of dry and processed signals.

5.5.2 Transient Detection Section

Transients may be very different from instrument to instrument (see section 2.4). To give the user control over the transient detection, a range of parameters are available. The two input signals have separate parameters. It is important to note that the transient analysis has an adaptive threshold, based on the RMS of the signals.

Rel is the release time of the envelope analysis which the transient detection is based on. Low values allow for more frequent transient detection, high values allow for more stable RMS-analysis for the dynamic threshold.

Thresh sets the threshold for transient detection. This threshold is relative to the RMS level of the signal, meaning that a threshold of 0 dB would result in transients being detected all the time, and a higher threshold would require stronger amplitude changes for detection.

LowThresh sets a minimum amplitude for detection, regardless of the RMS level.

MinTime sets a minimum amount of time between transient detections.

TransMonitor lets the user hear a click every time a transient is detected. This click is based on the input signal, making it easier to adjust the detection parameters. A "LED" to the right of this knob also lights up every time a transient is detected.

5.5.3 Convolution Section

MaxSegLen is related to N_{max}, and sets the maximum L_{s}. It is given in seconds (calculated from the block size and sampling frequency) to be intuitive for musicians. Longer segments lead to more smearing in time, making the inputs less distinguishable.

BlockSize sets L_{B}. This mainly affects output delay, and also has an impact on processor use. The impact depends on segment length, so the optimal block size varies. Any block size below 2048 samples is lower than the JND (see section 2.5), which implies no noticeable delay for performers.

MaxProcs sets the maximum amount of parallel processes. Fewer processes lowers the load on the processor, and makes the output less complex and time-smeared. Higher amounts are closer to the sound of convolution. Note that the number of needed processes never exceeds $\log_2 N_{\text{max}}$, so increasing this parameter past that has no noticeable effect, see section 6.4.1.

Chapter 6

Discussion

The ultimate goal for this thesis was to make a sound effect that uses an algorithm which performs convolution between two audio signals in real-time. As described in section 2.1.1, general convolution between two signals x_1 and x_2, will result in a signal $y(n)$, computed as

$$y(n) = \sum_{k=-\infty}^{\infty} x_1(k)x_2(n-k). \tag{6.1}$$

The equation is restated here for practical reasons. When seen in context with the goals of this thesis, stated in section 1.1, usage of equation 6.1 in the algorithm implies some inherent constraints.

The limits in the summation span from $-\infty$ to ∞. Thus, computation with equation 6.1 implies usage of every sample from start to end, in both x_1 and x_2, whenever a sample is to be sent out of the audio effect. Because the audio effect is to operate in real time, it is subject to the constraint that future samples are not available, and the straightforward usage of equation 6.1 is therefore impossible.

A natural way to handle the fact that future samples are not available, would be to completely omit the usage of future samples. The output signal $y(n)$ could then be computed as,

$$y(n) = \sum_{k=-\infty}^{n} x_1(k)x_2(n-k). \tag{6.2}$$

Straightforward usage of equation 6.2 implies the use of every former sample from the input. The consequence is that the computation power needed will eventually grow beyond what is available. In addition, output produced by all the input samples up to this point would be intolerably muddy for all but the most ethereal performances, and put little emphasis on what is happening in the present moment. The usage of all previous samples is therefore not applicable.

One way to ensure that the computational power is kept within an acceptable limit, as well as putting a stronger emphasis on what is currently happening, is to omit some of the former samples every time new samples are included in the computation. All algorithms explored in this thesis employ some variation of this idea.

6.1 Preliminary Algorithm

The preliminary algorithm is an implementation that was done in the beginning of this project to gain insight into real-time convolution in general and to identify future problems that might be encountered. An explanation on how the preliminary algorithm works can be found in section 4.1. It was shown early in the process that the preliminary algorithm fails to meet the criteria stated in section 1.1. The following sections describe the problems inherent in this method.

6.1.1 Why the Preliminary Algorithm Fails

Tests with single sines of equal frequency and phase on both of the inputs were sufficient to show that the preliminary algorithm fails to meet the solution criteria. They showed that the buffer sizes have to be adjusted to the frequency of the input signals, for the effect give acceptable output. Convolving two sines of equal frequency should ideally result in a sine identical to the input, but this was not the case.

The files *440AsInput_BufferX.wav* are generated with the preliminary algorithm using buffer sizes of X samples. From listening to these it is clear that the results generated with buffer sizes of 100, 300 and 500 samples are most similar to what we expected. Fig. 5.1a and 5.2a show that the outputs with buffer sizes of 100 samples and 300 samples, respectively, is similar to a sine. Fig. 5.3a and the associated sound file show that the output generated with a buffer size of 350 samples is far from a sine.

If F_s is the sampling frequency, one period of a sine tone with frequency F will contain

$$SiP = \frac{F_\text{s}}{F} \text{ [samples]}. \tag{6.3}$$

Since a 440 Hz sine tone contains $\frac{44100}{440} = 100.2273 \approx 100$ samples in each period, all of the files generated with buffer sizes close to integer multiples of a period of the input sines, sound as expected. However, when changing the buffer size to not being an integer multiple of a period of the input signals, the results change dramatically. The result with buffer size of 350 samples exemplifies this. Heavy side lobes are produced in the output. To avoid this, the buffer size would have to be adjusted to be a multiple of the period of the input signals. Musical signals rarely only contain single frequencies, and the buffer lengths would therefore need to be adaptive to all the frequency components in the input signals, which is infeasible. An attempt to deal with this was fading, described in the following section.

6.1.2 Independent Buffer Sizes, Overlap on Output and Fading of Overlap

The preliminary algorithm has the ability of having independent buffer sizes on the two input signals. The idea was that the buffer sizes would be determined by a transient analysis in later versions, and the signals could have transients in different locations. In such an implementation, different buffer sizes could occur, and therefore it was desirable that the algorithm was able to handle it. The preliminary algorithm provides two different modes on how samples are drawn out of the input signals and put in the buffers, namely the `SkipOnSmall` mode and the `OverlapOnLarge`. It also has different modes on how to put convolution results on the output. The `overAdd_small` mode was meant to be used along with the `OverlapOnLarge` buffer mode, and the `overAdd_large` mode was meant to be used along with the `SkipOnSmall`.

`SkipOnSmall` was quickly abandoned, as it would mean throwing away parts of the signal with the shortest block length before it was ever processed. `OverlapOnLarge`, however, caused clicking with a period equal to the large block size. This can be heard in *440AsInput_B_1_2000_B_2_100_NoFade.wav*. To avoid this, as well as avoiding the side lobes mentioned in section 6.1.1, fading between output buffers was introduced, see section 4.1.3. The fading that gave the best results was `expFade2`, which can be heard in *440AsInput_B_1_2000_B_2_100_expFade2.wav*. The clicking was strongly attenuated, but the side lobes remained if the short block size was not a multiple of the input sine period. An example of this is *440SinesAsInput_B_1_2000_B_2_150_expFade2.wav*.

6.2 Algorithm Version 1

To make the convolution more efficient, a frequency domain algorithm was introduced. The algorithm in this version was based on Istvan Varga's `ftconv`. It is a hybrid of straight time domain convolution and traditional frequency domain methods. It has several advantages over both of these approaches. The downfall of time domain convolution is its processing time of $O(n^2)$, while the problem of traditional frequency domain multiplication is that the entire signal must be buffered up before it can be Fourier transformed and subsequently multiplied. See section 2.2 for more informations. `ftconv` finds a middle-ground between these, allowing one to partition the signals into blocks before the FFT is performed, and resulting in a delay of only the block size. This allows for long segments to be convolved efficiently, and with little output delay. It will also be shown in section 6.4 that this method, combined with the extensions of later versions, lead to the *convolution events* defined in section 4.4.1.

One disadvantage is that for the FFT to be maximally efficient, the blocks in the segments must have a length that is 2^n. This is because we use a radix-2 FFT. This limits the flexibility and ease of use of the algorithm, as it is not intuitive for a musician why the block sizes are so severely limited. It also demands that we have two settings for the segment length, the block size L_B and the block number N. Ideally the segment length would be given in seconds, which is a unit anyone can relate to. In the GUI for the final implementation, the segment length is given in seconds, and the block number is rounded to the closest fit, see section 5.5.3.

`ftconv` is meant to be used with one constant impulse response stored in a table, and a live signal being continuously sent into the opcode. Because `ftconv` originally convolves inputs of equal length for every iteration, it was deemed impractical to allow for segments of independent length. Unlike what is allowed in the preliminary algorithm, it was decided that the segments for both signals would have equal length.

When the impulse response is replaced by a changing live signal, it is not immediately obvious how the segments should be updated. The first attempted approach was to have a fixed L_s, and throw the oldest block pairs out every time a new pair is inserted. Testing revealed several disadvantages to this naive method; *delayed changes*, *indistinct transients*, and *destructive interference* leading to side lobes.

6.2.1 Delayed Change

Each time a new FT block pair enters the segment, they are first multiplied with the oldest block pair, and are not multiplied with each other until they reach the middle of the segment after about $L_\mathrm{s}/2$ samples. We can also see that we have a delay of $L_\mathrm{s}/2$ samples before there is output in fig. 5.6 and 5.5. Not only does this make transitions between chords slow and gradual, but with long segments this could lead to harmonically distinct parts, such as parts in different keys, being convolved, resulting in disharmonic output. While not necessarily an unwanted result, it would be unfortunate if it was very difficult for musicians to avoid this without forcing them to have a very low block size. An example may be heard in *disharmonyFromDelayedChange.wav*, where *synth.wav* is convolved with itself.

6.2.2 Indistinct Transients

The lack of a dynamic segment length has a great effect on the distinctiveness of the transients in a signal. This is apparent in *indistinctTransientsSynthDrumloop2.wav*, where *synth.wav* and *drumloop2.wav* are used as input. The result is plotted in fig. 5.6.

Another situation in which transients would be indistinct is when a signal of approximately constant amplitude, for example a synthesizer pad, enters the segment. When the signal first enters the segment, most blocks will again be zero. Assuming that the other channel is for this period filled with a constant signal, the output will keep growing in amplitude until the first channel's segment has been filled with the new signal. Instead of getting a sharp attack and a constant amplitude out, the output amplitude would not be stabilized until after L_{s} samples. This is apparent in fig. 5.5, where a sharp and distinct transient on the input rises slowly on the output. It can be heard in the beginning of *disharmonyFromDelayedChange.wav*.

6.2.3 Destructive Interference

The perhaps most alarming problem with this naive approach is the heavy destructive interference it introduces. The interference was observed to have the form of an amplitude modulation with a frequency of $1/L_{\mathrm{B}}$ samples, which is similar to the distortion in the preliminary algorithm. The reason for this can most easily be illustrated with a simple sine as an example. As can be seen in fig. 5.7, even if the input signals are two identical sines which are in phase, the output blocks are not necessarily in phase. The phase of the output is dependent on the block size, resulting in interference if the input period does not match the block length. This is of course the case in most instances, as a musical signal is often a harmonically complex signal containing many frequencies at one time. Fig. 5.8 shows that the results are in phase when the sines have a period with a rational relationship to the block size.

6.3 Transient Detection (Algorithm Version 2)

To deal with delayed change and indistinct transients, transient detection was introduced. With this addition, the slow rise of amplitude described in 6.2.2 is strongly diminished, while the disharmonic convolution described in 6.2.1 is completely done away with, assuming that harmonically distinct sections are separated by a transient in one of the signals. This is also important to give performers immediate feedback.

It is obvious by comparing fig. 5.9 with fig. 5.6 that the transients on the output are much more similar to the input. This can also be heard in *drumloop2_synth_version2_transDet.wav*.

What does remain is the destructive interference. It was however observed that the interference only occurred when the segments reached a full state before a new transient was detected. This is audible in *Git1Akk_Syn1Akk_ver2_blockSize512_blockNum100.wav*, where one can hear that $L_\mathrm{B}N = 51200$ samples (1.16 seconds) after the transient, disharmony is introduced. In this state, this version functions exactly as version one. That is, it takes in one block pair, and throws away the segments' oldest block pair. Once a transient is detected, the interference disappeared until the full state was reached once again. It was also observed that the throwing away of all old blocks when a transient occurs resulted in a thinner, if clearer, sound than version 1. It is also problematic, with respect to the stated criteria, that the blocks are thrown away without finishing their processing, as it entails a sound that is less close to convolution. For this reason, parallel processes were introduced.

6.4 Parallel Processes (Algorithm Version 3)

To find a more sophisticated way to handle blocks thrown after a transient detection, several methods were explored. After testing, `ThrowAll` was deemed to be the best choice. It will be shown that using this throwing algorithm will lead to the realization of convolution events, which is mathematically equivalent to convolution of segments between transients. See section 2.3 for a mathematical proof of this.

6.4.1 Alternative 1: `ThrowAll` (Used in Final Implementation)

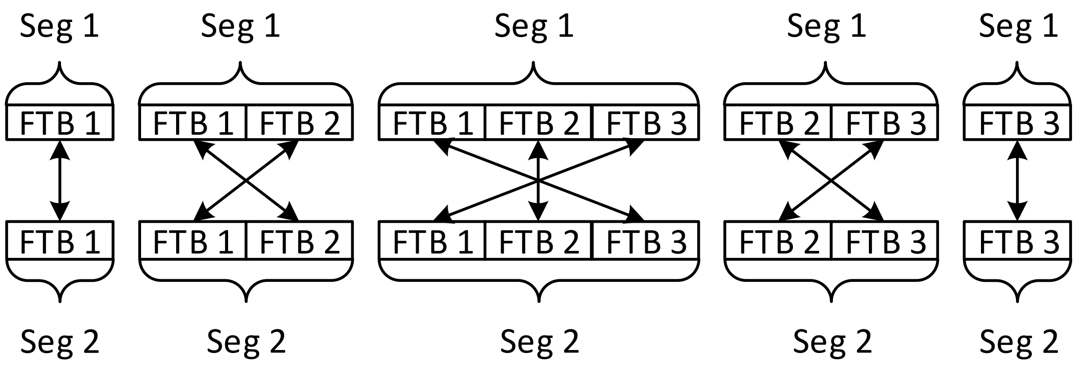


FIGURE 6.1: Example of a process where a transient is detected after three blocks have entered. The arrows denote multiplications. Notice that FT Block pair 1 exits the process first, followed by FT Block pair 2, etc. This illustrates five iterations.

As shown in section 2.3, in the blockwise convolution of a segment of finite length, the oldest blocks are convolved first, and the newest blocks are convolved last. Once the length of a segment has been established, i.e. when a new transient is detected, it follows that the oldest blocks will eventually have been convolved with every block of the other signal, and may be discarded. As it turns out, this point is reached exactly when a transient occurs, that is T_{N+1} is exactly one sample after $E_{1,N}$ (from eq. (2.15) and (2.16)). This means that block 1 may be discarded immediately after a transient is detected. Note that the proof in section 2.3 operates in the time domain, but since convolution in the time domain is equivalent to multiplication in the frequency domain, this is not a problem. Since we use a frequency domain algorithm, we will use multiplication of FT blocks interchangeably with convolution of time domain blocks in this particular section. See section 2.1.2 for more information on this.

Fig. 6.1 shows an example of a process's lifespan, in this case a transient is detected after three blocks have entered the process. Comparing it with the terminology stated in section 2.3, this is equivalent to a segment with $N = 3$ blocks. At first, we multiply only the first FT block pair, then the second FT block pair enters and is multiplied with the first FT block pair. Once the first FT block pair has been multiplied with all the other pairs, it may be discarded. This happens exactly when a transient is detected and the process is made semi-active. Once all FT block pairs have been multiplied with all the other pairs, the convolution is complete, and the process becomes inactive.

This algorithm is equivalent to the blockwise convolution described in 2.3. That blockwise convolution is mathematically equivalent to the convolution of two segments, and the limits of these segments are decided by transients. Consequently, we have arrived at a convolution event.

There is, however one special case in which a necessary precaution must be taken. The user sets N_{max}, which is the maximum number of blocks allowed in a segment. There must be a limit, since if no transients occur, the segments would grow infinitely in length, and both infinite memory and processing power would be necessary. Therefore, if N_{max} is reached, the active process is turned into a semi-active process, and a new active process is created. This has two benefits: The processing power used can be limited by the user, and since we never reach a state in which one block pair is thrown out every time a one block pair is taken in, the destructive interference described in all previous versions is highly attenuated. This can be heard in *440HzSine_FinalVersion_512_BS_100B.wav*, as well as *gitar1akkord_synth1akkord_FinalVersion.wav*. In fig. 5.13 and 5.14, one can see an AM distortion. However, in contrast with previous versions, it has a period of L_s. This is usually far below the audible frequency range ($<$ 20 Hz)[18, p. 1], and causes no sidelobes, as one can see from the FFT plot. This sort of AM distortion can be mostly done away with by using a compressor, and is not a big problem with complex signals containing several frequencies, like *gitar1akkord_synth1akkord_FinalVersion.wav*. The lack of sidelobes ensures that no unnatural disharmony is introduced.

Since a convolution is not complete until after a transient is detected[1], there may be several convolutions going on in parallel. This is handled by creating semi-active processes. The process in fig. 6.1 becomes semi active after it receives a transient, and then starts throwing blocks.

The maximum amount of needed processes is equal to $\log_2 N_{\max}$, because all processes(except the active process) throw one block per iteration. The maximum size a process can attain before becoming semi-active is $N_{\max}$. To get the maximum amount of parallel processes, one has to get each process as long as possible; this happens when one process is filled to the max, then a transient occurs when the following process is half full (resulting in two processes with $N_{\max}/2$ blocks. Then a transient occurs when the next process has $N_{\max}/4$ blocks, and so forth. This leads to maximally $\log_2 N_{\max}$ processes in parallel.

The transients are much clearer than in Version 1, but a little less clear than in version 2. However, since we include a parameter limiting the maximum amount of processes, the user may choose to use only 1 process, and the result is the same as Version 2, without the destructive interference. The difference can be seen in fig. 5.15 and 5.16, and heard in *drumloop2_synth_FinalVersion_10Process.wav* and *drumloop2_synth_FinalVersion_1Process.wav*

[1]The length of a convolution result is twice as long as the input, see eq. (2.2)

Combining the parallel processes and the process handling method used in `ThrowAll`, a method for live convolution which meets all the technical goals proposed in section 1.1 has been developed!

6.4.2 Alternative 2: `ThrowLast`

`ThrowLast` is an alternative way of handling the parallel processes. It's functionality is described in chapter 4.4.2. It differs from the `ThrowAll` version in that it throws away FT blocks more rarely. As opposed to the `ThrowAll` version, which throws away one FT block pair from every semi-active process for every iteration, the `ThrowLast` version only throws away one FT block pair from the oldest semi-active process. If transients occur frequently, several processes will be running in parallel in both of the versions. If several processes are running in parallel in the `ThrowAll` version, several FT blocks will be thrown away while only one new FT block pair is added. The `ThrowLast` version was developed because it was not obvious why the algorithm should throw away more FT blocks than the amount added. If blocks were thrown away more rarely, the output would be generated based on more information. We believed that an output based on more information would be more likely to sound fuller and more musically pleasing. At least it would increase the probability of having overlap in frequency between the two input signals.

Tests with the `ThrowLast` version showed that it has a clear weakness. If several processes are running in parallel, the segments in the semi-active processes will not be altered before their associated process become the oldest one. This results in that they contribute to the output with the exact same convolution result in subsequent iterations. This leads to unwanted periodicity which is audible and which dominates the output. The section 5.4.2 presents some plots and where to find audio files which show this phenomenon. The unwanted periodicity has a fundamental frequency whose period is the block size L_B. For example, in *ThrowLast256+SingleSine172Hz.wav*, we get

$$f_{\text{fundamental}} = \frac{F_{\text{s}}}{L_{\text{B}}} = \frac{44100[\frac{\text{samples}}{\text{s}}]}{256[\text{samples}]} = 172.265625[\text{Hz}]. \tag{6.4}$$

6.4.3 Alternative 3: `TwoProc`

Before the final real-time implementation was done, we thought that memory management in the `ThrowAll` version and the `ThrowLast` version would be unnecessarily complicated, due to several processes running in parallel. We thought that a version with maximally two processes running in parallel would simplify the memory management significantly, and perhaps not negatively affect the output. Special cases of the `ThrowAll` version or the `ThrowLast` version where the maximum allowed processes is set to two, would both accomplish this, but they would both also result in versions where processes are prematurely ended if transients occur frequently. The idea with the `TwoProc` version was to have an implementation that only threw away one FT Block pair for every pair added, while only having maximally two processes running in parallel.

One reason for choosing the `ThrowAll` version over the `TwoProc` version was the mathematical analysis that was done in section 2.3. This mathematical analysis proves that `ThrowAll` realizes actual convolution of segments between transients in the input signals, or in the jargon of this thesis, the realization of *convolution events*. The way the `TwoProc` version handles transients results in a convolution method that suddenly jumps forward in time, resulting in a messy output. To understand this concept, imagine that in one iteration, both of the processes in `TwoProc` contains several FT Blocks. If a transient is present during the next iteration, the oldest FT Blocks in the old process will be multiplied with FT Blocks associated with a point in time that is way beyond the point in time associated with the FT Blocks with which they were multiplied in the last iteration.

Another reason for not choosing the `TwoProc` version came from simple tests with a drum loop and a synthesizer as input signals, and with transient detection on the drum loop to ensure that new processes were started frequently. See section

5.4.3 for information on the test results. When listening to these audio files, it is clear that the old process dominates the output too much if the segment size is large. The larger the segment size is, the more dominating the old process will be. `TwoProc` could have been usable if strict restrictions were set on the segment size. This would obviously lead to an effect with fewer possibilities. It was therefore concluded that the reduced complexity of the memory management in a real-time implementation of `TwoProc` would not be worth the reduction of possibilities it would involve when comparing them with the possibilities of `ThrowAll`.

6.4.4 Level Control and Normalization

In convolution, there is a build up and a fading out of amplitude, called the "tails". In most applications, the impulse response one convolves a signal with is significantly shorter than the signal. The tails occur when there isn't a complete overlap between the signal and the impulse response, and have lengths approximately equal to the impulse response. One usually normalizes the impulse response so that there is no amplification of the output, something which may be done because the impulse response is known and unchanging.

In our algorithm, none of these assumptions apply. The impulse response is neither shorter than the signal, nor known in advance. In fact, the "impulse response" has equal length to the signal, resulting in output which is basically two tails, with no midsection where the amplitude stabilizes. An example can be seen in the bottom of fig. 5.7. One can see the amplitude increasing and decreasing. The output amplitude may become very large as well, since normalization of the "impulse response" cannot be performed beforehand.

To deal with these problems, we have developed the *Total Block Normalization* method, as described in 4.4.4, and illustrated in fig. 4.18. With this method all FT Block pairs will have equal contribution to the output. More specifically, they will be weighted with a factor of $1/B_{\text{Tot}}$, where B_{Tot} is the total number of FT Block pairs on which the output is based.

Using this method, the output amplitude does not build up as the number of blocks in a process increases. In addition, this keeps the amplitude from growing unchecked.

6.5 Computational Complexity

To understand how the different parameters affect the processing time, an analysis of the computational complexity is needed. In the derivation of the complexity, the transient analysis and handling of processes is ignored, as the main load on the processor is the multiplication and addition of blocks.

The following pseudocode gives a simplified overview of the total algorithm:

```
Perform FFT on new input block pair

for (all blocks in active and semi-active processes)
    for (every sample in an FT block)
        perform multiplication and addition of FT block sample
    end
end

Perform IFFT on output block
```

A radix-2 FFT has a computational complexity of $(N/2)\log_2 N$ complex multiplications, and $N\log_2 N$ complex additions [5, p. 519-526]. Each block has L_B samples,but they are also padded with L_B zeros, resulting in a length of $2L_\mathrm{B}$. There are two input blocks which must be Fourier-transformed, ergo the complexity of this operation is $2((2L_\mathrm{B}/2)\log_2 2L_\mathrm{B}) = 2L_\mathrm{B}\log_2 2L_\mathrm{B}$ complex multiplications and $2(2L_\mathrm{B}\log_2 2L_\mathrm{B}) = 4L_\mathrm{B}\log_2 2L_\mathrm{B}$ complex additions.

The multiplication and addition of block samples happens B_Tot times, where B_Tot is the sum of the number of block pairs contained in all of the active and semi-active processes. There are $2L_\mathrm{B}$ complex samples in each FT block which will all be multiplied and added, so this for-loop has a complexity of $2B_{\mathrm{Tot}}L_{\mathrm{B}}$ complex multiplications and $2B_{\mathrm{Tot}}L_{\mathrm{B}}$ complex additions.

The IFFT must only be performed on a single block, and has the same complexity as an FFT, and therefore the complexity of the last operation is $L_{\mathrm{B}}\log_2 2L_{\mathrm{B}}$ complex multiplications and $2L_{\mathrm{B}}\log_2 2L_{\mathrm{B}}$ complex additions.

The total complexity is $3L_{\mathrm{B}}\log_2 2L_{\mathrm{B}} + 2B_{\mathrm{Tot}}L_{\mathrm{B}}$ complex multiplications and $6L_{\mathrm{B}}\log_2 2L_{\mathrm{B}} + 2B_{\mathrm{Tot}}L_{\mathrm{B}}$ complex additions. $B_{\mathrm{Tot}} \leq N_{\mathrm{max}}$, so the total complexity C is

$$\begin{aligned} C \leq\ & 3L_{\mathrm{B}}\log_2 2L_{\mathrm{B}} + 2N_{\mathrm{max}}L_{\mathrm{B}}[\text{complex multiplications}] \\ & + \\ & 6L_{\mathrm{B}}\log_2 2L_{\mathrm{B}} + 2N_{\mathrm{max}}L_{\mathrm{B}}[\text{complex additions}]. \end{aligned} \tag{6.5}$$

The computational complexity is therefore $O(L_{\mathrm{B}}\log L_{\mathrm{B}} + N_{\mathrm{max}}L_{\mathrm{B}})$.

6.5.1 Computational Complexity Versus Output Delay

The time available to the processor, every time it is waiting for a new block pair to be buffered, is $L_{\mathrm{B}}/F_{\mathrm{s}}$ seconds. Dividing this by the operation number stated in eq. (6.5), and assuming that a multiplication takes k times as long as an addition, we get

$$T(L_{\mathrm{B}}, N_{\mathrm{max}}) = \frac{1}{F_{\mathrm{s}}((3k+6)\log_2 2L_{\mathrm{B}} + (2k+2)N_{\mathrm{max}})}[\text{s/operation}]. \tag{6.6}$$

To best see how the block size affects processing time, it is appropriate to consider $L_{\mathrm{Smax}} = N_{\mathrm{max}}L_{\mathrm{B}}$, as the segment length is, in fact, what is relevant to a musician. The segment length, after all, is what decides how many seconds of audio can be convolved at a time. Substituting N_{max} with $L_{\mathrm{Smax}}/L_{\mathrm{B}}$, we get

$$T(L_{\mathrm{B}}, L_{\mathrm{Smax}}) = \frac{L_{\mathrm{B}}}{F_{\mathrm{s}}((3k+6)L_{\mathrm{B}}\log_2 2L_{\mathrm{B}} + (2k+2)L_{\mathrm{Smax}})}[\text{s/operation}]. \tag{6.7}$$

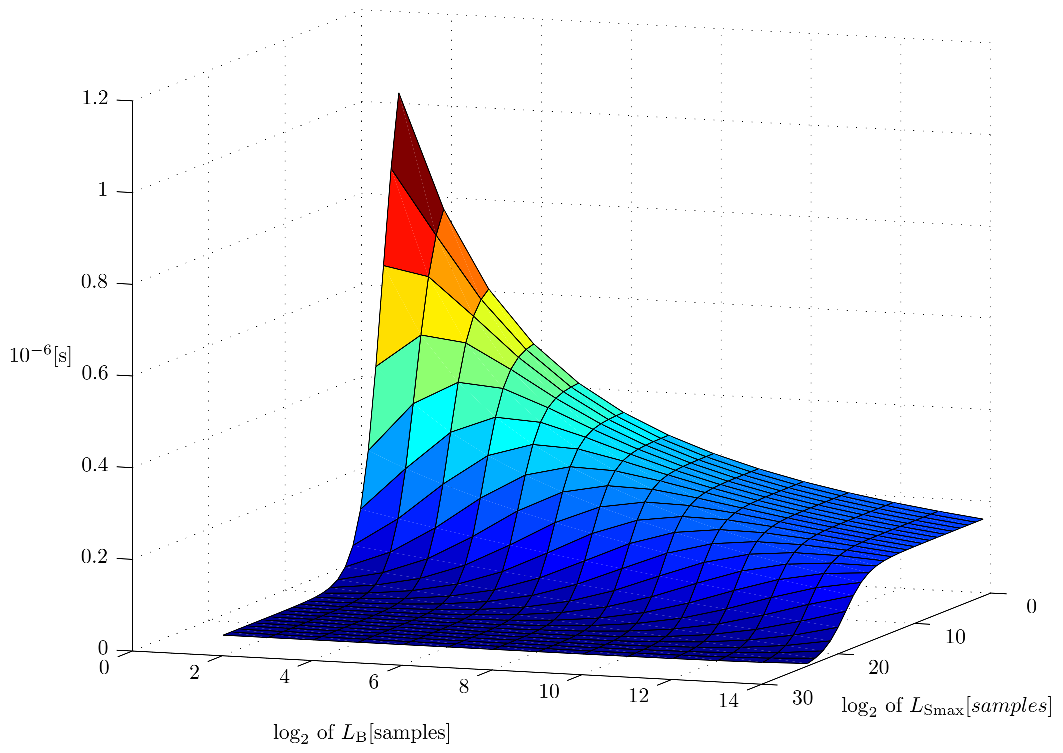


FIGURE 6.2: Plot of time available to the processor per operation, with logarithmic axes, $\log_2 L_\text{Smax}$ versus $\log_2 L_\text{B}$, generate with eq. (6.7).

This equation has been graphically represented in fig. 6.2, assuming $k = 1$, which is reasonable for an x86-processor. Here, high values are good, i.e. the processor has more time available per operation. From fig. 6.2, one can see that a higher block size improves the runtime only when $L_\text{Smax} > 2^5 = 32$, and even above that, the block size only lightens the CPU load up to a certain size, after which it starts increasing again. Typical values for L_Smax range from 2^{12} to 2^{18} samples (0.1 to 6 seconds), ideal block sizes range from $2^{12} = 512$ to $2^{14} = 16384$ samples. Still, even block sizes down to 64 samples leave time in the order of 10^{-7} seconds per operation, which is in the MFLOPS range, a fairly low load in these times. Even on the most extreme settings, on a five year old laptop, no more than 40% of the CPU is used if $L_\text{B} = 256$.

6.6 Esthetic Considerations

This section gives a qualitative description of the effect, as well as suggestions for areas of use.

6.6.1 Characteristics of the Effect

The frequency-multiplicative properties of convolution have several consequences which are important to the sonic characteristics of the output.

Much like cross synthesis[13, p. 518], our cross convolver typically results in more energy in the lower frequency bands. This is due to most tonal instruments having overtones which have less energy than the fundamental frequency. Because the frequencies of the two input signals are multiplied, high frequency content which is low in amplitude becomes even lower on the output. To deal with this, we suggest using an equalizer, which alleviates this symptom considerably.

The dynamic range of the output is also considerable. We get very high amplitude output if the fundamental frequencies of the input signals overlap. Conversely, if there is only overlap with overtones/subharmonics, or fundamentals which do not quite match, the output can be very low. We suggest using a compressor or limiter on the output to alleviate this problem.

Another important characteristic is the time-smearing effect of convolution. In our algorithm, the convolution settings (block size, segments length and maximum process number) can be used to control the smearing. With longer segment lengths and higher numbers of processes there is a greater degree of smearing. Transient detection also affects time smearing, and frequent detections lead to a less smeared output.

6.6.2 Areas of Application

One obvious application in which the effect can be used is in Live Electronics, a performance category which is associated with improvisation and experimentation on new ways to generate and process sound. One example of performers operating within this category is T-EMP (Trondheim ensemble for Electroacoustic Music Performance) consisting of people from The Music Technology section at NTNU Department of Music. They have already been experimenting with the use of live convolution [3].

We also believe that the effect can be used in pop-music. In pop-music, a great deal of the final result is done through post-processing. Even though the effect has been designed to operate in real-time, there is no inherent limitations in the effect that prevents it from being used in post-processing.

Another application in which the effect can be used is in art installations. The effect has great possibilities of making floating sound landscapes, like the sound landscapes found in ambient music. This also leads us to believe that the effect can be used to make abstract sound effects in movies and radio plays.

6.6.3 The Effect in Action

Several examples of musicians using the effect have been contributed, and can be found in the *Music Created Using The Effect* folder in the digital appendix.

Thomas Etholm-Kjeldsen's music falls within the pop genre, using the effect with guitars, drums and synthesizers. The effect input/output has also been put in separate files, so that one may hear the contributions the effect has made to the music. The effect was applied in post-production.

Jakob Eri Myhre and Olaf Mundal's work is improvisational, live electronics, using a trumpet (Myhre) and a guitar (Mundal). These only use the output of the effect in the final work. The effect was used live.

Jakob Eri Myhre's solo contributions are fleeting soundscapes, which could be used in film or art installations. They use piano, flutes, and synthesizers, among other instruments. The effect was applied in post-production.

Our own contribution, as *Conwolves*, is a short pop tune, using vocals, guitar, drums, synthesizer, and violin as input to the effect. The effect was applied in post-production.

Chapter 7

Future Work

The algorithm developed in this thesis provides a framework for live convolution. It is conceptually divided into the following parts: Input buffering, transient analysis, signal segmentation, convolution engine, and output buffer. In our implementation, choices have been made for how these parts act and interact, choices which may or may not be suitable for every application. There is still room for experimentation and improvements, some of which are suggested here.

7.1 Independent Segment Length

In our final implementation, the convolution engine works with segments that are separated from each other by the transient analysis component. It was decided to keep the segment length of both signals equal in each process, and therefore start a new segment in both signals each time a transient is detected on either signal, mainly for practical implementation reasons. This need not necessarily be the case. There may be a relevant way in which to keep the segment lengths independent. This might lead to more rhythmically interesting output.

7.2 MIDI-Controlled Segmentation

An alternative to transient analysis is using MIDI trigger signals to segment the signals. A possible solution is to synchronize the segmentation to the DAW's clock, performing a segmentation at every 1/4 beat, 1/6 beat, or any other time signature. This could give interesting rhythmic effects to two stationary inputs.

7.3 Zero-Delay FFT-Based Convolution

A method for zero-delay FFT-based convolution has been proposed [19]. This technique can be used in combination with our block convolver at longer block sizes to provide zero delay, and keeping the efficient nature of FFT-based convolution.

7.4 Automatic Gain Control

A problem with the current algorithm is that output amplitudes may vary wildly, depending on the frequency content of the input. If there is frequency overlap only on the overtones of one of the signals, or if the overlap is imperfect (e.g. two tones which are close, but not exactly the same), output will be weak. If the fundamental frequencies of periodic signals align, output amplitudes may be a thousandfold higher. This is not intuitive for musicians, who are more used to additive frequency interaction between instruments than multiplicative. It would be advantageous to find a way to control the dynamic range of the output. At this point, we have had reasonable success with using a compressor on the output, but perhaps a more tailored solution would be in order.

7.5 Input Amplitude Thresholding for Computational Efficiency

As should be obvious by this point, if there is zero input on one channel, there is zero output. To save unnecessary computations, one could stop frequency multiplications once the input level on one or both of the channels falls below a threshold, potentially saving lots of power, which could be useful in a battery driven hardware implementation.

Chapter 8

Conclusion

An algorithm for live convolution has been developed. At the outset of this thesis, several goals were set. The algorithm was to:

- Use convolution, and sound like convolution
- Run in real time
- Be intuitively usable for musicians

The algorithm was mathematically proven to give out the convolution of live signals in real time, given that the signals are segmented somehow. We segment the signals using transient analysis, which offers musically relevant segmentation, and allows musicians to control the segmentation simply by playing their instruments. The output delay may be less than 1ms at the extreme, and performs reasonably well with about 6ms of delay, which is well below what most people are able to distinguish. The two first goals have been reached.

We have created a GUI, and implemented the effect as a VST plug-in. These are both important steps for making it accessible to musicians. As shown by the music examples attached to this thesis, the effect has been successfully used in a several contexts, including post-processing for popular music, and experimental work, as well as live electroacoustical improvisation. The musicians were given minimal instructions, and we therefore have reason to believe that the last goal seems to have been reached, although more user tests are needed.

We hope that this effect will be of use to someone, and that convolution has been revealed to have another useful trick up its sleeve.

Appendix A

Final Implementation

A.1 Csound Code

In the following code, the Cabbage code ranges from line 1-39. The transient detection section is from line 114-152, and the opcode is called on line 200.

```
1  <Cabbage> ;Code within the Cabbage tags describe the GUI
2  form size(623, 310), caption("LaivConv"), pluginID("Laiv")
3  ;form size(623, 435), caption("LaivConv"), pluginID("Laiv")
4  image bounds(0, 0, 623, 310), file("background.jpg"), ...
       shape("round")
5
6  groupbox bounds(0, 0, 80, 310), text("Input Gain"), ...
       fontcolour("white")
7       rslider channel("input1Gain"),  bounds(0,51,70,70), ...
       text("Gain 1"), range(0,1, 0.5, 1, 0.01), tracker(113,171,236)
8       rslider channel("input2Gain"),  bounds(0,238,70,70), ...
       text("Gain 2"), range(0,1, 0.5, 1, 0.01), tracker(113,171,236)
9  groupbox bounds(81, 0, 380, 310), text("Transient Detection"), ...
       fontcolour("white")
10
11 groupbox bounds(86, 27, 370, 19), colour(0,0,0,0), ...
       text("Channel 1"), fontcolour("white")
```

```
     rslider channel("transRelease1"),   bounds(90,51,70,70), ...
    text("Rel [s]"), range(0.1, 0.4, 0.3, 1, 0.01), ...
    tracker(113,171,236); envelope follower release
     rslider channel("transThresh1"),    bounds(160,51,70,70), ...
    text("Thresh [db]"), range(1, 15, 7, 1, 0.1), ...
    tracker(113,171,236) ; attack threshold (in dB)
     rslider channel("transLowThresh1"),     ...
    bounds(230,51,70,70), text("LowThresh"), range(0.1, 1, 0.5, ...
    1, 0.01), tracker(113,171,236) ; lower threshold for ...
    transient detection (adaptive)
     rslider channel("transMintime1"),   bounds(300,51,70,70), ...
    text("MinTime [s]"), range(0.01, 1, 0.05, 1, 0.01), ...
    tracker(113,171,236) ; minimum duration between events, ...
    (double trig limit)
     rslider channel("transMonitor1"),   bounds(370,51,70,70), ...
    text("TransMonitor1"), range(0, 1, 0, 1, 0.01), ...
    tracker(113,171,236) ; Volume of transient monitor1
groupbox bounds(86, 214, 370, 19), text("Channel 2"), ...
    fontcolour("white")
     rslider channel("transRelease2"),   bounds(90,238,70,70), ...
    text("Rel [s]"), range(0.1, 0.4, 0.3, 1, 0.01), ...
    tracker(113,171,236) ; envelope follower release
     rslider channel("transThresh2"),    bounds(160,238,70,70), ...
    text("Thresh [db]"), range(1, 15, 7, 1, 0.1), ...
    tracker(113,171,236) ; attack threshold (in dB)
     rslider channel("transLowThresh2"),     ...
    bounds(230,238,70,70), text("LowThresh"), range(0.1, 1, ...
    0.5, 1, 0.01), tracker(113,171,236) ; lower threshold for ...
    transient detection (adaptive)
     rslider channel("transMintime2"),   bounds(300,238,70,70), ...
    text("MinTime [s]"), range(0.01, 1, 0.05, 1, 0.01), ...
    tracker(113,171,236) ; minimum duration between events, ...
    (double trig limit)
     rslider channel("transMonitor2"),   bounds(370,238,70,70), ...
    text("TransMonitor2"), range(0, 1, 0, 1, 0.01), ...
    tracker(113,171,236) ; Volume of transient monitor
groupbox bounds(462, 0, 80, 310), text("Convolution"), ...
    fontcolour("white")
```

```
     rslider channel("MaxNumBlocks"),    bounds(467,25,70,70), ...
    text("MaxSegLen"), range(0.1, 6, 1.5, 1, 0.01), ...
    tracker(113,171,236) ;Maximum number of blocks
     combobox channel("BlockSize"),      bounds(462,105,80,70), ...
    caption("Block Size"),value(5), items(16, 32, 64, 128, 256, ...
    512, 1024, 2048, 4096, 8192, 16384) ;Maximum number of blocks
     rslider channel("MaxNumProcs"),     bounds(467,185,70,70), ...
    text("MaxProcs"), range(1, 10, 10, 1, 1), ...
    tracker(113,171,236) ;Maximum number of blocks
button channel("reInit"),           bounds(467,265,70,20), ...
    text("Reinit"),  colour(255,255,0), value(0), fontcolour("red")

groupbox bounds(543, 0, 80, 310), text("Output Mix"), ...
    fontcolour("white")
     rslider channel("input1OutGain"),   bounds(548,25,70,70), ...
    text("Dry Input 1"), range(0, 1, 0, 1, 0.01), ...
    tracker(113,171,236) ;Gain on dry output 1
     rslider channel("input2OutGain"),   bounds(548,95,70,70), ...
    text("Dry Input 2"), range(0, 1, 0, 1, 0.01), ...
    tracker(113,171,236) ;Gain on dry output 2
     rslider channel("convOutGain"),     bounds(548,165,70,70), ...
    text("Conv Output"), range(0, 10, 0.5, 1, 0.01), ...
    tracker(113,171,236) ;Gain on convolution output
     rslider channel("totalOutGain"),    bounds(548,235,70,70), ...
    text("Total Gain"), range(0, 1, 0.5, 1, 0.01), ...
    tracker(113,171,236) ;Gain on total mix

checkbox channel("transientDisplay1"),      ...
    bounds(442,75,10,10), value(0)
checkbox channel("transientDisplay2"),      ...
    bounds(442,262,10,10), value(0)

;csoundoutput bounds(0,315, 623, 120), text("Output")
</Cabbage>

<CsoundSynthesizer>

<CsOptions>
```

```
—d
—n
</CsOptions>

<CsInstruments>

sr = 44100 ;sampling rate

ksmps = 8 ;control rate

0dbfs = 1 ;max volume ref.

nchnls = 2

chn_a "input1", 1 ;input channel 1
chn_a "input2", 1 ;input channel 2
chn_a "MasterOut", 2 ;output channel
chn_a "TransientMonitor1", 2 ;Transient Monitor 1 channel
chn_a "TransientMonitor2", 2 ;Transient Monitor 1 channel

;—————————————————————————————————————————————————————————————————
;—————Instrument that takes in signal from the microphone ————————
;—————————————————————————————————————————————————————————————————
instr 1 ;Takes in input signals and send them to their ...
    respective channels
aIn1, aIn2 ins
        chnmix aIn1, "input1"
        chnmix aIn2, "input2"
endin

instr 98 ;take in signals, do processing

aIn1    chnget "input1"
aIn2    chnget "input2"
kGain1  chnget "input1Gain"
kGain2  chnget "input2Gain"

iResponse = 50 ; response time in milliseconds
```

```

aIn1    =   aIn1*kGain1
aIn2    =   aIn2*kGain2

;Parameters for transient detection of channel 1


    kAttack1    =   0.0001          ; envelope follower attack
    kRelease1   chnget "transRelease1"      ; envelope ...
   follower release
    kAtck_db1   chnget "transThresh1"       ; attack threshold ...
   (in dB)
    kLowThresh1 chnget "transLowThresh1"    ; lower threshold ...
   for transient detection (adaptive, relative to recent ...
   transient strength)
    kdoubleLimit1   chnget "transMintime1"      ; minimum ...
   duration between events, (double trig limit)
    kTransMon1  chnget "transMonitor1"

;Parameters for transient detection of channel 2

    kAttack2    =   0.0001          ; envelope follower attack
    kRelease2   chnget "transRelease2"      ; envelope ...
   follower release
    kAtck_db2   chnget "transThresh2"       ; attack threshold ...
   (in dB)
    kLowThresh2 chnget "transLowThresh2"    ; lower threshold ...
   for transient detection (adaptive, relative to recent ...
   transient strength)
    kdoubleLimit2   chnget "transMintime2"      ; minimum ...
   duration between events, (double trig limit)
    kTransMon2  chnget "transMonitor2"

;reinitialize i-rate variables for transient detection every ...
   second.
start:
    timout 0, 1, continue
    iLowThresh1 = i(kLowThresh1)
    idoubleLimit1 = i(kdoubleLimit1)
    iLowThresh2 = i(kLowThresh2)
    idoubleLimit2 = i(kdoubleLimit2)
    reinit start
continue:
;analyze transients. Transient analysis part mostly written by ...
    Ã¿yvind Brandtsegg
#define analyzeTransients(N)#
    aFollowIn$N follow2 aIn$N, kAttack$N, kRelease$N
    kFollowIn$N downsamp aFollowIn$N
    kFollowdbIn$N    = dbfsamp(kFollowIn$N)          ; convert ...
    to dB
    kFollowDelIn$N  delayk  kFollowdbIn$N, iResponse/1000   ; ...
    delay with response time for comparision of levels

    kTrigIn$N   init 0
    kLowThresh$N    init 0
    kTrigIn$N   = ((kFollowdbIn$N > kFollowDelIn$N + ...
    kAtck_db$N) ? 1 : 0)   ; if current rms plus threshold is ...
    larger than previous rms, set trig signal to current rms

    ; avoid transient detection of very soft signals (adaptive ...
    level)
    if kTrigIn$N > 0 then
    kLowThresh$N    = (kLowThresh$N *0.7)+(kFollowIn$N ...
    *0.3)        ; (the coefficients can be used to adjust ...
    adapt rate)
    endif

    kTrigIn$N   = (kFollowIn$N > kLowThresh$N *iLowThresh$N ? ...
    kTrigIn$N : 0)

    ; avoid closely spaced transient triggers (first trig ...
    priority)
    kDouble$N   init 1
    kTrigIn$N   = kTrigIn$N * kDouble$N
    if kTrigIn$N > 0 then
    reinit double$N
    endif
    double$N:
    kDouble$N   linseg  0, idoubleLimit$N, 0, 0, 1, 1, 1


    ; amplitude
    kamp$N      = kFollowIn$N

    ; delay triggers to sync with amp analysis
    ;kTrigIn$N  delayk  kTrigIn$N, 0.02
    kTransient$N    = kamp$N * kTrigIn$N
;; split transients and sustain
    aTransient$N    upsamp kTrigIn$N
    aTransEnv$N follow2 aTransient$N, 0.001, 0.2;   a$Ndel  ...
       delay a$N, iResponse/1000
;   asplitTrans$N   = a$Ndel * aTransEnv$N
;   asplitSustain$N = a$Ndel * (1-aTransEnv$N)
#

$analyzeTransients(1)
$analyzeTransients(2)
rireturn

chnset kTransMon1*aIn1*aTransEnv1, "TransientMonitor1"
chnset kTransMon2*aIn2*aTransEnv2, "TransientMonitor2"

;initial settings for convolution knobs
kBlockSizeSamp  init    256
kBlockSize      init    5
kMaxNumProc     init    10
kBlocks         init    258
kNormMode       init    1

kBlockSize  chnget "BlockSize"
kSegLen     chnget "MaxNumBlocks"
kMaxNumProc chnget "MaxNumProcs"
kButton     chnget "reInit"
kNormMode   chnget "NormMode"
```

```



;convert from combobox output to blocksize. Convert from ...
    seconds to block number
kBlockSizeSamp  pow 2,kBlockSize+3
kBlocks      =   round(kSegLen*44100/kBlockSizeSamp)

;initial settings for laivconv opcode
iBlockSize  init    256
iBlockNum   init    258
iMaxNumProc init    10
iNormMode   init    1
iSkipInit   =   1


kParamChange changed kButton
if (kParamChange == 1) then
    reinit beginReinit
endif

beginReinit:
iBlockSize  =   i(kBlockSizeSamp)
iBlockNum   =   (i(kBlocks) > 1 ? i(kBlocks) : 1)
iMaxNumProc =   i(kMaxNumProc)
iNormMode   =   i(kNormMode)


;run laivconv opcode
aOut        laivconv    aIn1, aIn2, iBlockSize, iBlockNum, ...
    iMaxNumProc, kTransient1+kTransient2, iSkipInit

rireturn

    chnmix aOut, "MasterOut"



;Control segment for LEDs displaying transient detection:
```

```
kTransButton1   changed kTransient1
kTransButton2   changed kTransient2
kBEnv1 init 0
kBEnv2 init 0

if kTransButton1 > 0 then
    reinit letsGo1
endif

if kTransButton2 > 0 then
    reinit letsGo2
endif

chnset kBEnv1, "transientDisplay1"
chnset kBEnv2, "transientDisplay2"

letsGo1:
kBEnv1  linseg 1,0.15,1,0,0,0
rireturn

letsGo2:
kBEnv2  linseg 1,0.15,1,0,0,0
rireturn

endin

instr 99 ;Output mixer
aOut    chnget  "MasterOut"
aIn1    chnget  "input1"
aIn2    chnget  "input2"
aMon1   chnget  "TransientMonitor1"
aMon2   chnget  "TransientMonitor2"

kIn1Amp chnget  "input1OutGain"
kIn2Amp chnget  "input2OutGain"
kCnvAmp chnget  "convOutGain"
kTAmp   chnget  "totalOutGain"

```

```
a0  = 0

;sends mixed output to output channel
outs    ...
    3*kTAmp*(kCnvAmp*aOut+kIn1Amp*aIn1+kIn2Amp*aIn2+aMon1+aMon2) ...
    , 3*kTAmp*(kCnvAmp*aOut+kIn1Amp*aIn1+kIn2Amp*aIn2+aMon1+aMon2)
    chnset a0, "MasterOut"
    chnset a0, "input1"
    chnset a0, "input2"
endin
</CsInstruments>

<CsScore>

i1 0 3600
i98 0 3600
i99 0 3600

</CsScore>

</CsoundSynthesizer>
```

LISTING A.1: The Csound code for the final implementation.

A.2 Opcode `laivconv`

```
/*
    ftxconv.c:

    Copyright (C) 2005 Istvan Varga

    This file is part of Csound.

    The Csound Library is free software; you can redistribute it
    and/or modify it under the terms of the GNU Lesser General ...
   Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later ...
   version.

    Csound is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied ...
   warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General ...
   Public
    License along with Csound; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
    02111-1307 USA
*/

#include "csdl.h"
#include <math.h>

/*
** ftxconv - data structure holding the internal state
*/

typedef struct {
  int start1;
  int end1;

  int start2;
  int end2;
  int length;
}pIndex;

typedef struct {

/*
```

```
**  Input parameters given by user
*/
  OPDS    h;
  MYFLT   *aOut;                  // output buffer
  MYFLT   *aIn1;                   // input buffer 1
  MYFLT   *aIn2;                   // input buffer 2

  MYFLT   *iBlockLen;              // length of impulse ...
    response partitions (latency <-> CPU usage)
  MYFLT   *iMaxNumBlocks;                // Number of blocks in ...
    segment
  MYFLT   *iMaxNumProcesses;

  MYFLT   *kTransDet;               //For signaling if a ...
    transient has been detected
  MYFLT   *iSkipInit;             // skip initialization ...
    without failure (optional, default 0)


 /*
 ** Internal state of opcode maintained outside
 */
  int     initDone;                    /* flag to indicate ...
    initialization */
  int     cnt;                         /* buffer position, 0 to ...
    blockLen - 1        */
  int     maxNumBlocks;                 /* number of convolve ...
    partitions             */
  int         maxNumProcesses;          /*Maximum number of ...
    processes running in parallel*/
  int     blockLen;                    /* partition length in ...
    sample frames (= iBlockLen as integer) */
  int     rbCnt1;                       /* ring buffer index 1, ...
    0 to maxNumBlocks - 1  */
  int     rbCnt2;                       /* ring buffer index 2, ...
    0 to maxNumBlocks - 1  */
  int         transDetected;                 /*Initialized to ...
    0, set to 1 if transient is detected (then returned to 0)*/
  int          oldestProcess;                /*Process number ...
    for the oldest active process*/
  int          newestProcess;                /*Process number ...
    for the newest active process*/

  /* The following pointer point into the auxData buffer */
  MYFLT   *tmpBuf;                        /* temporary buffer for ...
    accumulating FFTs   */
  MYFLT   *ringBuf1;                       /* ring buffer of FFTs ...
    of input partitions - these buffers are now computed during ...
    init */
  MYFLT   *ringBuf2;                       /* impulse responses ...
    (scaled)       */
  MYFLT   *outBuf;                        /* output buffer ...
    (size=blockLen*2)  */
  pIndex    *processIndex;

  AUXCH   auxData;                        /* Aux data buffer ...
    allocated in init pass */
} ftxconv;

/*
**  Function to multiply the FFT buffers
**    outBuf - the output of the operation (called with ...
    tmpBuf), single channel only
**    ringBuf - the partitions of the single input signal
**    IR_data - the impulse response of a particular channel
**    blockLen - size of partition
**    maxNumBlocks - number of partitions
**    ringBuf_startPos - the starting position of the ring buffer
**                          (corresponds to the start of the ...
    partition after the last filled partition)
*/
static void multiply_fft_buffers(MYFLT *outBuf, MYFLT ...
    *ringBuf1, MYFLT *ringBuf2, int blockLen, int maxNumBlocks, ...
    int ringBuf1_startPos, int ringBuf2_startPos, int length)
{
  MYFLT   re, im, re1, re2, im1, im2;
MYFLT   *rbPtr1, *rbPtr2, *outBufPtr, *outBufEndPm2, ...
 *rbEndP1, *rbEndP2;

/* note: blockLen must be at least 2 samples */
blockLen <<= 1; /* locale blockLen is twice the size of the ...
 partition size */
outBufEndPm2 = (MYFLT*) outBuf + (int) (blockLen - 2);  /* ...
 Finding the index of the last sample pair in the output ...
 buffer */
rbEndP1 = (MYFLT*) ringBuf1 + (int) (blockLen * ...
 maxNumBlocks);   /* The end of the ring buffer 1*/
rbEndP2 = (MYFLT*) ringBuf2 + (int) (blockLen * ...
 maxNumBlocks);   /* The end of the ring buffer 2*/
rbPtr1 = &(ringBuf1[ringBuf1_startPos*blockLen]);             ...
                /* Initialize ring buffer 1 pointer */
rbPtr2 = &(ringBuf2[ringBuf2_startPos*blockLen]);             ...
                /* Initialize ring buffer 2 pointer */
outBufPtr = outBuf;                                           ...
  /* Initialize output buffer pointer */

/* do { */
/*   *(outBufPtr++) = FL(0.0); */
/*   *(outBufPtr++) = FL(0.0); */
/* } while (outBufPtr <= outBufEndPm2); */

/*
** Multiply FFTs for each partition and mix to output buffer
** Note: IRs are stored in reverse partition order
*/
do {
  /* wrap ring buffer position */
  if (rbPtr1 >= rbEndP1)
    rbPtr1 = ringBuf1;
  if (rbPtr2 >= rbEndP2)
    rbPtr2 = ringBuf2;

  outBufPtr = outBuf;
```

```
    *(outBufPtr++) += *(rbPtr1++) * *(rbPtr2++);    /* ...
   convolve DC - real part only */
    *(outBufPtr++) += *(rbPtr1++) * *(rbPtr2++);    /* ...
   convolve Nyquist - real part only */
    re1 = *(rbPtr1++);
    im1 = *(rbPtr1++);
    re2 = *(rbPtr2++);
    im2 = *(rbPtr2++);

    /*
    ** Status:
    ** outBuf + 2, ringBuf + 4, irBuf + 4
    ** re = buf + 2, im = buf + 3
    */

    re = re1 * re2 - im1 * im2;
    im = re1 * im2 + re2 * im1;
    while (outBufPtr < outBufEndPm2) {
      /* complex multiply */
      re1 = rbPtr1[0];
      im1 = rbPtr1[1];
      re2 = rbPtr2[0];
      im2 = rbPtr2[1];
      outBufPtr[0] += re;
      outBufPtr[1] += im;
      re = re1 * re2 - im1 * im2;
      im = re1 * im2 + re2 * im1;
      re1 = rbPtr1[2];
      im1 = rbPtr1[3];
      re2 = rbPtr2[2];
      im2 = rbPtr2[3];
      outBufPtr[2] += re;
      outBufPtr[3] += im;
      re = re1 * re2 - im1 * im2;
      im = re1 * im2 + re2 * im1;
      outBufPtr += 4;
      rbPtr1 += 4;
      rbPtr2 += 4;
```

```
        /*
        ** Status:
        ** outBuf + 2 + 4n, ringBuf + 4 + 4n, irBuf + 4 + 4n
        ** re = buf + 2 + 4n, im = buf + 3 + 4n
        */
      }
      outBufPtr[0] += re;
      outBufPtr[1] += im;

    } while (--length);
}
static inline int buf_bytes_alloc(int blockLen, int ...
    maxNumBlocks, int maxNumProcesses)
{
      int nSmps;

      nSmps = (blockLen << 1);                          /* ...
    tmpBuf      */
      nSmps += ((blockLen << 1) * maxNumBlocks);          ...
    /* ringBuf1     */
      nSmps += ((blockLen << 1) * maxNumBlocks);          ...
    /* ringBuf2     */
      nSmps += ((blockLen << 1));
    /* outBuf */

      return ((int) sizeof(MYFLT) * nSmps + (int) ...
    sizeof(pIndex)*maxNumProcesses);
}
static void set_buf_pointers(ftxconv *p, int blockLen, int ...
    maxNumBlocks)
{
      MYFLT *ptr;

      ptr = (MYFLT*) (p->auxData.auxp);
      p->tmpBuf = ptr;
      ptr += (blockLen << 1);
      p->ringBuf1 = ptr;
      ptr += ((blockLen << 1) * maxNumBlocks);
    p->ringBuf2 = ptr;
    ptr += ((blockLen << 1) * maxNumBlocks);
    p->outBuf = ptr;
  ptr += (blockLen << 1);
  p->processIndex = (pIndex*) ptr;

}

static void throwBlock(pIndex* process, int maxNumBlocks){

  (*process).start1++;
  (*process).start2--;

  /*Check that start indexes are not out of bounds*/
  if ((*process).start1 >= maxNumBlocks) (*process).start1 = 0;
  if ((*process).start2 < 0) (*process).start2 = maxNumBlocks-1;
  (*process).length--;
}

static void addBlock(pIndex *process, int rbCnt1, int rbCnt2){

  (*process).end1=rbCnt1;
  (*process).end2=rbCnt2;
  (*process).length++;
}

static void updateOldBuffers(pIndex* processIndex, int ...
    *oldestProcess, int newestProcess, int maxNumBlock, int ...
    maxNumProcesses){
  //kaster en blokk fra alle semiaktive. sjekker om oldest er ...
    blitt tom, og setter den saa til en ny oldest/newest.
  int index = (*oldestProcess);

  while (index != newestProcess){
    if (processIndex[index].length != 0){
      throwBlock((processIndex+index),maxNumBlock); //Update ...
    start, decrement length
    }
```

```
      if (processIndex[index].length == 0 && index == ...
      *oldestProcess) {
        *oldestProcess = (*oldestProcess +1)%maxNumProcesses;
      }
      index = (++index)%maxNumProcesses;
    }
}

static void startNewProcess(int *newestProcess, int ...
    *oldestProcess, pIndex* processIndex, int maxNumProcesses, ...
    int rbCnt1, int rbCnt2){

  /*Increment newestProcess index*/
  *newestProcess= (*newestProcess + 1)%maxNumProcesses;

  if(*newestProcess == *oldestProcess)
    oldestProcess++;

  /*Update indexes, set length to 1*/
  processIndex[*newestProcess].length = 0; //length will be ...
    set to one in addblock
  processIndex[*newestProcess].start1 = rbCnt1;
  processIndex[*newestProcess].start2 = rbCnt2;
  addBlock((processIndex+*newestProcess), rbCnt1, rbCnt2);

}

static int ftxconv_init(CSOUND *csound, ftxconv *p)
{
  int     n, nBytes;



  if(p->initDone) csound->Warning(csound, Str("Init kjort ...
    etter at init ble satt!"));

  /* set p->blockLen to the initial partition length, ...
    iBlockLen */
  p->blockLen = MYFLT2LRND(*(p->iBlockLen));
  if (UNLIKELY(p->blockLen < 4 || (p->blockLen & (p->blockLen ...
    - 1)) != 0)) {  // Must be a power of 2 at least as large ...
    as 4
    return csound->InitError(csound, Str("ftxconv: invalid ...
    impulse response "
                                         "partition length"));
  }


  /* Calculate the total length  */
  n = MYFLT2LRND(*(p->iBlockLen) * *(p->iMaxNumBlocks));
  if (UNLIKELY(n <= 0)) {
    return csound->InitError(csound,
                             Str("ftxconv: invalid length, or ...
    insufficient"
                                 " IR data for convolution"));
  }

  // Compute the number of partitions (total length / ...
    partition size)
  p->maxNumBlocks = MYFLT2LRND(*(p->iMaxNumBlocks));
  p->maxNumProcesses = MYFLT2LRND(*(p->iMaxNumProcesses));

  /*
  ** Calculate the amount of aux space to allocate (in bytes) ...
    and allocate if necessary
  ** Function of partition size and number of partitions
  */
  nBytes = buf_bytes_alloc(p->blockLen, p->maxNumBlocks, ...
    p->maxNumProcesses);
  if (nBytes != (int) p->auxData.size)
    csound->AuxAlloc(csound, (int32) nBytes, &(p->auxData));
  else if (p->initDone > 0 && *(p->iSkipInit) != FL(0.0))
    return OK;    /* skip initialisation if requested */

  /*
  ** From here on is initialization of data
  */


  /* initialise buffer pointers */
  set_buf_pointers(p, p->blockLen, p->maxNumBlocks);

  /* clear ring buffers to zero */
  n = (p->blockLen << 1) * p->maxNumBlocks;
  memset(p->ringBuf1, 0, n*sizeof(MYFLT));
  memset(p->ringBuf2, 0, n*sizeof(MYFLT));

  /* initialise buffer index */
  p->cnt = 0;
  p->rbCnt1 = 0;
  p->rbCnt2 = 0;

  /* clear output buffers to zero (why not use memset here?) */
  memset(p->outBuf, 0, (p->blockLen << 1)*sizeof(MYFLT));

  /*Initialize process indexes to 0*/
  memset(p->processIndex, 0, p->maxNumProcesses*sizeof(pIndex));

  /*Initialize transient detection*/
  p->transDetected = 0;
  p->oldestProcess = 0;
  p->newestProcess = 0;

  /*
  ** After initialization:
  **    Buffer indexes are zero
  **    tmpBuf is filled with rubish
  **    ringBuf and outBuf are filled with zero
  **    IR_Data is filled with the FFT of the impulse response
  */
  printf("FFTScale is: %f", ...
    csound->GetInverseRealFFTScale(csound, (p->blockLen << 1)));
  p->initDone = 1;
  return OK;
```

```
}

static int ftxconv_perf(CSOUND *csound, ftxconv *p)
{
    MYFLT         *x, *rBuf1, *rBuf2;
    int           i, n, nSamples, normalizeFactor;
    int           m = csound->ksmps;    // Size of audio buffer

    MYFLT         FFTscale;

    normalizeFactor = 0;
    /* Only continue if initialized */
    if (p->initDone <= 0) goto err1;

    /*Check for transients*/
    if (*(p->kTransDet) != FL(0.0)){
      printf("!!Transient detected!! it is %f\n", ...
    *(p->kTransDet));
      p->transDetected = 1;
    }



    /*If maxNumBlocks is reached, throw process*/
    if (p->processIndex[p->newestProcess].length >= ...
    p->maxNumBlocks){
      p->transDetected = 1;
    }

    nSamples = p->blockLen;   /* Length of partition */
    rBuf1 = &(p->ringBuf1[p->rbCnt1 * (nSamples << 1)]); /* ...
    Pointer to a partition of ring buffer 1*/
    rBuf2 = &(p->ringBuf2[p->rbCnt2 * (nSamples << 1)]); /* ...
    Pointer to a partition of ring buffer 2*/

    /* FFT amplitude scale, trivial function */
    //FFTscale = csound->GetInverseRealFFTScale(csound, ...
    (p->blockLen << 1)); //Why is this always == 1.000?
    //FFTscale = 1000/(*(p->iBlockLen));
```

```
 //printf("FFTscale is: %f\n\n",FFTscale);


 /* For each sample in the audio input buffer (length = ...
ksmps) */
 for (n = 0; n < m; n++) {

   /* store input signal in buffer, scale for FFT */
   rBuf1[p->cnt] = p->aIn1[n];//*FFTscale;
   rBuf2[p->cnt] = p->aIn2[n];//*FFTscale;

   /* copy output signals from buffer (contains data from ...
previous convolution pass) */
   p->aOut[n] = p->outBuf[p->cnt];

   /* is input buffer full ? */
   if (++p->cnt < nSamples)
     continue;                    /* no, continue with next ...
sample */

   /* Now the partition is filled with input --> start ...
calculate the convolution */
   p->cnt = 0; /* reset buffer position */

   /* pad input in ring buffer with zeros to double length */
   for (i = nSamples; i < (nSamples << 1); i++){
     rBuf1[i] = FL(0.0);
     rBuf2[i] = FL(0.0);
   }

   /* calculate FFT of input */
   csound->RealFFT(csound, rBuf1, (nSamples << 1));
   csound->RealFFT(csound, rBuf2, (nSamples << 1));

   if (p->transDetected){

     /*A transient is detected */
```

```
        /*Initialize start- and end-indexes, and newestProcess ...
    index*/

        //Inc newestprocessindex and Initialize start- and ...
    end-indexes, set length to 1, (check maxNumProcess)(sjekk ...
    om det skrives over en aktiv proc)
        startNewProcess(&(p->newestProcess), ...
    &(p->oldestProcess), p->processIndex, p->maxNumProcesses, ...
    p->rbCnt1, p->rbCnt2);

        //Throw one block from all semiactive processes, i.e ...
    update start indexes, and update oldestProcess
        updateOldBuffers(p->processIndex, &(p->oldestProcess), ...
    p->newestProcess, p->maxNumBlocks, p->maxNumProcesses);


      }else{
       /*A transient is not detected*/
        if(p->oldestProcess==p->newestProcess){
        /*Only one active process*/
        /*When here, newestProcess can still be full or empty*/
        /*therefore check length against maxNumOfBlocks*/

          ...
    if(p->processIndex[p->newestProcess].length==p->maxNumBlocks){

            /*Active process is full*/

            /*update end indexes, i.e. add one more block*/
            addBlock((p->processIndex+ p->newestProcess), ...
    p->rbCnt1, p->rbCnt2);

            /*update start indexes, i.e. throw out one block*/
            throwBlock((p->processIndex + ...
    p->newestProcess),p->maxNumBlocks);


          }else{
```

```

          /*Active process is not full, update only ...
end−indexes*/
          /*We only need to take in a fftblock in active ...
process*/

          addBlock((p−>processIndex + p−>newestProcess ), ...
p−>rbCnt1, p−>rbCnt2);
        }
      }else{

        /*Active process is not full, update only end−indexes*/
        /*We only need to take in a fftblock in active process*/

        addBlock((p−>processIndex + p−>newestProcess), ...
p−>rbCnt1, p−>rbCnt2);

        //Throw one block from all semiactive processes, i.e ...
updatere start indexes
        updateOldBuffers(p−>processIndex, ...
&(p−>oldestProcess), p−>newestProcess, p−>maxNumBlocks, ...
p−>maxNumProcesses);
      }
    }

    /* update ring buffer position */
    p−>rbCnt1++;
    p−>rbCnt2−−;
    if (p−>rbCnt1 >= p−>maxNumBlocks)
      p−>rbCnt1 = 0;
    if (p−>rbCnt2 < 0)
      p−>rbCnt2 = p−>maxNumBlocks−1;

    /* clear output buffer to zero */
    memset(p−>tmpBuf, 0, sizeof(MYFLT)*((nSamples << 1)−2));

    /* multiply complex arrays −−> multiplication in the ...
frequency domain */
```

```
    if (p->oldestProcess == p->newestProcess){

      multiply_fft_buffers(p->tmpBuf, p->ringBuf1, ...
p->ringBuf2, nSamples, p->maxNumBlocks, ...
p->processIndex[p->newestProcess].start1, ...
p->processIndex[p->newestProcess].end2, ...
p->processIndex[p->newestProcess].length);
      normalizeFactor = ...
p->processIndex[p->newestProcess].length;
    }else{
      i = p->oldestProcess-1;
      do{
        i = (++i)%p->maxNumProcesses;
        if (p->processIndex[i].length){
          multiply_fft_buffers(p->tmpBuf, p->ringBuf1, ...
p->ringBuf2, nSamples, p->maxNumBlocks, ...
p->processIndex[i].start1, p->processIndex[i].end2, ...
p->processIndex[i].length);
          normalizeFactor += p->processIndex[i].length;
        }

      }while (i != p->newestProcess);
    }

    /* inverse FFT */
    csound->InverseRealFFT(csound, p->tmpBuf, (nSamples << 1));



    /*
    ** Copy IFFT result to output buffer
    ** The second half is left as "tail" for next iteration
    ** The first half is overlapped with "tail" of previous ...
block
    */
    x = &(p->outBuf[0]);
    for (i = 0; i < nSamples; i++) {
        x[i] = ...
    (p->tmpBuf[i]/*FFTscale*/)/((MYFLT)normalizeFactor) + x[i + ...
    nSamples]; //Normalize output by number of blocks added ...
    together (normalizeFactor)
        x[i + nSamples] = (p->tmpBuf[i + ...
    nSamples]/*FFTscale*/)/((MYFLT)normalizeFactor);
      }
    p->transDetected = 0;
    }
    return OK;

 err1:
    return csound->PerfError(csound, Str("ftxconv: not ...
    initialised"));
}

/* module interface functions */

static OENTRY localops[] = {
    {
        "laivconv",             // name of opcode
        (int) sizeof(ftxconv),  // data size of state block
        TR|5,                   // thread
        "a",                    // output arguments
        "aaiiiOp",             // input arguments
        (int (*)(CSOUND *, void *)) ftxconv_init,    // init ...
    function
        (int (*)(CSOUND *, void *)) NULL,           // k-rate ...
    function (?)
        (int (*)(CSOUND *, void *)) ftxconv_perf   // a-rate ...
    function
    }
};

LINKAGE
```

Listing A.2: The C code for the opcode implementation.

Appendix B

Matlab Implementations

This appendix contains the Matlab implementations. The files can also be found in the digital appendix delivered with this thesis. In the digital appendix, the different versions are arranged into separate folders, one folder for each version. Some of the versions use the same files, but each folder contain all the files needed to run each version so there is no need to move files around. For the version that uses transient detection, the script transDet2.m has to be run on the audio files before the main script is run.

B.1 Preliminary Algorithm

The preliminarty algorithm can be run with the main script in listing B.1. It uses the `buffBlend` function shown in listing B.2.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Preliminary algorithm           %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%——————————————————————————————%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%June 2013                       %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Authors: Antoine Henning Bardoz  %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%         Lars Eri Myhre         %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```matlab

% Reads wav from file
% Uses function "buffblend" defined in buffblend.m

%buffMode 'skipOnSmall' should be used with blendMode
%'overAdd_small'
%buffMode 'overlapLarge' should be used with blendMode
%'overAddLarge','expfade', 'expfade2' or 'linfade'
buffMode = 'overlapLarge'; %'overlapSmall',
blendMode = 'overAdd_large'%'overAdd_small',
%'expfade', 'expfade2' or 'linfade';

%output will be played if set
playsound=1;

%Take inn signals
inSig = cell(1,2);
[inSig{1},fs] = wavread('synth.wav');
inSig{2} = wavread('drumloop2.wav');

%Convert to mono
inSig{1} = inSig{1}(:,1)';
inSig{2} = inSig{2}(:,1)';

%Length of signals,
len1 = length(inSig{1})
len2 = length(inSig{2})

%Initialize output
output = [];

%Initsialize buffers (length in samples):
bLen = {2000, 2000};
buffer = cell(1,2);
buffer{1} = zeros(1,bLen{1});
buffer{2} = zeros(1,bLen{2});

%Iteration variable for main loop
```

```matlab
i = 1;

%Main loop, simulating real-time
while ~(i+bLen{1} > len1 ||...
        i+bLen{2} > len2)

    %Buffer up signals
    switch buffMode
        case 'skipOnSmall'
            buffer{1} = inSig{1}(i:i+bLen{1}-1);
            buffer{2} = inSig{2}(i:i+bLen{2}-1);
            i = max(i+bLen{1},i+bLen{2});
        case 'overlapLarge'
            buffer{1} = inSig{1}(i:i+bLen{1}-1);
            buffer{2} = inSig{2}(i:i+bLen{2}-1);
            i = min(i+bLen{1},i+bLen{2});
        case 'oneSample'
            buffer{1} = inSig{1}(i:i+bLen{1}-1);
            buffer{2} = inSig{2}(i:i+bLen{2}-1);
            i = i + 1;
        otherwise
            error('Not a valid buffer mode')
    end

    %Perform convolution on buffers
    temp = conv(buffer{1}, buffer{2});

    %Put output block(temp) on existing output
    output = buffBlend(output, temp, bLen{1}, bLen{2}, blendMode);

end

%play output
if(playsound)
    soundsc(output,fs)
end
```

Listing B.1: Main script for the preliminary algorithm.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%BuffBlend                        %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%---------------------------------%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%June 2013                        %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Authors: Antoine Henning Bardoz  %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%         Lars Eri Myhre          %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Function puts a newly generated output block on the existing
%output
function [ out] = buffBlend( oldOut, newOut, len1, len2, mode )

%Length of smallest buffer
small = min(len1,len2);
%Length of largest buffer
large = max(len1,len2);

if small <= 1
    out = [oldOut newOut];
elseif isempty(oldOut)
    out = newOut;
else
    switch mode

        case 'overAdd_small'
        %newOut is put on output with an overlap equal to the ...
    length
        %of the shortest buffer, i.e min(len1,len2)
            oldOut(length(oldOut)-small+1:end) = ...
                oldOut(length(oldOut)-small+1:end)+ ...
    newOut(1:small);
            out = [oldOut newOut(small+1:end)];

        case 'overAdd_large'
        %newOut is put on output with an overlap equal to the ...
    length
        %of the longest buffer, i.e max(len1,len2)
            oldOut((length(oldOut)-large)+1:end)=...
                ...
    oldOut((length(oldOut)-large)+1:end)+newOut(1:large);
            out = [oldOut newOut((large+1):end)];

        case 'linFade'
        %Uses overlap similar to 'overAdd_large', and linear ...
    fading
            sizes = sort([len1, len2]);
            blendCurve = linspace(1,0,sizes(2)-1);
            oldOut(length(oldOut)-sizes(2)+2:end) = ...
                ...
    blendCurve.*oldOut(length(oldOut)-sizes(2)+2:end) + ...
                              ...
    fliplr(blendCurve).*newOut(1:sizes(2)-1);
            out = [oldOut newOut(sizes(2)+2:end)];

        case 'expFade'
        %Uses overlap similar to 'overAdd_large', and ...
    exponential fading
            ampModOld = logspace(0,-2,(large-1));
            ampModNew = logspace(-2,0,(large-1));
            out = [oldOut(1:end-large+1) ...
                (oldOut(end-large+2:end).*ampModOld +...
                newOut(1:large-1).*ampModNew)...
                newOut(large:small+large-1)];

        case 'expFade2'
        %Uses overlap similar to 'overAdd_large', and ...
    exponential fading
            ampModOld =1-logspace(-100,0,large-1);
            ampModNew =1-logspace(0,-100,large-1);
            out = [oldOut(1:end-large+1) ...
                (oldOut(end-large+2:end).*ampModOld +...
                newOut(1:large-1).*ampModNew)...
                newOut(large:small+large-1)];

        otherwise
```

```
            error('Not a valid blending mode')

    end
end
```

Listing B.2: buffBlend function used in the main script of the preliminary algorithm.

B.2 Algorithm Version 1

Algorithm Version 1 can be run with the script *mainAVersion1*, shown in listing B.3. It uses the functions `fftMulAndAdd`, `expandOutput` and `generateWeight` shown in listing B.4, B.5 and B.6, respectively.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%mainAVersion1                          %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%---------------------------------%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%June 2013                              %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Authors: Antoine Henning Bardoz   %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%         Lars Eri Myhre           %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%This script use the functions "fftMulAndAdd",
%"expandOutput" and "genereteWeight, their .m-files
%have to be in the same folder as this script for
%the script to run

% hear generated sound file-> playSound = 1;
playSound = 1;

%Different weightmodes used in a test which failed. The mode
%'const' does nothing, but have to be set for the code to run.
weightMode = 'const';

%Sampling frequency
```

```
fs = 44100;

%test transient location
transLoc = 68700;

%Block length
blockLen = {512, 512};

%Number of blocks
maxNumOfBlocks = 100;
numOfBlocks = maxNumOfBlocks;

%Load test signals
%include path for files, or put files in same folder as script
inSig = cell(1,2);
[inSig{1},fs] = wavread('drumloop2.wav');
inSig{2} = wavread('synth.wav');

%Convert to Mono
inSig{1} = inSig{1}(:,1)';
inSig{2} = inSig{2}(:,1)';

%Shortest signal
minLen=min(length(inSig{1}),length(inSig{2}));
inSig{1} = inSig{1}(1:minLen);
inSig{2} = inSig{2}(1:minLen);

%Size of ffts must account for length of convolution
fftLen = blockLen{1}+blockLen{2}-1;

%Segment length, should be an integer times block length
segLen = {blockLen{1}*numOfBlocks, blockLen{2}*numOfBlocks};

%Initialize blocks
block = cell(1,2);

%Initialize fft blocks
fftBlock = cell(1,2);
```

```matlab

%Initialize segments
segment = cell(1,2);
segment{1} = zeros(1,segLen{1});
segment{2} = zeros(1,segLen{2});

%Initialize fft segments
fftSegment = cell(1,2);
fftSegment{1} = zeros(1,(numOfBlocks)*fftLen);
fftSegment{2} = zeros(1,(numOfBlocks)*fftLen);

%Initialize output
output = [zeros(1,fftLen)];

%Generate weight vector, used in the failed test. Does nothing if
%weightMode = 'const'
wVector = generateWeight(weightMode,numOfBlocks);

i=1;
while ~((i+blockLen{1} > length(inSig{1})) ||...
        (i+blockLen{2} > length(inSig{2})))

    %Fill up blocks
    block{1} = inSig{1}(i:i+blockLen{1}-1);
    block{2} = inSig{2}(i:i+blockLen{2}-1);

    i = max((i+blockLen{1}),(i+blockLen{2}));

    %fft on blocks
    fftBlock{1} = fft(block{1},fftLen);
    fftBlock{2} = fft(block{2},fftLen);

    %Fill upp fftSegment
    %First fftBlock -> out, second fftBlock -> first fftBlock,
    %third fftBlock -> second fftBlock ...... and so on
    %new fftBlock -> last fftBlock
    fftSegment{1} = [fftSegment{1}(fftLen+1:end) fftBlock{1}];
    fftSegment{2} = [fftSegment{2}(fftLen+1:end) fftBlock{2}];

    %Multiply fftSegments.
    %Function takes inn two fftSegments with numOfBlocks
    %fft blocks each. First block in fftSegment1 is multiplied
    %with last block in fftSegment2 First block in fftSegment2
    %is multiplied with last block in fftSegment1 ...and so on
    %real(ifft(mul_res)) are added together in out vector.
    %out vector is fftSegment1/numOfBlocks long
    temp=fftMulAndAdd(fftSegment{1}, fftSegment{2}, wVector,...
        numOfBlocks,fftLen);

    %Put result on output with buffblend function
    output=expandOutput(output,temp,blockLen{1});

    %Update info on the number of blocks in the segments
    if numOfBlocks < maxNumOfBlocks
        numOfBlocks = numOfBlocks+1;
    end
end

%Play Sound
if playSound
    soundsc(output,fs)
end
```

Listing B.3: Main script for algorithm version 1.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%fftMulAndAdd                    %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%-----------------------------------%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%June 2013                         %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Authors: Antoine Henning Bardoz   %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%         Lars Eri Myhre          %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function out = fftMulAndAdd(fftSegment1, fftSegment2,...
                                wVector, numOfBlocks,fftLen)

%Function takes inn two fftSegments with numOfBlocks fft ...
    results each.
%First block in fftSegment1 is multiplied with last block in ...
    fftSegment2
%First block in fftSegment2 is multiplied with last block in ...
    fftSegment1
%All multiplication results are added together in out vector.
%outvector is fftSegment1/numOfBlocks long

%Initialize output
out=zeros(1,fftLen);

%Number of blocks in segment
maxNumOfBlocks = length(fftSegment1)/fftLen;

%Iterate and multiply segments
for i=1:numOfBlocks
    start = maxNumOfBlocks−numOfBlocks;
    out=out+wVector(i)*real(ifft((fftSegment1(...
         ((start+i−1)*fftLen+1):(start+i)*fftLen)...
              .*fftSegment2(((maxNumOfBlocks−i)*fftLen+1):...
              (maxNumOfBlocks−i+1)*fftLen))));
end

%Normalize,i.e divide by number of blocks in segment
out = out/numOfBlocks;
```

Listing B.4: fftMulAndAdd function used in main script of algorithm version 1.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%expandOutput                                %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%June 2013                                   %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Authors: Antoine Henning Bardoz            %%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%          Lars Eri Myhre          %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Function puts newly generated output on existing output

function out = expandOutput(oldOut, newOut, blockLen)

%Length of existing output
lenOld=length(oldOut);

%New output block overlaps existing ouput
oldOut(lenOld-blockLen+2:lenOld) = ...
    oldOut(lenOld-blockLen+2:lenOld)+newOut(1:blockLen-1);

out = [oldOut newOut(blockLen:blockLen*2-1)];

end
```

Listing B.5: expandOutput function.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%mainAVersion1                      %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%----------------------------------%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%June 2013                          %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Authors: Antoine Henning Bardoz    %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%          Lars Eri Myhre          %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% a function used in a failed test. Should not be considered,
%and does nothing in the 'const' mode. The different modes
%provides different weight for the segments.

function wVector = generateWeight(mode, numOfBlocks)

switch mode
    case 'v_function'
        for i = -(numOfBlocks/2):numOfBlocks/2
```

```
                    wVector(i+numOfBlocks/2+1)=abs(i)+numOfBlocks;
        end
        wVector=(1/max(wVector))*wVector;

    case 'exp_exp'
        if(mod(numOfBlocks,2)==0)
        wVector = [logspace(-1,0,numOfBlocks/2)...
                    logspace(0,-1,numOfBlocks/2)];
        else
        wVector = [logspace(0,-1,floor(numOfBlocks/2)) 10^(-1)...
                    logspace(-1,0,floor(numOfBlocks/2))];
        end
    case 'lin'
        wVector = linspace(0,1,numOfBlocks);
    case 'lin_lin'
        if(mod(numOfBlocks,2)==0)
        wVector = [linspace(1,0,numOfBlocks/2)...
                    linspace(0,1,numOfBlocks/2)];
        else
        wVector = [linspace(1,0,floor(numOfBlocks/2)) 10^(-1)...
                    linspace(0,1,floor(numOfBlocks/2))];
        end
    case 'const'
        wVector = ones(1,numOfBlocks);
end


end
```

Listing B.6: generateWeight function.

B.3 Algorithm Version 2

Algorithm Version 2 can be run with the script *mainAVersion2.m*, shown in listing B.7. It uses the functions `fftMulAndAdd`, `expandOutput` and `generateWeight` shown in listing B.4, B.5 and B.6, respectively. It also uses transient vectors, called `transVec` and `transVec2` for the respective audio files. They can be generated with the script *transDet2.m* shown in listing B.8.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%mainAVersion1                     %%%%%%%%%%%%%%
%%%%%%%%%%%%%%---------------------------------%%%%%%%%%%%%%%%
%%%%%%%%%%%%%June 2013                         %%%%%%%%%%%%%%
%%%%%%%%%%%%%Authors: Antoine Henning Bardoz   %%%%%%%%%%%%%%
%%%%%%%%%%%%%%         Lars Eri Myhre          %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%This scrupt use the functions "fftMulAndAdd", "expandOutput"
%and "genereteWeight, their .m-files have to be in the
%same folder as this script for the script to run

%It also uses vectors transVec and transVec1 that can be
%generated with transDet2.m

% hear generated sound file-> playSound = 1;
playSound = 1;

%Different wightmodes used in a test which failed. The mode
%'const' does nothing, but have to be set for the code to
%run.
weightMode = 'const';

%Sampling frequency
fs = 44100;

%Block length
blockLen = {512, 512};

%Number of blocks
maxNumOfBlocks = 100;
numOfBlocks = maxNumOfBlocks;

%Load test signals
```

```
%include path for files, or put files in same folder as script
inSig = cell(1,2);
[inSig{1},fs] = wavread('drumloop2.wav');
inSig{2} = wavread('synth.wav');

%Load transVec for drumloop2 generated with transdet2.m
load 'drumloop2.mat'

%Don't use transients from inSig{2}
transVec2 = zeros(1,length(inSig{2}));

%Convert to Mono
inSig{1} = inSig{1}(:,1)';
inSig{2} = inSig{2}(:,1)';

%Shortest signal
minLen=min(length(inSig{1}),length(inSig{2}));
inSig{1} = inSig{1}(1:minLen);
inSig{2} = inSig{2}(1:minLen);

%Size of ffts must account for length of convolution
fftLen = blockLen{1}+blockLen{2}-1;

%Segment length, should be an integer times block length
segLen = {blockLen{1}*numOfBlocks, blockLen{2}*numOfBlocks};

%Initialize blocks
block = cell(1,2);

%Initialize fft blocks
fftBlock = cell(1,2);

%Initialize segments
segment = cell(1,2);
segment{1} = zeros(1,segLen{1});
segment{2} = zeros(1,segLen{2});

%Initialize fft segments
```

```
fftSegment = cell(1,2);
fftSegment{1} = zeros(1,(numOfBlocks)*fftLen);
fftSegment{2} = zeros(1,(numOfBlocks)*fftLen);

%Initialize output
output = [zeros(1,fftLen)];

%Generate weight vector, used in the failed test. Does nothing if
%weightMode = 'const'
wVector = generateWeight(weightMode,numOfBlocks);

%Iteration variable for real-time loop
i=1;

%Loop that simulates real-time
while ~((i+blockLen{1} > length(inSig{1})) ||...
        (i+blockLen{2} > length(inSig{2})))

    %Fill up blocks
    block{1} = inSig{1}(i:i+blockLen{1}-1);
    block{2} = inSig{2}(i:i+blockLen{2}-1);

    %If a transient is detected, shorten numOfBlocks to one ...
       block, then
    %keep adding blocks every iteration until you either reach ...
       a new
    %transient or reach maxNumOfBlocks
    if sum(transVec(i:i+blockLen{1}-1))>=1
        numOfBlocks = 1;
    end
    if sum(transVec2(i:i+blockLen{1}-1))>=1
       numOfBlocks = 1;
    end

    %Update iteration variable
    i = max((i+blockLen{1}),(i+blockLen{2}))

    %fft on blocks
```

```
        fftBlock{1} = fft(block{1},fftLen);
        fftBlock{2} = fft(block{2},fftLen);

        %Fill upp fftSegment
        %First fftBlock -> out, second fftBlock -> first fftBlock,
        %third fftBlock -> second fftBlock ...... and so on
        %new fftBlock -> last fftBlock
        fftSegment{1} = [fftSegment{1}(fftLen+1:end) fftBlock{1}];
        fftSegment{2} = [fftSegment{2}(fftLen+1:end) fftBlock{2}];

        %Multiply fftSegments.
        %Function takes inn two fftSegments with numOfBlocks fft ...
       results each.
        %First block in fftSegment1 is multiplied with last block ...
       in fftSegment2
        %First block in fftSegment2 is multiplied with last block ...
       in fftSegment1
        %..and so on
        %real(ifft(mul_res)) are added together in out vector.
        %out vector is fftSegment1/numOfBlocks long
        temp=fftMulAndAdd(fftSegment{1}, fftSegment{2}, wVector, ...
       numOfBlocks,fftLen);

        %Put result on output with buffblend function
        output=expandOutput(output,temp,blockLen{1});

        %Update info on the number of blocks in the segments
        if numOfBlocks < maxNumOfBlocks
            numOfBlocks = numOfBlocks+1;
        end
    end

    %play the output
    if playSound
        soundsc(output,fs)
    end
```

Listing B.7: Main script for algorithm version 2.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%transDet2                              %%%%%%%%%%%%%%
%%%%%%%%%%%%%%---------------------------------%%%%%%%%%%%%%%
%%%%%%%%%%%%%%June 2013                              %%%%%%%%%%%%%%
%%%%%%%%%%%%%%Authors: Antoine Henning Bardoz  %%%%%%%%%%%%%%
%%%%%%%%%%%%%%         Lars Eri Myhre          %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Generates a vector transVec. TransVec will contain info on
%when transients occur in inSig

%clear all;
close all;

%sampling freq
fs = 44100;

%Length of vector that decides the threshold (by calculating rms)
rmsVecLen = 20000;

%Minimum number of samples between transients
minSampBetTrans = (100/1000)*fs;

%Load test signals
inSig = wavread('../../Testsignaler/Drumloop2.wav');

%Convert to Mono
inSig = inSig(:,1)';

inSig=[inSig];
%inSig= inSig(1:length(transVec2));
%Length of test signal
inSigLen = length(inSig);

%inSig = inSig(1:inSigLen/2);
%inSigLen = length(inSig);
%Initialize vector
```

```
rmsVec= zeros(1,rmsVecLen);

%Initialize counter
counter=1;

%Initialize transient vector
transVec = zeros(1,inSigLen);

%Initial rms value
rmsVal = 0;

%Flag that indicates a resent transient
transFlag = 0;
%Counter
transCounter = 0;

while((counter)< inSigLen)

    rmsVec=[rmsVec(2:rmsVecLen) inSig(counter)];

    rmsVal=sqrt(mean(rmsVec.^2));

    if ((transFlag == 0))
        if(abs(inSig(counter))>1.5*rmsVal)
            transVec(counter)=1;
            transFlag = 1;
        end
    elseif ((transFlag == 1) && (transCounter<minSampBetTrans))
        transCounter = transCounter+1;
    else
        transCounter = 0;
        transFlag = 0;
    end

    counter = counter + 1;

end
```

Listing B.8: Script for generating transient vector.

B.4 ThrowAll

The ThrowAll version can be run with the script *mainThrowAll.m*, shown in listing B.9. It uses the functions fftMulAndAdd, expandOutput and generateWeight, shown in listing B.10, B.5, B.6, respectively. ThrowAll also uses transient vectors, called transVec and transVec2 for the respective audio files. They can be generated with the script *transDet2.m* shown in listing B.8.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%mainThrowAll                    %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%---------------------------------%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%June 2013                       %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Authors: Antoine Henning Bardoz  %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%          Lars Eri Myhre         %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%This script uses the functions fftMulAndAdd, expandOutput,
%generateWeight, which can be found in their .m-files.

%It also uses the vectors transVec and transVec2 which can
%be altered with transDet2.m

%If set, the sound file generated will be played
playSound = 1;

%Different wightmodes exist. They were used in a test which
%failed. The mode 'const' does nothing, but have to be set
%for the code to run.
weightMode = 'const';

%Block length
```

```
blockLen = {512, 512};

%Number of blocks4
maxNumOfBlocks = 200;
numOfBlocks = maxNumOfBlocks;

%Number of maximum concurrent processes
maxNumProc = 10;

%Load test signals
inSig = cell(1,2);
[inSig{1},fs] = wavread('drumloop2.wav');
inSig{2} = wavread('synth.wav');

%Load transient vector,transVec,
%for drumloop2, or use transDet2 to generate
%transVec
load 'drumloop2.mat'
%Don't use transient detection for inSig{2}
transVec2 = zeros(1,length(transVec));

%Convert to Mono
inSig{1} = inSig{1}(:,1)';
inSig{2} = inSig{2}(:,1)';

%Normalize
inSig{1}=inSig{1}/max(inSig{1});
inSig{2}=inSig{2}/max(inSig{2});

%Size of ffts must account for length of convolution
fftLen = blockLen{1}+blockLen{2}-1;

%Segment length, should be an integer times block length
segLen = {blockLen{1}*numOfBlocks, blockLen{2}*numOfBlocks};

%Initialize blocks
block = cell(1,2);

```

```
%Initialize fft blocks
fftBlock = cell(1,2);

%Initialize fft segments
fftSegment = cell(maxNumProc,2);
    fftSegment{1,1} = zeros(1,fftLen);
    fftSegment{1,2} = zeros(1,fftLen);
for i = 2:maxNumProc
    fftSegment{i,1} = [];
    fftSegment{i,2} = [];
end

%RunningProcVec is a structarray that contains info about
%the number of running processes 2 -> running, 1 -> running
%but counting down blocks, 0 -> not running the other value
%in each cell is number of blocks in the segment representing
%that process
runningProcVec = struct;

%Initially, one process is active
runningProcVec(1).state=2;
runningProcVec(1).numOfBlocks=1;

%Initialize other processes to zero
for j=2:maxNumProc
    runningProcVec(j).state=0;
    runningProcVec(j).numOfBlocks=0;
end

%Counter that counts the number of recent transients
transCount=1;

%Initialize output
output = zeros(1,blockLen{1});

%Iteration variable
i=1;

```

```
%Number of processes running in paralell
processCounter = 1;

while ~((i+blockLen{1} > length(inSig{1})) ||...
        (i+blockLen{2} > length(inSig{2})))

    %Fill up blocks
    block{1} = inSig{1}(i:i+blockLen{1}−1);
    block{2} = inSig{2}(i:i+blockLen{2}−1);

    %Checks if active process is full. If it is it says that
    % a transient has occured.
    for j=1:maxNumProc
        if(runningProcVec(j).state==2)
            if(runningProcVec(j).numOfBlocks==maxNumOfBlocks)
                transVec(i+10)=1;
            end
        end
    end

    %Check if there is a transient somwhere in these blocks
    %If there is a transient, increment number of processes
    if (sum(transVec(i:i+blockLen{1}−1))>=1) ||...
            (sum(transVec2(i:i+blockLen{1}−1))>=1)
        transCount=mod(transCount,maxNumProc)+1;
        for j = 1:maxNumProc
            if(runningProcVec(j).state==2)
                runningProcVec(j).state=1;
            end
        end
        runningProcVec(transCount).state=2;
        runningProcVec(transCount).numOfBlocks=0;
        processCounter = processCounter+1;
    end

    i = max((i+blockLen{1}),(i+blockLen{2}));

    %fft on blocks
    fftBlock{1} = fft(block{1},fftLen);
    fftBlock{2} = fft(block{2},fftLen);

    %Fill up fft segments according to the info in runningProcVec
    for j = 1:maxNumProc
        switch runningProcVec(j).state
            case 2
                %Process is active and not full. Append segments
                %with newest FT Block pair
                if(runningProcVec(j).numOfBlocks<maxNumOfBlocks)
                    fftSegment{j,1}=[fftSegment{j,1} fftBlock{1}];
                    fftSegment{j,2}=[fftSegment{j,2} fftBlock{2}];
                    runningProcVec(j).numOfBlocks=...
                        runningProcVec(j).numOfBlocks+1;
                else
                %Process is active and full. Append segments with
                %newest FT Block pair. Throw away oldest FT block
                %pair
                    ...
   fftSegment{j,1}=[fftSegment{j,1}(fftLen+1:end)...
                        fftBlock{1}];
                    ...
   fftSegment{j,2}=[fftSegment{j,2}(fftLen+1:end)...
                        fftBlock{2}];
                end
            case 1
                %Process is semi—active.
                runningProcVec(j).numOfBlocks=...
                    runningProcVec(j).numOfBlocks—1;
                if(runningProcVec(j).numOfBlocks==0)
                    %Process is empty, set state to not active
                    %decrement number of processes
                    runningProcVec(j).state=0;
                    processCounter = processCounter—1;
                else
                %Throw away oldest FT block pair
                    ...
   fftSegment{j,1}=[fftSegment{j,1}(fftLen+1:end)];
```

```
                        ...
    fftSegment{j,2}=[fftSegment{j,2}(fftLen+1:end)];
                end
            otherwise
                fftSegment{j,1}=[];
                fftSegment{j,2}=[];
        end
    end

    temp = 0;
    totalBlockNum = 0;
    %Send all semi-aciteve processes and active process to
    %fftMulAndAdd and save all results in temp
    for j = 1:maxNumProc
        if(runningProcVec(j).state~=0)
            wVector = generateWeight(weightMode,...
                runningProcVec(j).numOfBlocks);
            temp=temp+(1/runningProcVec(j).numOfBlocks)*...
                fftMulAndAdd(fftSegment{j,1},fftSegment{j,2}...
                ,wVector, runningProcVec(j).numOfBlocks,fftLen);
            totalBlockNum = totalBlockNum +...
                runningProcVec(j).numOfBlocks;
        end
    end

    %Put new output block on existing output
    output=expandOutput(output,temp,blockLen{1});

end

%play the output
if playSound
    soundsc(output,fs)
end
```

Listing B.9: Main script for the `ThrowAll` version.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%fftMulAndAdd                     %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%------------------------------------%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%June 2013                         %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Authors: Antoine Henning Bardoz  %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%         Lars Eri Myhre          %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function out = fft_MulAndAdd(fftSegment1, fftSegment2,...
                            wVector, numOfBlocks,fftLen)


%Function takes inn two fftSegments with numOfBlocks fft ...
    results each.
%First block in fftSegment1 is multiplied with last block in ...
    fftSegment2
%First block in fftSegment2 is multiplied with last block in ...
    fftSegment1
%All multiplication results are added together in out vector.
%outvector is fftSegment1/numOfBlocks long

%Initialize output
out=zeros(1,fftLen);

%Maximum allowed number of blocks
maxNumOfBlocks = length(fftSegment1)/fftLen;

%Iterate, multiply
for i = 1:numOfBlocks
    out=out+...
    wVector(i)*real(ifft(fftSegment1(((i-1)*fftLen+1):i*fftLen)...
    .*fftSegment2((fftLen*numOfBlocks-i*fftLen+1)...
    :fftLen*numOfBlocks-(i-1)*fftLen)));
end
```

Listing B.10: fftMulAndAdd function.

B.5 ThrowLast

The `ThrowLast` version can be run with the script *mainThrowLast.m*, shown in listing B.11. It uses the functions `fftMulAndAdd`, `expandOutput` and `generateWeight`, shown in listing B.10, B.5, B.6, respectively. `ThrowLast` also uses transient vectors, called `transVec` and `transVec2` for the respective audio files. They can be generated with the script *transDet2.m* shown in listing B.8.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%mainThrowLast                       %%%%%%%%%%%%%%
%%%%%%%%%%%%%%-----------------------------------%%%%%%%%%%%%%%
%%%%%%%%%%%%%%June 2013                           %%%%%%%%%%%%%%
%%%%%%%%%%%%%%Authors: Antoine Henning Bardoz     %%%%%%%%%%%%%%
%%%%%%%%%%%%%%         Lars Eri Myhre            %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%This script uses the functions fftMulAndAdd, expandOutput,
%generateWeight, which can be found in their .m-files.

%It also uses the vectors transVec and transVec2 which can
%be altered with transDet2.m

%If set, the sound file generated will be played
playSound = 1;

%Different wightmodes exist. They were used in a test which
%failed. The mode 'const' does nothing, but have to be set
%for the code to run.
weightMode = 'const';

%Block length
blockLen = {512, 512};

%Number of blocks4
maxNumOfBlocks = 200;
numOfBlocks = maxNumOfBlocks;
```

```

%Number of maximum concurrent processes
maxNumProc = 200;

%Load test signals
inSig = cell(1,2);
[inSig{1},fs] = wavread('drumloop2.wav');
inSig{2} = wavread('synth.wav');

%Load transient vector,transVec,
%for drumloop2, or use transDet2 to generate
%transVec
load 'drumloop2.mat'
%Don't use transient detection for inSig{2}
transVec2 = zeros(1,length(transVec));

%Convert to Mono
inSig{1} = inSig{1}(:,1)';
inSig{2} = inSig{2}(:,1)';

%Normalize
inSig{1}=inSig{1}/max(inSig{1});
inSig{2}=inSig{2}/max(inSig{2});

%Size of ffts must account for length of convolution
fftLen = blockLen{1}+blockLen{2}−1;

%Initialize blocks
block = cell(1,2);

%Initialize fft blocks
fftBlock = cell(1,2);

%Initialize fft segments
fftSegment = cell(maxNumProc,2);
    fftSegment{1,1} = zeros(1,fftLen);
    fftSegment{1,2} = zeros(1,fftLen);
for i = 2:maxNumProc
```

```
    fftSegment{i,1} = [];
    fftSegment{i,2} = [];
end

%RunningProcVec is a structarray that contains info about
%the number of running processes, it has variables state
%and numOfBlocks for each process. State will be either
%3,2,1 or 0. 3 -> the active process. Append segments with
%newest FT Block pair. 2->semi-active, but not the oldest.
%dont modify segments. 1-> oldest semi-active process, throw
%away oldest FT Block pair. 0->not running.
runningProcVec = struct;

%Initially, one process is active
runningProcVec(1).state=3;
runningProcVec(1).numOfBlocks=0;
%Initialize other processes to zero
for j=2:maxNumProc
    runningProcVec(j).state=0;
    runningProcVec(j).numOfBlocks=0;
end

%Variables that point to the active and
%tossing (oldest sem-active) process
activeProcess=1;
tossingProcess=0;

%Initialize output
output = [zeros(1,fftLen)];

%Generate weight vector
wVector = generateWeight(weightMode,numOfBlocks);

%Iteration variable for main loop
i=1;

%Variable with info on total number of paralell processes
while ~((i+blockLen{1} > length(inSig{1})) ||...
        (i+blockLen{2} > length(inSig{2})))

    %Fill up blocks
    block{1} = inSig{1}(i:i+blockLen{1}−1);
    block{2} = inSig{2}(i:i+blockLen{2}−1);

    %Check if there is a transient somwhere in these blocks
    %If there is a transient, increment number of processes
    if (sum(transVec(i:i+blockLen{1}−1))>=1) ||...
            (sum(transVec2(i:i+blockLen{1}−1))>=1)
        if(runningProcVec(mod(activeProcess+maxNumProc−2,...
                maxNumProc)+1).state == 0)
            %Only one process running. Active process is now
            %oldest semi−active process
            runningProcVec(activeProcess).state=1;
            tossingProcess=activeProcess;
        else
            %Several processes are running.
            %Set active process to semi−active (not oldest)
            runningProcVec(activeProcess).state=2;
        end
        %Start new active process
        activeProcess=mod(activeProcess,maxNumProc)+1;
        runningProcVec(activeProcess).state=3;
        fftSegment{activeProcess,1}=[];
        fftSegment{activeProcess,2}=[];
    end
    if(tossingProcess˜=0 &&...
            runningProcVec(tossingProcess).numOfBlocks==0)
        %tossing process is empty, set oldest process to not
        %running, set second oldest semi−active to oldest
        %semi−active
        if(runningProcVec(mod(tossingProcess,...
                maxNumProc)+1).state˜=3)
            runningProcVec(tossingProcess).state=0;
            tossingProcess=mod(tossingProcess,maxNumProc)+1;
            runningProcVec(tossingProcess).state=1;
        else
```

```
                tossingProcess=0;
            end
        end

        i = max((i+blockLen{1}),(i+blockLen{2}));

        %fft on blocks
        fftBlock{1} = fft(block{1},fftLen);
        fftBlock{2} = fft(block{2},fftLen);

        %Fill up fft segments according to the info in runningProcVec
        for j = 1:maxNumProc
            switch runningProcVec(j).state
                case 3
                    if(runningProcVec(j).numOfBlocks<maxNumOfBlocks)
                        fftSegment{j,1}=[fftSegment{j,1} fftBlock{1}];
                        fftSegment{j,2}=[fftSegment{j,2} fftBlock{2}];
                        runningProcVec(j).numOfBlocks=...
                            runningProcVec(j).numOfBlocks+1;
                    else
                        ...
       fftSegment{j,1}=[fftSegment{j,1}(fftLen+1:end)...
                            fftBlock{1}];
                        ...
       fftSegment{j,2}=[fftSegment{j,2}(fftLen+1:end)...
                            fftBlock{2}];
                    end
                case 1
                    if runningProcVec(j).numOfBlocks
                        runningProcVec(j).numOfBlocks = ...
                            runningProcVec(j).numOfBlocks-1;
                    end
                    if(runningProcVec(j).numOfBlocks==0)
                        runningProcVec(j).state=0;
                    end
                        fftSegment{j,1}=...
                            [fftSegment{j,1}(fftLen+1:end)];
                        fftSegment{j,2}=...
```

```
                                [fftSegment{j,2}(fftLen+1:end)];
                case 2
                    %Do nothing
                otherwise
                    fftSegment{j,1}=[];
                    fftSegment{j,2}=[];
            end
        end

        %temporary vector for output block
        temp = 0;

        %Send processes to fftMulAndAdd, sum up the results in ...
       temp vector
        for j = 1:maxNumProc
            if(runningProcVec(j).state~=0)
                wVector = generateWeight(weightMode,...
                    runningProcVec(j).numOfBlocks);
                temp=temp+(1/runningProcVec(j).numOfBlocks)...
                    *fftMulAndAdd(fftSegment{j,1},fftSegment{j,2},...
                    wVector,runningProcVec(j).numOfBlocks,fftLen);
            end
        end

        %Put output block (temp) on existing output
        output=expandOutput(output,temp,blockLen{1});

    end

    %Play the output
    if playSound
        soundsc(output,fs)
    end
```

Listing B.11: Main script for the `ThrowLast` version.

B.6 TwoProc

The `TwoProc` version can be run with the script *mainTwoProc.m*, shown in listing B.12. It uses the functions `fftMulAndAdd`, `expandOutput` and `generateWeight`, shown in listing B.10, B.5, B.6, respectively. `TwoProc` also uses transient vectors, called `transVec` and `transVec2` for the respective audio files. They can be generated with the script *transDet2.m* shown in listing B.8.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%mainTwoProc                      %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%---------------------------------%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%June 2013                        %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Authors: Antoine Henning Bardoz  %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%         Lars Eri Myhre          %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%This script uses the functions fftMulAndAdd, expandOutput,
%generateWeight, which can be found in their .m-files.

%It also uses the vectors transVec and transVec2 which can
%be altered with transDet2.m

%If set, the sound file generated will be played
playSound = 1;

%Different wightmodes exist. They were used in a test which
%failed. The mode 'const' does nothing, but have to be set
%for the code to run.
weightMode = 'const';

%Block length
blockLen = {512, 512};

%Number of blocks
maxNumOfBlocks = 100;
numOfBlocks = maxNumOfBlocks;
```

```matlab

%Load test signals
inSig = cell(1,2);
[inSig{1},fs] = wavread('drumloop2.wav');
inSig{2} = wavread('synth.wav');

%Load transient vector,transVec,
%for drumloop2, or use transDet2 to generate
%transVec
load 'drumloop2.mat'
%Don't use transient detection for inSig{2}
transVec2 = zeros(1,length(transVec));

%Convert to Mono
inSig{1} = inSig{1}(:,1)';
inSig{2} = inSig{2}(:,1)';

%Normalize
inSig{1}=inSig{1}/max(inSig{1});
inSig{2}=inSig{2}/max(inSig{2});

transVec = [transVec zeros(1,length(inSig{1}))];
transVec2 = zeros(1,length(transVec));

inSig{2} = [inSig{2} inSig{2}];

%Size of ffts must account for length of convolution
fftLen = blockLen{1}+blockLen{2}-1;

%Initialize fftSegments
fftSegmentOld = cell(1,2);
fftSegmentOld{1} = [];
fftSegmentOld{2} = [];

fftSegmentNew = cell(1,2);
fftSegmentNew{1} = zeros(1,fftLen*maxNumOfBlocks);
fftSegmentNew{2} = zeros(1,fftLen*maxNumOfBlocks);

```

```
%Variable with info on numOfBlocks in the segments
currentNumOfBlocksNew = 0;
currentNumOfBlocksOld = 0;

%Initialize output
output = [zeros(1,fftLen)];

%Generate weight vector
wVector = generateWeight(weightMode,numOfBlocks);

i=1;
rundeteller=0;
processCounter = 1;
while ~((i+blockLen{1} > length(inSig{1})) ||...
        (i+blockLen{2} > length(inSig{2})))

    %Fill up blocks
    block{1} = inSig{1}(i:i+blockLen{1}-1);
    block{2} = inSig{2}(i:i+blockLen{2}-1);

    %Fft on blocks
    fftBlock{1} = fft(block{1},fftLen);
    fftBlock{2} = fft(block{2},fftLen);

    if( (currentNumOfBlocksNew+currentNumOfBlocksOld) == ...
    maxNumOfBlocks)
        maxNumBlo = 1;
        %Total number of blocks has reached maxNumOfBlocks. ...
    One block
        %should be thrown away when a new block is added.
        if (sum(transVec(i:i+blockLen{1}-1))>=1) ||...
                (sum(transVec2(i:i+blockLen{1}-1))>=1)
            trans = 1;
            %A transient is detected. Add the blocks in new
            %to old. Put newest block in new. Remove one block
            %from old because total number of blocks has reached
            %maxNumOfBlocks
            fftSegmentOld{1} = [fftSegmentOld{1}(fftLen+1:end) ...
```

```
                fftSegmentNew{1}];
            fftSegmentOld{2} = [fftSegmentOld{2}(fftLen+1:end) ...
                fftSegmentNew{2}];
            fftSegmentNew{1} = [fftBlock{1}];
            fftSegmentNew{2} = [fftBlock{2}];
            currentNumOfBlocksOld = currentNumOfBlocksOld−1+...
                currentNumOfBlocksNew;
            currentNumOfBlocksNew = 1;
        elseif (currentNumOfBlocksOld˜=0)
            trans=0;
            %No transient detected. Remove oldest block in old.
            %Add newest block to new process.
            fftSegmentOld{1} = [fftSegmentOld{1}(fftLen+1:end)];
            fftSegmentOld{2} = [fftSegmentOld{2}(fftLen+1:end)];
            fftSegmentNew{1} = [fftSegmentNew{1} fftBlock{1}];
            fftSegmentNew{2} = [fftSegmentNew{2} fftBlock{2}];
            currentNumOfBlocksOld = currentNumOfBlocksOld − 1;
            currentNumOfBlocksNew = currentNumOfBlocksNew + 1;
        else
            trans=0;
            %No transient detected. Old is empty. Remove oldest
            %block from new and add newest block to new.
            fftSegmentNew{1} = [fftSegmentNew{1}(fftLen+1:end) ...
                fftBlock{1}];
            fftSegmentNew{2} = [fftSegmentNew{2}(fftLen+1:end) ...
                fftBlock{2}];
        end
    else
             maxNumBlo = 0;
            %Total number of blocks is less than maxNumOfBlocks.
            %No blocks should be thrown.
        if (sum(transVec(i:i+blockLen{1}−1))>=1) ||...
                (sum(transVec2(i:i+blockLen{1}−1))>=1)
            trans=1;
            %Transient detected.
            fftSegmentOld{1} = [fftSegmentOld{1} ...
   fftSegmentNew{1}];
```

```
        fftSegmentOld{2} = [fftSegmentOld{2} ...
fftSegmentNew{2}];
        fftSegmentNew{1} = [fftBlock{1}];
        fftSegmentNew{2} = [fftBlock{2}];
        currentNumOfBlocksOld = currentNumOfBlocksOld+...
            currentNumOfBlocksNew;
        currentNumOfBlocksNew = 1;
    else
        trans = 0;
        %No transient detected.
        fftSegmentNew{1} = [fftSegmentNew{1} fftBlock{1}];
        fftSegmentNew{2} = [fftSegmentNew{2} fftBlock{2}];
        %Consider throwing one block if old is active,
        %even if maxNumOfblocks is not reached.
        fftSegmentOld{1} = fftSegmentOld{1}(fftLen+1:end);
        fftSegmentOld{2} = fftSegmentOld{2}(fftLen+1:end);
        currentNumOfBlocksOld;
        currentNumOfBlocksNew = currentNumOfBlocksNew + 1;
    end
end

i = max((i+blockLen{1}),(i+blockLen{2}));
temp = 0;

if(currentNumOfBlocksOld~=0)
        wVector = ...
generateWeight(weightMode,currentNumOfBlocksOld);
        temp=temp+fftMulAndAdd(fftSegmentOld{1},...
            fftSegmentOld{2},wVector,...
            currentNumOfBlocksOld,fftLen);
        wVector = ...
generateWeight(weightMode,currentNumOfBlocksNew);
        temp=temp+fftMulAndAdd(fftSegmentNew{1},...
            fftSegmentNew{2},wVector,...
            currentNumOfBlocksNew,fftLen);
else
      wVector = ...
generateWeight(weightMode,currentNumOfBlocksNew);
```

```
            temp=temp+fftMulAndAdd(fftSegmentNew{1},...
                fftSegmentNew{2},wVector,...
                currentNumOfBlocksNew,fftLen);
    end

    output=expandOutput(output,temp,blockLen{1});%/processCounter;
end

if playSound
    soundsc(output,fs)
end
```

Listing B.12: Main script for the `TwoProc` version.

Appendix C

Transient Detection Algorithm

The code for the algorithm was written mainly by Øyvind Brandtsegg, and is on lines 114-152 in the Csound implementation file, see Appendix A. A flowchart of the functionality is included below for the convenience of the reader.

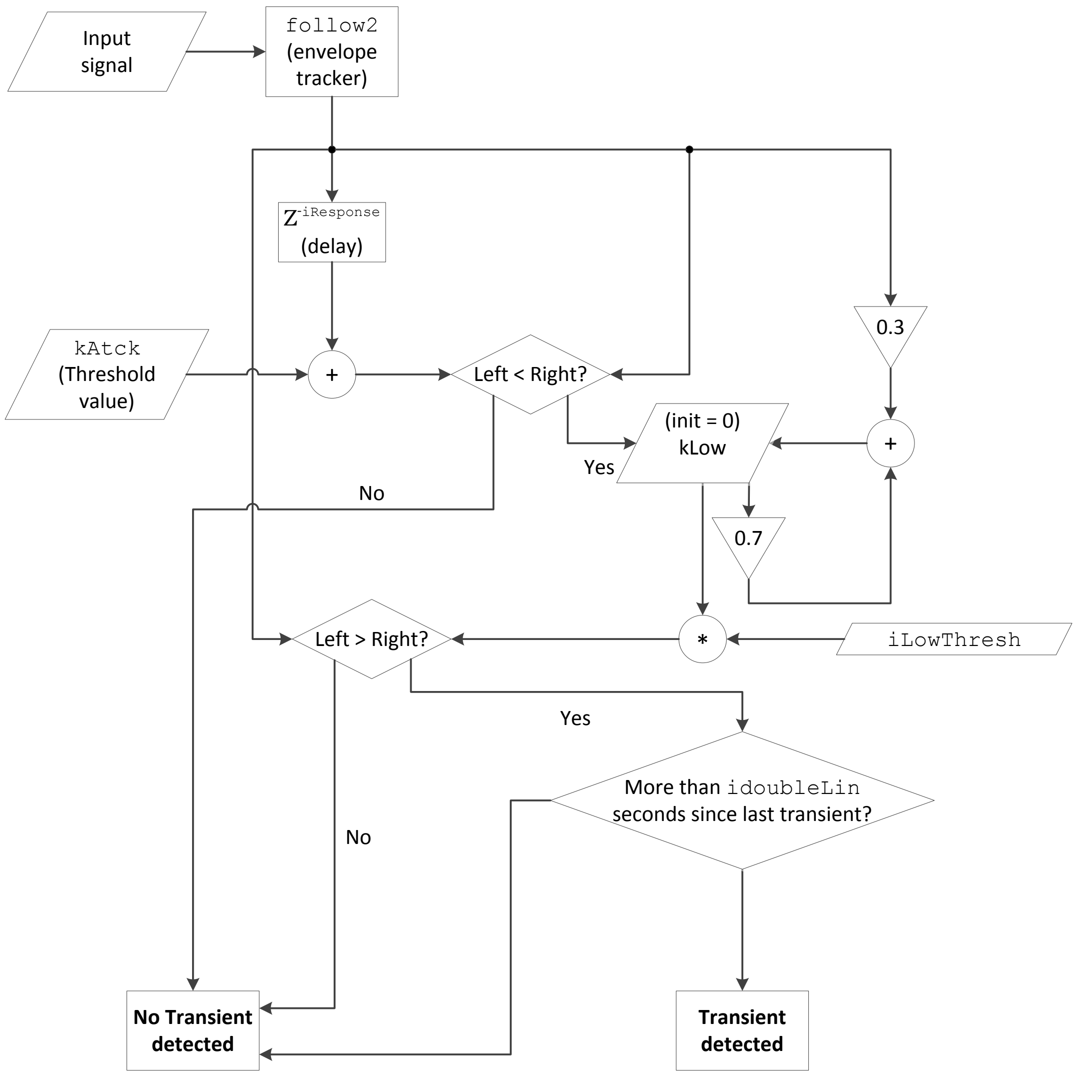


Figure C.1: Flowchart of Transient Detection algorithm.

Index

Bibliography

[1] C. Leider. *Digital Audio Workstation, ISBN 0-07-142286-2*. 2004. ISBN 0-07-142286-2.

[2] T. Engum. Real-time control and creative convolution. *Proceedings og the International Conference on New Interfaces for Musical Expression*, 72(12): 519–522, December 2001. URL http://link.aip.org/link/?RSI/72/4477/1.

[3] Ø. Brandtsegg S. Saue. Experiments with dynamic convolution techniques in live performance. In *Linux Audio Conference*, IEM, Graz, Austria, 2013. Institute of Electronic Music and Acoustics, University for Music and Performing Arts Graz, Austria. URL http://lac.linuxaudio.org/2013.

[4] A. Dominguez-Torres. The origin and history of convolution i: Continuous and discrete convolution operations (last visited in march 2013). 2010. URL http://www.slideshare.net/Alexdfar/origin-adn-history-of-convolution.

[5] J.G. Proakis and D.G. Manolakis. *Digital Signal Processing, Fourth Edition, ISBN 0-13-187374-1*. ISBN 0-13-187374-1.

[6] E. Kreyszig. *Advanced Engineering Mathematics, Ninth Edition, ISBN 978-0-471-72897-9*. ISBN 978-0-471-72897-9.

[7] W.H. Press et. al. *NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING, ISBN 0-521-43064-X*. 1986-1992. ISBN 0-521-43064-X.

[8] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation 19*, pages 297–301, 1965.

[9] K.R. Ramakrishnan B. Thoshkahna, F. X. Nsabimana. A transient detection algorithm for audio using iterative analysis of stft. *12th International Society for Music Information Retrieval Conference (ISMIR 2011)*, 2011. URL http://ismir2011.net/papers/PS2-6.pdf.

[10] T. Maki-Patola and P. Hamalainen. Latency tolerance for gesture controlled continuous sound instrument without tactile feedback. In *Proc. International Computer Music Conference (ICMC)*, 2004.

[11] A. Sæbø P. Svensson S. Farner, A. Solvang. Ensemble hand-clapping experiments under the influence of delay and various acoustic environments. In *121st Convention of the Audio Engineering Society*, 2009.

[12] Mathworks. Matlab homepage (last visited in june 2013). URL http://www.mathworks.se/products/matlab/.

[13] R. Boulanger et. al. *The Csound Book, First Edition, ISBN 0-262-52261-6*. ISBN 0-262-52261-6.

[14] R. Boulanger J. Clements. Csound homepage (last visited in june 2013). URL http://www.csounds.com.

[15] R. Walsh. Cabbage audio plugin framework. In *Proceedings of the International Computer Music Conference*, University of Huddersfield, UK, 2011. URL http://www.icmc2011.org.uk/.

[16] R. Walsh. The cabbage foundation homepage (last visited in june 2013). URL http://thecabbagefoundation.org/.

[17] I. Varga. ftconv (last visited in june 2013), 2005. URL http://www.csounds.com/manual/html/ftconv.html.

[18] L.E. Kinsler et.al. *Fundamentals of Acoustics, Fourth Edition, ISBN 978-0-471-84789-2*. ISBN 978-0-471-84789-2. URL http://www.wiley.com/college.

[19] W.G. Gardner. Efficient convolution without input/output delay. In *97th Convention of the Audio Engineering Society*, San Fransisco, CA, 1994.